Plan-execution health repair in a multi-agent system*

Femke de Jonge Nico Roos

IKAT, Universiteit Maastricht,
P.O. Box 616, NL-6200 MD, Maastricht

Abstract

This paper presents an algorithm for *plan health repair* in multi-agent plan execution. Plan-execution health repair aims at avoiding conflicts that might arise due to disruptions in the execution of a plan. This can be achieved by adjusting the executions of tasks instead of replanning the tasks. For this purpose, established methods from the domains of planning, discrete event systems, model-based diagnosis, and constraint satisfaction problems have been combined.

1 Introduction

Creating a conflict-free plan for a multi-agent system in a complex and dynamic environment is a difficult task. Firstly, the planner needs to have realistic expectations on the possible world changes to be able to take the consequences of parts of the plan into account. Secondly, the planner needs a certain amount of abstraction when referring to the smallest parts of the plan, because not every detail can be planned. Nevertheless, these details might influence the effects of the plan. Therefore, even when a perfect conflict-free plan is created, we have no guaranty that the plan remains conflict free during the execution of the plan. The conflicts that might arise during the execution of the plan can be caused by unforeseen changes in the environment, or unexpected (or unplanned) behaviour of the agents that execute the plan. In the normal cause of events, after detecting a conflict, a replanning is applied to adjust the plan such that the plan is again conflict free. In this article, we suggest a method that is less rigid. To regain a conflict-free planning, we try to adjust the plan execution within the margins of the plan. Most of the times, these adjustments will be preferable because the conflict is solved locally, while replanning usually involves a lot of consequences for the environment. If however, making adjustments within the margins of the plan is impossible or too complex to be determined efficiently, then, replanning should be applied.

The following example illustrates the occurrence of conflicts during the execution of a plan in a multi-agent system. It will be used as a running example throughout this article. Consider a small airport with only one runway used for both arrival and departure. It is a small but busy airport, so plans are tight. Assume that two aircraft agents, agent A and agent B, have agreed on the mutual plan in which B lands before A takes off. Here, the

*This research is supported by the Technology Foundation STW, applied science division of NWO and the technology programme of the Ministry of Economic Affairs. Project DIT5780: Distributed Model Based Diagnosis and Repair

smallest planned actions are the departure and arrival of the aircraft, which we will refer to as tasks. The execution of such tasks is a reactive process in the sense that (sub)actions during the execution of a task (for example manoeuvring or changing speed) are not part of the planning. Although the plan is agreed on, still, small changes in the execution (mostly caused by external influences) can cause conflicts. For instance, assume that A is a bit early as the aircraft speeded up while taxiing, and B is a bit delayed because of heavy head wind during its flight. This might cause a conflict in the use of the runway because of violating the security constraints on the distance that should be kept between two aircraft. We would like to detect such conflicts as early as possible, so that the agents can agree on a set of reactive actions (repairs) that will prevent the conflicts to occur in the future. For instance, agent A could wait a while before take off. Note that these reactive adjustments are not typified as replanning since the changes in executions will remain within the margins of the planned tasks. We will call restoring the health of a plan by adjusting the execution of tasks *plan-execution health repair*.

Below, we will introduce a model for a multi-agent system for plan-execution health repair. Based on this model, agents can detect (future) conflicts in advance. Next to presenting such a model, our goals are to apply diagnosis on the model and the anticipated conflicts, in order to determine causes and to regain plan-execution health.

We model the reactive nature of tasks by introducing a set of states for each task and a set of corresponding events that cause state changes within a task. Diagnosing conflicts in this model is related to diagnosis on Discrete Event Systems (DES), e.g. [3], in which agents try to determine occurrence of fault-states based on a partially observable sequence of events. Diagnosis on DES differs from our approach with respect to the diagnostic task. We focus on determining combinations of events that cause violations of constraints between states of different tasks and determining events causing state changes that resolve the constrain violations. In this respect our approach is closer related to Model-Based Diagnosis (MBD). For an overview of MBD in a 'single agent context', see [7]. More recently, MBD has also been studied in a multi- agent context [10]. To our best knowledge, MBD has not been applied to plan health repair, only to diagnosis of planners [1]. Other diagnostic approaches in a multi-agent context we would like to mention are Social Diagnosis, for finding causes for social disruptions in multi-agent systems [6], and diagnosis based on a causal model for achieving multi-agent adaptability [5].

Plan health repair can be viewed as a part of distributed continual planning, which addresses the adaptation of plans during plan execution in a multi agent system (for an overview, see [4]). In general, continual planning consists of two parts. On the one hand, agents monitor the plan execution and the dynamic world. On the other hand, based on these observations, a planning technique can be applied to either prevent conflicts, or to improve the current planning to fit the preferences better. For both the monitoring and the planning part, a consideration of the costs should be made. More detailed or up-to-date observations of world changes and more optimal plan adjustments require higher computational costs (and in a multi-agent system, probably higher communication costs as well). High computational costs lead to a slower system, which is not preferable in a highly dynamic environment. Established planning techniques for plan repair through replanning are summarized in [11, 12]. An approach of plan improvements to handle uncertainty in plan execution is given by Raja et al. [8].

desJardins et al. [4] state that the most preferred continual planning technique uses a

hierarchical plan. Initially, an abstract plan is made, and as the execution approaches, the plan is being refined. A drawback of this approach is that the plan execution might fail because of planning decisions made in higher, more abstract parts of the plan which apparently cannot be refined without any occurring conflict. To prevent this, agents should be able to backtrack into the hierarchy of plan refinement. The approach of refining the plan during its execution is not always preferable for two reasons. First, for several application domains, such as Air Traffic Control, it is unacceptable that some parts of the plan (for instance, use of runways or ground handling) will be decided on at a very last moment. Furthermore, often, there is enough time and means to create an optimal initial plan. Applying plan refinement would possibly result in a loss of optimality. Second, in many application domains, not every part of the plan can be planned in full detail. Most certainly within the domain of Air Traffic Control, the tasks within a plan are mostly reactive of nature. For instance, the task of 'taxi' will be appointed a execution time, but how the pilot should execute this task is not specified. It is for the above mentioned types of application domains that we present our approach of plan-execution health management. What sets our approach apart from continual planning and replanning in particular, is the application of health repair within the margins of the current plans.

The outline of this paper is as follows. In section 2, we introduce a model that agents can use to represent a multi-agent system for plan-execution health repair. Based on this model, agents are able to detect whether conflicts occur in the future. In section 3, we define diagnoses that agents can apply on the conflicts in order to find out what has caused them. A formal definition of both weak and strong plan-execution health repair is given in section 4. In section 5 we present an algorithm for agents to regain plan-execution health based on a transformation to a constraint satisfaction problem. In section 6 we provide conclusions and suggest future work on diagnosing and resolving conflict in multi-agent system for plan-execution health repair.

2 Model description

A multi-agent plan $MAP = (A, PD, R, Cst)$ consists of a set of agents A, a set of plan descriptions PD, containing one plan description for each agent: $PD = \bigcup_{i=1}^{|A|} PD_i$, a set of common rules R specifying the execution of the plan in general, and a set of constraints Cst between the agents' plans. We assume that each agent has its own plan, and that all plans are combined within MAP.

A *plan description* $PD_i = (P_i, \mathcal{S}_i, \mathcal{E}_i, \tau_i, \sigma_i)$ describes how the plan of agent i will be executed. The base of the plan description is the sequence of tasks $P_i = \langle t_{i,0}, t_{i,1}, ..., t_{i,n} \rangle$ which the agent wants to execute in this specific order. We use $\overline{P_i}$ to denote the corresponding set of all tasks in sequence P_i. Note that for simplicity reasons, we assume P_i to be totally ordered. However, our approach is also applicable on partially ordered sequences of tasks. To describe the health of a task, the sets $\mathcal{S}_i$ and $\mathcal{E}_i$ contain for each task a set of states and a set of events respectively. The functions τ_i and σ_i formalize, combined with the common rules R, the execution of tasks within a plan (we will specify this further on).

During the execution of an agent's plan, a task is in a certain *state*. Each task has its own set of possible states: $S_{i,j} \in \mathcal{S}_i$. We distinguish three types of states: pending, active,

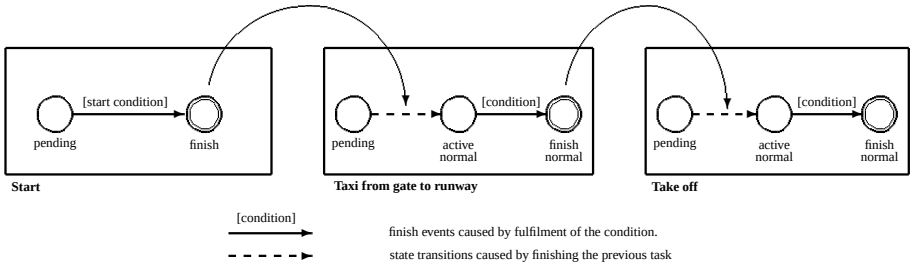

Figure 1: Normal plan execution of agent A.

and finish states. For each task $t_{i,j}$ holds: $S_{i,j} = \{s_{i,j}^{pending}\} \cup S_{i,j}^{active} \cup S_{i,j}^{finish}$. There is only one pending state for each task, this is the state in which the task is awaiting before it is being executed. By finishing the previous task, the next task will become active by changing from the pending to an active state (which state that is, depends on the execution of the previous task). Finally, when the task is completed, the task changes from an active state to a finish state and consequently, the next task is triggered.

Each plan has one start task: $t_{i,0}$, with $S_{i,0} = \{s_{i,0}^{pending}, s_{i,0}^{finish}\}$. The start task has only one pending and one finish state. When the start conditions are fulfilled, this start task will change from the pending to the finish state, which will cause the next task to begin execution (viz. go from the pending state to an active state).

State changes are caused by *events*. Each task $t_{i,j}$ has its own set of events: $E_{i,j} \in \mathcal{E}_i$, with $E_{i,j} = E_{i,j}^{finish} \cup E_{i,j}^{disrupt} \cup E_{i,j}^{repair}$. Finish events are triggered when pre-defined conditions are fulfilled and change tasks from an active to a finish state. Disruption events are externally caused and represent unexpected changes in the execution of a task that might effect the plan-execution health. And finally, the repair events are executed by the agent to regain the plan-execution health when necessary. A task's state is the result of the sequence of events during the plan execution, and will be represented by predicate $ts(t_{i,j}, s, E)$, where $t_{i,j} \in \overline{P_i}$ is the task for which event sequence $E = \langle e_1, ..., e_k \rangle$ leads to state $s \in S_{i,j}$. We use the predicate $ats(t, s)$ to denote that task t will achieve state s during the actual plan execution, i.e. the past, current and expected events lead to state s.

Figure 1 illustrates the normal execution of a plan of the departing agent A in our running example. The plan consists of three tasks: $P_1 = \langle t_{1,\text{Start}}, t_{1,\text{Taxi}}, t_{1,\text{Take\_off}} \rangle$, and has event sequence $\langle e_{\text{finish\_Start}}, e_{\text{finish\_Taxi}}, e_{\text{finish\_Take\_off}} \rangle$.

We model the *execution of tasks* within an agent's plan by partial functions τ_i and σ_i, and the set of common rules R from MAP. The partial function τ_i maps a state and an event to a new state: $\tau_i : \overline{P_i} \times \bigcup_j S_{i,j} \times \bigcup_j E_{i,j} \nrightarrow \bigcup_j S_{i,j}$. τ_i is defined such that only events in $E_{i,j}$ can change the state of a task $t_{i,j}$ into a new state in $S_{i,j}$. We assume that there is exactly one finish event for each task. A task can, by definition of τ_i, reach different finish states depending on de previous state the task is in. The partial function σ_i returns the new state in the next task based on the finish state of the previous task: $\sigma_i : \overline{P_i} \times \bigcup_j S_{i,j}^{finish} \nrightarrow \bigcup_j S_{i,j}$. The functions τ_i and σ_i are defined per agent (instead of being the same for all agents), since they represent different types of plan-execution behaviour for different types of agents.

The set of common rules R in MAP consists of three rules. The first rule in R

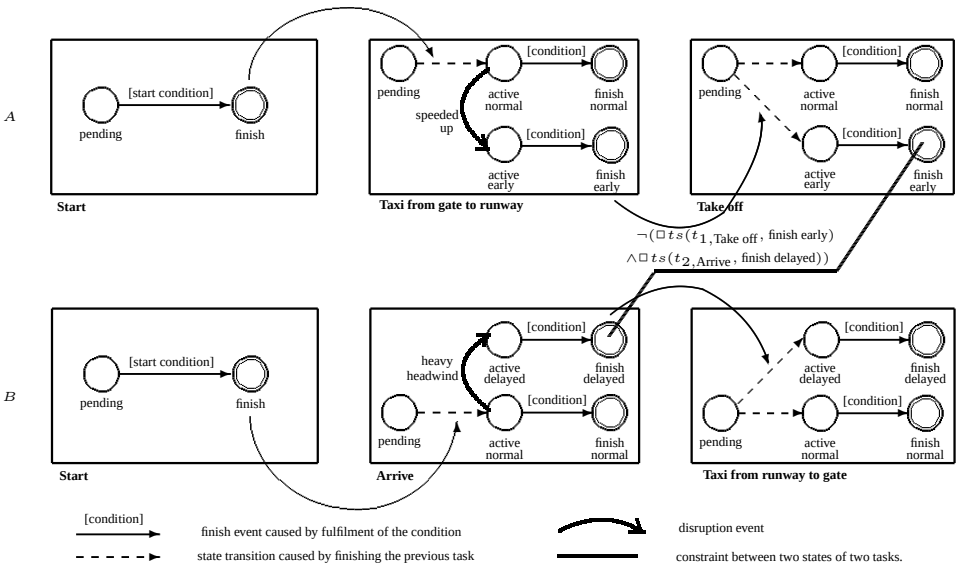

Figure 2: Disturbed plan executions of agents A and B.

describes how a state transition of a task is caused by an event e_k:

$$(ts(t_{i,j}, s, \langle e_1; ...; e_{k-1}\rangle) \wedge \tau_i(t_{i,j}, s, e_k) = s') \rightarrow ts(t_{i,j}, s', \langle e_1; ...; e_k\rangle) \quad (1)$$

The second rule in R describes the activation of the next task when the previous task is finished:

$$(ts(t_{i,j}, \text{pending}, \langle e_1; ...; e_k\rangle) \wedge \sigma_i(t_{i,j-1}, s) = s') \rightarrow ts(t_{i,j}, s', \langle e_1; ...; e_k\rangle) \quad (2)$$

The third rule in R defines which states will or will not be reached during the plan execution. We use the predicate $Events(\{E_1, ..., E_m\})$ to denote that these sequences of events will occur (a sequence E_i for each P_i).

$$\exists e_1; ...; e_k(Events(\{\langle e_1; ...; e_k; ...e_n\rangle_i, ...\}) \wedge ts(t_{i,j}, s, \langle e_1; ...; e_k\rangle)) \leftrightarrow ats(t_{i,j}, s) \quad (3)$$

The set Cst in MAP is the set of *constraints*, with each constraint composed of predicates $ats(,)$ and logic symbols $\{\vee, \wedge, \neg\}$. Moreover, constraints are only defined on finish states, as they can be viewed as a summary of the execution of a task. The constraint violations can be repaired during the active states of a task. An example of a constraint is $cst = \neg(ats(t, s) \wedge ats(t', s')) \vee ats(t'', s'')$, in which s, s', s'' are finish states. The constraints are 'demands' on the plan execution that should be fulfilled. A constraint violation or conflict occurs when the expected execution is inconsistent with a certain constraint. We will assume that when plans are executed normally (only finish events occur), all constraints will hold and the plan-execution is in good health. Consequently, the constraint violations are caused by disruption events, and might be solved by repair events to regain the plan-execution health. In addition, we assume that the constraints represent all interdependencies that exist between plans of different agents.

Figure 2 illustrates a disrupted execution of the plans of the departing agent A and arriving agent B in our running example. Both plans consist of three tasks: $P_1 = \langle t_{1,\text{Start}}, t_{1,\text{Taxi}}, t_{1,\text{Take\_off}} \rangle$, and $P_2 = \langle t_{2,\text{Start}}, t_{2,\text{Arrive}}, t_{2,\text{Taxi}} \rangle$. The event sequences of the plan execution are $\langle e_{\text{finish\_Start}}, e_{\text{Speeded\_up}}, e_{\text{finish\_Taxi}}, e_{\text{finish\_Take\_off}} \rangle_1$ and $\langle e_{\text{finish\_Start}}, e_{\text{Heavy\_head\_wind}}, e_{\text{finish\_Arrive}}, e_{\text{finish\_Taxi}} \rangle_2$. In this setting, the constraint $\neg(ats(t_{1,\text{Take\_off}}, \text{finish\_early}) \wedge ats(t_{2,\text{Arrive}}, \text{finish\_delayed}))$ between the two plans is violated.

In general, we assume that each agent has knowledge of its individual plan description PD_i, of the common rules R, of the constraints $Cst_i \subseteq Cst$ that are relevant for its plan, and of the other agents to whose plans the constraints Cst_i apply. During the execution of a plan, an agent notices when disruption events occur (for instance through its sensors). Based on this, an agent can construct the sequence of past events (up to and including the current or latest events) in the so-called current event history CEH_i (with $CEH = \bigcup_i CEH_i$). We assume that in the future, from current task $t_{i,j}$ on, no disruption or repair events will occur. Hence, for each task in the remaining plan, one finish event will occur. The resulting sequence of events $FE_i = \langle e_j, e_{j+1}, ..., e_n \rangle$, with $e_x \in E_{finish}$ and $FE = \bigcup_i FE_i$, will be called the future event sequence. The current event history can be combined with the future events sequence into the future event history: $FEH_i = CEH_i \circ FE_i$ (with $\circ$ denoting a concatenation of the two sequences). Using the future event history, an agent can determine the possible consequences of the disruption events. If the constraints possibly get violated, the agent contacts the other agents involved to verify this. This way, conflicts are detected as early as possible.

3 Event diagnosis

Through diagnosis we wish to find out for each violated constraint, which set of disruption events causes the violation. With the help of these events we can also establish which states are responsible for the violations. These states might be helpful to prevent new constraint violations in much earlier phases by recognizing state patterns.

We define two types of Event Diagnosis: Responsible Event Diagnosis (Δ_R) and Constraint Satisfaction Event Diagnosis (Δ_{CS}). Both diagnoses are a subset of the disruption events that occur in the future event history; $\Delta \subseteq E_{disrupt} \cap \overline{FEH}$, with $\overline{FEH}$ the corresponding set of all events in the sequences in FEH. Moreover, both diagnoses are defined on one constraint cst^* that is violated, but can as well be extended to sets of violated constraints. Below, we provide two formal definitions of the diagnoses. Responsible event diagnosis gives us a minimal set of disruption events that causes the violated constraint cst^*. To achieve this, we remove as many disruption events as possible, such that the constraint violation still holds.

Definition 1 *Responsible Event Diagnosis is a minimal diagnosis* $\Delta_R \subseteq \{E_{disrupt} \cap \overline{FEH}\}$ *s.t.* $Events(FEH - \{E_{disrupt} \setminus \Delta_R\}) \cup PD \vdash \neg cst^*$.

With $FEH - X$ we denote that the events in X are removed from the sequences in FEH.

The constraint satisfaction event diagnosis gives us a minimal set of disruption events, such that when these events are left out of the future events history, the violated constraint will hold again.

Definition 2 *Constraint Satisfaction Event Diagnosis is a minimal diagnosis* $\Delta_{CS} \subseteq \{E_{disruption} \cap \overline{FEH}\}$ *s.t.* $Events(FEH - \Delta_{CS}) \cup PD \vdash cst^*$.

The two diagnoses are related as follows: each minimal hitting set on the set of all possible responsible event diagnoses, is a constraint satisfaction diagnosis, and vice versa.

In our example, all possible event diagnoses are:
$\Delta_R = \{e_{\text{Speeded\_up}}, e_{\text{Heavy\_head\_wind}}\}$, $\Delta^1_{CS} = \{e_{\text{Speeded\_up}}\}$ and $\Delta^2_{CS} = \{e_{\text{Heavy\_head\_wind}}\}$.

4 Plan-execution health repair

Once the agents have detected the constraint violations that will arise because of (some of) the occurred disruption events, the agents should adjust the execution of the plans such that no constraint violations will occur in the future and the plan-execution health is restored. To achieve this, each agent can insert repair events in the future event history in order to create new state paths in its plan execution. By inserting repair events, the anticipated constraint violations can be avoided.

A weak plan-execution health repair FER^- is a set of event sequences containing all future event sequences with some repair events inserted, such that by applying FER^-, all anticipated constraint violations will dissolve and no new violations will be created.

Definition 3 *A weak plan-execution health repair* FER^- *is a set of sequences* $FER^- = FE \Cup RE$*, where* RE *is a minimal subset of* E_{repair} *s.t.* $Events(CEH \circ FER^-) \cup PD \cup Cst \nvdash \bot$.

We use $FER^- = FE \Cup RE$ to denote that the events in RE are placed at specified places within the sequences collected in FE. Note that for the same FE and RE different sets $FER^- = FE \Cup RE$ are possible, depending on the placement of the repair events in the sequences in FE. With a minimal RE we limit the subsets of RE to those which have no subset that will construct a (weak) plan-execution health repair as well.

A strong plan-execution health repair FER^+ differs from the weak version in that FER^+ ensures that all constraints hold.

Definition 4 *A strong plan-execution health repair* FER^+ *is a set of sequences* $FER^+ = FE \Cup RE$ *where* RE *is a minimal subset of* E_{repair} *s.t.* $Events(CEH \circ FER^+) \cup PD \vdash Cst$.

Proposition 1 *A weak plan-execution health repair is a strong one and vice versa.*[†]

In our example, we can introduce an event e_{Wait}, which changes the state of task $t_{1,\text{Take\_off}}$ from 'active_early' into 'active_normal'. Then, an example of a plan execution health repair is $FER = \{\langle e_{\text{fin\_Start}}, e_{\text{Speeded\_up}}, e_{\text{fin\_Taxi}}, e_{\text{Wait}}, e_{\text{fin\_Take\_off}}\rangle_1, \langle e_{\text{fin\_Start}}, e_{\text{Heavy\_head\_wind}}, e_{\text{fin\_Arrive}}, e_{\text{fin\_Taxi}}\rangle_2\}$.

[†]The proof is omitted because of limited space. It is available on request.

5 An algorithm for plan-execution health repair

Both the weak and strong plan-execution health repair are strongly related to model-based diagnosis. The weak plan-execution health repair corresponds with consistency-based diagnosis, as formalized by Reiter [9]. The strong plan-execution health repair is a type of abductive diagnosis, as defined by Console and Torasso [2]. Since both types of diagnosis are known to be NP-hard, in general, our plan-execution health repair is NP-hard as well.

To enable the agents to find a plan-execution health repair, we formulate the plan-execution health repair as a constraint satisfaction problem: $PR_{csp} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$. The set variables $\mathcal{V}$ contains a variable for each task in MAP: $\mathcal{V} = \{v_{i,j} | t_{i,j} \in \overline{P_i}\}$. $\mathcal{D}$ contains for each variable a domain of possible values, in this case the set of finish states: $\mathcal{D} = \{S_{i,j}^{finish} | t_{i,j} \in \overline{P_i}\}$. The set of constraints, $\mathcal{C}$, is divided into plan constraints, $\mathcal{C}_{plan}$, and conflict constraints, $\mathcal{C}_{conflict}$. The plan constraints represent the execution of the plans, as described by PD. A plan constraint between two successive tasks is true, if there is an event path from the value assignment (or finish state) of the first task, to the value assignment (or finish state) of the second task. The possible paths depend on CEH and FER (the future event sequences combined with repair events). Therefore, the set of plan constraints can be constructed as follows.

$$\mathcal{C}_{plan} = \{c_{v_{i,j}, v_{i,j+1}}(s_1, s_2) | Events(CEH \circ FER) \cup PD \vdash ats(t_{i,j}, s_1) \wedge ats(t_{i,j+1}, s_2)\} \quad (4)$$

The set of conflict constraints $\mathcal{C}_{conflict}$ is a direct mapping of the set Cst in MAP onto the variables $v_{i,j}$.

$$\mathcal{C}_{conflict} = \{c_{v_{i,j}, ..., v_{k,l}}(s_1, ..., s_p) | ats(t_{i,j}, s_1) \wedge .... \wedge ats(t_{k,l}, s_p) \vdash cst\} \quad (5)$$

The problem of finding a plan-execution health repair after detecting violated constraints is now transformed into the problem of assigning values to the variables from their domains such that all constraints are met. The following three stage algorithm for finding such a value assignment is based on representing the constraint satisfaction problem in a constraint graph. Summarized, the first stage is an initiation stage in which each agent creates its own individual constraint graph which is a subgraph of the whole constraint graph. In the second stage, agents achieve arc-consistency on the whole constraint graph, i.e. the domains are maximally reduced into valid values based on the constraints. Based on this domain reduction, the agents assign the values to the variables in the third stage.

Stage 1: Initially, each agent creates an individual constraint graph for its own plan. An individual constraint graph consists of two types of nodes, viz. so-called internal and external nodes. The internal nodes represent the variables of the agent's plan, these are the variables that the agent itself can influence. The external nodes represent the variables of other agents' plans that are linked to the variables of the internal nodes through the conflict constraints. The values of the external variables can not be assigned by the agent, but have influence on the possible value assignments of the internal nodes. Each node in the graph is labeled with its variable's domain. The domains of the external nodes are communicated by the agent that owns the nodes (i.e. which tasks or variables are represented). There are two types of arcs that connect the nodes. The bidirectional arcs between the internal nodes represent the plan constraints. These arcs work in both ways:

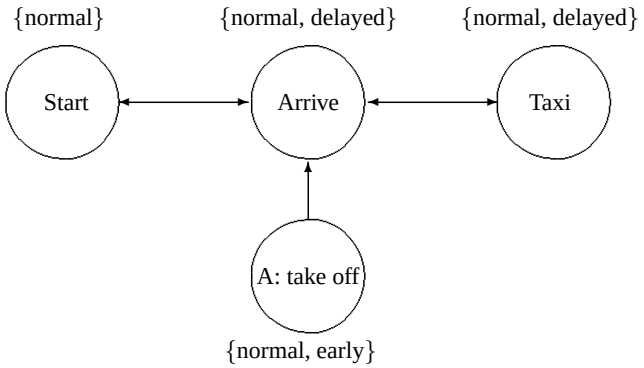

Figure 3: Individual constraint graph of agent B.

the two linked nodes influence each others values (through the plan constraint). The unidirectional arcs connect the internal nodes with the external nodes. They represent the conflict constraints and can be hyperarcs. Through these arcs, the possible value assignment of the internal nodes is influenced by the possible value assignment of the external nodes, but not vice versa. Note that when we consider the whole constraint graph (the individual graphs combined), these arcs will be bidirectional as well. Because of the sequential nature of plans, the individual subgraph solely based on the internal nodes and the bidirectional arcs, will be linear. The individual constraint graph (the subgraph combined with the external nodes and unidirectional arcs) itself is a non-cyclic graph.

The individual constraint graph of agent B from our example is presented in figure 3. Each task in A's plan $P_2 = \langle t_{2,\text{Start}}, t_{2,\text{Arrive}}, t_{2,\text{Taxi}} \rangle$ has its own node. These nodes are connected through bidirectional arcs denoting the dependencies between the two tasks (i.e., the event paths that are possible). The node 'A: take off' is the external node, connected to the 'Arrive' node by the constraint $\neg(ats(t_{1,\text{Take\_off}}, \text{finish\_early}) \wedge ats(t_{2,\text{Arrive}}, \text{finish\_delayed}))$. Moreover, each node has a domain of possible values: the possible finish states of the task.

Before the agents will apply domain reduction on the individual constraint graphs, some variables will be locked to increase search efficiency. In the first place, the values of the variables that correspond to tasks in the past are fixed to their occurred value. In the second place, in order to attempt to solve the violated constraints locally at first, all agents that are not involved in the detected constraint violation, lock the values of their variables into the current or (under normal circumstances) expected value.

Stage 2: By repeating two steps, arc-consistency on the whole constraint graph is achieved. First, the agents reduce the domains of their variables by applying arc-consistency on their individual constraint graphs. This can be achieved in linear time, provided that the domain reduction is started with the unidirectional arcs connected to the external nodes. Note that for this part of the algorithm no communication with other agents is required.

Figure 4 shows the individual constraint graph of agent B from our example, after applying arc-consistency (task 'Arrive' is the current task). The domains of tasks 'Arrive' and 'Taxi' are both reduced, since i) there is no event sequence that causes a state transition to '(finish) normal' in the task 'Arrive', and ii) based on the state-domain of task 'Arrive', there is no event sequence that will lead to state '(finish) normal' in task 'Taxi'.

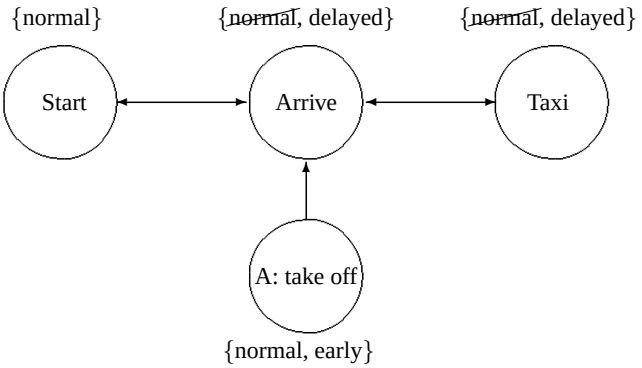

Figure 4: Arc-consistent individual constraint graph of agent B.

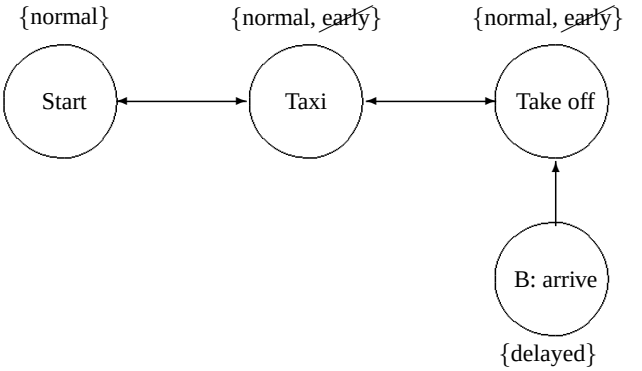

Figure 5: Arc-consistent individual constraint graph of agent B.

Second, for each internal node with a changed domain that is involved in a conflict constraint, the corresponding agent communicates the new domain to the other agents involved (represented by the external nodes). Subsequently, the latter agents adjust the domains of the corresponding external nodes to the communicated values. Then again, the agents apply arc-consistency on the updated individual constraint graphs, which is followed by communication on the altered domains. The two steps are repeated until no domains change anymore. Then, the whole constraint graph is arc consistent and the domains are maximally reduced.

In our example, agent B communicates the new domain of node 'Arrive' to agent A, which updates the label at the external node in his individual constraint graph. Consequently, applying arc-consistency on its graph results in altered domains for agent A (which should be communicated to agent A and so on). See figure 5.

Stage 3: Based on the restricted domains, the agents search for a value assignment. The agents firstly agree on an order in which they will search for an assignment. Then, the first agent in the order searches for a value assignment for its variables within the restricted domains and communicates these to the agents involved in its conflict constraints. These agents adjust the corresponding domains of the external nodes according to this value assignments and thereupon, all agents apply arc-consistency on their adjusted graphs, as in

stage 2. If arc-consistency on the whole constraint graph is found, this means that there possibly still is a legitimate value assignment left. In that case, the second agent in the order searches for a value assignment of its variables, and so on. But when during the arc-consistency stage a domain becomes empty, no value assignment for this particular variable is possible, given the chosen values until now. Then, the search process backtracks and the agent that made the last assignment searches for an other value assignment. If that does not succeed (i.e. still, empty domains occur during the arc-consistency stage), the process backtracks to the previous agent in the order, and so on.

Eventually, a solution for the constraint satisfaction problem may be found when all non-fixed variables have been assigned a value. Since a value assignment represents a state of a task, we know that this state can be reached from the assigned state of the previous task, following an event sequence containing repair events. The set of all future event sequences FER, is the solution to our plan-execution health problem: by applying the repairs in this event sequence, no constraint violations will occur.

In our example, the only possible value assignment might be for A to assign value 'normal' to all its tasks, and for B to assign value 'normal' to task 'Start' and value 'delayed' to both tasks 'Arrive' and 'Taxi'. This assignment corresponds to the plan execution health repair as described in the previous section, in which agent A applies the repair event 'wait' during the execution of the task 'Taxi'.

When no solution is found, the search space can be increased by unlocking variables that are not directly involved in the constraint violation. When no solution is found and all variables are unlocked, there exists no plan-execution health repair and changes in the planning (replanning) have to be made to avoid a the detected conflict.

There are two situations in which replanning might also be applied. The first situation arises when finding a plan-execution health repair takes too much time, and an adjustment in the plan itself (instead of an adjustment in the plan execution) can be found much faster. For this purpose, a comparative assessment of both methods (plan-execution health repair and replanning) is required. On the one hand, the expected complexity and duration of finding repair events and replanning should be compared. To obtain an accurate view on the expected duration of finding repair events, additional research is required. On the other hand, the costs of plan-execution health repair and replanning should be compared. In most application areas, local, small changes caused by repair events are much less expensive than widespread, large changes which are generally common for replanning.

The second situation in which the agents should fall back on replanning techniques is when tasks reach states that cannot be changed by applying repair events. For instance, when during arrival an aircraft is running out of fuel and needs to land immediately. Such 'emergency states' should trigger the agents to solve their problems more drastically, by changing their plans.

The third stage of the algorithm can be changed such that the assignment of the variables (and indirectly which repair events are inserted) becomes negotiable. The chosen value assignments will be presented as proposals, and agents can make counterproposals. To enable the agents to value the different proposals and thus to make choices, some kind of utility function should be implemented. However, then, the algorithm becomes more complex and possibly more time is needed to find a solution that all involved agents agree on.

6 Conclusions and future work

In this paper, we introduced a model for reactive execution of plans in a multi-agent context. The model forms a basis for adequate handling of conflicts that can arise during the execution of the plans. Using this model, agents may perform appropriately model-based diagnosis to resolve conflicts and regain plan-execution health by inserting small repair actions (the repair events) in the execution of the tasks.

Still, several topics are to be examined in the future. First, the model should be extended to a probabilistic model in which the chances that a disruption event will occur in the future are taken into account. Second, the heuristics to improve the efficiency of the described algorithm for multi-agent plan-execution health repair should be tested and further investigated.

References

[1] L. Birnbaum, G. Collins, M. Freed, and B. Krulwich. Model-based diagnosis of planning failures. In *AAAI*, pages 318–323, Boston, July 1990.

[2] L. Console and P. Torasso. A spectrum of logical definitions of model-based diagnosis. *Computational Intelligence*, pages 133–141, 1991.

[3] R. Debouk, S. Lafortune, and D. Teneketzis. Coordinated decentralized protocols for failure diagnosis of discrete-event systems. *Journal of Discrete Event Dynamical Systems: Theroy and Application*, 10:33–86, 2000.

[4] M.E. desJardins, E.H. Durfee, Jr. C.L. Ortiz, and M.J. Wolverton. A survey of research in distributed, continual planning. *AI Magazine*, 4:13–22, 2000.

[5] Bryan Horling, Victor Lesser, Regis Vincent, Ana Bazzan, and Ping Xuan. Diagnosis as an Integral Part of Multi-Agent Adaptability. *Proceedings of DARPA Information Survivability Conference and Exposition (see also UMASS CSTR 1999-03)*, pages 211–219, 2000.

[6] M. Kalech and G.A. Kaminka. On the design of social diagnosis algorithms for multi-agent teams. In *IJCAI*, 2003.

[7] I. Mozetic. Model-based diagnosis: an overview. In *Advanced Topics in Artificial Intelligence*, pages 419–430. Springer-Verlag (LNAI 617), 1992.

[8] A. Raja, V. Lesser, and T. Wagner. Towards robust agent control in open environments. In *Proceedings of 5th International Conference of Autonomous Agents*, 2000.

[9] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[10] N. Roos, A. ten Teije, A. Bos, and C. Witteveen. An analysis of multi-agent diagnosis. In *AAMAS*, 2002.

[11] S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach.* Prentice Hall, 1995.

[12] R. van der Krogt, M. de Weerdt, and C. Witteveen. A resource based framework for planning and replanning. *Web Intelligence and Agent Systems*, 1(3-4), 2003.