Improving Belief Management for High-Level Robot Programs by Using Diagnosis Templates

Michael Reip¹, Gerald Steinbauer¹, and Alexander Ferrein²

¹ Institute for Software Technology, Graz University of Technology, Graz, 8010, Austria
mreip@ist.tugraz.at
steinbauer@ist.tugraz.at

² FH Aachen University of Applied Sciences, 52066 Aachen, Germany
ferrein@fh-aachen.de

ABSTRACT
Agents or robots acting in the real world generally face the problem that the actions they execute are error-prone, that their perception is noisy and exogenous events change the world in unexpected ways. This may lead to situations, where the robot has to abort a high-level mission because of inconsistent knowledge about the world. In (Gspandl et al., 2011), an approach for belief management following model-based principles was developed, which was able to deal successfully with such problems. Within that approach, all hypotheses about alternative courses of actions were generated and tested for consistency. This had a considerable negative effect on the scalability of the approach.

In this paper we extend that approach by only generating a selected set of hypotheses. It turns out that in many cases it is sufficient to only inspect hypotheses with special properties to get back to a consistent world state. Therefore, we introduce so-called Diagnosis Templates, which are action sequences that change a property in a desired way. For finite domains these Diagnosis Templates can be pre-computed and later matched against an inconsistent course of actions. We will show that this heuristic approach is complete and a significant speedup is achieved in the majority of cases. Experimental results from an indoor delivery robot domain show a significant improvement of the run-time compared to previous approaches.

INTRODUCTION
Agents or robots acting in the real world generally face the problem that the actions they execute are error-prone, that their perception is noisy and exogenous events change the world in unexpected ways. If agents or robots are controlled by high-level programs written in programming and plan languages like IndiGolog (Giacomo et al., 2009) this may lead to situations where they have to abort high-level missions or draw wrong decisions due to inconsistent knowledge about the world. These phenomena have a negative impact on the performance and dependability of agents and robots acting in the real world, because they fail to fulfill their mission or endanger themselves or the environment.

Consider an office delivery robot which has to deliver a classified document to the office of the department’s head. Because of some shortcomings in its navigation actions the robot accidentally enters the wrong destination, namely a student’s office. If the robot does not recognize that mistake and continues to hand over the document to the person in the room, somebody will be in trouble. However, by using some models and reasoning the robot is able to detect and correct this issue. First the robot might expect some kind of furniture in the head’s office like a leather couch and a particular size of the office. If instead, the robot perceives simple wooden chairs in a small office it is able to detect the inconsistency and may initiate some kind of repair, e.g., reinitialization of the navigation system or revising the robot’s knowledge base.

The latter repair action we refer to as belief management. In (Gspandl et al., 2011), an approach for belief management following model-based principles was developed. It is able to deal successfully with the problems mentioned above. Moreover, the management system was integrated into a reference implementation of an IndiGolog interpreter. Within the approach, inconsistent situations are detected either by a mismatch of expected and received sensing or by checking additional invariants. Once an inconsistent situation is detected, all hypotheses about alternative courses of actions that might explain the inconsistency were generated and tested if they resolve the inconsistency. There is a very high number of hypotheses as they resemble all possible combinations of exogenous events and alternative action outcomes. This had a considerable negative effect on the scalability.

In this paper we present an extension to the approach suggested in (Gspandl et al., 2011). The basic idea is to reduce the number of explanation hypotheses generated. We observed that in the majority of cases an inconsistency is caused by a single faulty action. Thus the repair step handles one sin-
gle fault at a time which is a common assumption in the diagnosis literature. Please note that a single faulty action can cause a significant fraction of the knowledge base to be inconsistent. In independent single fault settings we show how to reduce the number of hypotheses by analyzing which actions are able to influence which facts. Repaired situations that become inconsistent after a while can be repaired again. This means, that we restrict ourselves to unrelated single faults. The idea of dependency analysis has already been successfully applied to speed up automated planning (Hoffmann and Nebel, 2001; Helmert, 2006). It turned out that a number of generated hypotheses are irrelevant and can be omitted because they do not influence the knowledge in question. For instance for the robot example, it makes no sense to not influence the knowledge in question. The remainder of this paper is organized as follows. First, we introduce the situation calculus, IndiGolog and the basic belief management approach. After introducing the running example, we formalize the Diagnosis Templates and show how they can be generated off-line. The application of Diagnosis Templates to a particular inconsistency is presented in the next section. In the experimental result section we present results for a robot delivery domain and compare them with the original approach. We conclude with a discussion and an outlook for future research.

PREREQUISITES

The situation calculus (McCarthy, 1963) serves as the theoretical framework for this work. It is a sorted second order language with equality that allows to reason about actions and their effects. Starting in the initial situation \( S_0 \), histories can be evaluated to define the current situation \( s \). This is done by the special function symbol \( \text{do} : \text{action} \times \text{situation} \to \text{situation} \) denoting situation \( s' \) after performing action \( a \) in situation \( s \) \( (s' = \text{do}(a,s)) \). \( s \) is called a sub-situation of \( s' \) that is a transitive relation. Relational and functional fluents denote situation dependent properties. In Reiter’s variant of the situation calculus (Reiter, 2001), action precondition axioms \( D_{ap} \) of the form \( \text{Poss}(\alpha(x),s) \equiv \Pi_n(x) \) and so-called successor state axioms \( D_{sa} \) of the form \( F(\tilde{x},\text{do}(a,s)) \equiv \varphi^+(\alpha,\tilde{x},s) \lor F(\tilde{x},s) \land \varphi^-(\alpha,\tilde{x},s) \) define the validity effects of actions in situation \( s \), where \( \varphi^+ \) and \( \varphi^- \) are expressions that either fulfill or negate fluent \( F \). In combination with foundational axioms \( \Sigma \), unique name axioms for actions \( D_{una} \), and some axioms about the initial situation \( D_{si0}, D_{isa} \) and \( D_{ap} \) form the so-called basic action theory \( D = \Sigma \cup D_{isa} \cup D_{ap} \cup D_{una} \cup D_{si0} \).

For more details, we refer the reader to (Reiter, 2001). Diagnosis generation follows the approach of History Based Diagnosis (Iwan, 2002). Iwan distinguishes between action variations (\( \text{Varia} \)) and exogenous action insertions (\( \text{Inser} \)) as explanations for inconsistencies between an observation \( \phi \) (a sensor value) and an action history \( \delta \) (a situation term). \( \text{Varia}(a, A(\tilde{x}), s) \equiv \Theta_A(a, \tilde{x}, s) \) and \( \text{Inser}(a, s) \equiv \Theta(a, s) \) state under which conditions \( \Theta_A \) action \( A \) is a variation of action \( a \) and under which condition \( \Theta \) some action \( a \) is a valid insertion into the action history. We use the notion of executable action histories \( \text{Exec}(\text{do}([\alpha_1, \ldots, \alpha_n], S_0)) \equiv \bigwedge_{j \in \{1, \ldots, n\}} \text{Poss}(\alpha_j, \text{do}([\alpha_1, \ldots, \alpha_{j-1}], S_0)) \). \( \text{IndiGolog} \) is a robot programming language that builds upon the situation calculus (Giacomo et al., 2009). The basic action theory \( D \) is extended by a set \( C \) that contains axioms for reifying programs as terms and a set \( \text{Senses} \) collecting all sensing axioms. This yields the extended basic action theory \( D^* = D \cup C \cup \text{Senses} \). The interpreter’s main cycle is responsible for executing programs (incorporating sensing and exogenous events, check program for termination, perform single step transformation). This main cycle was extended in (Gspandl et al., 2011) with a belief management system that maintains a consistent belief. That approach is incorporated into our belief management system we use in this paper.

Additionally the definition of \( \text{Cons} \) (Gspandl et al., 2011) is needed to denote consistent situations. Situations are consistent iff every sensing value \( \text{RealSense} \) is equal to the expected one based on the current history \( SF \) and iff invariants hold at every sub-situation.

Definition 1 (Consistency) Let \( \delta \) be an action sequence and \( a \) a single action. A situation \( \sigma \) is consistent iff \( D^* \models \text{Cons}(\sigma) \) with \( \text{Cons}(\cdot) \) inductively defined as:

1. \( \text{Cons}(\epsilon) \equiv \text{Invaria}(S_0) \)
2. \( \text{Cons}(\delta, a) \equiv \text{Cons}(\delta) \land \text{Invaria}(\delta, a) \lor [SF(\alpha, \delta) \land \text{RealSense}(\alpha, \delta) \lor \neg SF(\alpha, \delta) \land \neg \text{RealSense}(\alpha, \delta)] \)

RUNNING EXAMPLE

Throughout the paper a delivery domain serves as the running example. The task is to deliver objects from room to room, by applying the actions \text{pickup}, \text{goto} and \text{putdown}. We have \text{pickupNothing}, \text{pickWrongObject} and \text{putdownNothing} as variations of the given actions. \text{snatch} and \text{exogMoveObject} form the set of insertions. The set of fluents consists of \( a t, \_is\_at \) and has\_\_object. Additional predicates room and obj are used to enumerate all rooms and objects. Preconditions are...
whether or not fluent \( F \) directly depends on action \( a(\vec{y}) \) in situation \( s \) if at least one of the fluents \( F_1, \ldots, F_n \) depend on action \( a \) and denote it

\[
\text{DepEffect}_{E,F}(\vec{x}, a(\vec{y}), s) \doteq \\
\bigwedge_{i=1}^{n} \text{DepAct}_{F_i}(\vec{x}, a(\vec{y}), s)
\]

with \( \vec{x} \) is an abbreviation for the formal parameters \( \vec{x}_1, \ldots, \vec{x}_n \).

Note that in the above definition, we restrict ourselves to the positive fluent case for readability reasons, i.e., we do not define \( \text{DepAct}_{\neg F} \) for cases where \( \text{DepAct}_{\neg F} \) hold.

To continue our example, \( at(r, s) \) directly depends on action \( goto(r) \). So every set containing fluent \( at(r, s) \) directly depends on action \( goto(r) \). As \( \text{DepAct}_{\neg F}(r, goto(r), s) \) holds, also \( \text{DepAct}_{\neg F}(r, object, goto(r), s) \) holds in \( s \).

By now, we defined if a fluent is dependent on an action, i.e., if the fluent value changes immediately after performing action \( a \). We also need to determine, whether a change of a fluent value renders the execution of an action in the successive situation impossible. Assume, a number of special actions \( \text{switch}_{\text{val}} F_i \) is included in the basic action theory, which allows us to switch the value of a particular fluent \( F_i \). This auxiliary action is needed for axiomatisation.

**Definition 4** For a given fluent \( F(\vec{x}) \), \( \text{DepPoss}_{F}(\vec{x}, a(\vec{y}), s) \) defines whether or not the execution of action \( a(\vec{y}) \) depends on \( F(\vec{x}) \).

\[
\text{DepPoss}_{F}(\vec{x}, a(\vec{y}), s) \doteq \\
\left( \begin{array}{l}
\text{Poss}(a(\vec{y}), s) \land F(\vec{x}, s) \\
\neg \text{Poss}(a(\vec{y}), s) \lor \text{do}(\text{switch}_{\text{val}} F, s)
\end{array} \right)
\]

We write \( \text{DepPoss}_{\neg F} \) to denote \( \text{DepPoss}_{\neg F}(\vec{x}, a(\vec{y}), s) \doteq \text{Poss}(a(\vec{y}), s) \land \neg F(\vec{x}, s) \land \neg \text{Poss}(a(\vec{y}), \text{do}(\text{switch}_{\text{val}} F, s)) \)

**Definition 5** \( \text{DepEffect}_{E,F} \) denotes iff fluent \( E \) is an effect of action \( a \) and depends on fluent \( F \).

\[
\text{DepEffect}_{E,F}(\vec{x}, \vec{y}, a(\vec{z}), s) \doteq \\
\left( \begin{array}{l}
F(\vec{y}, s) \land E(\vec{x}, a(\vec{z}), s) \\
\neg E(\vec{x}, \text{do}(\text{switch}_{\text{val}} F, a(\vec{z})), s)
\end{array} \right)
\]

For instance, fluent \( is\_at(\text{obj}, r, s) \) for an object \( \text{obj} \) and room \( r \) is changed by action \( goto(r) \) iff \( has\_object(\text{obj}) \) holds in the preceding situation. So \( \text{DepEffect}_{\neg, \text{at, has}\_object}(\text{obj}, r, goto(r), s) \) holds for a suitable \( s \).

Definition 6 holds if any fluent \( E_1, \ldots, E_n \) depends on fluent \( F \) via action \( a \), either by precondition or effect.
Definition 6: Given fluent $F$ with parameters $y$, and fluents $E_1, \ldots, E_n$ with parameters $x_1, \ldots, x_n$, effected by action $a(z)$ in situation $s$, $DepFl_{E_1, \ldots, E_n, F}(x'_1, \ldots, x'_n, y, a(z), s)$ is defined as:

$$DepFl_{E_1, \ldots, E_n, F}(x'_1, \ldots, x'_n, y, a(z), s) = \exists y, z. \text{Varia}(a, a_1(z), s) \wedge \text{Inser}(a, s)$$

where $F_1, \ldots, F_n$ contain all fluents with $\Phi$ abbreviating the formula $\exists x'_1, \ldots, x'_m, y'_1, \ldots, y'_n, z, a(z) \wedge \text{DepFl}_{E_1, \ldots, E_n, F_1}(x'_1, \ldots, x'_m, y'_1, z, a(s)) \wedge \ldots \wedge \text{DepFl}_{E_1, \ldots, E_n, F_n}(x'_1, \ldots, x'_m, y'_1, z, a(s))$.

For readability reasons we omit the negative fluent cases in Definition 5 and 6.

Definition 7: Assume $F$ is a fluent. An edge-labeled and node-labeled non recursive, fully expanded tree $T_F$ is a fluent dependency tree for $F$ iff the following properties hold:

1. The root is labeled with $F$.
2. If $n$ is a node of $T_F$ with label $F_1, \ldots, F_n$, then every edge from node $n$ to successor node $n_{\text{succ}}$ is labeled with an action term $a$ with $\exists s, x'_1, \ldots, x'_n, y, a(z). \text{DepAct}_{E_1, \ldots, E_n, F_1}(x'_1, \ldots, x'_n, y, a(z), s)$.
3. The successor node $n_{\text{succ}}$ with edge $a$ of an arbitrary node $n$ labeled $E_1, \ldots, E_m$ in the tree is labeled the following way:

$$n_{\text{succ}} = \begin{cases} \$ & \exists s, \bar{z}. \text{Varia}(a, a_1(\bar{z}), s) \\ F_1, \ldots, F_n & \Phi \end{cases}$$

where $F_1, \ldots, F_n$ contain all fluents with $\Phi$ abbreviating the formula $\exists x'_1, \ldots, x'_m, y'_1, \ldots, y'_n, \bar{z}, a(z)$.

4. If $n$ is labeled with $\$, it is a leaf.

5. Let $n$ be a node in $T_F$. Then, define $H(n)$ as the set of edge labels from $n$ to the root.

The fluent dependency tree $T_F$ is defined analogously, but left out for space reasons.

Note that if $n$ is a leaf in $T_F$, $H(n)$ contains exactly one variation or insertion. This change occurs at the first entry of $H(n)$. An example for expanding a node is $\text{has\_object}(o, s)$ that forms a correct expansion of node $\text{is\_at}(o, r, s)$ via action $\text{goto}(r)$, as the robot has to hold object $o$ in order to achieve a truth value change of fluent $\text{is\_at}(o, r, s)$ by action $\text{goto}(r)$. Hence, fluents that hold in node $n$ either make the execution of the action that labels the edge to the parent possible, or cause the action leading to the parent’s fluents. Fig. 1 shows an example fluent dependency tree for fluent $\text{is\_at}(o, r, s)$.

Fluent dependency trees form explanations for a single fluent $F$. Edges to the root contain actions that directly lead to the root fluent in some situation $s$. The first node level consists of fluents necessary to achieve the root fluent via the action of the corresponding edge. Edges on the next level contain again actions that lead to its parent node. This interplay between fluents and actions is continued until there is no valid expansion anymore.

Algorithm 1: gen_dependency_tree(root, $V$, $E$)

```
1 foreach a \in Actions do
2  if a \in DepAct(E_1, \ldots, E_n) then
3    if a \in Changes then
4      V \leftarrow V \cup \{node_{a}\};
5      E \leftarrow E \cup \{\text{root, node}_{a}, a\};
6      continue;
7    child \leftarrow \$;
8    foreach f \in Fluents do
9      if f \in DepFl(E_1, \ldots, E_n, a) then
10        child \leftarrow child \cup \{f\};
11      if child \neq \$ \wedge \neg \text{Cycle}(child) then
12        V \leftarrow V \cup \{\text{node}_{child}\};
13        E \leftarrow E \cup \{\text{root, node}_{child, a}\};
14        gen_dependency_tree(node_{child}, V, E);
15    if |root| = 1 then // no child node
16      delete root;
```

Fluent dependency tree generation is shown in Algorithm 1. Besides the target fluent, several sets are given as input that allow simple inclusion checks. The set $\text{Actions}$ of all actions and the set $\text{Fluents}$ of all
fluent formulas directly follow from the basic action theory. Note that we assume finite domains here. The set Changes of all insertions and variations is based on the diagnosis principles in use defined by (Iwan, 2002). The set DepAct(E₁, ..., Eₙ) and the set DepFl(E₁, ..., Eₙ, a) follow from the corresponding predicates. The former set contains all actions that can lead to E₁ ∨ ... ∨ Eₙ and the latter DepFl(E₁, ..., Eₙ, a) resembles all fluents that action a depends on leading to E₁ ∨ ... ∨ Eₙ.

The algorithm applies depth-first search and receives the target fluent labeled node as input. This node is expanded recursively until a valid leaf is detected, a loop occurred or there exists no valid expansion anymore. The algorithm starts by selecting those actions from the set of all actions that yield the required fluent set {E₁, ..., Eₙ}. Note that in the first iteration, this set contains the fluent F only. If this action is already a variation or insertion (l. 3), then a valid leaf is found and added to the given root node. In this case, the algorithm proceeds with the next action.

If no valid leaf is found the label of the successor node is built by iterating over all fluents and selecting those that influence at least one of the fluents E₁, ..., Eₙ by action a (l. 7-8). If the resulting label is non-empty and does not occur in the path to the root (i.e., it is not in H(n)), it is added to the parent node by edge a and the algorithm is called recursively on the new node (l. 11-14). In order to avoid non-terminating recursion the predicate Cycle checks if the node’s label is already part of the current path. If the given node (parent) could not be expanded, it is deleted (l. 15). Due to the recursive behavior of the algorithm, this deletes non-expandable paths, resulting in a tree that is either empty or contains valid leaves only.

To illustrate the algorithm, we show how the fluent dependency tree $T_{is\_at(obj, r)}$ is expanded. The algorithm starts with a root node labeled is\_at(obj, r). Out of all actions, the action goto(r) is the first that influences the root (DepAct(is\_at(obj, r, goto(r), s)). As it is neither a variation or insertion, a new child node is built containing all fluents that condition the parent label by applying action goto(r). In our case, the only conditioning fluent is has\_object(obj) (if object obj is not in room r before but after going there, the robot must hold object obj). So, the child node has\_object(obj) is added to the parent by edge goto(r) and the algorithm is called recursively.

The first action influencing has\_object(obj) in the recursive call is pickupWrongObject(obj). This is a variation of action pickup(obj), so it is added directly to the parent with label $\exists$ and we found the first valid leaf (shown in Fig. 2). The node has\_object(obj) can further be expanded by the endogenous action pickup(obj) leading to node is\_at(obj, r₁), at(r₁), ¬has\_object(obj). Though this node is influenced by action goto(r₁) and putdown(obj) it cannot be expanded by any of them, because their equivalent child label has\_object(obj) already occurred along the path to the root. In contrast the exogenous action exogMoveObject(obj, r₁) forms a valid expansion, leading to the second leaf. This intermediate step is shown in Fig. 3. Nodes that have to be deleted are drawn with dashed lines. Finally after returning from the recursive calls the root node can be expanded by another action, namely exogMoveObject leading to a third leaf. The final tree is shown in Fig. 1.

```
Figure 2: Dependency tree after first leaf is found.
```

For a node n of $T_{F(\bar{x})}$, $H(n)$ is defined to be the sequence of edge labels on the path in $T_{F(\bar{x})}$ from node n to the root. If n is a leaf, $H(n)$ contains exactly one variation or insertion (cf. Def. 7).

**Definition 8** Let $T_{F(\bar{x})}$ be a dependency tree for fluent $F(\bar{x})$. For every leaf node $n_1, \ldots, n_m$ of $T_{F(\bar{x})}$, $dt_{F(\bar{x})}(n_i) = H(n_i)$ with $i \in \{1, \ldots, m\}$ is a Diagnosis Template. DT$^{F(\bar{x})}$ denotes the set of all Diagnosis Templates for a fluent $F$.

For Diagnosis Template $[a_1, \ldots, a_n]$, by definition, action $a_1$ is always a variation or an insertion. Actions $a_2, \ldots, a_n$ act as conditions for action $a_1$ to possibly fulfill $F(\bar{x})$. For instance, if is\_at(obj, r) does not hold, it is achieved through pickupWrongObject(obj) only if pickupWrongObject(obj) is followed by goto(r). The Diagnosis Templates for $T_{is\_at(obj, r)}$ are

$$
\{[\text{pickupWrongObject(obj), goto(r)]}, [\text{ezogMoveObject(obj), pickup(obj), goto(r)]}, [\text{ezogMoveObject(obj)]}
$$

In the following theorem we show that every action sequence that consists of exactly one variation or insertion and leads to a fluent $F(\bar{x})$, can be reduced to a Diagnosis Template $dt_{T_i}^{F(\bar{x})} \in DT^{F(\bar{x})}$. The theorem guarantees that all possible explanations for $F(\bar{x})$ are represented in the Diagnosis Templates.

```
Figure 3: Fluent Dependency Tree before deleting nodes.
```
Such a sequence is called a minimal explanation, if it is derived the following way: an executable consistent action sequence \([a_1, \ldots, a_n]\) deriving fluent \(F(\vec{x})\) with \(3s, \neg F(\vec{x}, s) \land F(\vec{x}, do(a_1, \ldots, a_n), s)\) is first reduced to \([a_k, \ldots, a_m]\) with \(k \geq 1\) and \(m \leq n\). \(a_k\) is further required to be an insertion or a variation and \(a_m\) has to lead to the fluent \((3s, \neg F(\vec{x}, do(a_1, \ldots, a_{n-1}), s)) \land F(\vec{x}, do(a_1, \ldots, a_n), s))\). Second, we remove all actions \(a_i\) with \(i\) from \(m\) to \(k\) that are not necessary to lead to fluent \(F(\vec{x})\), denoted by \(\exists F(\vec{x}, do(a_1, \ldots, a_{n-1}, a_{n+1}, \ldots, a_m), s)\). The resulting sequence is still required to be executable and consistent. \(m\) is chosen in a way that the final length of the sequence is minimal. Thus, such sequences start with an insertion or variation and consist of actions only that are necessary to fulfill fluent \(F(\vec{x})\).

**Theorem 1** The set of Diagnosis Templates \(DT^{F(\vec{x})}\) contains all minimal action sub-sequences with exactly one insertion or variation leading to situation \(s^*\), where \(F(\vec{x}, s^*)\) holds. Thus, every action sequence \([a_1, \ldots, a_n]\) with exactly one variation or insertion that leads to fluent \(F(\vec{x})\) without any sub-sequence being a Diagnosis Template. We show that the sequence \([a_2, \ldots, a_n]\) that is assumed to be a minimal explanation already, either contains an action that is not part of the fluent dependency tree \(T_F(\vec{x})\) or that \([a_1, \ldots, a_n]\) does not cover a full path within \(T_F(\vec{x})\).

The root of the fluent dependency tree is labeled \(F(\vec{x})\) (rule 1 of Definition 7). The last action \(a_n\) directly leads to \(F(\vec{x})\), so it fulfills \(\exists s. DepAct(\vec{x}, a_n, s)\). Thus, there exists an edge from the tree root labeled \(a_n\) (rule 2 of Definition 7). The node following this edge is labeled with fluents \((3s, \neg F(\vec{x}, s) \land F(\vec{x}, do(a_1, \ldots, a_n), s))\). The proof is by contradiction. We try to find an executable and consistent action sequence \([a_1, \ldots, a_n]\) with exactly one variation or insertion that fulfills \(F(\vec{x})\) without any sub-sequence being a Diagnosis Template. We show that the sequence \([a_2, \ldots, a_n]\) that is assumed to be a minimal explanation already, either contains an action that is not part of the fluent dependency tree \(T_F(\vec{x})\) or that \([a_1, \ldots, a_n]\) does not cover a full path within \(T_F(\vec{x})\).

**Algorithm 2:** fulfill\(_{F(\vec{x})}\)(\(DT^{F(\vec{x})}, \vec{a}, \vec{v}, S_0\))

**Input:** a set of diagnosis templates \(DT^{F(\vec{x})}\), the current action sequence \([a_1, \ldots, a_n]\), the visited actions sequence \([a_1, \ldots, v_n]\) an initial situation \(S_0\)

**Output:** a set of diagnosis hypotheses \((H_1, \ldots, H_n)\)

1. \(H \leftarrow \emptyset; \quad //\) hypothesis candidates
2. \(dtSet \leftarrow \emptyset; \quad //\) diag templates next iter
3. \(s_{-1} \leftarrow do(a_1, \ldots, a_{m-1}), S_0); \quad //\) prev. sit
4. foreach \(dt_i \in DT(\vec{x})\) do
5. \(\text{if } |dt_i| = 1\) then
6. \(\text{if } \exists F. Varia(dt_i, a_1, a_m(s), s_{-1}) \text{ then} \)
7. \(H \leftarrow H \cup \text{applyVaria}(s_{-1}, dt_i, a_1, \vec{v});\)
8. \(\text{else if } \text{Inser}(dt_i, a_1, do(a_m, s_{-1})) \text{ then} \)
9. \(H \leftarrow H \cup \text{applyInser}(s_{-1}, dt_i, a_1, \vec{v});\)
10. \(\text{else if } |dt_i| > 1\) then
11. \(\text{if } \exists F. Varia(dt_i, a_1, a_m(s), s_{-1}) \text{ then} \)
12. \(\text{dtSet } \leftarrow \text{dtSet } \cup \{dt_i \setminus dt_i, a_1\}\)
13. \(\text{else} \)
14. \(\text{dtSet } \leftarrow \text{dtSet } \cup dt_i\)
15. \(\text{if } |\vec{a}| > 1\) then
16. \(H \leftarrow H \cup \text{fulfill}_{F(\vec{x})}(\text{dtSet}, [a_1, \ldots, a_{m-1}], [a_m, \vec{v}], S_0);\)
17. return \(H\)

Recalling Def. 1 about consistency, we see that an inconsistency either results from a mismatch between a sensed fluent and its value in the predecessor situation or from an invariant that does not hold in the current situation. In the former case, there are two possibilities. We can either replace the sensing action by a variation, or we can change the action sequence in such a way that it leads to the sensed fluent value, i.e., leading to a situation where the sensed fluent value holds after performing this action sequence. Similarly, for the latter case of an invalid invariant, we can identify a single fluent or a set of fluents whose values need to be altered in order to fulfill the invariant.

Thus, we can narrow down the task of diagnosis generation to replacing a sensing actions with a variation (which is trivial) on one hand, and altering the given action sequence to fulfill a given fluent (or a set of fluents), on the other hand.

According to the formalism of successor state axioms \(F(\vec{x}, do(\alpha, s)) \equiv \varphi^+(\alpha, \vec{x}, s) \lor F(\vec{x}, s) \land \neg \varphi^-(\alpha, \vec{x}, s)\) a fluent can either be achieved by executing an action that leads to this fluent value, or omitting actions that negate the fluent. In order to change an action sequence to yield a particular fluent value we execute Algorithm fulfill to find an action sequence enabling the fluent and on the other hand we have an Algorithm maintain that avoids that a fluent value is negated. The resulting hypotheses set is the union of a possible sensing variation and the result sets of Algorithms 2 and 3. All consistent hypotheses form the diagnosis set. So the diagnosis set is a subset of the hypothesis set that is again a subset of all hypotheses ap-
Algorithm 3: maintain\(_{F}(\bar{\alpha}, \bar{\nu}, S_0)\)

**input**: the current action sequence \([a_1, \ldots, a_m]\), an action sequence of visited actions \([v_1, \ldots, v_p]\), an initial situation \(S_0\)

**output**: a set of diagnosis hypotheses \((H_1, \ldots, H_n)\)

1. \(HSet \leftarrow \emptyset\);  // hypothesis candidates
2. \(a_{last} \leftarrow [a_1, \ldots, a_{m-1}]\);  // prev. action seq
3. \(s \leftarrow do([a_1, \ldots, a_m], S_0);\)  // current sit
4. \(s_{-1} \leftarrow do([a_1, \ldots, a_{m-1}], S_0);\)  // prev. sit
5. if \(F(\bar{x}, s_{-1}) \land \neg F(\bar{z}, s)\) then
6. \(\text{if } \exists \alpha, \bar{y}. \text{Varia}(\alpha, a_{m}(\bar{y}), s_{-1}) \land \text{Exec}(\alpha, s_{-1})\) then
7. \(\text{foreach } f \in \text{Fluents do}\)
8. \(\text{if } \exists \bar{y}, \bar{z}. \text{DepEffect}_{F,fl}(\bar{x}, \bar{y}, a_{m}(\bar{y}), do(\bar{a}, S_0))\) then
9. \(HSet \leftarrow HSet \cup [a_1, \ldots, a_{m-1}, \alpha, v_1, \ldots, v_p]\)
10. \(\text{fulfill}_{f(\bar{y})}([\bar{a}_{last}, [a_m, \bar{v}]], S_0);\)
11. \(HSet \leftarrow H \cup [\text{maintain}_{F}(\bar{a}_{last}, [a_m, \bar{v}]), S_0];\)
12. else
13. \(\text{if } |\bar{a}| > 1\) then
14. \(HSet \leftarrow HSet \cup \text{maintain}_{F}(\bar{a}_{last}, [a_m, \bar{v}]), S_0);\)
15. return \(HSet\)

Applied in (Gspandl et al., 2011). The Algorithms fulfill and maintain are called for every fluent that has to be changed.

Algorithm 2 iterates recursively over the given action sequence and tries to apply the Diagnosis Templates as defined in the previous section. It takes the initial situation \(S_0\), the action sequence \([a_1, \ldots, a_m]\) the visited action sequence \([v_1, \ldots, v_p]\) and the set of Diagnosis Templates \(DT^{-F}(\bar{x})\). First, the algorithm initializes the hypothesis set and the new Diagnosis Template set and sets abbreviations for the current situation \(s\) and the last situation \(s_{-1}\). Then every Diagnosis Template is processed. If the length of the template equals one (l. 5; so solely the insertion or variation is remaining), all possible applications on the remaining action sequence are computed by applyVaria or applyInsair, respectively. The resulting hypotheses are added to the resulting hypothesis set. If the remaining length of the template is greater than one, the current action is checked for unification against the last template action. This means a further condition of the template is fulfilled and the template is reduced by the last action (l. 11). Otherwise the whole template is added to the template set for the next iteration. Finally, the action sequence is reduced by the last action, which is added to the set of visited actions. Before returning the resulting hypothesis set, the algorithm is called recursively if the new action sequence \([a_1, \ldots, a_{m-1}]\) is not empty (l. 16).

Algorithm 3 maintains a fluent value if it already held, but was changed by some action. Similarly to Algorithm 2, it takes the initial situation \(S_0\), the current action sequence \([a_1, \ldots, a_m]\) and the sequence of visited actions \([v_1, \ldots, v_p]\) as input. First, the hypothesis set is initialized, followed by abbreviations for the current and the predecessor situation \(s\) and \(s_{-1}\). Then it is checked if the current action changes the fluent value \(F(\bar{x}, s)\) to false (l. 5). If there exists an executable action variation it is added to the hypotheses set (l. 6). Furthermore the effect of the fluent negating action is tried to be avoided. Therefore, relevant fluents to enable the effect are identified in l. 9. If such a fluent is found, it is prevented by maintaining the negative fluent in l. 10 and by fulfilling the negative fluent in l. 11. In both cases resulting hypotheses are added to the hypothesis set. If the main fluent \(F(\bar{x}, s)\) is not negated by the current action and the remaining action sequence is not empty, the algorithm is called recursively (l. 14).

**EXPERIMENTAL RESULTS**

Similarly to (Gspandl et al., 2011) we evaluated the performance of the presented system in a simulated delivery domain. Every agent had to pickup and deliver 3 objects and finally return to a predefined position. Every scenario was executed 100 times with a timeout of 2 hours.

The fault types are the same as in the most difficult scenario of (Gspandl et al., 2011) where a robot picks up the wrong object (20%) or fails to pick it up totally (20%), is unable to put down an object (30%), or the object is snatched from the agent (15%). The probabilities of these faults are shown in brackets. Additionally, we applied another exogenous event exogMoveObject(obj, room) where an object obj is moved invisibly to room. The probability is 2% for each object. Thus, in case of 50 objects, every 2 actions an object is moved on average. The only condition constraining this exogenous action is that the object is not carried by the robot. Compared to the other changes, this exogenous action is more difficult to handle, because it is almost always admissible and depends directly on the number of rooms and objects. Every three execution steps the robot receives sensing if it is carrying some object and it senses for three objects if they are located in the same room. The average number of sensed ghost objects (an object that is sensed in the same room though it is located in another room) is 3.32 across all scenarios. Every agent operated with a limited pool size of 250 diagnosis.

In order to analyze run-times and success rates we simulated 4 scenarios with different object room combinations. A contest following our approach taken in (Gspandl et al., 2011) but applying single fault repairs (named G) was confronted by a contestant with the proposed system (named T). Though the templates could be generated offline, it is done at the beginning of each of T’s run, as it takes a few milliseconds only. The different settings, which are used in our experiments are the following: 3 objects and 20 rooms (20r), 10 objects and 29 rooms (29r), 29 objects and 50 rooms (50r) and 60 objects and 71 rooms (71r).

The run-times of the contestants are shown in Table 1(a). Table 1(b) presents the number of runs that ran into a timeout in percent. The run-times include successful runs only (all 3 tasks could be established).
Table 1: Experimental results: (a) Average run times per setting in seconds. (b) Average number of timeouts in percent. (c) Average number of successfully executed tasks (maximum is 3). (d) Average number of successfully executed missions in percent.

<table>
<thead>
<tr>
<th></th>
<th>20r</th>
<th>29r</th>
<th>50r</th>
<th>71r</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>417</td>
<td>27/0</td>
<td>28</td>
<td>314</td>
</tr>
<tr>
<td>T</td>
<td>29</td>
<td>40</td>
<td>165</td>
<td>274</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>20r</th>
<th>29r</th>
<th>50r</th>
<th>71r</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>0</td>
<td>19</td>
<td>91</td>
<td>95</td>
</tr>
<tr>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

As contestant G ran into timeout in complex scenarios nearly all the time, those run-times have to be taken with a pinch of salt. If we include all runs, contestant G needs 5323s, 6647s and 6827s for 29, 50 and 71 rooms. These run-times are obviously bounded by the 2 hours timeout, whereas the other run-times do not change significantly.

Table 1(c) comprises the number of successfully executed tasks, where the maximum of executed tasks equals 3. The average number of runs where all 3 tasks were finished is shown in Table 1(d). In contrast to the approach taken in (Gspandl et al., 2011), the proposed system is able to keep the performance while increasing the number of rooms and objects. Thus, we can conclude that the presented system is able to speed up run-times tremendously while keeping the success rate on a stable level.

CONCLUSION AND FUTURE WORK

According to (McIlraith, 1999), belief management focuses on the question “What happened” in terms of actions instead of “What went wrong” identified by faulty components. In the context of plan diagnosis (Roos and Witteveen, 2009), the authors aim to find a set of abnormally qualified actions that maintains the consistent, maximum informative belief. Our approach differs in two ways. First, it seeks to maintain a full consistent belief (no fluent turns undefined). Second, the algorithm aims for repairing the faulty action sequence, allowing the agent to continue its mission with a consistent belief. (Fritz, 2009) proposed a framework on plan generation and plan execution including monitoring, but paid less attention to the subject of correcting knowledge. In order to tackle the problem of “What happened” several approaches were proposed in the literature. (Grastien et al., 2007) established a connection to satisfaction problems, whereas (Sohrabi et al., 2011) defined the problem as a planning problem. Within the planning domain several pre-computed, domain specific data structures are used to speed up planning times. This exploitation was first proposed by (Hoffmann and Nebel, 2001) and later refined by (Helmer, 2006). Building up on these strategies we show that Diagnosis Templates originating from a domain description lead to huge run-time savings and improvements in the success rate. For future work we plan to overcome the basic assumption made throughout this paper, to deal with unrelated single faults only. Extending Diagnosis Templates to multiple faults is expected to solve this issue. Furthermore, we are interested to do further research in analyzing the domain’s impact on the performance of the system. In the future we will also investigate how the execution of additional discriminating actions can resolve ambiguous diagnoses.

REFERENCES


