Fundamenta Informaticae 57 (2003) 345–370
IOS Press

Monitoring Agents using Declarative Planning [*]

Jürgen Dix[†]
Institut für Informarik, Technische Universität Clausthal
Julius-Albert-Straße 4
D-38678 Clausthal, Germany
dix@tu-clausthal.de

Thomas Eiter, Michael Fink, Axel Polleres
Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria
{*eiter, fink, polleres*}*@kr.tuwien.ac.at*

Yingqian Zhang
Department of Computer Science, University of Manchester
M13 9PL, UK
zhangy@cs.man.ac.uk

Abstract. In this paper we consider the following problem: Given a particular description of a multi-agent system (*MAS*), is it implemented properly? We assume that we are given (possibly incomplete) information about the system and aim at refuting its proper implementation. In our approach, agent collaboration is described as an action theory. Action sequences reaching the collaboration goal are computed by a planner, whose compliance with the actual *MAS* behaviour allows to detect possible collaboration failures. The approach can be fruitfully applied to aid in offline testing of a *MAS* implementation, as well as in online monitoring.

Keywords: knowledge representation, multi agent systems, planning

[*]A preliminary version of this paper appeared in: Proc. 26th Annual German Conference on Artificial Intelligence (KI 2003), LNCS 2821, pages 646–660, Springer, 2003. This work was supported by FWF (Austrian Science Funds), projects P14781, P-16536-N04, 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.

[†]Address for correspondence: Institut für Informarik, Technische Universität Clausthal, Julius-Albert-Straße 4, D-38678 Clausthal, Germany

1. Introduction

Multi-agent systems have been recognised as a promising paradigm for distributed problem solving. Indeed, 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 [22]. A major problem that agent developers face with many platforms is to verify that a suite of implemented agents collaborate well in order to reach a certain goal (e.g., in supply chain management). Tools for automatic verification[1] 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 as intended. 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" with 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 is based on this fact and involves the following steps:

(1) The intended collaborative behaviour of the agents is modelled as a *planning problem*. More precisely, knowledge about the actions performed by the agents (specifically, messaging) and their effects is formalised 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 behaviour, 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 (or user, respectively), pinpointing 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 supports both during testing, and in the operational phase of the system. Among the benefits of this approach are the following:

- It allows to deal with collaboration behaviour 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 in the agent messaging by the system designer may be transparently incorporated to the action theory T, without further need to adjust the monitoring process.

[1]By well-known results, verification is impossible in general, even in simple cases when details of some agents (e.g., in heterogenous environments) are missing.

- Furthermore, T forms 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., determining the messages needed and their order, in a (simple) collaboration.

In the following, we detail our approach and illustrate it with 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. In Section 3 we describe how to model the intended behaviour of a multi-agent system as an abstract planning problem, and instantiate this for our example system using the action language $\mathcal{K}$ [7, 5]. Our approach to agent monitoring is then discussed in Section 4; some fundamental properties are investigated in Section 5. After a brief discussion of the implementation in Section 6 and a review of related works in Section 7, we conclude in Section 8 with an outlook on further research.

2. Message Flow in a Multi-Agent System

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

As opposed to verification, monitoring a *MAS* does not require a complete specification of the behaviour 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 each 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 $\mathsf{A} = \{\mathsf{a}_1, \dots, \mathsf{a}_n\}$ of collaborating agents 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 $\mathtt{send\_msg}$ 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 2.1. (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 $\mathcal{M}_{log} = t_1{:}m_1, t_2{:}m_2, \dots, t_k{:}m_k$ 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

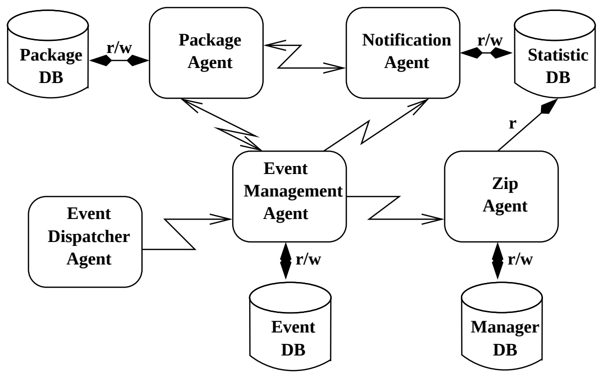

Figure 1. The *Gofish* post office system.

utterance. These commands may range from ask/tell primitives to application specific commands fixed during system specification.

Often, an agent 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 will 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 determine 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 services 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 is informed through 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) that maintains an event database. Information about packages is stored 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, which are stored in a manager database, about zip codes not being well served.

Example 2.1. (Simple *Gofish*)
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, $A = \{\mathsf{pa}, \mathsf{em}, \mathsf{disp}\}$. The messages concerning agent notify will be discussed in the extended version of the example at the end of Section 4.

Agent $\mathfrak{disp}$ informs agent $\mathfrak{em}$ about a package (identified by a unique identifier) being dropped off at the post office, 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 the package by herself: $\mathsf{C}_{\mathfrak{disp}} = \{\mathsf{dropOff}, \mathsf{distCenter}, \mathsf{truck}, \mathsf{delivery}, \mathsf{pickup}\}$. Agent $\mathfrak{em}$ instructs agent $\mathfrak{pa}$ to add a package to the package database after the drop off, as well as to update the delivery time after delivery or customer pickup: $\mathsf{C}_{\mathfrak{em}} = \{\mathsf{addPackage}, \mathsf{setDelivTime}\}$. The package agent here only receives messages, thus $\mathsf{C}_{pa} = \{\}$.

3. Modelling Agent Behaviour via Declarative Planning

We now discuss how to formalise the intended collaborative behaviour of agents as an action theory for planning that encodes a 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 [9], PDDL [11] or HTN [8] 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 modelling control knowledge and plans for agents is Golog [21]. 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 $\mathcal{K}$ [7, 5]. While our approach does not rely on $\mathcal{K}$, we have chosen it because of its declarative nature and its capabilities of dealing with incomplete knowledge and nondeterminism.

3.1. Modelling Intended Behaviour of a *MAS*

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

Step 1: ***Actions (Act).*** Declare a set of *actions* such that corresponding *actions* for each message $m = \langle \mathfrak{s}, \mathfrak{r}, \mathsf{c}, d \rangle$ in our domain, i.e., we have $\mathsf{c}(\mathfrak{s}, \mathfrak{r}, d) \in \mathit{Act}$ (see Def. 2.1). Again, if the message repertoires C_i are pairwise disjoint and each message type c has a unique recipient, we simply write $\mathsf{c}(d)$. 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,[2] $Fl_{\mathfrak{a}}$, of a particular agent $\mathfrak{a}$ 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 behaviour 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* $\mathcal{P} = \langle \mathit{Act}, \mathit{Fl}, T, G \rangle$, where $\mathit{Fl} = \bigcup_{\mathfrak{a} \in A} \mathit{Fl}_{\mathfrak{a}} \cup \mathit{Fl}_{ext},$ whose *solutions* are a set of $\mathcal{P}$*-Plans.* Note that the precise formulation of these notions depends on the underlying

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

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 description is a generic formulation suitable for many planning frameworks. We shall consider planning at an abstract level in these frameworks in Section 5. In the remainder of this section, we turn to the particular planning framework built around the language $\mathcal{K}$.

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 $\mathcal{K}$. Rather than giving a detailed review of the language $\mathcal{K}$ and its planning framework, we describe here for space reasons only the key features and refer to [7, 5] for further details.

The language $\mathcal{K}$ (where $\mathcal{K}$ stands for planning with *knowledge* states) is a member of a family of logic-based action languages in the area of knowledge representation and reasoning. These languages aim at providing a flexible, declarative formalism for reasoning about actions and their effects, on which planning systems might be built. Prominent languages in this family are the languages $\mathcal{A}$ [10] and $\mathcal{C}$ [13]. Compared with these languages $\mathcal{K}$ is closer to logic programming than to classical logic, since it includes respective features (e.g., default negation and strong negation). In a nutshell, $\mathcal{K}$ offers the following distinguishing features:

- *handling incomplete knowledge:* for a fluent f, in a state neither f nor $\neg f$ may be known.
- *nondeterministic effects:* actions may have multiple possible outcomes.
- *optimistic and secure (conformant) planning:* construction of “credulous” plans or “sceptical” plans which work in all cases.
- *parallel actions:* more than one action may be executed simultaneously.

An operational prototype of a planning system for $\mathcal{K}$, $\texttt{DLV}^{\mathcal{K}}$, built as frontend on top of the `DLV` system [4], is available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.

In $\mathcal{K}$, an action domain is defined by the static *background knowledge* BK, which specifies a finite set of static facts through a non-monotonic logic program in a function-free first-order language, and a dynamic action description, $\mathcal{AD}$. Actions and fluents, p, are defined by declarations of the form

$$p(\overline{X})\ \texttt{requires}\ bk_1(\overline{Y_1}), \ldots, bk_m(\overline{Y_m})$$

where $\overline{X} = X_1, \ldots, X_n$ is a list of parameters, each of which must be *typed* by some predicates $bk_1, \ldots,$ bk_m, which are defined in BK. In addition, to specify action executions and effects, $\mathcal{K}$ allows to state axioms of the following forms:

(1) $\texttt{caused}\ f\ \texttt{if}\ \alpha\ \texttt{after}\ \beta.$
(2) $\texttt{total}\ f\ \texttt{if}\ \alpha\ \texttt{after}\ \beta.$
(3) $\texttt{inertial}\ f.$
(4) $\texttt{executable}\ a\ \texttt{if}\ \beta.$
(5) $\texttt{nonexecutable}\ a\ \texttt{if}\ \beta.$

Here f is a fluent literal, a an action, α a set of (possibly default negated) fluent literals, and β is a set of (possibly default negated) actions and fluent literals:

(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, by (4) and (5) we can express under which circumstances an action is *executable* or *nonexecutable*.

A planning problem $\mathcal{P}^{\mathcal{K}}$ in $\mathcal{K}$ may then be formalised, according to the general schema from Section 3.1 above, 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 sorts introduced above), and *G* is the goal, i.e. a set of ground fluent literals.

The semantics of $\mathcal{K}$ is defined through *transitions* $t = \langle s, A, s'\rangle$ from states s to states s' by simultaneous execution of a actions A, where a *state* s is any consistent set of ground fluent literals.[3] Roughly, the action description yields a non-monotonic logic program which computes the possible successor states s' from s and A in its models.

A *trajectory* Tr is then any initial state s_0 (which must comply with the integrity constraints in the planning theory) or sequence $t_1, \ldots, t_n$ of transitions $t_i = \langle s_{i-1}, A_i, s_i\rangle$, $i \in \{1, \ldots, n\}$, starting in an initial state s_0.[4] An *(optimistic) plan* for goal G is $P = \langle\rangle$, or the projection $P = \langle A_1, \ldots, A_n\rangle$ of a trajectory Tr, such that G holds in s_0, or s_n, respectively.

Example 3.1. (Simple *Gofish* cont'd)
In the *Gofish* example, the following $\mathcal{K}$ actions (corresponding to the possible messages) and fluents are defined (in DLV$^{\mathcal{K}}$ notation [5]):

```
actions : dropOff(P) requires pkg(P).                    )
          addPkg(P) requires pkg(P).                     |
          distCenter(P) requires pkg(P).                 |
          truck(P) requires pkg(P).                      } Act
          delivery(P) requires pkg(P).                   |
          pickup(P) requires pkg(P).                     |
          setDelivTime(P) requires pkg(P).               )
fluents : pkgAt(P,Loc) requires pkg(P),loc(Loc).         )
          delivered(P) requires pkg(P).                  |
          recipAtHome(P) requires pkg(P).                } Fl
          added(P) requires pkg(P).                      |
          delivTimeSet(P) requires pkg(P).               )
```

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

A possible package (e.g., a generic p_1) and its locations form the 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 DLV$^{\mathcal{K}}$ notation) as follows:

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

[4] In [7, 5], states and transitions occurring in possible trajectories are called *legal*, in order to distinguish them from other transitions which are meaningless for executions and plans wrt. to the domain of discourse. Here, we just omit this distinction.

```
initially : recipAtHome(p1).
always :    noConcurrency.
 inertial pkgAt(P, Loc).
 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 the truck is already on the way. Finally, the goal is $G = \texttt{delivTimeSet}(p_1)$.

The following (optimistic) plans reach G:

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

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: We assume the following entries in the message log $\mathcal{M}_{log}$ = 0:$\langle \mathit{disp}, \mathit{em}, \text{dropOff}, p_1\rangle$, 5:$\langle \mathit{em}, \mathit{pa}, \text{addPackage}, p_1\rangle$, 13:$\langle \mathit{disp}, \mathit{em}, \text{distCenter}, p_1\rangle$, 19:$\langle \mathit{disp}, \mathit{em}, \text{truck}, p_1\rangle$, 20:$\langle \mathit{disp}, \mathit{em},$

$\mathsf{pickup}, p_1\rangle$. According to the message history in $\mathcal{M}_{log}$, we can 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 ($\mathtt{monitor}$) is to *aid in 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 4.1. (Structure of the monitoring agent)
The agent $\mathtt{monitor}$ loops through the following steps:

1. Read and parse the message log $\mathcal{M}_{log}$. If $\mathcal{M}_{log} = \emptyset$, the set of all possible 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 on these tasks.

Checking *MAS* behaviour: $\mathtt{monitor}$ continually keeps track of the *messages sent between the agents*. They are stored in the message log, $\mathcal{M}_{log}$, which is accessible by $\mathtt{monitor}$. Thus for $\mathtt{monitor}$, the behaviour 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 $\mathtt{em}$, $\mathtt{disp}$, $\mathtt{pa}$ are given by $\mathsf{C}_{\mathtt{em}}$, $\mathsf{C}_{\mathtt{disp}}$, and $\mathsf{C}_{\mathtt{pa}}$ (see Section 2).

Intended behaviour and compatibility: The desired collaborative *MAS* behaviour is formalised 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 $\mathtt{monitor}$ is exactly doing that.

In general, not all $\mathcal{P}$-Plans may be admissible, as constraints may apply (derived from the intended collaborative behaviour).[5] E.g., some actions ought to be taken in a fixed order, or actions may be penalised 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: $\mathtt{monitor}$ can pinpoint the precise place indicating which messages have 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 behaviour is due to a coding or design error. However, the info given by $\mathtt{monitor}$ will aid the agent designer in detecting the real cause.

Messages from $\mathtt{monitor}$: Agent $\mathtt{monitor}$ 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), $\mathtt{monitor}$ 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 unreachable, (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* behaviour 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 that 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; $\mathtt{monitor}$ detects this and gives a warning. Inspection of the actual code may show that the command of 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 $\mathtt{monitor}$, this flaw 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 makes 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.

Only plan P_1 remains to reach the goal. However, there is *no guarantee of success*, if Sue is not back home in time. This situation can be easily captured in the framework of [7, 5], where we have the notion of a *secure* plan. An (optimistic) plan is *secure* (or *conformant* [14]), 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_2 and P_3 are secure plans, while P_1

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

is not secure. Thus, a design error is detected, if delivering the package must be guaranteed under all circumstances.

Based on a generic planning problem $\mathcal{P}$, we now define compatible plans as follows. Let $\mathcal{P}$-Plans denote the set of all plans for $\mathcal{P}$.

Definition 4.2. ($\mathcal{M}_{log}$ compatible plans)
Let the planning problem $\mathcal{P}$ model the intended behaviour 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 $m_1, \ldots, m_n$.

In a planning framework with different notions of plans, $\mathcal{P}$-Plans is assumed to comprise the most liberal notion of plan. For example, in DLV$^{\mathcal{K}}$, the planner for $\mathcal{K}$, optimistic and secure plans can be computed for any problem $\mathcal{P}^{\mathcal{K}}$, and $\mathcal{P}$-Plans would consist of all optimistic plans.

Definition 4.3. (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 $\langle t_n{:}m_n, \textit{idle}\rangle$ if (i) applies and $\langle t_n{:}m_n, \textit{timeout}\rangle$ (resp. $\langle \textit{timeout}\rangle$ if $\mathcal{M}_{log}$ is empty) otherwise.

Initially, $\mathcal{M}_{log}$ is empty and thus C-Plans$(\mathcal{P}, \mathcal{M}_{log}, 0) =$ 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. Agent $\mathtt{monitor}$ can thus check 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 the problematic action.

Running scenario: Under guaranteed delivery (i.e., under secure planning), agent $\mathtt{monitor}$ writes Culprit$(\mathcal{M}_{log}, \mathcal{P})$ = $\langle 20{:}m_5, \textit{idle}\rangle$ (the $pickup(p_1)$ message) in 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 $\mathtt{monitor}$ would not detect a problem at this stage.

Example 4.1. (Simple *Gofish* extended)
We now consider the extension of the previous simple *Gofish* example by adding a customer notification service. That is, the *Gofish* postal service performs mail tracking in order to be able to notify customers about the status of mail delivery. Each recipient of a package is notified about the arrival of the package at the distribution center and when the package has been loaded on a truck for delivery.

The realisation of the notification service in our *MAS* brings the notification agent ($\mathtt{notify}$) into play. The notification agent is informed by the event management agent ($\mathtt{em}$) about the arrival of a package, as well as its loading on a truck. In both cases agent $\mathtt{notify}$ contacts the package agent ($\mathtt{pa}$) in order to obtain the required customer information. For simplicity, we subsume both messages – from $\mathtt{em}$ to $\mathtt{notify}$ and from $\mathtt{notify}$ to $\mathtt{pa}$ – into a single message getRecipInfo which is parameterised by the corresponding event `dist` or `truck`, i.e., $\mathsf{C}_{\mathtt{notify}} = \{$getRecipInfo$\}$. The package agent replies to the requests of $\mathtt{notify}$ with the corresponding information to notify the customer, e.g., the email address of the recipient in case of `dist` and the phone number in case of `truck`. Thus now, $\mathsf{C}_{\mathtt{pa}} = \{$recipInfo$\}$.

In order to reflect this extension in our model, we add the following actions for the newly introduced messages and another fluent:

```
actions : getRecipInfo(P,Loc) requires pkg(P), loc(Loc).
          recipInfo(P,Loc) requires pkg(P), loc(Loc).
```

```
fluents : informed(P,Loc) requires pkg(P), loc(Loc).
```

The new external fluent `informed` captures the state of the customer concerning her knowledge about the package status. The effects and executability conditions of the new actions are defined as follows:

```
inertial informed(P,Loc).
inertial -informed(P,Loc).

executable getRecipInfo(P,dist) if pkgAt(P,dist).
executable getRecipInfo(P,truck) if pkgAt(P,truck).
caused -informed(P,Loc) after getRecipInfo(P,Loc).
nonexecutable getRecipInfo(P,Loc) if informed(P,Loc).
nonexecutable getRecipInfo(P,Loc) if -informed(P,Loc).

executable recipInfo(P,Loc) if pkgAt(P,Loc), -informed(P,Loc).
caused informed(P,Loc) after pkgAt(P,Loc), recipInfo(P,Loc).
nonexecutable recipInfo(P,Loc) if informed(P,Loc).
```

Furthermore, we modify the axioms for the delivery and the pickup action:

```
executable delivery(P) if pkgAt(P,truck), informed(P,truck),
                          not delivered(P).

executable pickup(P) if pkgAt(P,dist), not informed(P,truck),
                        not delivered(P).
```

The customer must be informed about the package being delivered by a truck before delivery, thus she will no longer show up at the distribution center for picking up the package.

In general, because of the “nondeterminism” of the external event of a customer showing up at a distribution center for picking up a package, we will now obtain more plans that reach the goal G=`delivTimeSet`(p_1). For example, the following (optimistic) plans are variants of the plan P_1:

$$P_{1,1} = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist}); \texttt{truck}(p_1); \texttt{getRecipInfo}(p_1, \text{truck}); \texttt{pickup}(p_1); \texttt{recipInfo}(p_1, \text{truck}); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle$$

$$P_{1,2} = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist}); \texttt{truck}(p_1); \texttt{pickup}(p_1); \texttt{getRecipInfo}(p_1, \text{truck}); \texttt{recipInfo}(p_1, \text{truck}); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle$$

Note that still the customer may show up at the distribution center after the package has been loaded on a truck. However, this can no longer be the case after the customer has been notified. Moreover, it is assumed that the customer notification takes place while or shortly after loading the package on the truck.

Running scenario: Consider a run of our scenario in the extended *MAS* and suppose the following sequence of messages in the message log $\mathcal{M}_{log}$: $m_1 = \langle$dropOff, $p_1\rangle$, $m_2 = \langle$addPackage, $p_1\rangle$, $m_3 = \langle$distCenter, $p_1\rangle$, $m_4 = \langle$getRecipInfo, p_1, dist$\rangle$, $m_5 = \langle$recipInfo, p_1, dist$\rangle$, $m_6 = \langle$truck, $p_1\rangle$, $m_7 = \langle$getRecipInfo, p_1, truck$\rangle$, and $m_8 = \langle$recipInfo, p_1, truck$\rangle$. Then, everything is fine even under secure planning, i.e. guaranteed delivery, since pickup cannot occur after Sue has been notified that her package has been loaded on truck for delivery. That is, $\mathcal{M}_{log}$ is compatible with the secure plan

$$P_{2,1} = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist}); \texttt{truck}(p_1); \texttt{getRecipInfo}(p_1, \text{truck}); \texttt{recipInfo}(p_1, \text{truck}); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle.$$

However, if Sue had shown up at the distribution center before notification, i.e. m_8 = $\langle$pickup, $p_1\rangle$ say again at time 20, the package would no longer be guaranteed delivered on time, and monitor would again write Culprit($\mathcal{M}_{log}, \mathcal{P}$) = $\langle$20:m_8, *idle*$\rangle$ to a file to indicate the problematic situation.

To sum up, by extending our postal service with customer notification we reduced the probability of unsuccessful deliveries but we did not achieve guaranteed delivery. We remark, however, that in our example setting this could easily be obtained by disallowing customer pickups.

5. Properties

In this section, we take a closer look at our agent monitoring approach and show that it has some desirable properties. To this end, we shall need some preliminary definitions in order to make the notions we have used above formally more precise.

As for the underlying planning framework, referred to as $\mathcal{PF}$, we assume that the basic element for the semantics of plan execution is given by trajectories in the planning world formed by state transitions, similar to the semantics of the planning language $\mathcal{K}$. That is, we assume that there is a set of possible world states, $S_{\mathcal{PF}}$ (where world states s are described e.g. by fluents) and a set of actions $A_{\mathcal{PF}}$ in $\mathcal{PF}$, as well as a set $I_{\mathcal{PF}} \subseteq S_{\mathcal{PF}}$ of initial states. Furthermore, there is a partial, multi-valued transition function $tr_{\mathcal{PF}} : S_{\mathcal{PF}} \times A_{\mathcal{PF}} \to 2^{S_{\mathcal{PF}}}$ which assigns a set of possible successor states $tr_{\mathcal{PF}}(s, A)$ to a state $s \in S_{\mathcal{PF}}$ and an action $a \in A_{\mathcal{PF}}$ to be executed in s; the transition $tr_{\mathcal{PF}}$ might be undefined, however, or no successor state may exist.

Definition 5.1. A *trajectory* T in $\mathcal{PF}$ is a sequence $s_0, a_1, s_1, \ldots, a_n, s_n$ of states $s_i \in S_{\mathcal{PF}}$ and actions $a_i \subseteq a_{\mathcal{PF}}$, $n \geq 0$, such that $s_0 \in I_{\mathcal{PF}}$ and $s_i \in tr_{\mathcal{PF}}(s_{i-1}, a_i)$, for every $i \in \{1, \ldots, n\}$.

We view plans for reaching a goal G in $\mathcal{PF}$, which in general is some constraint on the desired states, from a semantical perspective, as structures corresponding to the trajectories in the planning world which are compatible with them. More formally,

Definition 5.2. Any plan P in $\mathcal{PF}$ is an object which has associated with it a nonempty set of trajectories in $\mathcal{PF}$, $Sem(P)$, such that G holds in state s_n for each $T \in Sem(P)$ where $T = s_0, a_1, s_1, \ldots, a_n, s_n$.

By way of illustration, in the $\mathcal{K}$ planning framework, the transition function $tr_{\mathcal{K}}$ is implicit by the definition of state transitions, viz. $tr_{\mathcal{K}}(s, A)$ is defined for a state s and a set of actions A (which we can view as a single compound action) iff, in the terminology of $\mathcal{K}$, A is executable wrt. s and $tr_{\mathcal{K}}(s, A) = \{s' \mid \langle s, A, s'\rangle$ is a state transition$\}$ in this case. An optimistic plan P in $\mathcal{K}$ for the goal G is then semantically characterised by the condition that there is a sequence of actions, $\langle A_1, \ldots, A_n\rangle$, $n \geq 0$, such that (i) each $T \in Sem(P)$ is of form $s_0, A_1, s_1, \ldots, A_n, s_n$ and (ii) each trajectory $s_0, A_1, s_1, \ldots, A_n, s_n$ which establishes the goal G is in $Sem(P)$. Furthermore, a secure plan P is in $\mathcal{K}$ an optimistic plan which satisfies in addition that (iii) for each trajectory $T' = s_0, A_1, s_1, \ldots, A_m, s_m$ with $m \leq n$, the goal G is established if $m = n$, and $tr(s_m, A_{m+1})$ is defined and nonempty otherwise.

As for the multi-agent system M in question, we assume that its collaborative behaviour is governed by some strategy, $\mathcal{S}$. We take here also a pure semantical view and project $\mathcal{S}$ to the set of possible runs which might be observed during execution (in particular, by the agent tester). Formally, a run is defined as follows.

Definition 5.3. Given a *MAS* M, we assume there is an underlying set of possible system states, S_M. A *run* is a sequence $R = S^0, m_1, S^1, \ldots, m_k, S^k$, $k \geq 0$, of (global) system states S^i and messages m_j, where S^0 is the initial state (resp., from the set of possible initial states in case of indeterminism).

Informally, upon message m_i, the system transits from state S^{i-1} to state S^i; here, we abstract from concrete time points when messages are sent. We denote the set of possible runs under obedience of strategy $\mathcal{S}$ by $Runs(\mathcal{S})$, which is assumed to be nonempty. Each such run R must establish the collaboration goal, which is assumed to be expressed by a success-predicate on global states (that is, S^k must satisfy it given $R = S^0, m_1, S^1, \ldots, m_k, S^k$).

In order to account for the case that we do not know the precise collaboration strategy $\mathcal{S}$ adopted by M (e.g., this could be negotiated in a startup phase), we model the intended behaviour by a nonempty set $\mathcal{IS}(M)$ of possible strategies $\mathcal{S}$. A run R is *admissible*, if it is possible for some $\mathcal{S} \in \mathcal{IS}(M)$, i.e., $R \in \bigcup_{\mathcal{S} \in \mathcal{IS}(M)} Runs(\mathcal{S})$.

The planning framework, $\mathcal{PF}$, and the *MAS* M are linked by the basic *Modelling Assumption* that runs in M gracefully correspond to trajectories in $\mathcal{PF}$ and vice versa such that $\mathcal{PF}$ models evolutions in M, and that the planning goal corresponds to the collaboration goal. This is made precise as follows.

Modelling Assumption:

1. There is a one-to-one correspondence, f, between messages m and actions a, $f(m) = a$, and a correspondence, g, (not necessarily one-to-one) between states $S \in S_M$ in the agent system and states $s \in S_{\mathcal{PF}}$ in the planning framework, $S \overset{g}{\rightleftharpoons} s$, such that the initial states in M and $\mathcal{PF}$ correspond among each other;
2. there is a fixed planning goal, G, defining a planning problem, $\mathcal{P}$, such that the states in $\mathcal{PF}$ fulfilling G and the states in M establishing the collaboration goal correspond among each other; and
3. the correspondence homomorphically extends to transitions in runs in M and transitions in trajectories in $\mathcal{PF}$, respectively. That is, for any S^{i-1}, m_i, S^i in a run, we have that $g(S^{i-1}), f(m_i), g(S^i)$ is part of a trajectory in $\mathcal{PF}$, and conversely, for any s_{i-1}, a_i, s_i in a trajectory, $g^{-1}(s_{i-1}), f^{-1}(a_i), g^{-1}(s_i)$ is part of a run of M.

Conditions 2 and 3 of the Modelling Assumption aim at allowing abstraction in the encoding of the multi-agent system in the planning formalism; note that no one-to-one correspondence between trajectories and runs is requested. For example, fluents in the multi-agent system might be disregarded in the planning formulation, such that states in the multi-agent system with different fluents values correspond to the same state in the planning world. On the other hand, the planning formulation might include fluents which do not correspond to fluents in the multi-agent system and whose value is immaterial for the intended monitoring task. These fluents can be projected away, leading to a possible many-to-one correspondence from states in the planning world to states of the multi-agent system. We emphasise that some assumption on the correspondence between, on the one hand, states and runs in the multi-agent system and, on the other hand, states and trajectories is mandatory for proving meaningful results.

We shall denote the solutions (plans) for the planning problem $\mathcal{P}$ by $\mathcal{P}$-Plans. Furthermore, we shall occasionally simply write $m \rightleftharpoons A$, $S \rightleftharpoons s$, and $R \rightleftharpoons T$ for appropriate objects corresponding via g and f.

The correspondence $\rightleftharpoons$ induces a notion of similarity $\simeq_{\rightleftharpoons}$ (for short, $\simeq$) among runs by $R \simeq_{\rightleftharpoons} R'$ if and only if there is some trajectory T in $\mathcal{PF}$ such that $R \rightleftharpoons T$ and $R' \rightleftharpoons T$. In order to get meaningful results, we assume that collaboration strategies $\mathcal{S}$ are closed under similarity $\simeq$; that is, whenever $R \in Runs(\mathcal{S})$ and $R \simeq R'$, then also $R' \in Runs(\mathcal{S})$ holds. As a consequence, each trajectory corresponds either only to admissible runs or to no admissible run.

After these preliminary definitions, our first result concerns the soundness of the monitoring approach. Let us say that a plan $P \in \mathcal{PF}$ *models* a strategy $\mathcal{S}$, iff each run $R \in Runs(\mathcal{S})$ corresponds to some trajectory $T \in Sem(P)$ and vice versa, and that a set $MP = \{P_i \mid i \in I\}$ of plans models a set of strategies $\mathcal{SS} = \{\mathcal{S}_i \mid i \in I\}$, if each P_i models $\mathcal{S}_i$.

Theorem 5.1. (Soundness)
Suppose that the set $\mathsf{I\text{-}Plans}(\mathcal{P}) \subseteq \mathcal{P}\text{-}\mathsf{Plans}$ of intended plans for the planning problem $\mathcal{P}$ in $\mathcal{PF}$ models the intended collaborative behaviour of the *MAS* M, $\mathcal{IS}(M)$. Let $\mathcal{M}_{log}$ be a message log. Then, M is implemented incorrectly if $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists.

Proof:
Let $R = S^0, m_1, S^1, \ldots, m_k, S^k$ be the run which produces $\mathcal{M}_{log} = t_1{:}m_1, \ldots, t_k{:}m_k$. Consider the two different types of $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$. Suppose first that it is of form $\langle t_n{:}m_n, \mathit{timeout}\rangle$ or $\langle \mathit{timeout}\rangle$. Then, a time-out has been detected and we have $n = k$ or $k = 0$, respectively. This means that either R has terminated or that M is stuck. The monitor agent expects, supported by a trajectory $T \in Sem(P)$ for some $P \in \mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n)$ ($\neq \emptyset$), such that T is compatible with the messages $m_1, \ldots, m_n$ in $\mathcal{M}_{log}$, that the execution of M continues, i.e., some message m_{n+1} follows. Hence in both cases (whether M is terminated or stuck), $R \notin \bigcup_{\mathcal{S} \in \mathcal{IS}(M)} Runs(\mathcal{S})$. Hence, M is not implemented correctly.

Suppose next that $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P}) = \langle t_n{:}m_n, \mathit{idle}\rangle$. Then, we have $0 < n \leq k$. By the definition of culprit, we have that $\mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, n) = \emptyset$. This means that there is no trajectory $T = s_0, a_1, s_1, \ldots, a_{k'}, s_{k'}$ in the planning framework $\mathcal{PF}$ such that $T \in Sem(P)$ for some $P \in \mathsf{I\text{-}Plans}(\mathcal{P})$ with the property that the prefix $T' = s_0, a_1, s_1, \ldots, a_n, s_n$ of T and the prefix $R' = S^0, m_1, S^1, \ldots, m_n, S^n$ of R satisfy $R' \rightleftharpoons T'$. Hence, $R \notin \bigcup_{\mathcal{S} \in \mathcal{IS}(M)} Runs(\mathcal{S})$, which again means that M is not implemented correctly. This proves soundness. $\Box$

The soundness result of the monitoring approach can be generalised to a setting in which the intended collaborative behaviour $\mathcal{IS}(M)$ of the agents is not exactly modelled by some intended plans in the planning framework, but just cautiously approximated. This is in particular useful if the strategies governing the collaborative behaviour in the *MAS* M amount to an expressive notion of plans.

For example, the *MAS* might employ a conditional plan P^M [24] in which depending on conditions $c_1, \ldots, c_k$ on the current state, suitable actions $a_1, \ldots, a_k$ are executed, respectively, and a similar strategy is recursively applied on each case. Conditional plans are very important, since sensing information (observations from the world) can be suitably respected. They are more liberal than secure plans, which do not allow for branching on conditions.

However, the planning framework $\mathcal{PF}$ which we employ for agent monitoring might not be capable of conditional planning, such that we can not model P^M as a respective intended plan; for example, the $\mathcal{K}$ planning framework does not support conditional planning. Despite this obstacle, we might employ the planning framework $\mathcal{PF}$ fruitfully for error detection as follows.

Definition 5.4. Given a planning framework $\mathcal{PF}$ and a *MAS* M, we say that a set CP of plans in $\mathcal{PF}$ *covers* the intended collaborative behaviour of M, $\mathcal{IS}(M)$, if for each run $R \in \bigcup_{\mathcal{S}\in\mathcal{IS}(M)} Runs(\mathcal{S})$, there exists some plan $P \in CP$ and trajectory $T \in Sem(P)$ such that $R \rightleftharpoons T$.

Agent $\mathtt{monitor}$ then uses in Step 3 of its procedure from Definition 4.1 the cover CP instead of the intended plans $\textsf{I-Plans}(\mathcal{P})$. We then write $\textsf{I-Plans}(CP)$, $\textsf{C-Plans}(CP, \mathcal{M}_{log}, n)$, $\textsf{Culprit}(\mathcal{M}_{log}, CP)$ etc.

We have the following result:

Theorem 5.2. (Soundness of Covering)
Suppose the planning problem $\mathcal{P}$ in $\mathcal{PF}$ is such that its accepted solutions, $CP \subseteq \mathcal{P}^{\mathcal{K}}\textsf{-Plans}$, cover the intended collaborative behaviour of the *MAS* M, given by $\mathcal{IS}(M)$. Let $\mathcal{M}_{log}$ be a message log. Then, the *MAS* is implemented incorrectly if $\textsf{Culprit}(\mathcal{M}_{log}, CP)$ exists.

Proof:
The proof is similar to the proof of Theorem 5.1, where $\textsf{I-Plans}(\mathcal{P})$ is replaced with CP. □

As an immediate corollary, we obtain soundness of agent $\mathtt{monitor}$ via optimistic plans in the $\mathcal{K}$ planning framework:

Corollary 5.1. (Soundness of $\mathcal{P}^{\mathcal{K}}$ Cover)
Let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem, such that the set $OP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of optimistic plans covers the intended collaborative behaviour of the *MAS* M. Let $\mathcal{M}_{log}$ be a message log. Then, *MAS* is implemented incorrectly if $\textsf{Culprit}(\mathcal{M}_{log}, OP)$ exists.

In particular, if nothing is known about the collaboration strategy of M, the set OP might safely be set to $\mathcal{P}^{\mathcal{K}}\textsf{-Plans}$, i.e., all optimistic plans. Then, any behaviour will be covered, including intended behaviour governed by a conditional plan, or by a more restrictive secure plan.

As for completeness of the monitoring method, there is clearly no converse of the soundness result for covers CP of the intended behaviour in general, since CP might include a plan P which has an associated trajectory that masks an inadmissible run of the *MAS* M; this is the price to pay for overestimating the intended behaviour.

On the other hand, if all trajectories of plans in the cover CP correspond to admissible runs of the *MAS*, then the cover allows to unveil an incorrect *MAS* implementation, provided certain conditions are met.

As for a general completeness result, we adopt the following assertions. The first is that successful runs can not grow arbitrarily long, i.e., they have a (known) upper bound on their length. The second assertion concerns the evolution of the *MAS* with respect to the particular mechanism of message logging we build on, which does not foresee recording state information about M. From the messages in $\mathcal{M}_{log} = t_1{:}m_1, \ldots, t_k{:}m_k$ alone, it is in general impossible to infer the state of the agent system M. We thus assert for M the property that any runs $R = S^0, m_1, S^1, \ldots, m_k, S^k$ and $\bar{R} = \bar{S}^0, m_1, \bar{S}^1, \ldots, m_k, \bar{S}^k$ are similar, i.e., $R \simeq \bar{R}$ holds; we say that M has *one-way logging*. For example, one-way logging is guaranteed in agent systems with deterministic message effects and a single initial state.

Let us call a cover CP for $\mathcal{IS}(M)$ *exact*, if for each $P \in CP$ and each $T \in Sem(P)$, there exists some strategy $\mathcal{S} \in \mathcal{IS}(M)$ and run $R \in Runs(\mathcal{S})$ such that T corresponds to R.

Theorem 5.3. (Completeness)
Let the planning problem $\mathcal{P}$ in $\mathcal{PF}$ be such that the set $CP \subseteq \mathcal{P}$-plans of selected plans exactly covers the intended collaborative behaviour of a *MAS* M, given by $\mathcal{IS}(M)$, where all admissible runs are bounded. If M has one-way logging and is implemented incorrectly, then either (i) $CP = \emptyset$ or (ii) there is some message log $\mathcal{M}_{log}$ such $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists.

Proof:
Suppose that M is incorrectly implemented. That is, the intended collaborative behaviour is violated, and there is a run $R = S^0, m_1, S^1, \ldots, m_k, S^k$ witnessing this fact, i.e., $R \notin \bigcup_{\mathcal{S} \in \mathcal{IS}(M)} Runs(\mathcal{S})$. If k exceeds the length bound, then any trajectory T such that $R \rightleftharpoons T$ exceeds the length bound as well, and thus $\mathsf{C\text{-}Plans}(\mathcal{P}, \mathcal{M}_{log}, k) = \emptyset$ must hold; hence, $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists in this case. Thus, for the rest assume that k is within the limit. Let $\mathcal{M}_{log} = t_1{:}m_1, \ldots, t_k{:}m_k$ be the message log produced by R; notice that because of one-way logging, $\mathcal{M}_{log}$ is produced only by runs R' such that $R' \simeq R$. Towards a contradiction, suppose that $CP \neq \emptyset$ and $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ is not found by agent $\mathtt{monitor}$. Thus, there is no time-out detected (and thus M is not judged terminated or stuck), and there must exist some trajectories $T \in Sem(P)$ for some plans $P \in CP$ of form $T = s_0, m_1, s_1, \ldots, m_k, s_k$. By the Modelling Assumption, there exists some runs R' of M such that $R' \rightleftharpoons T$. Since CP is an exact cover, at least one R' among them is admissible, i.e., $R' \in Runs(\mathcal{S})$ for some $\mathcal{S} \in \mathcal{IS}(M)$. Since the correspondence f between action sets and messages is one-to-one and M has one-way logging, it follows that $R' \simeq R$. However, by closure of strategies under $\simeq$, it follows $R \in Runs(\mathcal{S})$ and thus $R \in \bigcup_{\mathcal{S} \in \mathcal{IS}(M)} Runs(\mathcal{S})$, which is a contradiction. □

In particular, the above theorem holds if CP models the intended behaviour $\mathcal{IS}(M)$ (i.e., $CP = \mathsf{I\text{-}Plans}(\mathcal{P})$). In (i), we can conclude a design error, while in (ii) a design or coding error may be present. Again, we obtain an easy corollary for the $\mathcal{K}$ planning framework:

Corollary 5.2. (Completeness of Exact $\mathcal{P}^{\mathcal{K}}$ Cover)
Let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem, such that the set $OP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of optimistic plans exactly covers the intended collaborative behaviour of the *MAS* M. If M has one-way logging and is implemented incorrectly, then either (i) $CP = \emptyset$ or (ii) there is some message log $\mathcal{M}_{log}$ such $\mathsf{Culprit}(\mathcal{M}_{log}, \mathcal{P})$ exists.

Notice that Theorem 5.3 allows us to detect incorrectness despite a mismatch of the structure of strategies $\mathcal{S}$ used in M and the structures of plans supported in $\mathcal{PF}$.

We may dispose some of the assertions if the strategies used in the *MAS* satisfy certain properties. An example is the case in which the collaborative behaviour is governed by a *conformant strategy* $\mathcal{S}$, which means that $\mathcal{S}$ semantically corresponds to a conformant (i.e., secure) plan; that is, starting from any possible initial state, the same sequence $m_1, m_2, \ldots, m_n$ of messages is expected to appear (and always lead to success), regardless of how the global state evolves. If we model the intended behaviour by secure plans in the planning framework, then we can drop the one-way logging assertion. We formulate the result here for the particular formalism $\mathcal{K}$, for which we described secure plans more detailed after Definition 5.2 above. In fact, it turns out that exact covers are tantamount to secure plans.

Lemma 5.1. Let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem, such that the set $CP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of optimistic plans exactly covers the intended collaborative behaviour $\mathcal{IS}(M)$ of M. Suppose that each $\mathcal{S}$ in $\mathcal{IS}(M)$ is conformant. Then, each plan $P \in CP$ is secure.

Proof:
Indeed, suppose $P \in CP$ is not secure. Every trajectory $T \in Sem(P)$ corresponds to some admissible run R of a strategy $\mathcal{S} \in \mathcal{IS}(M)$, and some such R and $\mathcal{S}$ must exist. On the other hand, insecurity of $P = \langle a_1, a_2, \ldots, a_n \rangle$ implies that there exists some trajectory $T' = s_0, a_1, s_1, \ldots, a_m, s_n$ in $\mathcal{P}^{\mathcal{K}}$ violating condition (iii) for a secure plan above. That is, either $m = n$ and the goal is not established, or $m < n$ and $tr(s_m, a_{m+1})$ is either undefined or empty. By the Modelling Assumption, T' corresponds to a run R' in M which does not establish the collaboration goal at termination of M or that M is stuck. Since R' and R have the same message sequence $m_1, \ldots, m_k$ (where $f(m_i) = a_i$, $1 \leq i \leq k$), this R' compromises that $\mathcal{S}$ is conformant. This is a contradiction. □

Theorem 5.4. (Completeness of $\mathcal{P}^{\mathcal{K}}$ for Secure Strategies)
Let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem, such that the set $CP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of optimistic plans exactly covers the intended collaborative behaviour $\mathcal{IS}(M)$ of M. Suppose that M has bounded runs and a conformant collaboration strategy, $\mathcal{S}_M$. Then, if M is implemented incorrectly, there is some message log $\mathcal{M}_{log}$ such that either (i) $CP = \emptyset$, or (ii) Culprit$(\mathcal{M}_{log}, CP)$ exists.

Proof:
The argument is similar to that in the proof of Theorem 5.3. In arguing towards a contradiction, rather than concluding that $R \simeq R'$ must hold, we use that R must correspond to some trajectory $T' = s'_0, a_1, s'_1, \ldots, a_k, s'_k$, where $a_i = f(m_i)$, $i \in \{1, \ldots, k\}$, which does not reach the planning goal G. This trajectory means that the plan $P = \langle a_1, \ldots, a_n \rangle$ given by the trajectory $T = s_0, a_1, s_1, \ldots, a_n, s_n$, is not secure. However, this is a contradiction to Lemma 5.1. □

For example, in our running scenario, a design error is detected for conformant plans as *MAS* collaborative behaviour formalism, if we use secure plans in $\mathcal{K}$. The culprit vanishes if we move to a cover which contains in addition the (non-secure) plan P_1, since it is compatible with $\mathcal{M}_{log}$.

We can deploy the $\mathcal{K}$ planner also for checking C-Plans$(\mathcal{P}, \mathcal{M}_{log}, n) \neq \emptyset$ or whether Culprit$(\mathcal{M}_{log}, \mathcal{P})$ exists. In particular, if the intended behaviour is expressed by optimistic/secure plans in $\mathcal{K}$, deciding C-Plans$(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n) \neq \emptyset$ is tantamount to optimistic/secure plan existence; the plan prefix given by $\mathcal{M}_{log}, n$ can easily be encoded in the planning problem itself by adding corresponding constraints.

Let $\mathcal{M}_{log} = t_1{:}m_1, t_2{:}m_2, \ldots, t_n{:}m_n$, and let $\mathcal{P}^{\mathcal{K}}$ be a $\mathcal{K}$ planning problem modelling the *MAS* M at hand. Let $\mathcal{P}^{\mathcal{K}}_{\mathcal{M}_{log}}$ be the problem obtained from $\mathcal{P}^{\mathcal{K}}$ by adding the following to the $\mathcal{K}$ program (cf. [7] for details on the semantics):

```
initially : step$_0$.
always :    caused false after not $m_1$, step$_0$.
            caused step$_1$ after $m_1$.
            caused false after not $m_2$, step$_1$.
            caused step$_2$ after $m_2$.
            ⋮
            caused false after not $m_n$, step$_{n-1}$.,
```

where $\texttt{step}_0, \ldots, \texttt{step}_{n-1}$ are newly added propositional fluents. Intuitively, this modified planning encoding enforces the "execution" of the messages in $\mathcal{M}_{log}$ and only plans which comply with these messages are computed. We therefore obtain the following result:

Proposition 5.1. Suppose the set $OP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of optimistic plans for $\mathcal{P}^{\mathcal{K}}$ (respectively, the set $SP \subseteq \mathcal{P}^{\mathcal{K}}$-plans of secure plans) covers $\mathcal{IS}(M)$. Let $\mathcal{M}_{log} = t_1{:}m_1, t_2{:}m_2, \ldots, t_n{:}m_n$ be a message log. Then, C-Plans$(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n) \neq \emptyset$ iff $\mathcal{P}^{\mathcal{K}}_{\mathcal{M}_{log}}$ has an optimistic (resp. secure) plan.

As easily seen, the encoding $\mathcal{P}^{\mathcal{K}}_{\mathcal{M}_{log}}$ can therefore be used to check the existence of Culprit$(\mathcal{M}_{log},\mathcal{P}^{\mathcal{K}})$ of form $\langle t_n{:}m_n, \mathit{idle}\rangle$, while in general we cannot use a planner for detecting a timeout in the *MAS*.

This encoding is not restricted to our particular formalism. For instance, the computation of plans compatible with a prefix $\mathcal{M}_{log}$ can be achieved in any planning formalism which allows for a similar modelling of domains such that certain actions can be fixed. Thus, we can apply planning also to check whether C-Plans$(\mathcal{P}^{\mathcal{K}}, \mathcal{M}_{log}, n) \neq \emptyset$ or whether a Culprit$(\mathcal{M}_{log}, \mathcal{P}^{\mathcal{K}})$ of form $\langle t_n{:}m_n, \mathit{idle}\rangle$ exists.

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 the expressive planning language; the corresponding complexity results and a discussion of complexity issues can be found in [7]. We remark that, like for satisfiability (SAT), NP-hardness (or even worse, if secure plans are required) is a theoretical worst-case measure. Nevertheless, solutions for many instances can be found quickly, especially if only optimistic planning is required. Moreover, there are problem classes which are polynomial time solvable and for which $\texttt{DLV}^{\mathcal{K}}$ is guaranteed to compute plans in polynomial time. This highly depends on the requirements of intended plans and how complicated the corresponding planning problem gets.

For small domains, where the number of plans is moderate, I-Plans (or C-Plans, resp.) might be computed offline or simply be cached such that checking against $\mathcal{M}_{log}$ becomes simple.

6. Implementation

To demonstrate the proposed method, a running example has been implemented. In our implementation, the *Gofish MAS* and agent monitor are developed within IMPACT (*Interactive Maryland Platform for Agents Collaborating Together*). Each agent consists of a set of *data types*, *API functions*, *actions*, and an *agent program* that includes some rules prescribing the behaviour. Since $\texttt{DLV}^{\mathcal{K}}$ [5] is used as the planner, a new connection module has been created so that monitor is able to access the $\texttt{DLV}^{\mathcal{K}}$ planner. In this way, before the *Gofish MAS* operates, we feed $\mathcal{P}^{\mathcal{K}}_{\mathit{Gofish}}$ into monitor, which then calls $\texttt{DLV}^{\mathcal{K}}$ to compute all potential plans including both secure and optimistic plans.

Running scenario: The *Gofish* post office guarantees package delivery within 24 hours from dropOff. Consider the case that Sue receives an email at time 16 informed that her package (p_1=0x00fe6206c.1) has arrived at the distribution center. Sue decides to pick up the package herself. Unfortunately, when she reaches the distribution center at time 20, the clerk tells her that the package has been loaded on the truck at time 19 and it is now on the way to her home.

Because of the guaranteed delivery requirement, agent monitor computes secure plans for the purpose of monitoring:

$$\begin{aligned} P_1 \;=\; & \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist}); \\ & \texttt{truck}(p_1); \texttt{getRecipInfo}(p_1, \text{truck}); \texttt{recipInfo}(p_1, \text{truck}); \texttt{delivery}(p_1); \\ & \texttt{setDelivTime}(p_1)\rangle. \end{aligned}$$

$$P_2 = \langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist}); \texttt{pickup}(p_1); \texttt{setDelivTime}(p_1)\rangle.$$

Therefore after the action "pickup", a mistake was immediately detected by $\mathtt{monitor}$. In the end, the $\mathtt{monitor}$ agent generated a log file as follows:

Problematic action:
$20{:}\texttt{pickup}(0x00fe6206c.1), \texttt{idle}$

Actions executed:
$0{:}\texttt{dropOff}(0x00fe6206c.1); 5{:}\texttt{addPkg}(0x00fe6206c.1);$
$13{:}\texttt{distCenter}(0x00fe6206c.1); 15{:}\texttt{getRecipInfo}(0x00fe6206c.1, \text{dist});$
$16{:}\texttt{recipInfo}(0x00fe6206c.1, \text{dist}); 19{:}\texttt{truck}(0x00fe6206c.1);$

Possible plans before problematic action:
$\langle \texttt{dropOff}(p_1); \texttt{addPkg}(p_1); \texttt{distCenter}(p_1); \texttt{getRecipInfo}(p_1, \text{dist}); \texttt{recipInfo}(p_1, \text{dist});$
$\texttt{truck}(p_1); \texttt{getRecipInfo}(p_1, \text{truck}); \texttt{recipInfo}(p_1, \text{truck}); \texttt{delivery}(p_1); \texttt{setDelivTime}(p_1)\rangle$

More information about our running example can be found at the project homepage.[6] In addition, we refer to [26] for the details of IMPACT.

7. Related Work

In contrast to research on plan generation, there has been relatively little work on the use of plans to debug a *MAS* and to monitor the execution of agents. As plans can be executed by one agent or by many collaborative agents, in this section, we review related work in (1) single-agent settings, and (2) multi-agent environments.

7.1. Monitoring in Single-Agent Settings

Interleaving monitoring with plan execution has been addressed in the context of single agent environment in [12], where the authors present a situation calculus-based account of execution monitoring for robot programs written in Golog. A situation calculus specification is given for the behaviour of Golog programs. Combined with the interpretation of Golog programs, an execution monitor detects the discrepancy after each execution of a primitive action. Once a discrepancy is found, the execution monitor checks whether it is *relevant* in the current state, that is, whether preconditions of the next action still hold with the effect of an exogenous action. If this exogenous action does matter, a *recover* mechanism will be invoked. The method of recovering is based on planning. A new plan (or program) is computed whose execution will make things right by way of leading the current state to the desired situation, had the exogenous action not occurred. In their work, declarative representations have been proposed for the entire process of plan-execution, -monitoring and -recovery. Similar to our method, the approach is completely formal and capable of monitoring arbitrary programs. The authors have addressed the problem of recovering from failure, which is not included in our system for the moment. However, their approach must know in advance all exogenous events in order to specify appropriate *relevance* checks and *recover*

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

mechanisms. In addition, they do not explore in-depth how to properly define *Relevant* and *Recover*. The framework was later expanded in [25] by introducing tracing and backtracking into the process of online monitoring of Golog programs.

To enable generation of plans in dynamic environments, Veloso et al. [28] introduce *Rationale-Based Monitoring* based on the idea of planning as decision making. Rationale-based monitors encode the features that are associated with each planning decision. The method is used for sensing relevant (or potentially relevant) features of the world that likely affect the plan. Moreover, it investigates the balance between sensitivity to changes in the world and stability of the plans. Although this approach provides the planner opportunities to optimise the plans in a dynamic environment during plan generation, as opposed to our approach, they have not studied the issue of execution monitoring.

As the methods mentioned above address the problem of a *single agent* acting in an uncertain environment, the techniques focus on monitoring of environment and verifying plans. While our approach could be directly applied to single agent domains, these approaches need extra work in order to handle monitoring the collaboration of multiple agents.

7.2. Monitoring of Multi-Agent Coordination

Teamwork monitoring has been recognised as a crucial problem in multi-agent coordination. Jennings proposes two foundations of multi-agent coordination in [17]: *commitments* and *conventions*. Agents make commitments, and conventions are a means of monitoring of the commitments. The monitoring rules, i.e., what kind of information is monitored and the way how to perform monitoring, are decided by conventions. Jennings illustrates the method by some examples, but does not investigate how to select such conventions. Different from his idea, our approach avoids the problem of monitoring selectivity.

Myers [23] introduces a continuous planning and execution framework (CPEF). The system's central component is a plan manager, which directs the processes of plan-generation, -execution, -monitoring, and -repair. Monitoring of the environment is carried out at all time during plan generation and execution. Furthermore, execution is tracked by the plan manager by comparing reports of individual action outcomes with the temporal ordering relationships of actions. Several types of event-response rules have been concerned: (1) *failure monitors* encode suitable responses to potential failures during plan execution, (2) *knowledge monitors* detect the availability of information required for decision making, and (3) *assumption monitors* check whether assumptions that a given plan relies on still hold. The idea of assumption monitors helps early detection of potential problems before any failure occurs, which can also be achieved in our system with a different approach. Based upon CPEP, Wilkins et al. present a system in [29]. The execution monitoring of agent teams is performed based on communicating state information among team members that could be any combination of humans and/or machines. Humans make the final decision, therefore, even if unreliable communications exist, the monitoring performance may not be degraded much with the help of human experience.

Another interesting monitoring approach in multi-agent coordination is based on *plan-recognition*, by Huber [15], Tambe [27], Intille and Bobick [16], Devaney and Ram [1], Kaminka et al. [18, 20]. In this approach, an agent's intentions (goals and plans), beliefs or future actions are inferred through observations of another agent's ongoing behaviour.

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

Intille and Bobick [16] construct a probabilistic framework that can represent and recognise 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 the 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 action. The approach relies on all *coordination constraints* among the agents. Once an agent fails, it may not be able to recognise the plans.

Another line of work has been pursued by Kaminka et al. [18, 20], who developed the *OVERSEER* monitoring system building upon work on multi-agent plan-recognition in [16, 27]. The authors 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, *social knowledge*, i.e. relationships and interactions among agents, is utilised for better predicting the behaviour 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.

While our objective is (1) to debug *offline* an implemented *MAS*, and (2) to monitor *online* the collaboration of multiple agents, the plan-recognition approaches described above mainly aim to inferring (sub-)team plans and future actions of agents. The *MAS* debugging issue is not addressed. 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.

However, plan recognition is suitable for various situations: if communication is *not* possible, agents exchanging messages are not reliable, or communications must be secure. It significantly differs from our approach in the following points:

(1) If a multi-agent system has already been deployed, or if 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 [20] 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. Although our approach is NP-hard in the worst case, we did not encounter any difficulties in the scenarios we have dealt with. (3) IMPACT does not yet have implemented capabilities for dealing with probabilistic, temporal and spatial reasoning, but 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. [19] introduce 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 behaviour 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.

8. 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 upon its complexity.

Our approach works for arbitrary agent systems and can be tailored to any planning formalism that is able to express the collaborative behaviour 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 6).

Of course, our approach is not yet mature, and in this paper we focused on presenting the conceptual idea. Several issues remain to be addressed in further work. One issue concerns the modelling of multi-agent systems in a planning framework. This seems to be particularly important for complex multi-agent systems, since modelling such a system is not easy in general and requires a thought-through methodology. The methodology described in this paper merely provides some rules of thumb. Clearly, developing a good methodology is not a simple task, and will require quite some efforts.

Another issue is scalability of our approach. The example which we have considered in this paper is of moderate size, and for larger multi-agent systems, the planning tasks will become more difficult to solve. It remains to be explore to which size of systems our approach remains feasible, depending on different underlying planning techniques. However, already in multi-agent systems of moderate size (which are not unrealistic in practice) verifying the correct behaviour of agents may be difficult and laborious, and tool support will be acknowledged.

There are also several extensions to our basic approach. We mention just some of the planned future research in this direction:

(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 assigned? And, what is the optimal cost and how does the corresponding behaviour look like? This would allow us to assess the quality of an actual agents behaviour and to select cost-effective strategies. To keep the exposition simple, we have omitted that $\mathtt{DLV}^{\mathcal{K}}$ 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 costs [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 realisation of our approach, also economic behaviour 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 $\textsf{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, n') \subseteq \textsf{C-Plans}(\mathcal{P}, \mathcal{M}_{log}, n)$, 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 to above, whether from the states possibly reached by the actions in $\mathcal{M}_{log}$, the goal can still be established.

Investigating the above and further issues is part of our ongoing and future research, which will be carried out in the context of the project “An Answer Set Programming Framework for Reactive Planning and Execution Monitoring” which is funded by the Austrian Science Funds (FWF).

References

[1] Devaney, M., Ram, A.: Needles in a Haystack: Plan Recognition in Large Spatial Domains Involving Multiple Agents, *Proceedings 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference, AAAI 98*, 1998.

[2] Dix, J., Kuter, U., Nau, D.: *HTN Planning in Answer Set Programming*, Technical Report CS-TR-4332, CS Department, Univ. Maryland, 2002, Submitted to *Theory and Practice of Logic Programming*.

[3] Dix, J., Munoz-Avila, H., Nau, D., Zhang, L.: Theoretical and Empirical Aspects of a Planner in a Multi-Agent Environment, *Proceedings of Journees Europeens de la Logique en Intelligence artificielle (JELIA ’02)* (G. Ianni, S. Flesca, Eds.), LNCS 2424, Springer, 2002.

[4] Eiter, T., Faber, W., Leone, N., Pfeifer, G.: Declarative Problem-Solving Using the DLV System, in: *Logic-Based Artificial Intelligence* (J. Minker, Ed.), Kluwer Academic Publishers, 2000, 79–103.

[5] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A Logic Programming Approach to Knowledge-State Planning, II: The $\text{DLV}^{\mathcal{K}}$ System, *Artificial Intelligence*, **144**(1-2), 2002, 157–211.

[6] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: Answer Set Planning under Action Costs, *Journal of Artificial Intelligence Research*, **19**, 2003, 25–71.

[7] Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity, *ACM Transactions on Computational Logic*, 2003, To appear.

[8] Erol, K., Hendler, J. A., Nau, D. S.: UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning, *Proceedings Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)* (K. J. Hammond, Ed.), AAAI Press, June 1994.

[9] Fikes, R. E., Nilsson, N. J.: STRIPS: A new Approach to the Application of Theorem Proving to Problem Solving, *Artificial Intelligence*, **2**(3-4), 1971, 189–208.

[10] Gelfond, M., Lifschitz, V.: Representing Action and Change by Logic Programs, *Journal of Logic Programming*, **17**, 1993, 301–321.

[11] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., Wilkins, D.: *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.

[12] Giacomo, G. D., Reiter, R., Soutchanski, M.: Execution Monitoring of High-Level Robot Programs, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR 98)*, 1998.

[13] Giunchiglia, E., Lifschitz, V.: An Action Language Based on Causal Explanation: Preliminary Report, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI '98)*, 1998.

[14] Goldman, R., Boddy, M.: Expressive Planning and Explicit Knowledge, *Proceedings Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, AAAI Press, 1996.

[15] Huber, M. J., Durfee, E. H.: On Acting Together: Without Communication, in: *Spring Symposium Working Notes on Representing Mental States and Mechanisms*, American Association for Artificial Intelligence, Stanford, California, 1995, 60–71.

[16] Intille, S. S., Bobick, A. F.: A Framework for Recognizing Multi-Agent Action from Visual Evidence, *Proceedings 16th National Conference on Artificial Intelligence and 11th Conference on on Innovative Applications of Artificial Intelligence (AAAI/IAAI 99)*, 1999.

[17] Jennings, N. R.: Commitments and Conventions: The Foundation of Coordination in Multi-Agent Systems, *The Knowledge Engineering Review*, **8**(3), 1993, 223–250.

[18] Kaminka, G. A., Pynadath, D. V., Tambe, M.: Monitoring Deployed Agent Teams, *Proceedings of the Fifth International Conference on Autonomous Agents (Agents-2001)*, ACM, 2001.

[19] Kaminka, G. A., Pynadath, D. V., Tambe, M.: Monitoring Teams by Overhearing: A Multi-Agent Plan-Recognition Approach, *Journal of Artificial Intelligence Research*, **17**, 2002, 83–135.

[20] Kaminka, G. A., Tambe, M.: Robust Agent Teams via Socially-Attentive Monitoring, *Journal of Artificial Intelligence Research*, **12**, 2000, 105–147.

[21] Levesque, H. J., Reiter, R., Lesperance, Y., Lin, F., Scherl, R. B.: GOLOG: A Logic Programming Language for Dynamic Domains, *Journal of Logic Programming*, **31**(1-3), 1997, 59–83.

[22] Luck, M., McBurney, P., Preist, C., Guilfoyle, C.: The AgentLink Agent Technology Roadmap Draft, 2002, AgentLink, available at `http://www.agentlink.org/roadmap/index.html`.

[23] Myers, K.: FPEF: A Continuous Planning and Execution Framework, *AI Magazine*, **20**(4), 1999, 63–69.

[24] Peot, M. A., Smith, D. E.: Conditional Nonlinear Planning, *Proceedings 1st International Conference on Artificial Intelligence Planning Systems (AIPS-92*, AAAI Press, 1992.

[25] Soutchanski, M.: Execution Monitoring of High-Level Temporal Programs, *Proc. IJCAI 99 Workshop on Robot Action Planning, July 31, 1999, Stockholm, Sweden*, 1999.

[26] Subrahmanian, V., Bonatti, P., Dix, J., Eiter, T., Kraus, S., Ozcan, F., Ross, R.: *Heterogeneous Agent Systems: Theory and Implementation*, MIT Press, 2000.

[27] Tambe, M.: Tracking dynamic team activity, *Proceedings 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.

[28] Veloso, M. M., Pollack, M. E., Cox, M. T.: Rationale-Based Monitoring for Planning in Dynamic Environments, *Proceedings 4th International Conference on Artificial Intelligence Planning Systems (AIPS-98)*, 1998.

[29] Wilkins, D., Lee, T., Berry, P.: Interactive Execution Monitoring of Agent Teams, *Journal of Artificial Intelligence Research*, **18**, 2003, 217–261.