Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-based Systems Group
Prof. Gerhard Lakemeyer, Ph. D.

Master's Thesis

Using Platform Models for a Guided Explanatory Diagnosis Generation for Mobile Robots

Daniel Habering

Supervisor: Prof. Gerhard Lakemeyer Ph.D.
Supervisor: Prof. Dr. Matthias Jarke

Advisor: Till Hofmann

August 19, 2019

Contents

1 Introduction

Any agent on a robotic platform, that operates in the real world, has to deal with all the unexpected events and failures that may happen. Most high-level control agents try to keep and maintain a belief about the current state of the world. The belief is then used to decide how to act in order to pursue its objectives.

Actions executed in the real world may fail randomly, actions have an unexpected outcome, unexpected events occur or hardware fails. To cope with these mishaps and stay operable, an agent has to be able to determine the exact cause and effect of the failure. This is called diagnosis and plays an important role in enabling robots to operate in the real world.

Consider a robot that operates machines in a factory. Its task is to retrieve parts from the output of a machine, transport it to a different machine and place it into the input, resulting in the following history:

$$H = \{\texttt{grab}(\texttt{R1}, \texttt{P}, \texttt{A}), \texttt{move}(\texttt{R1}, \texttt{A}, \texttt{B}), \texttt{place}(\texttt{R1}, \texttt{P}, \texttt{B}), \texttt{instruct}(\texttt{B})\}$$

After executing this task and transporting product P from machine A to B, the robot senses that machine B does not react as expected after putting something into the input. Ignoring this inconsistency may bring the production to a hold, so it is necessary that the robot determines the cause of the inconsistency and what effect it has on the world. There are several possible reasons, such as failing picking or putting the product, a failure in the navigation or even broken machines. Diagnosis algorithms aim at determining all the possible explanations and reason about these possible explanations such that the agent can make proper decisions afterwards.

In order to enable the agent to maintain a consistent belief about the state of the world, we are mainly interested in the actual state of the world and the cause of the failure. As mentioned above, an agent will try to execute a sequence of actions. History-based diagnosis aims to accommodate a history of executed actions by inserting and modifying actions, such that the history is consistent with the observation again. This is done by inserting fitting exogenous actions and replacing actions with faulty alternatives, such that resulting belief of the alternated history is consistent with the observations.

Following the previous example, the robot first executed a pick action to pick product P from machine A. It then drove to machine B and put the product into the input. A possible explanation for machine B not reacting to the product, would be that the `grab` action failed and that the product is still at the output of machine A. Therefore, the history of executed actions can be adapted by replacing the grab action with a failed variant, resulting in the adapted history H_1':

$$H_1' = \{\texttt{grab\_failed}(\texttt{R1}, \texttt{P}, \texttt{A}), \texttt{move}(\texttt{R1}, \texttt{A}, \texttt{B}), \texttt{place\_nothing}(\texttt{R1}, \texttt{P}, \texttt{B})\}$$

Another explanation would be that the robot dropped the product while driving to machine B. In this case, an exogenous action `drop` can be inserted into the history, resulting in the adapted history H_2':

$$H_2' = \{\texttt{grab}(\texttt{R1}, \texttt{P}, \texttt{A}), \texttt{move}(\texttt{R1}, \texttt{A}, \texttt{B}), \texttt{drop}(\texttt{R1}, \texttt{P}), \texttt{place\_nothing}(\texttt{R1}, \texttt{P}, \texttt{B})\}$$

In both cases propagating the effects of the adapted histories onto the initial situation would result in a consistent belief about the world state and the agent can decide on the next goals, e.g. removing the remaining product from machine A.

However, in order to find all possible consistent history adaptations for given observations and a given history of executed actions, one needs a description of all possible action variations and exogenous actions. With these descriptions, it is possible to use existing planning algorithms to generate diagnosis candidates. Existing history-based diagnosis approaches search for all possible replacements and insertions that are consistent with an observation. This is sufficient for reevaluating the outcome of a failed plan.

However, under the assumption that most failures are caused by malfunctioning hardware components, re-evaluating each action isolated is ignoring the fact that a malfunctioning hardware will also influence other actions. The malfunctioning hardware not only influences the actions of the executed history, but actions of future plans as well.

In this thesis, we want to extent existing history-based diagnosis approaches by incorporating knowledge about the hardware platform. A hardware platform consists of several component. The agent may already have components models in order to decouple the planning process from the hardware. Hofmann et al. (2018b) introduced a form of abstract planning, which uses hardware models to transform an abstract plan into an executable one. We want to use the component models and their constraints on actions in order to not only determine the state of the world after the failed plan, but also enable the agent to determine the underlying failure, recovering from it, if possible, and make more informed decisions. We also examine to which extent the

component models can be used for restricting the planning process used to generate diagnosis candidates. By keeping a history of state changes of each component, all possible action variations at a certain point of time can be determined. By encoding the platform models into the domain model and using them as a guide to create all needed action variations, this approach aims to tackle the drawbacks of the history-based diagnosis approach while keeping the beneficial outcome about what actually happened.

After creating diagnosis candidates that explain an observation, we want to use *active diagnosis*, which is a method for selecting sensing actions in order to rule out false diagnosis candidates. The knowledge about which component is more likely to fail can be used when deciding for a sensing action, in order to check the most likely cause first.

Eventually, the agent identified one or more valid diagnosis candidates and their effects onto the world without being able to reduce the set of diagnosis candidates any further. By trying to repair or reset possible faulty hardware components, the agent can try to recover from the detected failure and may continue the production. However, in some cases the recovery fails or the agent is not able to reset certain hardware components. In these cases, the agent can decide to disqualify itself from continuing the operation and call for human intervention. In multi-robot scenarios, the other robots will benefit from this decision, since the broken robot would possibly block resources that could be used by other functional robots.

We will first introduce the relevant background of planning, platform models and the domain in which the approach will be evaluated in Chapter 2. Afterwards related work and different approaches to the diagnosis problem will be presented in Chapter 3. The actual approach is described in more detail in Chapter 4. We will then evaluate our approach in the RoboCup Logistics League domain and present the results in Chapter 5, before we conclude in Chapter 6.

2 Background

2.1 Diagnosis

In the context of high-level control agents, diagnosis is the task of determining what is wrong. However, there are differences in different diagnosis approaches about "what" actually means. Given an observation, that contradicts normal or expected behaviour, diagnosis approaches search for consistent explanations of the inconsistency. In the context of a high-level control agent, which has a description of the controlled system available, diagnosis algorithms aim to determine components of the system, that could explain the observation if assumed to act abnormally (Reiter, 1987). These diagnosis approaches are described as model-based, since they utilize a model of the system. Of course, there may exist multiple possibilities for faulty components, that could explain the observation. So, rather than searching for one set of components, diagnosis algorithms determine multiple diagnosis candidates. A diagnosis candidate is a set of components assumed to act abnormally, consistent with the observation that contradicted the assumed behaviour.

In the context of an agent, that executes actions in order to interact with the environment and senses failures as observations in the environment, that are inconsistent with the expected effects of the executed actions, the actions can be viewed as components of a system. *Consistency-based* diagnosis approaches try to mark actions as abnormal in order to explain the inconsistent observations.

2.1.1 Explanatory Diagnosis

In dynamic environments, it a makes more sense to focus on action sequences, that modified the environment, than to determine faulty components of a system in order to explain failures (Mühlbacher and Steinbauer, 2016b). So, *explanatory diagnosis* approaches search for a sequence of actions, that explains an observation, leading to an explanation about *what happened*, rather than *what is wrong.* In the diagnosis of dynamic systems, this approach can help determining the actual resulting state of a system based on a contradicting observation (Mcllraith, 1999). By evaluating the conjectured actions and their effects on the world, an high-level agent can determine hypothetical world states that are consistent with the observation.

Therefore, in contrary to typical model-based diagnosis approaches, explanatory diagnosis approaches use a description of possible executable actions and their effects together with an observation and searches for sequences of actions, that, for a given initial state, are consistent with the observations. Note, that the initial state can be the result of an already executed sequence of actions. Therefore, explanatory diagnosis searches for possible consistent continuations for a given history of executed actions. However, the sequence of already executed actions is not re-evaluated in the hindsight of a unexpected observation. Mcllraith (1999) showed that calculating explanatory diagnosis candidates is analogous to planning.

Following the example from Chapter 1, a possible explanatory diagnosis candidate of the observed unexpected reaction of the machine B would be conjecturing a `knock_off` action to the end of the executed history, resulting in the continued history H_3'.

$$H_3' = \{\texttt{grab(R1,P,A)}, \texttt{move(R1,A,B)}, \texttt{place(R1,P,B)}, \texttt{knock\_off(R1,P,B)}\}$$

However, another explanation would be a failed *grab* action at machine A, resulting in the product still being at machine A. Explanatory diagnosis approaches are not able to generate such an explanation, since past actions do not get reevaluated.

2.1.2 History-based Diagnosis

For any high-level control agent that controls a robot in a dynamic environment by selecting sequences of actions to reach certain goals, the explanatory diagnosis approach is not sufficient to deal with any undetected failures during the action execution. Therefore, Iwan (2002) presented an approach that diagnoses a history of executed actions in the hindsight of an observation, which is inconsistent with the expected world state. The history-based diagnosis then searches for consistent alternations of the history, by replacing actions with faulty variations and inserting exogenous actions. A faulty variant of a grab action could model that the robot grabbed the wrong object, or no object at all. A diagnosis candidate of such a history-based diagnosis approach is given as an alternated version of the history of executed actions. The diagnosis candidates have to be consistent with the observation and executable given the description of actions, their preconditions, their effects and the initial state of the world. Note, that events that are not under the control of the agent are modelled as exogenous actions and can be inserted into the history at any point. It is shown that, similar to calculating explanatory diagnosis candidates, calculating history diagnosis candidates is analogous to planning as well (Sohrabi et al., 2010).

In our example, a possible explanation, generated by a history-based diagnosis approach, would be replacing the *grab* action with a faulty alternative with no effect, such that the position of the product is still at machine A in the resulting world state.

$$H'_4 = \{\texttt{grab\_nothing}(\texttt{R1}, \texttt{P}, \texttt{A}), \texttt{move}(\texttt{R1}, \texttt{A}, \texttt{B}), \texttt{place\_nothing}(\texttt{R1}, \texttt{P}, \texttt{B})\}$$

A different faulty action variation of the *grab* action may represent that the robot did knock the product off its position, resulting in the product to be lost.

$$H'_5 = \{\texttt{grab\_knock\_off}(\texttt{R1}, \texttt{P}, \texttt{A}), \texttt{move}(\texttt{R1}, \texttt{A}, \texttt{B}), \texttt{place\_nothing}(\texttt{R1}, \texttt{P}, \texttt{B})\}$$

Note, that in both cases the *place* action has to replaced by a faulty variation as well, since there is no product hold, that can be placed. Additionally, history-based diagnosis produce several explanations for what could have happened, but it lacks the ability to determine the cause of a failure. Assuming that most of the failures, that occur when executing an action, are caused by malfunctioning hardware components, we want to integrate knowledge about the underlying hardware platform into the diagnosis process in order to enable the diagnosis process to reason about the underlying hardware as well.

2.2 Active Diagnosis

One characteristic, that all of the diagnosis approaches have in common, is that they may generate multiple diagnosis hypotheses. There are different ways to decide which diagnosis hypothesis to choose. Most diagnosis approaches follow *Ockham's razor* and choose the diagnosis with the minimal amount of assumed faults. Likewise, domain knowledge about fault probabilities can be used to exclude unlikely hypotheses. But even then, there might exist situations where we have to decide between multiple, similarly likely diagnosis hypotheses.

To deal with this problem, *active diagnosis* can be used. *Active diagnosis* is an approach that chooses suitable sensing actions for excluding hypotheses from the set of valid diagnosis hypotheses. This is done by aggregating the shared knowledge of all diagnosis hypotheses, containing every piece of information that is true in every resulting world state of the diagnosis hypotheses. All sensing actions whose preconditions are fulfilled in the common belief and exclude at least one diagnosis candidate for each possible outcome, are suitable for reducing the set of possible diagnosis candidates. However, it is desirable to use a minimal number of sensing actions in order to find the true diagnosis as fast as possible.

Following Mühlbacher and Steinbauer (2014), selecting the most desirable executable sensing action is done by choosing the action that has the biggest impact onto the pool of diagnosis candidates. The impact is defined as the mutual information, which is defined in the terms of entropy implied by the probabilities of the different diagnosis candidates:

$$I(\alpha(\vec{x}); W(s)) = H(\alpha(\vec{x})) - H(\alpha(\vec{x})|W(s)) \tag{2.1}$$

where $I(\alpha(\vec{x}); W(s))$ denotes the information gain of the sensing action α, grounded with a parameter vector $\vec{x}$, in regard to the set of diagnosis candidates $W(s)$ in the situation s. $H(\alpha(\vec{x}))$ represents the impact of the sensing action on the pool of diagnosis candidates and $H(\alpha(\vec{x})|W(s))$ depicts the influence of sensor noise.

Any sensing action $\alpha(\vec{x})$ will split the pool of diagnosis candidates $W(s)$ into two parts (Ω and Λ). Ω contains all diagnosis candidates that are consistent with a positive outcome of $\alpha(\vec{x})$ and Λ contains all other candidates. Therefore, for given probabilities p_{s_i} of possible diagnosis candidates s_i, we can determine the probability for the sets $\Omega(s)_{\alpha(\vec{x})}$ and $\Lambda(s)_{\alpha(\vec{x})}$ by:

$$p_{\Omega(s)_{\alpha(\vec{x})}} = \sum_{\omega \in \Omega(s)_{\alpha(\vec{x})}} p_\omega \tag{2.2}$$

$$p_{\Lambda(s)_{\alpha(\vec{x})}} = \sum_{\lambda \in \Lambda(s)_{\alpha(\vec{x})}} p_\lambda \tag{2.3}$$

By using the entropy of the two probabilities $p_{\Omega(s)_{\alpha(\vec{x})}}$ and $p_{\Lambda(s)_{\alpha(\vec{x})}}$, we can then determine the influence of the sensing action by:

$$H(\alpha(\vec{x})) = -\left[p_{\Omega(s)_{\alpha(\vec{x})}}(s) \times ld(p_{\Omega(s)_{\alpha(\vec{x})}}(s)) + p_{\Lambda(s)_{\alpha(\vec{x})}}(s) \times ld(p_{\Omega(s)_{\alpha(\vec{x})}}(s))\right] \tag{2.4}$$

The second part of the equation takes the noise sensing actions are influenced by into account. With $p_{\alpha(\vec{x})}$ specifying the probability of the sensing action returning the correct result, we can define the influence of the noise for a given diagnosis candidate w as:

$$p(\alpha(\vec{x})|w) = \begin{cases} p_{\alpha(\vec{x})}, & \text{if } \alpha(\vec{x}) \text{ is consistent with w} \\ 1 - p_{\alpha(\vec{x})}, & \text{if } \alpha(\vec{x}) \text{ is not consistent with w} \end{cases} \tag{2.5}$$

Thus, the conditional entropy is given as:

$$H(\alpha(\vec{x})|w) = -\big[p(\alpha(\vec{x})|w) \times ld(p(\alpha(\vec{x})|w)) + p(\neg\alpha(\vec{x})|w) \times ld(p(\neg\alpha(\vec{x})|w))\big] \quad (2.6)$$

The conditional entropy of all diagnosis candidates can be combined to model the influence of sensor noise.

$$H(\alpha(\vec{x})|W(s)) = \sum_{w \in W(s)} (p_w \times H(\alpha(\vec{x}|w)) \quad (2.7)$$

In order to be able to select the sensing information with the highest information gain, we need to determine the set of executable sensing actions for the current set of diagnosis candidates $W(s)$. Mühlbacher and Steinbauer (2014) proposed two different algorithms to calculate the action with the highest information gain.

Algorithm 2: *computeNextStepSplitting*$(W(s))$

input : $W(s)$ ...a set of possible histories
output: return a grounded sensing action to perform next or $NULL$ if no grounded sensing action exists

```
1  T = {⟨Ω, Λ, α, v⟩|(Ω ⊆ W(s)) ∧ (|Ω| > 0) ∧ (Λ ⊆ W(s)) ∧ (|Λ| > 0) ∧ (Ω ∩ Λ = {}) ∧ (α ∈ A_S) ∧ (v = I(α(x̄); W(s)))}
2  Q = Insert(T)
3  while notEmpty(Q) do
4  |   ⟨Ω, Λ, α, v⟩ = pop(Q)
5  |   α(x̄) = findGrounding(⟨Ω, Λ, α, v⟩)
6  |   if α(x̄) ≠ NULL then
7  |   |   return α(x̄)
8  |   end
9  end
10 return NULL
```

Figure 2.1: An algorithm to calculate the sensing action with the highest information gain (Mühlbacher and Steinbauer, 2014).

The first algorithm calculates all grounded, executable sensing actions and calculates $I(\alpha(\vec{x}))$ for each of them. The action with the highest value is then returned. This algorithm assumes a finite number of domain object and thus a finite number of grounded sensing actions. For a infinite number of domain objects, this algorithm is undecidable.

The second algorithm, shown in Figure 2.1, first calculates all possible splittings of the set $W(s)$. For every possible split, the information gain is calculated and the splittings are added into a queue, ordered by the information gain. Note, that it is possible to calculate the mutual information without a grounded sensing action, if the splitting is given. The algorithm then removes the first splitting from the pool and searches for a grounded sensing action that will result in the given splitting of the diagnosis candidates. The search for a grounded sensing action can be done by using a first order theorem prover. If no grounding can be found, the next splitting will be examined. It is shown that the second algorithm is sound and complete if the

search for a grounded action is sound and complete (Mühlbacher and Steinbauer, 2014).

Knowledge about the probabilities of hardware state changes can be used to calculate the probability p_w of a diagnosis candidate w. Thus, we can use the knowledge of an engineer about the hardware platform to enhance the selection of sensing actions and therefore end up faster with the correct diagnosis. Since we assume a finite number of objects in our domain, we will use the first algorithm in this thesis.

2.3 Platform Model

Often, the additional constraints that are introduced by the hardware platform an agent is operating on, are ignored during the plan generation. On the one hand this is rational, since taking all constraints into account increases the problem size significantly. So, in terms of planning, it is desirable to plan with a platform-independent domain description. On the other hand, the platform may require additional effort and preparation before executing the desired actions.

In Hofmann et al. (2018b), a method for separating planning from the constraints that are enforced by the hardware platform was introduced. An agent would generate an abstract plan and adapt it by using a platform model, so it is executable on the hardware platform. One method for modelling components of the hardware platform, as proposed in Hofmann et al. (2018b), uses finite state machines with multiple types of edges. We can distinguish between two main types of edges: edges that are triggered by actions that are executable by the agent (e.g. a calibration action) and edges that represent exogenous actions not in control of the agent. Now, constraints can be defined for all actions that depend on this component of the hardware. A very basic example for a component model is shown in Figure 2.2. An action that depends on this component could have the constraint, that the component has to be in the `Calibrated` state. When converting the abstract plan into an executable one, the `calibrate()` action then has to be inserted somewhere into the sequence before the execution of the action, that depends on the hardware component. We assume a system that has such a hardware platform model for any reason. However, we do not use the proposed plan transformation in this thesis. Instead, we only want to reuse existing hardware models to improve the diagnosis

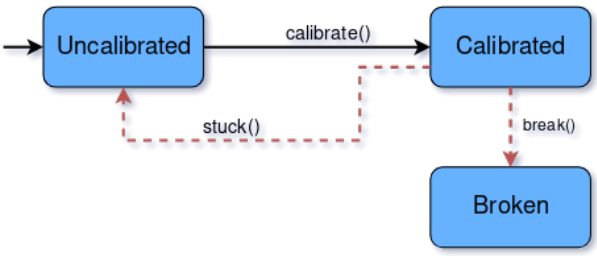

Figure 2.2: A very basic component model

process. This will be done by integrating the possible hardware state changes and the way an action behaves when executed with faulty hardware into the diagnosis candidate generation.

2.4 Planning

In general, classical planning is a graph search problem, where nodes are world states and edges are the actions that lead from one world state to another. A planning problem is then the task of finding a path from the initial state to a state in which a goal condition holds true. To make the graph search more efficient, most of these planners make use of some sort of heuristic. Most of them require a description of the domain and available actions to correctly determine a sequence of actions, that lead to a state in which some goal condition holds true. There exist a lot of extensions to this basic task, such as finding an optimal solution for a given metric, adding a temporal aspect to actions or including numeric fluents (Fox and Long, 2003).

We want to encode the diagnosis problems as planning problems and use existing planners to solve them in order to profit from the performance benefits that well researched planners have in comparison to searching the complete state space.

2.4.1 PDDL

One formal definition language for planning problems is the Planning Domain Definition Language (*PDDL*) (McDermott et al., 1998), which was first introduced by the Planning Competition Committee of the International Conference on Artificial Intelligence Planning Systems (AIPS-98). There exist not one single *PDDL* language, but different versions with different expressibility.

Among other features, it relies on the representation formalism as defined in the *STRIPS* formalism (Fikes and Nilsson, 1971). According to the *STRIPS* formalism, a world model is represented as a set of propositions and makes the closed world assumption. Thus, all propositions not in the set of the world model are assumed to be false. Actions consist of *parameters*, a *precondition* and an *effect*, where the *precondition* and the *effect* are defined by a set of propositions, interpreted as conjunctions.

A planning problem defined using *PDDL* is consists of two parts: the *domain description* and the *problem description*.

Domain description: The domain description defines all available types, constants, predicates and actions. Also, the domain defines the syntactic features used in this *PDDL* instance. For example, in order to use conditional effects, the requirement

`:conditional-effects` has to be added. The features used in a *PDDL* instance constraints applicable planning algorithms and influence the complexity of the described problem. Typically, the domain is defined once and then multiple problems can be defined for this domain. An action definition consists of a set of parameters, a precondition and a conjunction of literals that represents the effect of the action, as shown as an example in Listing 2.1. However, the effect and precondition can include several other expressions, such as disjunctive preconditions, quantifier and existential operators or equality conditions, depending on the defined requirements of the domain. Note, that disjunctive effects are not allowed. By this, preconditions are typically more expressive than effects.

Problem description: The problem description consists of an initial state that represents an instance of the world as defined in the domain description. The initial situation is defined by a set of literals believed to be true. All unmentioned predicates are assumed to be false. Additionally, the problem file defines all available objects, which are instances of the types defined in the domain description. Finally, the problem description contains a *goal*. A goal is defined by a precondition formula that has to hold true in a state of the world in order for this state to be a goal state.

```
1 (:action move
2          :parameters (?r- robot ?from - location
3                       ?to - location)
4          :precondition (at ?r ?from)
5          :effect (and (not (at ?r ?from))
6                       (at ?r ?to))
7              )
8 )
```

Listing 2.1: PDDL definition of a move action

PDDL2.1 *PDDL2.1* adds numeric fluents and expressions as well as durative actions and plan metrics (Fox and Long, 2003). Therefore, *PDDL2.1* allows temporal planning. A numeric fluent can be increased as part of an action effect, e.g. `(increase (total-cost) 1)`. In order to be able to plan cost optimal, a metric can be defined as part of the problem definition. Any planner that supports cost-optimal planning will then search for a plan that optimizes the metric. This extension of *PDDL* will be used in this thesis to encode the preference for explanations with less faults. By applying action costs to exogenous actions, that represent a hardware failure, we can force a planner to search for explanations with a minimum number of hardware failures.

PDDL+ *PDDL+* extents *PDDL* with a continuous aspect, removing the restriction of discrete changes of *PDDL* (Fox and Long, 2002). *PDDL+* allows a mix of discrete

and continuous change. Additionally, predictions of exogenous events can be modelled. *PDDL+* problems can be solved using Satisfiability Modulo Theories (SMT) solvers. This thesis will not make use of this extension.

2.4.2 Fast Downward

Fast Downward is a heuristic PDDL planner framework, that supports multiple search methods. Unlike other PDDL planners, Fast Downward translates the PDDL description of the planning problem into *mutli-valued planning tasks*, which are used to compute its heuristic function (Helmert, 2006). This is done by creating causal graphs, that represent the causal dependencies fluents have on each other. For example the fluent `(holding R1 P)`, that represents a robot R1 holding a certain product P, depends on the robot and the product being in the same position. Therefore, a causal dependency between the holding and the fluent `(at R1 ...)` can be inferred. By using the causal graph, Fast Downward can make use of the implicit constraints that are introduced by the propositional planning task. The actual planning process then can use different search algorithms on this decomposed planning task, such as LAMA (Richter and Westphal, 2010) or Greedy (Helmert, 2006). Fast Downward supports numerical fluents and temporal planning as well as some features introduced by *PDDL3* and *PDDL3.1*.

2.4.3 Top-K planning

One subset of planning problems, that are solvable by the Fast Downward framework, are cost-optimal planning problems. This is the task of not finding only one possible plan for a given planning problem but also minimizing the cost of the found plan. By adding costs to actions and therefore to generated plans, it is possible to encode preferences or likelihoods into the plan generation. However, in many cases we are not interested in a single optimal plan, but in the k best plans. For this, top-k planning can be used, which is the problem of finding a set of plans of size k while guaranteeing that no plan better than any member in the set is left out. Often, this is useful to evaluate possible optimal plans with any criteria that is too complicated to encode into the planning problem directly. We want to use such a top-k planner to generate a set of hypotheses for a given diagnosis problem. Sohrabi et al. (2016) proposed using the k-shortest path algorithm of Aljazzar and Leue (2011) to address this problem in their K^* algorithm.

K^* uses any heuristic search to find a single optimal plan. During the heuristic search, a portion of the search graph G gets constructed. After finding an optimal solution, the search graph is expanded even more, until the number of additionally expanded nodes reaches a certain percentage of the generated search graph. Sohrabi et al.

(2016) suggested continuing the heuristic search until the search graph is increased by 20% after finding the optimal solution. Afterwards, the shortest path tree T to the goal node is generated, by applying Dijkstra's algorithm on the expanded search graph G. By removing all edges in the shortest path tree from G, $G - T$ only consists of sidetrack edges, that can be assigned a detour cost for taking that edge. Based on this sidetrack graph, a path graph $P(G)$ can be constructed, in which nodes represent sidetrack edges. By applying Dijkstra's algorithm again on the path graph $P(G)$, it is possible to extract the k shortest paths. The worst case complexity of the k-shortest path algorithm for a given graph G with n nodes and m edges is given as $O(m+n\log n+kn)$. The bottleneck of the algorithm is the generation of the complete search graph G. K^* deals with this challenge by using a heuristic-guided search like A^* to expand only interesting portions of the search graph. By dynamically continuing the A^* search to further expand the search graph, if not enough shortest paths were found, the K^* algorithm is able to efficiently generate the search graph and therefore perform better than the k-shortest path algorithm of Aljazzar and Leue (2011). All in all, the complexity of finding the top-k plans is comparable to finding a single optimal plan, as stated in Sohrabi et al. (2016).

One disadvantage of the K^* planning algorithm, is that all plans are found almost simultaneously (Sohrabi et al., 2016). Therefore, it takes a long time until the first solution is reported. Also, the implementation of the K^* algorithm was done in a planner that supports the Stream Process Planning Language (*SPPL*) and not in *PDDL*. Recently, Katz et al. (2018) presented an iterative version of the K^* algorithm, that is implemented as a search engine of the Fast Downward framework, therefore supporting *PDDL* problems.

2.5 RoboCup Logistics League

The RoboCup Logistics League (RCLL) is a robotics competition that aims to support and evaluate research in the discipline of dynamic robotic-based production and to provide a regular competitive testbed (Niemueller et al., 2016a). The RCLL is part of the RoboCup (Kitano et al., 1997), an international initiative to support intelligent robotic research. The RCLL challenges the participating teams to develop a team of robots, that operates a smart factory and fulfils custom and dynamic orders. To accomplish this task, the robot team has to plan and execute the logistic flow of the factory by manually operating the machines. Mobile robotics itself raises a lot of challenges such as path planning, navigation and collision avoidance.

In the last years, two teams had to play on the same field, with a set of machines assigned to each team. A centralized refbox keeps track of the achieved points and generates random product orders during the game. Product orders have different

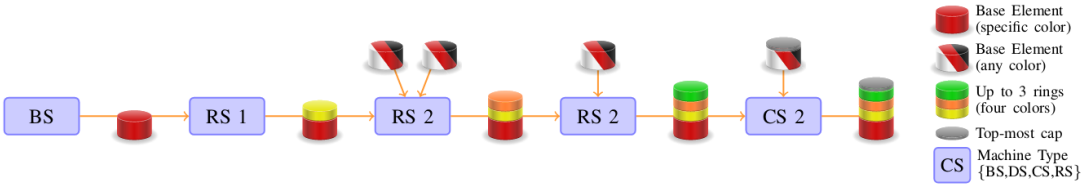

Figure 2.3: Production steps needed for a product of the highest complexity (Niemueller et al., 2015)

types of complexities, where a rising complexity implies a rising number of machine interactions needed and therefore increased risk and time. Generally, machine is operated by placing a intermediate product into the input side, instructing the machine by sending a message and then picking up the processed product from the output side.

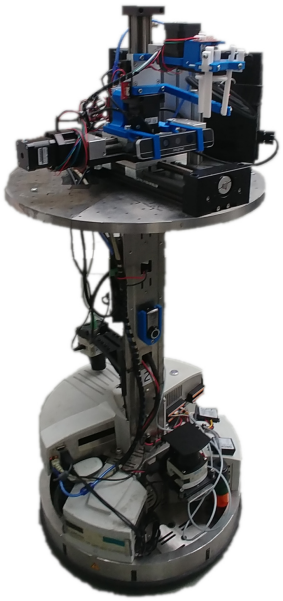

Figure 2.4: The robotino, as used by the Carologistics team in 2019 (Hofmann et al., 2019)

Each team can use six machines of four different types. The *base station* is the starting point of all available products, since it dispenses the base segment. There exist three different colors of bases and a product consists of one base, one cap and between zero and three rings. Caps can be mounted at the *cap station*. To mount a cap, the cap station has to be fed a cap carrier, from which it can retrieve and buffer the cap. There are two *cap stations* on the field, responsible for two different cap colors. Therefore, in order to produce a product of the lowest complexity, a cap has to be buffered at a cap station, the empty cap carrier has to be removed from the cap station, a base has to be retrieved from the base station and the buffered cap has to be mounted on the base at the cap station. Higher complexity products require the use of the two *ring stations*. Each ring station can mount two different ring colors. Additionally, bases and cap carriers can be filled into the ring stations and have to be used to pay for certain ring colors. For example, a ring station can mount a orange ring for the cost of one. Therefore, before mounting the orange ring, at least one base or cap carrier has to be filled into the ring station. The prices for each ring color are decided randomly by the refbox at the begin of each game. After mounting the cap onto a product, the final product has to be placed into the *delivery station*. The points of the product are only awarded after a successful delivery. An example for the steps needed for a product of the highest complexity is shown in Figure 2.3.

After a game has started, the robots have to deal with failures on their own in order to successfully finish the game and produce as many products as possible. All robots are built based of the same hardware platform: the Robotino platform by Festo Didactic. These platforms consist of omnidirectional locomotion, infrared distance sensors and interfaces for additional sensory. Each team may attach additional hardware, such as laser range sensors, webcams and grippers. For example, the robots of the Carologistics team are equipped with two Sick laser sensors, an additional laptop, a webcam, a depth camera and 3 linear drives that hold a parallel gripper. So the complete system consists of a lot of smaller components, which all have to operate correctly.

The RoboCup Logistics League is designed as a benchmark domain for planning, plan execution and mobile robotics. A team of robots has to be coordinated, while having to deal with the robots of the opponent team at the same time. Therefore, the RCLL domain it is *cooperative* and *competitive*. Actions are *non-deterministic*, since grasping and putting can fail at any time and collisions may occur with enemy robots. Also, the game evolves over time, since all actions have an influence on the state of the machines. The world is *dynamic*, with many possible causes for changes that the robot has no control over. All in all, the RCLL is a suitable benchmark domain for robotic planning and plan-execution and will be used for evaluating this diagnosis approach. This will be done by evaluating the ability of the robots to continue the production after hardware failures in a simulation of RCLL games.

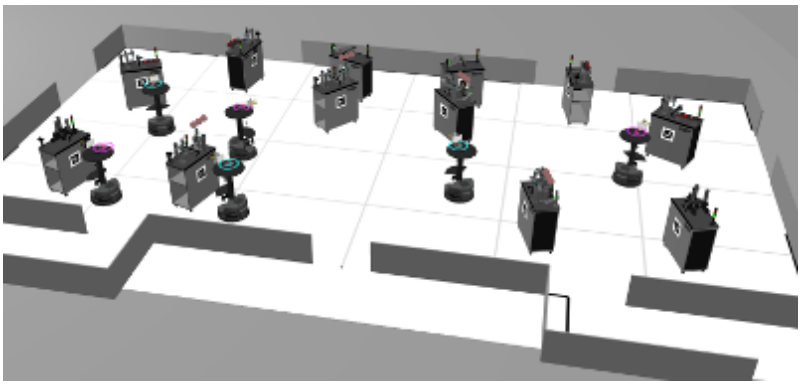

Figure 2.5: Simulated RCLL game in Gazebo

In order to effectively evaluate approaches in the RCLL domain, a simulation of the RCLL was developed, using the open source simulation framework *Gazebo*, (Koenig and Howard, 2004). The RCLL simulation, as presented in Niemueller et al. (2016b), aims for simulating complete RCLL games, support multi-robot strategy evaluation and simulate sensor data with optional noise. By this, we can closely resemble a realistic execution of actions in the simulated environment. Therefore, the simulation offers the possibility of evaluating agent and strategy developments close to reality without having to run real games.

2.6 Fawkes

Fawkes (Niemueller et al., 2010) is an open-source robotic framework used by the Carologistics team in the RoboCup Logistics League. It consists of several components, that are connected via well-defined interfaces, called blackboards (Niemueller et al., 2016c). The main structure, as used by the Carologistics team, consists of three layers, as shown in Figure 2.6. The lowest layer includes all components that deal with hardware, sensory and data analysis. The second layer combines several low level components to a more abstract *behaviour engine*. The *agent system* at the top is responsible for decision making, planning and cooperation. Actions executed by the agent are realised by the behaviour engine.

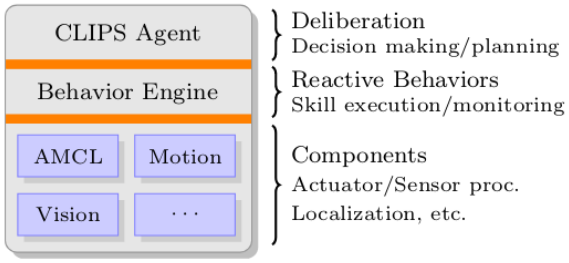

Figure 2.6: The three layers of fawkes (Niemueller et al., 2013)

Component-Based: Fawkes is a component-based software framework, which means that it consists of several units, combined by well-defined interfaces. The Fawkes framework offers dynamic loading of components, so-called plugins. It already comes with a huge variety of plugins that cover a wide range of functionalities from infrastructure to functional components, such as localization, perception or navigation. All these plugins are able to communicate by using well-defined interfaces, so called blackboards. Plugins are loaded in separate POSIX threads, which are managed by Fawkes. A thread can run in two different modes: *wait-for-wakeup* and *continuous*. A *wait-for-wakeup* thread is blocked until it is waken up, runs one iteration and then blocks again, whereas a *continuous* thread runs continously in the background. All threads are hooked into the main loop of Fawkes.The main loop is handled by the framework in a *sense-think-act* cycle. By doing so, it is guaranteed in which order and with what input all threads execute one iteration. A thread can be registered for a stage and will be woken up everytime the main loop reaches that stage. Especially in multi-core systems, Fawkes efficiently synchronizes multiple threads running in parallel, exploiting the systems resources as good as possible.

Blackboard: Blackboards are a hybrid blackboard/messaging system with XML-defined interfaces. Normally, each component acts as a writer for a specific blackboard, through which it shares its data with other components. These information are available for all interested components by reading from the blackboard. Other components can instruct the writing component or request data by sending messages.

Behaviour engine: The Lua-based behaviour engine combines several low-level

components to more abstract skills (Niemueller et al., 2009). It seperates the low-level sensor processing and control software from the agent. An individual behaviour - called skill - is implemented as a hybrid state machine (HSM). The nodes of the HSM represent the state of the execution and monitor the behaviour. Edges determine jump conditions to other states, implemented as boolean functions. A skill is finished if one of its final nodes is reached. Using the blackboard interfaces, a skill can monitor the data published by some low-level component and can send messages to instruct a low-level component. A table of variables can hold information in the scope of the skill, like numeric values of object positions. Skills are implemented using the extensible scripting language Lua. It is possible to have a hierarchy of skills, since a skill can call a subskill in any of its states. Therefore, it is possible to implement skills for basic behaviour and combine them into more complex behaviour.

For example, to grip from a machine, we need to make use of a gripper component, a vision component for detecting the workpiece and a laser component for detecting the machine. A skill can make local decisions to scope with basic failures and call subskills, and therefore act as an abstraction of basic functionalities. The skills are then used by the high-level agent.

2.7 CLIPS Executive

CLIPS is a rule-based production system (Wygant, 1989). The CLIPS rules engine consists of *facts*, *functions* and *rules*. *Facts* are pieces of information in a fact base. *Rules* are the core of the CLIPS rule engine. They consist of a left-hand side antecedent (LHS) and a right-hand side (RHS) consequent. The LHS consist of a set of conditions, typically patterns that formulate restrictions on which facts satisfy the conditions. If there exist a fact in the fact base for every pattern of the condition, the LHS is satisfied and the consequent is added to the agenda. Each consequent in the agenda is then executed after another. A consequent can assert, retract and modify facts of the fact base. Additional procedural knowledge is encoded in *functions*, that can be executed as part of a consequent. *Functions* can be implemented in other languages such as C++ or PYTHON and therefore can be used to communicate with components of the framework the CLIPS rule engine is integrated in.

The CLIPS Executive implements a goal lifecylce, as proposed in Roberts et al. (2014) and Aha (2018), shown in Figure 2.7. Currently, the CLIPS Executive is used by the Carologistics team as a high-level agent in the fawkes framework (Niemueller et al., 2019). The CLIPS Executive consists of a *domain model*, an *execution model* and stores a *world model* as part of the fact base (Niemueller et al., 2019). The *domain model* defines all available operators and predicates, the *execution model* holds all execution relevant information, such as sensed predicates and exogenous

actions and the *world model* holds all information about the world at a certain point of time. Each piece of information about the world is represented as a predicate, such as `(domain-fact (name at) (param-values R1 C-CS1))`, representing that the robot R1 is currently believed to be at the machine C-CS1.

2.7.1 Goal Lifecycle

Goals describe objectives which are desirable to pursue. The CLIPS Executive uses goals as the core data structure to organize its decision making process. There exist two types of goals: *achieve* and *maintain* goals. Achieve goals are meant to achieve a certain objective are maintain goals are used to maintain some condition or state. A goal also maintains a set of parameters, meta information and the resources needed to achieve the assigned objective.

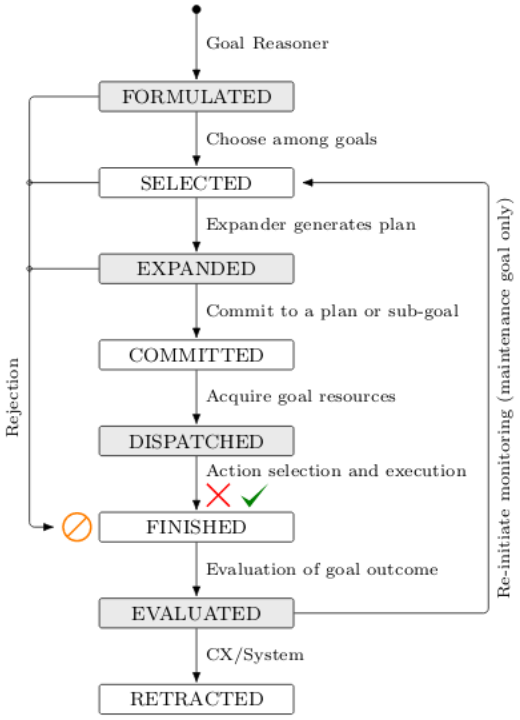

Figure 2.7: The goal life cycle (Niemueller et al., 2019)

Based on the world model, the CLIPS Executive *formulates* all reachable goals and follows the goal lifecycle, defined in Hofmann et al. (2018a). The most desirable goal gets *selected* and *expanded* by generating fitting action sequences that will lead to the goal objective based on the current state of the world, as represented in the *world model*. The expanded goal then *commits* to one of the generated plans and acquires all resources needed for the execution of the committed plan. When all resources are acquired, the goal finally gets *dispatched*. The action executer then takes care of executing the corresponding plan.

Each plan action is defined in a PDDL domain and contains of a set of preconditions, that have to hold true before executing, and a set of effects that are applied to the *world model* after executing the action. Most of these actions are tied to a middle-level skill of the *behaviour engine*, that is triggered upon the start of the

action execution. The plan action then follows the action execution cycle, which will be explained later.

After finishing the committed plan, regardless whether it failed or not, the goal switches to an *evaluation* state, in which the CLIPS Executive can reason about the success and outcome of the goal. Currently, at this point some goal-specific hard-coded evaluation rules are in place, that updates the world model after a failure. This work will replace this procedure with an automated diagnosis procedure. The ability to correctly identify the cause of the failure in this evaluation step and update the *world model* to correctly represent the real world is particularly useful when taking the goal lifecyle into account. After finishing the previous goal, the CLIPS Executive just starts with reformulating all goals again. However, with the information about the cause of the failure we might be able to recover from the failure and repair the previously tried goal instead of dropping it, losing the time and resources already invested. Thus, the ability to correctly diagnose the cause of failures and reasoning about the effects a failure have onto the world will result in a more robust and successful CLIPS Executive.

2.7.2 Plan Action Cycle

The CLIPS Executive uses a plan action cycle to control the execution of plan actions, as presented in Hofmann et al. (2018a) and shown in Figure 2.8. As soon as all grounded preconditions of the selected action are verified to hold true in the current state of the world model, the action is *pending*. The executor then starts the corresponding skill of the action and the action switches to *waiting*, until the skill reported the start of the execution.

The plan-action stays *running* until the skill reports the end of the execution, together with information about the success of the execution. Depending on the outcome of the skill, the effects are applied to the world model and

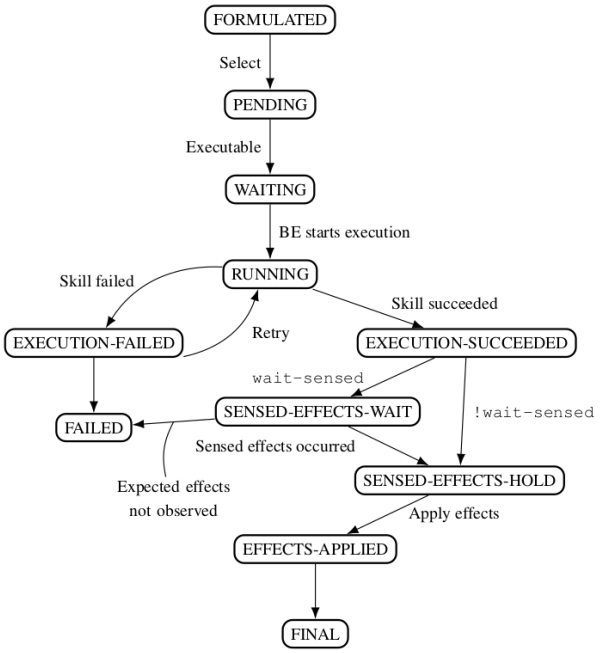

Figure 2.8: The action execution life cycle (Niemueller et al., 2018)

the agent continues with the next plan action. Niemueller et al. (2018) introduced an additional type of effects: *sensed effects*. These are effects, that can be directly observed in the world and the action waits for these effects to come into place. However, these sensed effects are only part of the execution model and are treated as normal effects regarding the PDDL definition.

2.7.3 Multi-agent Coordination

In the RCLL domain, three robots with separate instances of the CLIPS Executive are used, operating simultaneously. Instead of cooperating to fulfill a shared set of goals, the CLIPS Executive follows the *negotiated distributed planning* approach, where each instance pursues its own goals while coordinating with the other instances (Durfee et al., 1999). Most importantly, since all instances operate in the same world, any changes to the world model introduced by one instance of the CLIPS Executive has to be communicated to all other instances. This is done by using a shared world model, implemented in a distributed database with local instances on all CLIPS Executive instances (Niemueller et al., 2012). As soon as one instance updates a fact that is marked for syncing, it is updated in all instances of the distributed database.

To coordinate goal execution and prohibit multiple instances pursuing the same goal, resource locks are used. As mentioned in Subsection 2.7.1, goals can have resources assigned that determine which resources in the actual world are needed or going to be changed if an instance decides to execute the goal. By using mutex facts, that are owned by an instance and synced using the distributed database, it is guaranteed that only one instance can change a resource. Situations, that needs additional coordination between robots, e.g. prohibiting that two robots want to drive to the same location, can be resolved by temporarily locking the resource. This is done by using specific *lock* and *unlock* actions as part of the plans.

3 Related Work

In this chapter, we will present related work to our approach. Starting with different diagnosis approaches, we will then present related work to the active diagnosis and to solving diagnosis by planning.

3.1 Consistency-based Diagnosis

Consistency-based diagnosis was first introduced in Reiter (1987). Based on a description of the normal behaviour of a system and an observation that conflicts with the described behaviour, faulty components are identified. A consistency-based diagnosis then consists of a set of components of the system that behaves abnormally. Mühlbacher and Steinbauer (2016a) viewed executed actions as components of the system. The ultimate goal of their approach is to maintain a consistent belief about the world state. After an observation, that does not comply with the belief expected after the execution of the history, abnormal actions are identified and used to update the belief. The identification is made by using conflict-driven search to calculate diagnosis hypotheses. All fluents influenced by the failed actions are then marked as unknown, which results in a consistent but reduced belief of the world state. Our approach aims for maintaining a consistent belief as well, but by using multiple fault modes of actions we want to distinguish between the different ways an action can fail. Therefore, we end up with a more detailed belief, which enables the agent to make more informed decisions afterwards.

A similar approach is presented by Witteveen et al. (2005), in which actions are viewed as objects of a system, that can have several health modes. Based on an observation at any step of a plan execution, the executed plan actions are assigned a health mode. Based on a health mode assigned to an action scheme, they predict the outcome of future instances of this action and therefore use diagnosis to predict the outcome of plans. However, any fluent that depends on an action, that was diagnosed as abnormal, gets marked as unknown. This generally leads to a loss of information. Witteveen et al. (2005) assumed, that a failure of an action is not a single event, but enables us to predict the outcome of other actions as well. They assumed that a failure of an action gives information about other instances of the same action type, already assuming an underlying cause. However, they ignore the fact, that the cause

of a failure of one type of action can have an effect of a completely different type of actions as well, if they are influenced by the underlying cause as well.

3.2 Abduction-based Diagnosis

Abduction methods are also suitable for generating diagnoses, as stated in Poole (1989b). Abduction reasoning represents a form of logic inference, that seeks for the most likely explanation for a given observation. Poole (1989a) pointed out, that abduction can purely be used on a complete description of possible faults and their symptoms. Abductive diagnosis utilizes an axiomatisation how symptoms follow from causes. A diagnosis generated with abduction on a set of observed symptoms then consists of the minimal set of fault assumptions that explain the observations. Our approach resembles abductive diagnosing, since we search for hardware state changes that result in an adapted consistent history. However, rather than directly axiomatising the effects of a hardware state change, we define the behaviour of actions for given hardware changes. The description of the action variations will then be used to explain the observation.

3.3 History-based Diagnosis

Mcllraith (1999) introduced an approach, that searched for an explanation for a faulty component. This was done by implementing a consistency-based diagnosis, which conjectured actions to explain the observed fail. Our approach not only marks components as faulty, it actively maintained and altered a history of actions. Thus, this approach is capable of dealing with exogenous events that have to be integrated into an existing history. Based on this approach, Gspandl et al. (2011) proposed a method to revise a history of actions in context of an observation by replacing actions with a fitting faulty version or by integrating exogenous actions. However, their approach requires a description of all possible ways each action can fail. Diagnosis hypotheses are then generated by defining a search space over all possible replacements and insertions. Traversing the search space and finding adaptations ,that result in a consistent belief, is a time-consuming task due to the extensive state space. Iwan (2002) introduced the use of history-based diagnosis templates in order to cope with the problem, that there are multiple, similar diagnosis candidates that only differs in the grounding of parameters. In contrast to our approach, the search for history-based diagnosis candidates ignores the possible underlying root causes for a failure and may even produce impossible explanations by ignoring dependencies between actions. For example if an action is replaced by a faulty alternative and depends on a certain component of the hardware, we know that any other type of

action that also depends on this particular component is also likely to fail in some way. The knowledge about these types of dependencies is ignored in the approaches mentioned above.

3.4 Other Approaches

There are a lot of different approaches, that use diagnostic methods for a wide range of tasks. Roos (2018) uses reinforcement learning for learning the reliability of robots in a multi-robot environment. In de Jonge and Roos (2004), a model-based diagnosis approach is used to identify possible conflicts in future plans, where agents are operating simultaneously on shared resources. Another interesting diagnosis approach, that monitors the outcome of actions while executing them, is presented in Eiter et al. (2004). Upon detecting a state that does not lead to a goal state, their diagnosis approach searches for the last state in which a goal could be reached. However, this is based on the assumption that the agent has complete knowledge about the state of the system. This is nothing we want to assume, since a robot in a real world scenario may not be able to sense all aspects of its surroundings.

Another approach, closely related to diagnosis, is *explanation generation*. For given observations, explanation generation aims for finding sequences of actions and events that explain the observations. Approaches to explanation generation, like Bridewell and Langley (2011) or Molineaux and Aha (2015), differentiate events from actions and do not revisit knowledge about executed actions. Also, they assume that only partial states are observed instead of actions, which hugely deviates from our approach. In history-based diagnosis generation, the only secured knowledge is about the intentions of which actions should have been executed. The resulting state is only assumed, based on the assumed effects of an action.

3.5 Active Diagnosis

As mentioned above, we will use the approach to active diagnosis as proposed by Mühlbacher and Steinbauer (2014). They developed a framework for selecting sensing actions with the highest information gain on a set of diagnosis candidates, with the goal to narrow the set of possible diagnosis hypotheses. However, there exist some other, related approaches. Baral et al. (2000) also diagnoses dynamic systems by generating sensing plans. However, they represent the static behaviour of a system by static constraints and monitors the system by reading sensor values continuously. If a sensor reading is contradicting the static constraints, events that explain the sensor reading are searched. If more than one diagnosis is found, their approach generates a conditional plan with sensing actions to find the real diagnosis. Another

approach for active diagnosis is presented in Kuhn et al. (2008). They proposed the integration of diagnosis goals into production goals based on the assumption that there exist multiple ways of achieving production goals. Their concept of *pervasive diagnosis* then proposes the selection of the production strategy with the highest diagnosis information gain.

3.6 Diagnosis as Planning

As mentioned above, Gspandl et al. (2011) uses history-based diagnosis for maintaining a belief, by traversing a search space defined by all possible insertions and replacements for a given history of executed actions. This method closely resembles traditional planning methods, that aims for conjecturing actions until a goal state is reached. Sohrabi et al. (2010) showed the connection between diagnosis and planning in detail. Based on previous works on temporally extended goals (Baier and McIlraith, 2006) and temporally extended preferences in Baier et al. (2009), Sohrabi et al. (2010) uses a description of a diagnosis problem in situation calculus and transforms the formalized diagnosis problem into a *PDDL* planning problem. The history of executed actions and the conflicting observation are encoded as a temporal extended goal. However, the history is not directly encoded as action types. Instead, the resulting state after executing an action is encoded in special predicates. Also, in temporal extended goals the sequence of observations does not necessarily need to be strictly ordered, in contrast to the strictly order sequence of actions in our history. Therefore, temporal extended goals are not needed for implementing history-based diagnosis approaches.

4 Approach

The goal of this thesis is to utilize a model of the platform the agent operates on for an improved guided history-based diagnosis generation. Two of the drawbacks of history-based diagnosis are the manual creation of all possible action variations and the huge search space, that has to be traversed while searching for valid history adaptations. The primary contribution of this thesis is to extent the existing history-based diagnosis approach by reasoning about possible underlying hardware failures. Additionally, by using platform models we want to guide the engineers through the generation of action variations and therefore enforce that all action variations are properly defined for a successful diagnosis generation. Assuming, that a majority of failures are caused by the hardware platform, we expect the guided action variation generation to be sufficient for the most diagnosis tasks and enable the agent to make more informed decisions after a hardware failure.

After executing an action sequence and making an observation that contradicts the expected world state, we want to use the history of executed actions and the history of state transitions of the component models to generate alternated histories that result in consistent beliefs. This is going to be done by encoding the constraints given by the platform models and the sequence of executed actions into a PDDL domain and solving it using planning algorithms. If we end up with multiple diagnosis candidates, we want to perform *active diagnosis* to sense for literals, that will rule out diagnosis candidates. Ultimately, we want to improve the CLIPS Executive in terms of recovery after a failure by enabling it to properly evaluate the reasons for a failed goal. A sketch of the diagnosis pipeline in the goal evaluation step is shown in Figure 4.1.

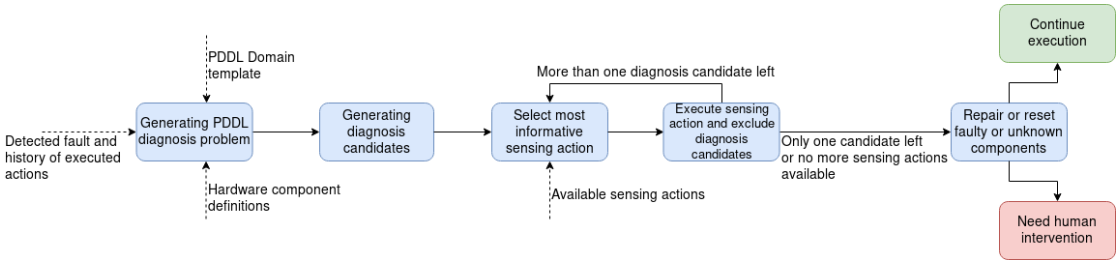

Figure 4.1: The diagnosis and recovery pipeline

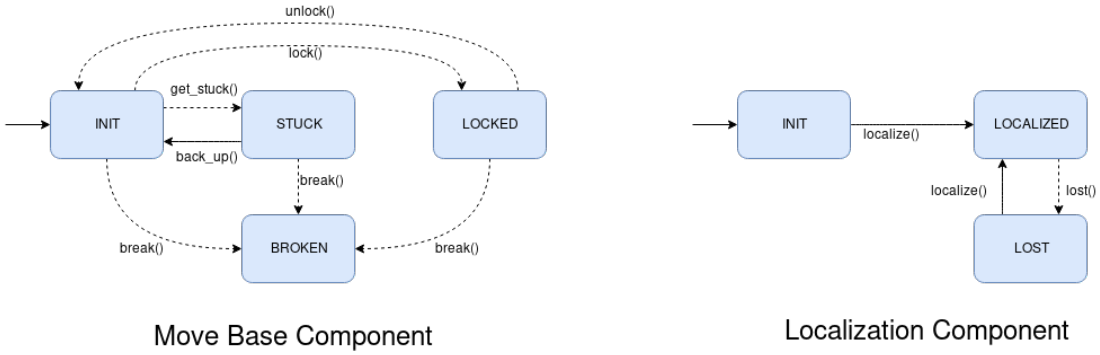

Figure 4.2: Two components with constraints for a move action. Left the move base, right the localization component

4.1 Generating Action Variations

Existing consistency-based diagnosis-approaches, like Mühlbacher and Steinbauer (2014), identify failed actions and investigate the influence of these actions on the world state. All fluents, that are directly or indirectly influenced by failed action, are then marked as unknown since the effect of a failed action is unknown. This approach is not really applicable in the CLIPS Executive, because there exists no way of modelling, that a fact is uncertain. However, by identifying the objects, that are influenced and their properties we can represent uncertainty about the property of an object by removing all facts describing this property.

Nevertheless, a single failed action may draw the outcome of subsequent actions unpredictable as well, which will result in a cascading loss of information. Of course, this will result in a consistent world model but will also have a huge negative impact on the possibilities of the agent to continue the production. For example, after a failed grabbing action we would have to mark the position of the workpiece we wanted to grab as unknown. Thus, the time and effort that was used to produce the workpiece would be lost, since the workpiece could no longer be processed due to the missing information about its location. To prohibit this, we want to reason about the different ways actions can fail.

In history-based diagnosis we are not merely interested in which actions may failed, but in which way exactly. By reasoning about the effects a failed action can have, the history-based diagnosis procedure will result in a refined world state with a lot more information than by using a consistency-based approach. In order to identify the way an action failed and what effects the failure has onto the world state, the diagnosis procedure requires a definition of all possible action variations.

By assuming, that the majority of action failures are caused by hardware errors

or wrong assumptions about the state of a platform component, we expect the platform model to be a sufficient guideline for the creation of action variations. As described before, we model the various components of the hardware platform the agent is operating on as finite state machines with multiple kinds of transitions. A component model may contain exogenous transitions, that represent component state changes, which can happen without being actively triggered by the agent. We assume, that most action failures are caused by a component unknowingly changing the state by following such an exogenous transition. Therefore, we require a description about the effect an action will have when executed with the component being in an *exogenously reachable* state. *Exogenously reachable* in this context stands for: reachable by following only exogenous transactions. We then need to take all component models into account, that enforce a state constraint onto an action and define action variations for exogenous reachable states in all of these component models.

Note, that we just have to define action variations for adjacent states, that can change unrecognised without failing the actions. For example, an action, that aligns the robot to some machine using laser scanners, needs the laser scanner component in a calibrated state. However, if the laser scanner got de-calibrated unnoticed by the agent, the action would succeed nevertheless. Therefore, we need to define an action variation of the alignment action for the laser scanner being uncalibrated. Then again, if the laser scanner would have turned off before or while executing the action, the action would fail without affecting the world, because it recognizes the missing laser data. Therefore, we do not need an action variation of the alignment action for the laser scanner component being turned off. By following these design guidelines, we end up with all necessary action variations for a successful diagnosis of hardware caused failures.

4.2 Formulating Diagnosis Problem in PDDL

Planning algorithms are a well researched topic and we want to utilize the performance benefits offered by well-known planning algorithms in comparison to simply traversing the search space. Thus, we want to encode the diagnosis problem as a PDDL problem and use a top-k planner to generate k diagnosis candidates.

As mentioned in in Subsection 2.4.1, a PDDL problem consists of several components: *predicates*, *types* and *operators*, which are part of the PDDL domain description, and *objects*, the *initial world state* and the *goal*, which are part of the PDDL problem description.

In order to diagnose a history of executed actions, we want to ensure that the planner only generates alternated versions of the executed history. In the following

sections the generation of the PDDL diagnosis problem will be described. This includes enforcing the generated plan to be an alternated version of the history, the integration of the hardware component constraints and the integration of the component model state transitions. Afterwards, we introduce the creation of a diagnosis problem description.

4.2.1 Domain Description

Typically, the *PDDL* domain description is created once and then reused for different problems. We use the action history to dynamically adapt the *domain description* in such a way, that the generated plan is an adapted version of the action sequence. Therefore, we have to adapt the domain to each diagnoses problem separately. In the following sections, the enforcement of the history is described as wells as the integration of the hardware component state changes and the restrictions onto the actions.

History Enforcement

```
1  (:action order_i
2           :parameter()
3           :precondition (and (last-ACT_i PARAM_i)
4                              (count ACT_i COUNT))
5           :effect (and (not (last-ACT_i PARAM_i))
6                        (next-ACT_i+1 PARAM_i+1)
7                        (not (count ACT_i COUNT))
8                        (count ACT_i COUNT+1)
9               )
10 )
```

Listing 4.1: Order action template

To force the planner to only create plans, that are variations of the executed history, we introduce a new type of *order* action, as shown exemplarily in Listing 4.2 for a grab action that succeeded a move action. For a given action sequence $h = \{a_1, \cdots, a_n\}$, for each $i \in \{0, \cdots, n\}$ an action $order_i$ is defined as shown in Listing 4.1.

- ACT_i is replaced with the action type of a_i and ACT_i+1 with the action type of a_{i+1}.
- PARAM_i is replaced with parameters of a_i. The same applies to PARAM_i+1.

In a similar way, ACT_0 is replaced with BEGIN, for which (last-BEGIN) is true initially and ACT_n+1 is replaced with FINISHED, for which (next-FINISHED) is part of the goal condition. Additionally, a new precondition and a new effect is added to each action and its variations, as shown exemplarily for the move action in Listing 4.3.

```
1 (:action order_1
2         :parameter()
3         :precondition (and (last-move R1 C-BS C-CS1)
4                            (count MOVE ONE))
5         :effect (and (not (last-move R1 C-BS C-CS1))
6                  (next-grab R1 WP1 C-CS1)
7                  (not (count MOVE ONE))
8                  (count MOVE TWO)
9             )
10 )
```

Listing 4.2: Grounded order action

```
1 (:action move
2         :parameter(?r - robot, ?from - location,
3                    ?to - location)
4         :precondition (and (next-move ?r ?from ?to)
5                            (at ?r ?from))
6         :effect (and (not (at ?r ?from))
7                      (at ?r ?to)
8                      (not (next-move ?r ?from ?to))
9                      (last-move ?r ?from ?to))
10            )
11 )
```

Listing 4.3: A non-faulty action with order predicates.

To reach a goal state, in which `(last-FINISH)` has to be true, the planner has to generate a sequence of of the form $h' = \{o_0, a'_1, o_1, \cdots, a'_n, o_n\}$, where o_i is an order action and a'_j is either $a_j \in h$ or one of its faulty variations. Additional exogenous actions are inserted at any place.

Determining which parameters have to be included in the (last-... and (next-... predicates is an essential aspect when creating the action descriptions. Thus, we have to differentiate between parameters that would have caused the action to fail in the first place if changed, and parameters that may change unknowingly. Think of a `grip` action executed at a certain machine after the robot moved to this machine.

Naturally, the action has a precondition that enforces the robot to be at the machine. After a failure one possible sequence could be, that the move failed and the robot ended up at a different machine. By enforcing the use of the old machine by adding it to the parameters in the ordering predicates, this sequence would have never been generated. Looking at the example, the grip action would succeed at any machine. Therefore, the machine ID is not a parameter, that has to be enforced by the ordering predicates. On the other hand, a move action on this particular platform may be known to fail, if the robot is not at the position he believed since it is not possible to generate a path to the target position. In this manner, we could enforce the starting position of the robot as a parameter. An example is shown in Listing 4.4 for the grab action as described above. The decision about which parameters have to be enforced in the history depends on the specific implementation and execution of the actions and can differ from platform to platform.

```
1  (:action grab
2           :parameter(?r - robot, ?wp - workpiece ?m -
               machine)
3           :precondition (and (next-get ?r) (at ?r ?m)
4                              (or (wp-at ?wp ?m)
5                                  (and (mps-side-free ?m)
6                                       (dummy-wp ?wp))
7                              )
8                      )
9           :effect (and (not (next-get ?r))
10                          (last-get ?r)
11                          (when (wp-at ?wp ?m)
12                                (and (not (wp-at ?wp ?m))
13                                     (holding ?r ?wp))
14                          )
15                )
16 )
```

Listing 4.4: A standard action with predicates distinguished between critical preconditions and conditional effects

Additionally, we have to differentiate between two different types of preconditions: *hard* preconditions and *soft* preconditions. A *hard* precondition has to be fulfilled in order to be able to execute the action, independent of the effect it has. *Soft* preconditions guard the effects an action can have, but do not lead to a failed execution if not fulfilled. Therefore, a *soft* precondition could be unsatisfied while executing the action without being noticed, whereas a *hard* precondition does lead to a failed execution, if being unsatisfied.

For example, a grip action would fail if the robot is not positioned at any machine. Therefore, `(at ?r ?m)` is a *hard* precondition of `grab`. The grab action will only result in a gripped workpiece, if there actually was a workpiece at the machine but will not abort the execution if there is no workpiece. Thus, `(wp-at ?wp ?m)` is only a *soft* precondition.

While generating diagnosis candidates we want to model all effects of an action, even the undesired ones. In general, this requirement is true for all *PDDL* models. However, in terms of planning, the distinction between desired and expected effects are often not made. This is because in planning you mostly assume a successful execution without failures or exogenous actions. Therefore, we potentially have to adapt a given *PDDL* model of the domain to properly distinguish between expected and desired behaviour. This is done by moving the *soft* preconditions to the effect as conditions for conditional effects. However, we still have to restrict the grounding of the parameters of the *soft* conditions that are not restricted by any other predicates of the precondition. Sticking to the example from above, having the `(wp-at ?wp ?m)` only as condition of an effect, the planner may ground `?wp` with any workpiece that is not located at the machine, ignoring any workpiece that actually is at the machine. Thus, the effect of picking up the workpiece would not be applied. Therefore, we add a disjunctive precondition, that is only used for grounding these parameters. In our example we could achieve this by adding the precondition `(or (wp-at ?wp ?m) (mps-side-free ?m))`. The `mps-side-free` predicate is true if there is not product at the input side of the machine. Following this precondition, `?wp` will be grounded with the workpiece that is at the machine, if there is one.

To prohibit the planner of generating grounded action instances with any object as `?wp` we introduce a dummy object for which `(dummy-wp ?wp)` holds true. Thus, we can adapt the disjunctive precondition to `(or (wp-at ?wp ?m) (and (mps-side-free ?m) (dummy-wp ?wp))`, resulting in only one possible grounding if the machine side is free.

By these adaptations of the *domain description*, the planner is forced to create a plan, that contains every action or one of its variants of the given history exactly in the order, they were executed.

Platform Constraints

Additionally to the history of executed actions, we want to keep track of the history of component states. As mentioned in Chapter 2, actions can have constraints for component models being in a certain state to be able to executable. By keeping track of the component states during the execution of a sequence of actions, we can use the history of component states to restrict the set of exogenous actions and action variants to be considered during the planning process.

Taking the two component models from Figure 4.2 as an example, we can adapt the definition of the `move` action from Listing 4.3 to correctly represent the fact, that the robot has to be localized and the move base has to be functional to correctly execute a move action.

```
1  (:action move
2      :parameter(?r - robot, ?from - location, ?to -
           location)
3      :precondition (and (next move ?r ?from)
4                         (at ?r ?from)
5                         (state MOVE_BASE INIT)
6                         (state NAVIGATION LOCALIZED))
7      :effect (and (not (at ?r ?from))
8                   (at ?r ?to)
9                   (not (next-move ?r ?from))
10                  (last-move ?r ?fri))
11             )
12 )
```

Listing 4.5: An action with hardware component constraints

Component State Changes

With action alternatives depending on the hardware states and the definition for all hardware components at hand, the last thing we need, are all exogenous actions representing uncontrollable hardware component state changes properly defined in PDDL. We want the planner to search for possible hardware state changes, determine the effects on the executed actions and return all hardware state changes, that would lead to an adapted history, that is consistent with the observations. We want to encode the fact, that an explanation based on a single hardware state change is more likely than two or more unrecognised state changes at the same time by introducing action costs to these exogenous actions. By using cost-optimal planning algorithms, like K^*, we then compute k explanations with the lowest number of hardware failures. An example exogenous action that represents the move base component, as defined in Figure 4.2, switching to the STUCK state is shown in Listing 4.6.

```
1 (:action get_stuck
2          :parameter()
3          :precondition (state MOVE_BASE INIT)
4          :effect (and (not (state MOVE_BASE INIT))
5                       (state MOVE_BASE STUCK))
6 )
```

Listing 4.6: An exogenous action

The planner can then replace an executed action with one of its variations by inserting the exogenous action, that leads to a state for which an action variation is defined. Applying a state change like Listing 4.6 to the component at a certain point in the history, forces the planner to exchange the `move` action with its variation `move_stuck`, as defined in Listing 4.7.

```
1 (:action move_stuck
2          :parameter(?r - robot, ?from - location, ?to -
              location)
3          :precondition (and (next-move ?r ?from)
4                             (at ?r ?from)
5                             (state MOVE_BASE STUCK))
6          :effect (and (not (next-move ?r ?from))
7                       (last-move ?r ?from))
8              )
9 )
```

Listing 4.7: An action variation of move

We enable the engineer to define the finite state machines of the hardware components in *YAML* files, which are then parsed and dynamically added to the *PDDL* domain. A definition of the *Localization* component, as shown in Figure 2.2, is given by the *YAML* file shown in Listing 4.8. Exogenous transitions are identified by specifying a probability value for an edge.

A diagnosis is then solved by applying possible state changes, following the effect which the changed component state have onto the executed history and verifying that the resulting belief is consistent with the observation.

Following the example from the beginning, we want to find an explanation for the product P not being at the machine B. A part of the planner run on this diagnosis problem is shown in Listing 4.9. The first plan assumes that the `move_base` component got stuck, prohibiting the robot from moving to machine B. Therefore, the product was placed back into machine A. The second plan assumes the gripper to be broken, resulting in the product still being at machine A.

```
localization:
        states:
                - INIT
                - LOCALIZED
                - LOST

        INIT:
                edges:
                        - LOCALIZED
                LOCALIZED:
                        transition: localize

        LOCALIZED:
                edges:
                        - LOST
                LOST:
                        transition: lost
                        probability: 0.2

        LOST:
                edges:
                        - LOCALIZED
                LOCALIZED:
                        transition: localize
```

Listing 4.8: YAML definition of the localization component

```
Plan: 1
(order_1)
(pick R-1 P A)
(order_2)
(get_stuck)
(move_stuck R-1 A B)
(order_3)
(put R-1 P A)
(order_4)
; cost = 1

Plan: 2
(order_1)
(break-gripper)
(pick-failed R-1 P A)
(order_2)
(move R-1 A B)
(order_3)
(put R-1 P B)
(order_4)
; cost = 1
```

Listing 4.9: Planner output for the example problem

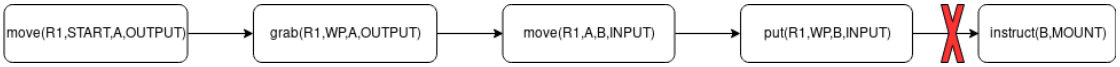

Figure 4.3: A plan that failed because of a failed instruct action

Domain Design Principles

While developing and testing, it became clear that one of the most critical points in terms of a successful diagnosis generation is the design of the *PDDL* domain. We identified some principles, that had to be followed in order to generate a *PDDL* domain, that is sufficient for a successful diagnosis candidate generation.

Restrict exogenous action insertion Using hardware models in the history-based diagnosis generation leads to the planner inserting hardware component state changes in some places into the history, exchanging plan-actions with faulty alternatives and checking if the generated history is consistent with the observation. However, there may exist several places in the history for an exogenous action to be inserted without changing the resulting world state. Due to the fact that *PDDL* planners generate totally ordered plans, this will result in a lot of solutions that only differ in the places in which an exogenous action was inserted. Only when comparing the resulting world states, it is possible to exclude duplicated diagnosis candidates. Additionally, in order to make sure that the top-k planner generates sufficient diagnosis candidates, we have to choose a rather large value for k in order to not loose any hypothesis due to the duplicated, optimal hypotheses, cluttering the top-k set.

To explain this problem in more detail, we want to examine an example of a failed plan, as shown in Figure 4.3. The plan failed, because the instruct action failed. We assume, that the instruct action has a single hardware component dependency, namely an established connection to properly send the instruct message. One explanation, among others is, that the robot lost the connection to the machine in some point of the plan. Since the previous executed actions did not depend on the connection, it is not clear at which point the connection was lost.

The planner now generates diagnosis candidates with the exogenous action that represents the connection loss, inserted at various places into the history, as shown in Figure 4.4. Note, that the introduction of the order actions, as shown in Section 4.2.1, doubles the places, in which an exogenous action can be inserted. All these diagnosis candidates result in the same world model and therefore do not contribute to the diagnosis.

To compensate the bloat of generated diagnosis candidates, we introduced an `(exogenous-possible)` predicate, that is added as an effect of each `order` action and retracted as an effect of each domain specific action. The `(exogenous-possible)` is then added to the precondition of every exogenous actions. Thus, an

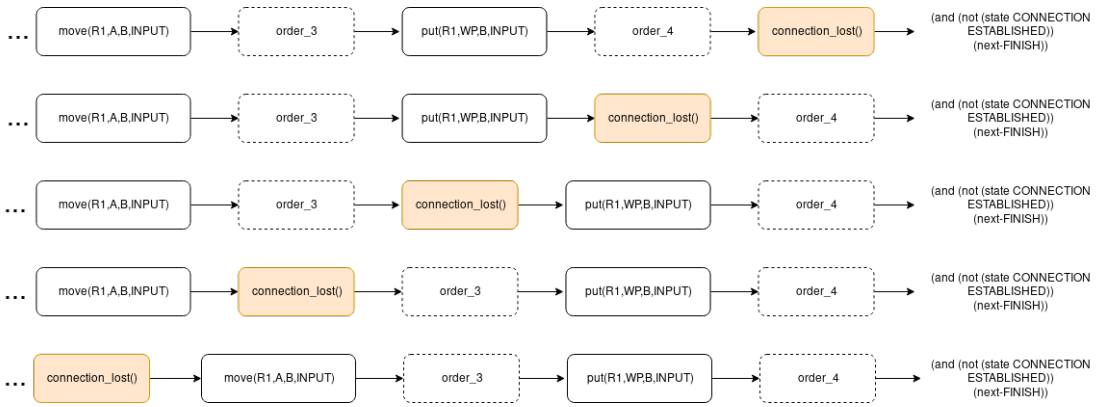

Figure 4.4: Multiple explanations, that result in the same world state

exogenous action can only be inserted after an `order` action and before an action of the history. By doing so, we can already remove half of the unnecessarily generated candidates. A proper solution would be, to allow the insertion of a hardware component state change action only directly before any action, that depends on this hardware component. This would properly model the fact, that we are sorely interested in the fact, that the state changed before executing the action. The exact point of time when the state change happened however is not of particular interest. However, this would drastically increase the complexity of the exogenous state change actions and any change of the normal actions would require changes in multiple places of the domain and probably lead to errors in an ongoing design process. Another solution would be the use of a planner that generates partially ordered plans. However, currently there exist no planner for the top-k planning problem that can generate partially ordered plans.

Multiple identical actions in a plan When using the order actions as proposed in Section 4.2.1, we ran into problems when diagnosing a failed plan, that contained multiple instances of the same action with the same parameters. Since the order actions only determine the next action in the history and some of its parameters, it is possible to skip a part of the history, when the same action appears twice or more in the history. To cope with this side-effect, we could use a counter predicate or any numerical fluent for each action/parameter combination, that is increased in the order actions. By adding the desired number of times an action/parameter combination appeared in the history to the goal condition, the planner is forced to generated adapted histories with the correct action sequence.

Model all side effects of an action When designing the *PDDL* definition of an action with the intention to use it for planning, it is sufficient to model the desired effects and preconditions of an action. Refer to Listing 4.10 for a `put` action that is sufficient for planning purposes. It correctly models the fact, that the `put` action

will move the workpiece, currently held by a robot, to the machine the robot is currently at, which is the way the `put` action should behave.

```
1 (:action wp-put
2   :parameters (?r - robot ?wp - workpiece ?m - mps)
3   :precondition (and (at ?r ?m INPUT)
4                      (wp-usable ?wp) (holding ?r ?wp)
5                      (mps-side-free ?m INPUT))
6   :effect (and (wp-at ?wp ?m INPUT)
7                (not (holding ?r ?wp))
8                (can-hold ?r)
9                (not (mps-side-free ?m INPUT))
10           )
11 )
```

Listing 4.10: Put action sufficient for planning purposes

In reality, putting a workpiece can be done even if the machine is currently occupied. Any workpiece that is located at the target position will be knocked away. Also, the robot may try to put without holding any workpiece. A first step to model the realistic behaviour of an action was presented before by differentiating between *hard* and *soft* preconditions. However, we still have to model possible, undesired side-effects as well. Holding a workpiece that was located at the machine is an optional but desired effect. Knocking a workpiece, that rested at the machine, away while putting another workpiece, is an undesired effect and is normally not included in the PDDL model of an action normally. Therefore, we identified some principles, which should be followed when modelling an action for diagnosis purposes:

- Use preconditions only for conditions, that have to be true in order for the action to complete the execution without failing and grounding of parameters. This contains all conditions, that are checked during the execution of the action.
- Any other conditions, that influence the effect of the action, have to be used as conditions for conditional effects.
- Sometimes an action can have arbitrary undesired side effects on other objects. In this case these objects have to be added to the parameters and grounded in the preconditions. Use a dummy object in cases where there is no additional object affected to prohibit bloated grounding of action instances.

Naturally, this will increase the complexity of an action model drastically. However, it is necessary for a complete diagnosis candidate generation. Refer to Listing 4.11 for an example `put` action that correctly models all effects, an executed put action can have.

```
(:action wp-put
  :parameters (?r - robot ?wp - workpiece ?m - mps
                          ?side - mps-side
                          ?there - workpiece)
  :precondition (and (state gripper CALIBRATED)
                     (state realsense ACTIVATED)
                     (state tag-camera ACTIVATED)
                     (state laser ACTIVATED)
                     (at ?r ?m ?side)
                     (or (and (mps-side-free ?m ?side)
                              (dummy-wp ?there))
                         (wp-at ?there ?m ?side)
                     )
                     (or (holding ?r ?wp)
                         (and (can-hold ?r)
                              (dummy-wp ?wp))
                     )
                 )
  :effect (and (not (exog-possible))
               (when (and (holding ?r ?wp))
                     (and (wp-at ?wp ?m ?side)
                          (not (mps-side-free ?m ?side))
                          (not (holding ?r ?wp))
                          (can-hold ?r)
                     )
               )
               (when (and (not (mps-side-free ?m ?side))
                          (wp-at ?there ?m ?side))
                     (and (not (wp-at ?there ?m ?side))
                          (not (wp-usable ?there))
                     )
               )
           )
)
```

Listing 4.11: Put action sufficient for diagnosis purposes purposes

Note the use of conditional effects to move a part of the preconditions used in the action desired for planning to the effects, following the second principle. For example, the put action will succeed, even if the robot does not hold a workpiece. However, if it holds a workpiece, the workpiece will be placed on the machine and the robot does not hold it any more.

Additionally, there is an optional object influenced by the put action, as mentioned above. The `?there` workpiece refers to a workpiece, that may already laying at the machine. This parameter is grounded with a dummy workpiece, if the machine is not already occupied and grounded with the workpiece that will be knocked down otherwise. The same method is used for the `?wp` workpiece, that will be placed onto the machine if the robot currently holds the workpiece and grounded with the dummy workpiece otherwise. Alternatively, we could have multiple `put` actions, modelling

the different situations in which a put action could be executed. So, a `put_nothing` action and a `put_knock_off` action would have to be added. The `put_nothing` action models a put action without holding a product and the `put_knock_off` action models a put action to a location where already a product rests.

Additional to the integration of the history and the platform models into the *PDDL* domain description, we have to generate a *PDDL* problem description. The *PDDL* problem description contains all available objects, the initial world state and a *goal* description.

4.2.2 Problem Description

There are two possible reasons why we want to perform a diagnosis task. The first and more straight forward reason is, that an action failed at some time during plan execution. Provided the action was correctly modelled, this implies that one of its preconditions was wrongly believed to be true. Thus, we can create a PDDL *problem description*, as described in Subsection 2.4.1, by adding the state of the world from before executing the plan and a goal description consisting of (next-FINISH) and the negation of the precondition (without the ordering predicates) of the failed action. The k-planner then generates all history adaptations, that lead to one or more of the preconditions of the failed actions to not be fulfilled.

```
1 (:goal (and (next-FINISH)
2              (not (and (at ?r ?from)
3                        (state MOVE_BASE  INIT)
4                        (state NAVIGATION LOCALIZED)))))
```

Listing 4.12: A diagnosis goal for a failed move action

The second situation, where we want to perform a diagnosis task is when some sort of monitoring process observes a fact that contradicts the current world model. In this case we want to reevaluate the last executed or currently running plan. Similar to the problem description generation after a failed action, we can generate the description by adding the contradicting observation to the goal.

4.3 Active Diagnosis

As described in Section 2.2, after generating diagnosis candidates, we have to narrow the set of diagnosis candidates in order to identify the correct diagnosis. For a given set of diagnosis candidates, a set of facts that are true in all diagnosis candidates can be determined, which is called the shared knowledge. For every candidate, that

is excluded by some sensing action, the shared knowledge will increase since at least one disagreement on a fact will be resolved. Following the *active diagnosis* approach of Mühlbacher and Steinbauer (2014) and introduced in Section 2.2, we select the sensing action with the highest mutual information $I(\alpha(\vec{x}); W(s))$ regarding the set of diagnosis candidates $W(s)$:

$$I(\alpha(\vec{x}); W(s)) = H(\alpha(\vec{x})) - H(\alpha(\vec{x})|W(s)) \tag{4.1}$$

We introduced a *active diagnosis* CLIPS environment in which the resulting world states are calculated and sensing results are integrated. After a failure was detected, the *PDDL* problem was generated and possible diagnosis candidates were calculated by using the K^* planner, we set up the *active diagnosis* environment by calculating the resulting world state of each diagnosis candidate. The CLIPS Executive then can request the information gain of a sensing action, which will be calculated by using the formula presented in Section 2.2. After executing a sensing action, the CLIPS Executive informs the *active diagnosis* environment about the sensing result, which then excludes all contradicting diagnosis candidates. Afterwards, the shared knowledge can be updated and any new information can be synced to the CLIPS Executive world model.

In order to calculate the impact $H(\alpha(\vec{x}))$ of a sensing action $\alpha(\vec{x})$, the probability p_w of each diagnosis candidate $w \in W(s)$ is used. Here, we incorporated the knowledge of an engineer, as given in the hardware model configurations, about the probabilities of certain hardware component state changes. Each diagnosis candidate, that assumes a hardware failure, consists of one or more exogenous actions, that model such a component state change. These exogenous actions represents the exogenous edges of the hardware models, as explained in Section 2.3. Thus, by assigning probabilities to exogenous transitions of the component models, as shown in Listing 4.8, we can determine the probability of a diagnosis candidate w as $p_w = \prod_{a \in h_w} p_a$, where a is an action in the history of w and p_a the probability of the action, if it is a state change action, or 1 if not. Note, that the probability values do not have to represent the actual probabilities of the real system, as long as the proportions between the probabilities are correct.

Since this thesis focusses on diagnosis candidate generation and recovery, we make some assumptions and adaptations, resulting in a simplified active diagnosis process.

Ignore sensing noise As described previously, the mutual information gain of an action depends on the entropy of the probabilities of the resulting splitting. The information gain $I(\alpha(\vec{x}); W(s))$ is decreased depending on the noise $H(\alpha(\vec{x})|W(s))$, a sensing action is subject to. However, since the work in this thesis concentrates on the diagnosis candidate generation and recovery behaviour, we will omit the

sensing noise and assume perfectly working sensing actions. Therefore, in this thesis is $H(\alpha(\vec{x})|W(s)) = 0$ for all sensing actions. Investigating the influence sensing noise will have onto the active diagnosis process is a task on its own and shall be subject of future work.

Single valued sensing result To ease the calculation of the diagnosis candidate set splitting, we assume that any sensing action only senses a single predicate. This is sufficient for the sensing actions in our domain and results in simple calculation of the splitting of the diagnosis set.

Assume finite amount of domain objects As mentioned in Section 2.2, there exist two different algorithms for calculating the next sensing action. The difference between these two algorithms is, that the first one requires a domain with finite number of domain objects guaranteed, whereas the second one can deal with infinite numbers of domain objects but requires a first order theorem solver. In general, we do not want to make the assumption of a finite amount of domain objects. In this thesis, we want to make this assumption in order to not have to integrate a first order theorem prover and to be able to concentrate on the diagnosis generation and repair.

4.3.1 Integrating Active Sensing Into CLIPS Executive

In order to integrate the active sensing into the CLIPS Executive, we had to make some adaptations to the CLIPS Executive. The adaptations are presented in the next sections, along with a more detailed explanation of the implementation of the active diagnosis.

Shared Knowledge

One major point of being able to perform active diagnosis is the knowledge about which sensing actions are executable. In order to be able to achieve this goal, we need to determine what is true in all diagnosis candidates. The resulting world state is called shared knowledge and contains every information that all diagnosis hypotheses can agree on. Therefore, we can not only use the shared world model for selecting sensing actions, but also update the CLIPS Executive world model every time the set of diagnosis candidates gets narrowed down, resulting in an online information gain during the active diagnosis process.

The diagnosis generation, as described in the preceding chapters, returns a set of diagnosis candidates. A diagnosis candidate as given as an adapted action sequence. To determine the resulting world state of each diagnosis candidate, we use a stored CLIPS Executive world model from the point of time when the execution of the

diagnosed plan started. By propagating the actions of a diagnosis candidate through applying all effects of the actions one after another, the resulting world state can be determined. After propagating the diagnosis candidate actions, we can exclude one of any diagnosis candidates, that result in the same world state. The set of facts believed to be true for a single diagnosis candidate d_i is then given as $K_{d_i} = \{\rho | \rho \text{ is true in } d_i\}$. For the active diagnosis at time t, the shared knowledge is defined as $K^t_{shared} = \cap_{d_i \in V^t} K_{d_i}$ with V^t the set of valid diagnosis candidates at time t.

Since the shared knowledge only will increase as more diagnosis candidates are excluded during the active diagnosis, we already can update the CLIPS Executive world model. Thus, the active diagnosis results in an online updating of the world model, which is especially useful in cases where the active diagnosis may take a longer amount of time. By incrementally updating the world state according to new findings, any other agent can already formulate goals on the partially updated world model.

However, synchronizing the CLIPS Executive world model with the shared knowledge is not trivial. The CLIPS Executive world model may change during the ongoing diagnosis, due to parallel running goals or updates from other agents. Simply deleting all facts, that are not in K_{shared} may delete facts that were not influenced by the diagnosed plan. To cope with this difficulty, we build up another set of facts that represents the assumed world state after executing the plan. This is simply done by applying all executed actions of the original history to the backup world state, resulting in $K_{assumed}$. Obviously, $K_{assumed}$ is known to be unconsistent with the real world, since it contradicts the observation that lead to the diagnosis in the first place. Now, we can determine all facts to be deleted from the CLIPS Executive world model by selecting all facts true in $K_{assumed}$ and not true in K_{shared}. Thus, any fact that was asserted to the CLIPS Executive world model and not influenced by the diagnosed plan will be ignored.

Asserting facts of the shared knowledge can be dangerous due to similar reasons. There might be facts in K_{shared}, that were not influenced by the diagnosed plan at all, but another agent deleted them. Blindly reasserting these facts would revert the changes introduced by another agent. Thus, we only assert facts, that are true in K_{shared}, but not in $K_{assumed}$. However, there is still one case that needs treatment. A fact $\rho \in K_{assumed}$ might be true in one or more $d_i \in V^t$, but there is at least one d_k, with $k \in V^t$ where $\rho \notin K_{d_k}$. Thus, at time t, syncing the shared knowledge to the CLIPS Executive world model would have removed ρ from the CLIPS Executive world model since $\rho \notin K^t_{shared}$. If the next sensing action excludes all diagnosis candidates in which ρ is not true, ρ will be in K^{t+1}_{shared} and have to be asserted to the CLIPS Executive world model again. Since ρ already is a member of $K_{assumed}$, the above condition is not met and we are in need of another condition for reasserting

deleted facts. This is solved by keeping track of all deleted facts by adding them to a set $K_{deleted}$ and also allowing the reinsertion of these facts. Thus, asserting a fact ρ to the CLIPS Executive world model is allowed if one of the following conditions is met.

- $\rho \in K_{shared}$ and $\rho \notin K_{assumed}$
- $\rho \in K_{shared}$, $\rho \in K_{assumed}$ and $\rho \in K_{deleted}$.

Both of these conditions for syncing the shared knowledge to the CLIPS Executive world model are based on the assumption, that no other agent is able to modify the facts that are influenced by the diagnosed plan. If this assumption is not valid, additionally checks for the syncing process are needed in order to guarantee a consistent world model.

Sensing actions

Currently, the CLIPS Executive does not have a concept of sensing actions. Therefore, we had to find a way to define sensing actions in order to enable the active diagnosis to determine all executable sensing actions and the information gain of each grounded sensing action. This is done by using a `domain-sensing-action` fact template, that defines the sensed predicate for a *PDDL* operator, as shown in Listing 4.13. Each sensing action is then defined in the *PDDL* domain, just like all normal actions, and the `domain-sensing-action` is added to define the sensed predicate and the connection between the parameters of the operator and the parameters of the predicates.

```
1 (deftemplate domain-sensing-action
2   "Defines an operator as sensing action. A successful
      executed sensing action results in the assertion or
      retraction of the grounded sensed predicate"
3    (slot operator (type SYMBOL))
4    (multislot param-names (type SYMBOL))
5    (slot sensed-predicate (type SYMBOL))
6    (multislot sensed-param-names (type SYMBOL))
7    (multislot sensed-constants)
8 )
```

Listing 4.13: CLIPS struct for sensing actions

Imagine an action, that takes pictures using a realsense camera and uses some sort of image recognition method to determine, whether there is a workpiece located at the machine, the robot is currently standing, or not. The action depends on the

realsense camera to be activated and the robot standing at the machine and does not have any direct effect, as shown in Listing 4.14.

```
1 (:action check-workpiece
2         :parameter(?r - robot, ?m - mps, ?side - mps-side)
3         :precondition (and (state REALSENSE ACTIVATED)
4                            (at ?r ?m ?side)
5         :effect ()
6 )
```

Listing 4.14: Check workpiece sensing action

For this operator, a `domain-sensing-action` fact is asserted, which determines, that the check-workpiece action will assert or retract the `mps-side-free` predicate. The `sensed-param-names` define the parameter names of the sensed predicate. The parameters are grounded with a constant, if the corresponding index of `sensed-constants` is not nil, and with the grounded parameter of the same name of a grounded check-workpiece action otherwise. Thus, the `check-workpiece(R-1 C-CS1 INPUT)` will assert or retract `(mps-side-free C-CS1 INPUT)` depending on the result of the sensing.

```
1 (domain-sensing-action
2    (operator check-workpiece)
3    (param-names r m side)
4    (sensed-predicate mps-side-free)
5    (sensed-param-names m side)
6    (sensed-constants nil nil)
7 )
```

Listing 4.15: domain-sensing-action fact for the check-workpiece operator

Regarding Incomplete Knowledge

As mentioned above, the CLIPS Executive makes the closed world assumption. That means, that every fact not explicitly known to be true, is treated as false. However, if active diagnosis fails to result in one single diagnosis hypothesis, there is at least one fact that is true in one diagnosis candidate and false in another one. Unfortunately, this is not directly representable in the CLIPS Executive. The current implementation of the RCLL domain in the CLIPS Executive uses a lot of mutually exclusive predicates, representing the state of an object. For example for any usable workpiece, there exist a fact `(domain-fact (name wp-at) (param-values WP M SIDE))`, representing that the workpiece WP is currently located at the SIDE side

of machine M. This basically resembles functional fluents, since we always assume exactly one predicate for each object and property. So, as soon as active diagnosis can not agree on the truth value of such a predicate, there will be no information about the position of an object at all. This indirectly models the uncertainty about the position.

However, with this indirect way of modelling incomplete knowledge about a property of an object, we will loose information gained by the active diagnosis. For example, we may end up with two diagnosis candidates, that can not agree on the position of a workpiece. But, since we only have two diagnosis candidates left, we know, that the workpiece has to be at one of two possible positions. However, the information about possible positions can not be transferred into the CLIPS Executive, since the CLIPS Executive world model is unable to express disjunctive knowledge. As mentioned above, the CLIPS Executive will remove the `wp-at` fact about the workpiece, loosing the information, that we know that the workpiece can only be at one of the two machines.

4.4 Repair

To really benefit from the integration of hardware models into the diagnosis approach, the robot needs to be able to take the information gained about possible hardware faults into account when deciding about future goals. Based on the findings of a diagnosed failure, the agent may decide to try to repair itself, if possible, or to exclude itself from the production and call for human intervention. In a multi-robot scenario it is better to give up and let other agents do the job, than to block resources while trying to reach a goal with broken hardware repeatedly.

So, after performing active diagnosis, we may end up with a single diagnosis candidate. If this diagnosis candidate blames a previously unrecognised hardware component state change, the agent now has a new insight about the state of its hardware platform. However, in some cases we may end up with more than one diagnosis candidate without being able reduce the set any further. Thus, the agent has to deal with incomplete knowledge about the exact hardware component that caused the failure or about the exact state change that occurred. Note, that the active diagnosis and the way we implemented the update of the CLIPS Executive world model, will only result in a component state for component c, if all diagnosis candidates agree on one state for c. So after finishing the active diagnosis, we end up with one or more components with unknown state or a single component being in a faulty state. To enable the CLIPS Executive to deal with these kind of situations, we introduced *repair* goals. For each component c, that is able to reset itself and where the success

of the reset is observable by the agent, a specific repair goal can be formulated if one of following conditions is true:

- The state of the component c is unknown.
- The component is in a state where a reset is possible.

Thus, as soon as the diagnosis of a failed goal finishes, the failed goal gets retracted and the CLIPS Executive starts over by formulating all reachable goals. If any component needs repair that has a repair goal, this repair goal gets formulated with the highest priority.

Take a failed gripping action as example. The robot drove to a machine and tried to grab a product from the output side. After finishing the gripping action, it senses that the gripper is empty. After generating all possible diagnosis candidates and performing active diagnosis, the explanations, that are still valid, are:

- The laser, used for aligning the with machine, has got decalibrated and therefore the robot was not aligned properly.
- The gripper has a broken axis, so it could not move.
- The gripper is decalibrated, resulting in missing the product when performing the gripping action.

The agent is not able to determine whether the laser is decalibrated or not, as well as it can not determine, if the axis is broken or just decalibrated. Therefore, it has to deal with two components, that could be faulty. The agent knows that it can try to repair the gripper by calibrating it. Calibrating a gripper will fail if the axes are broken. So after finishing the diagnosis process of the failed goal, the agent will formulate and select the `REPAIR-GRIPPER` goal, as shown in Listing 4.16. If this goal succeeds, the gripper component is known to be in the calibrated state again, and will be in a broken state otherwise.

```
1 (defrule goal-production-gripper-repair
2   (declare (salience ?*SALIENCE-GOAL-FORMULATE*))
3   (goal (id ?urgent-id) (class URGENT) (mode FORMULATED))
4   (or (not (domain-fact (name comp-state)
5                         (param-values gripper ?state)))
6       (domain-fact (name comp-state)
7                    (param-values gripper UNCALIBRATED)))
8   (not (goal (class REPAIR-GRIPPER)))
9   =>
10  (assert (goal (id (sym-cat REPAIR-GRIPPER (gensym*)))
11                (parent ?urgent-id)
12                (class REPAIR-GRIPPER)
13                (sub-type SIMPLE)
14          )
15  )
16 )
```

Listing 4.16: REPAIR-GRIPPER goal.

After attempting to repair all faulty components or resetting components with unknown state, the agent may try to continue the production if possible. However, by guarding the formulation of production goals with conditions that check for the required hardware components being functional, the agent can react on being partly broken. If no production goal is executable, the agent may decide to call for human intervention since it is not operable any more.

5 Evaluation

In this chapter we want to present the results and observations we obtained while testing our approach. Mainly, we want to evaluate two main questions:

- What effect does the integration of hardware models have onto the generation of history-based diagnosis candidates?
- To what extent does a team of robots benefit from the integration of hardware models into the diagnosis step?

The first question can be answered by comparing the runtime and the number of expanded states of the K^* planner for the same diagnosis problem, but once with and once without integrated hardware models. The second question will be answered by analysing the performance of a team of 3 robots in simulated RCLL games with simulated hardware failures. We will evaluate three different settings for the team of robots.

1. Equipped with diagnosis with integrated hardware models.
2. Equipped with diagnosis without integrated hardware models.
3. No diagnosis at all.

5.1 Simulation

As mentioned in Section 2.5, we are able to simulate complete RCLL games in Gazebo. The Gazebo simulation consists of several plugins, that simulate hardware components, processes and machines. Among others, a robot is simulated by a gripper plugin, a conveyor align plugin and several sensor plugins. The Gazebo simulation is able to simulate up to six robots, three for each team. Since we are sorely interested in the performance of a single team, we will only use 3 robots. Gazebo offers communication with external components via TCP/IP sockets, managed by Boost ASIO. These sockets establish topics, on which messages are send by a *publisher* to *subscribers* (Koenig and Howard, 2004). *Protobuf* messages are used to define the different message types, that are send on the topics.

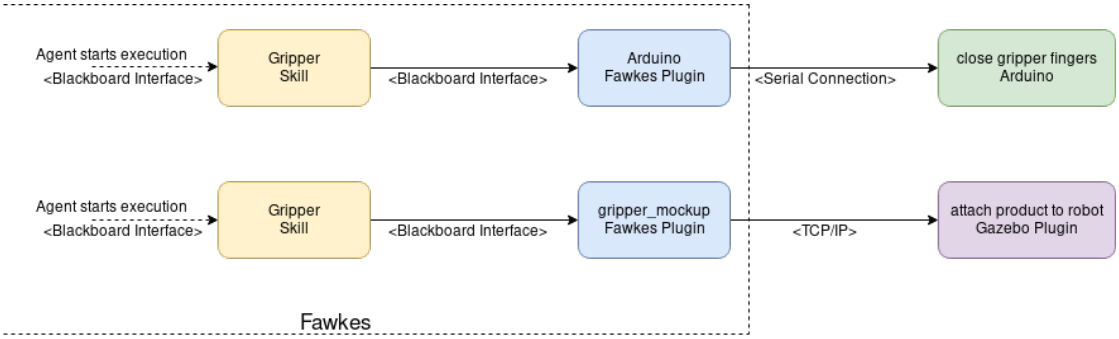

Figure 5.1: Closing gripper fingers in reality and simulation

Simulating Action Execution

As described in Section 2.6, any action, executed by the agent, that resembles a more complex behaviour, is implemented as a skill. A skill combines several low-level components to more abstract behaviours. In the simulation, these low-level components are simulated by Gazebo plugins. For example, when operating in the real world, grabbing a product utilizes an alignment and a gripping subskill. First, the alignment skill is used to align with the machine output and uses several different plugins in order to accomplish this task. Then, the gripper skill is used to control the gripper axes. In order to do so, the gripper skill instructs the Arduino plugin, which controls an Arduino board via a serial connection. The Arduino board then controls the motors of the gripper axes. When simulating, the Arduino plugin gets replaced by a mock up plugin, that interacts with a Gazebo plugin, responsible for simulating gripping and placing, as shown in Figure 5.1.

To simulate hardware failures, each Gazebo plugin, that simulates a hardware component, maintains a finite state machine with randomly triggered state changes. The probabilities of a state change can be configured and at certain points in time plugin will check if the state of the finite state machine should change, simulating a malfunctioning hardware. Thus, we can simulate unexpected component state changes and the effect this will have on to the real world. The agent than may come across unexpected observations or action failures determined by these simulated state changes.

Sensing actions are implemented by sending a *protobuf* request to the corresponding Gazebo plugin. Similar to sensing actions, the reset and repair actions are realized. Sensing results are transmitted via *protobuf* messages as well. For example, the agent may sense for the gripper component being calibrated and sends a *protobuf* request message. The Gazebo gripper plugin of this robot then checks the simulated state of the gripper and sends the response. We use the predefined request and response messages from Gazebo.

Simulated Hardware Components

The scenario we chose to evaluate the diagnosis approach consists of four hardware components: *gripper*, *realsense* 3D camera, *laser scanner* and *tag camera*. These components and their states are strongly influenced the system used by the team Carologistics at the RoboCup in the last years (Hofmann et al., 2019), (Hofmann et al., 2018a).

The *gripper* is used for gripping and placing products and has 3 fault states:

- The axes can become decalibrated, resulting in unprecise movement and eventually missing the product while trying to reach it.
- The axes can break, therefore loosing the ability to move at all.
- The gripper fingers can get stuck, resulting in the currently gripped product to not be released.

All three different fault states have different effects on the execution of the gripping and placing actions. For example, a placing action with a broken gripper axis will result in dropping the product to the floor, while trying to place the product with broken gripper fingers will result in the product still being grabbed and any product that may have been laying at the target position being knocked down. The agent may sense if the gripper is calibrated and if not, try to recover by recalibrating the gripper. Obviously, the recalibration will fail if the axis or the gripper fingers are broken.

The *realsense* is needed by the gripping and placing actions to determine the position of the conveyor belt, on which the products are located or have to be placed. The *realsense* sometimes either loses the connection or breaks completely. In both cases, the aligning of the grabbing action will fail. Sometimes, the robot is able to recover the realsense camera by disconnecting and connecting again.

A *tag camera* is used to detect the marker of the machines and therefore find the approximated position of the machine when aligning. In rare cases, the *tag camera* stops working and in this scenario we assume that the agent has no control or information about the state of the camera.

Similarly, the *laser scanner* is used for determining the machine position. Also, the laser scanner is used for localization purposes. Therefore, the state of the laser scanner is also a dependency for all actions that require localization or navigation.

With all these hardware components, we cover most of the failure types. Errors caused by the gripping actions are not detected right away and may be discovered indirectly when a machine fails to process due to a missing product in its input. These failures are the most complicated in regard to the world model, since the position

of the product, that was assumed to be moved, is important for the continuation of the production and therefore has to be re-evaluated. If the product still rests in the old position, it has to be removed. Otherwise, the product blocks the output of the machine it is resting in, drawing the whole machine unusable. The realsense has a direct effect on the execution of gripping and placing actions and cause a failure of the execution. But contrary to the tag camera, the state of the realsense can be sensed and in case of a lost connection, the connection can be re-established.

Restrictions

For a selected goal, the CLIPS Executive uses pre-defined plans with 5-17 actions in order to expand the goal. We introduce some restrictions to the simulated scenario, in order to reduce the complexity of the evaluation and to be able to concentrate on the performance of our approach:

- Following the principle of Ockham's razor, we ignore all explanations that do not imply a minimal number of exogenous state change actions. Therefore, if we find an explanation that assumes only one unrecognised state change, we ignore all explanations with two or more state changes.
- We assume perfect sensing actions without noise or failures. Theoretically, our approach also can diagnose failures of sensing actions, that are part of another ongoing diagnosis. However, this does not promise any new insights while increasing the complexity of the integration into the CLIPS Executive drastically.

5.2 RCLL Game Results

We simulated 16 games per scenario, in order to calculate the average number of points the team of robots was able to achieve. Points are awarded for successfully delivered products and buffering left-over products in the *ring stations*. Each 0.5 seconds, the plugins that simulate hardware components change their state with the configured probabilities. We can add all probabilities of state transisitions to states from which the robot can recover, e.g. the UNCALIBRATED state of the gripper, to $p_{\text{recoverable}}$. Similarly, all transitions to terminal states from which the robot can not recover can be summed up as p_{terminal}. Both combined represent the probability of any hardware failure: p_{failure}.

p_{failure}	No Diagnosis			Diagnosis without integrated hardware models			Diagnosis with integrated hardware models		
	Mean	Min	Max	Mean	Min	Max	Mean	Min	Max
0.12%	71.31	6	260	76.66	12	217	160.94	93	229
0.19%	58	4	142	68.44	12	151	156.6	91	295
0.27%	32.75	0	98	37.88	0	144	109.5	10	174

Table 5.1: Mean, minimum and maximum number of points a team of robots was able to achieve.

We evaluated the three different teams in three scenarios with different values for $p_{\text{recoverable}}$ and p_{terminal}, as shown in Table 5.1. In general, a team of robots with history-based diagnosis was not able to increase its performance in comparison to a team without diagnosis. Integrating hardware models into the diagnosis process resulted in a significant performance gain, as shown in Figure 5.4. However, all three teams are influenced by the probabilities of a hardware failure. Even though the teams with integrated hardware models were only stopped by hardware failures, from which the robot can not recover itself, the time needed to recognize a failure and performing the diagnosis had a negative impact onto the number of points gained.

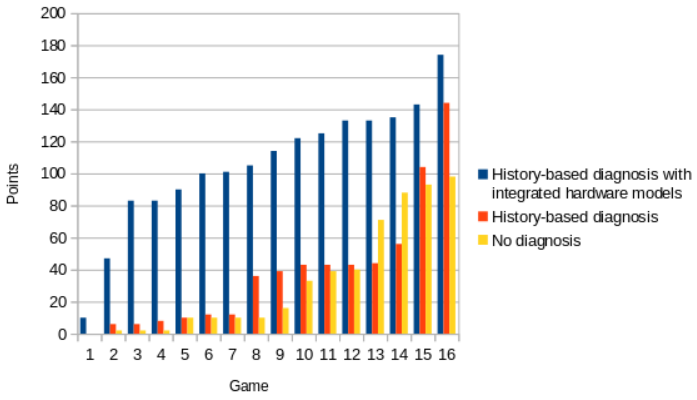

Figure 5.2: Ordered list of points gained in each game for $p_{\text{failure}} = 0.27\%$

In all three settings for the hardware failure probabilities, the teams with our approach were able to gain more than 100% more points in comparison to the other teams. In scenarios with a more unreliable hardware the performance gain is even higher, as shown in Figure 5.3.

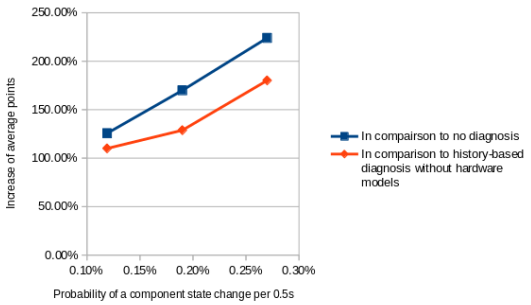

Figure 5.3: Increase of the points gained by a team equipped with diagnosis with integrated hardware models

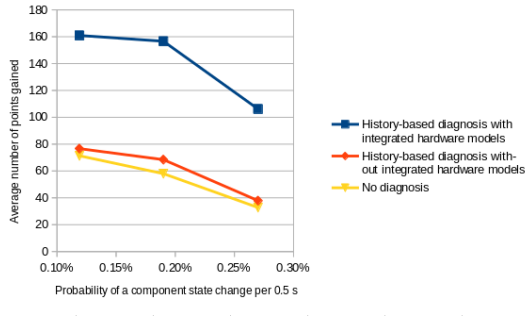

Figure 5.4: Average number of points gained of teams with different evaluation strategies

In order to understand why our approach enables a team to continue the production, we want to analyse the reasons for a production stop. Generally, a team is unable to gain any more points if all robots are broken or if a critical combination of machines is not usable any more. We identified two main causes for a full production stop, apart from all robots being not functional.

A goal failure is either caused by an observation that lead to the decision that the goal is not reachable any more or an action of its plan was failed. In both cases, the agent knows that its world model is inconsistent with the real world. As long as the world model is inconsistent, the agent is in danger of continuously failing to accomplish its task or even waste resources that could have been used by other robots. However, teams with history-based diagnosis are able to revisit an inconsistent world model. By performing active diagnosis, the world model can be updated to be consistent with the real world again.

In our evaluated scenario, a machine is only able to process a production step if it is in the IDLE state and not occupied by another product. So, if a robot fails to pick up a product from the output of a machine, this machine can not be used until the product gets removed eventually. This is critical, especially for the base station, which dispenses the ground product and is therefore needed as the initial step for every product. A robot team without diagnosing is not able to re-evaluate the picking action in the hindsight of a failed subsequent production step. Therefore, as soon as a pick failed at the base station or any other station critical for the production, the team is unable to continue. A team equipped with history-based diagnosis is able to determine the possibility of a leftover product.

The second cause for an agent to not be able to continue the production is missing awareness about its own incapability. Even if its world model is consistent with the real world after a failure, the underlying cause may have effect on future actions and goals. Therefore, determining the effect a failure had onto the world is not

sufficient to enable an agent to continue the production if the underlying cause was not determined and repaired. Only the teams equipped with history-based diagnosis and knowledge about the hardware components are able to identify the cause of the failure, potentially repairing the malfunctioning hardware and prohibit any blockade of the production. Any robot, that failed to recover from a hardware failure, excludes itself from the production. This enables the robots, that are still functioning correctly, to continue the production. Therefore, these teams are able to continue the production as long as at least one robot is not broken. This results in the drastic increase of the average number of points we observed in the simulation and shows the success of our approach.

After a failure was detected and a robot with history-based diagnosis determined that a product was left at a machine, the agent is able to decide to remove the product. However, if the failure was caused by a malfunctioning hardware and the same robot is tasked with removing the product from the output of a machine, it will continue to fail this task and probably keep retrying. Thus, the robot gets stuck in a loop of failing and restarting the task, effectively blocking this machine as well. A team, that only uses a classical history-based diagnosis approach, therefore will know about a product remaining at the output of a machine, but still fail to continue production if the robot tasked with the removal suffers from a broken hardware component. This explains, why the teams without integrated hardware models are not able to gain significantly more points than a team without any diagnosis procedure at all.

However, the hardware-based diagnosis assumes hardware failures as primary cause of action failures. Therefore, an action failure that is not caused by a hardware failure, e.g. failed to find a plan when moving to a new position, often leads to problems. The diagnosis process will likely generate explanations containing hardware component state changes. If the active diagnosis is not able to exclude all of these explanations, the robot is unable to decide that it is still fully functional. In some cases, he will wrongly decide that he is not functional and will exclude itself from the production. For example, if a `move` action failed, because no path could be found, the diagnosis procedure may blame the laser component as well, since the `move` action depends on a functioning laser scanner. However, if the agent is not able to sense the state of the laser scanner, there is no way of determining if the failure was random or caused by a hardware failure. Thus, the agent may decide, that the state of the laser is undetermined and therefore he should exclude himself from the production. This is preventable by making sure, that the state of every hardware component is observable. However, in some cases this is not possible and therefore the risk of a false exclusion from the game remains.

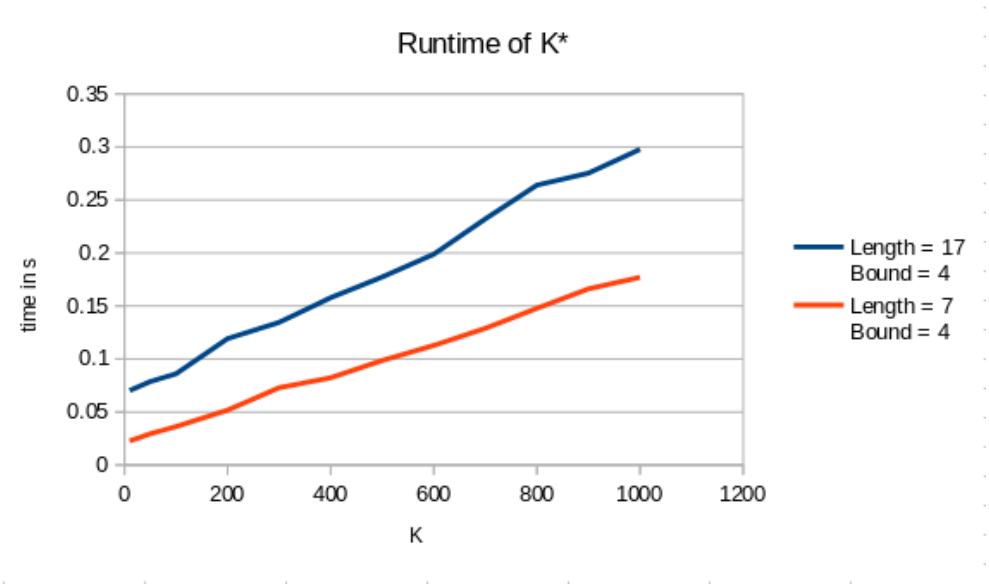

Figure 5.5: Runtime comparison of the K* planner for plans of length 7 and 17

5.3 Performance

The time a diagnosis of a fault takes is influenced by three factors: the generation and solving of the *PDDL* problem, the setup of the *active diagnosis* and the sensing and execution of the active diagnosis. In the following chapter, we want to evaluate the performance of the planner when solving the generated diagnosis problem and the performance of the active diagnosis.

5.3.1 Planning Performance

The initial assumption that restricting the action variations to component states will result in a reduction of the search space for diagnosis candidates proved to be wrong. In contrary, the state space that has to be searched while searching for valid history alternations is far bigger. Upon further investigation, it became clear that while we reduce the number of possible replacements in each step, the insertion of state change exogenous actions at any point into the plan results in a much bigger state space.

Given this drawback of our approach, the planning complexity and runtime of diagnosis generation is still an important aspect since the robot is forced to wait for the diagnosis process to finish before continuing with the production. K^* has two tunable parameters: the number of solutions k and the upper bound of the cost. As mentioned before, we assign a cost of one to each exogenous state change action.

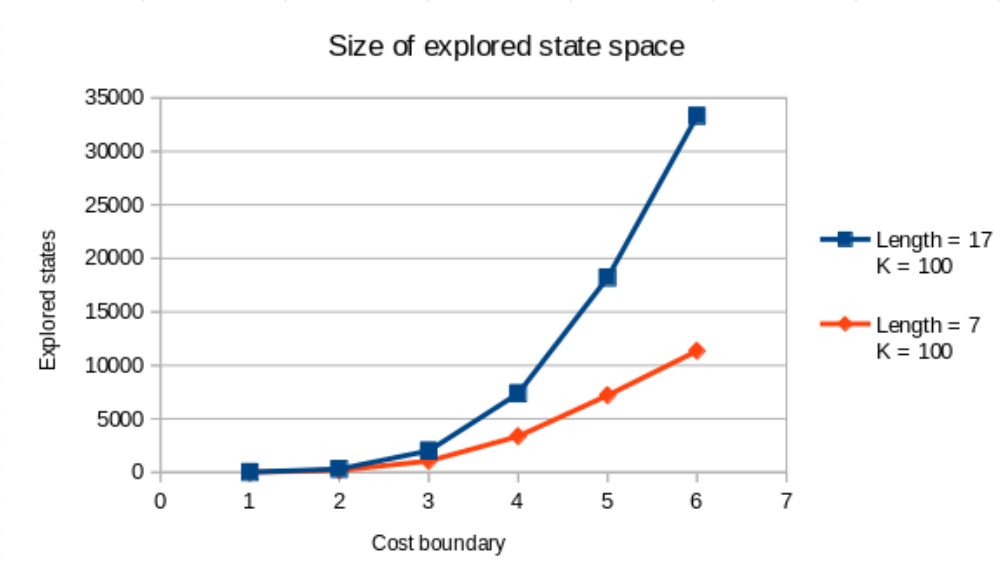

Figure 5.6: Number of explored states of the K* planner for given upper bound of the cost

Thus, the upper bound of the cost relates to the number of unknown state changes allowed in one diagnosis candidate. It is reasonable to assume, that most of the failures are caused by a single hardware state change. However, in rare cases two hardware failures may occur during one plan. As shown in Figure 5.6, the upper bound of the cost has a huge impact onto the number of explored states and therefore on the runtime. We experienced that an upper bound of three was sufficient for all scenarios we tested in the RCLL domain.

The number of generated diagnosis candidates is also important in order to guarantee that the correct diagnosis can be generated. We observed a linear increase in the runtime of K^* for an increased number of k, as shown in Figure 5.5.

5.3.2 Active Diagnosis Performance

Setting up the *active diagnosis* requires the generation of the resulting world models of all valid diagnosis candidates. This is done by applying the effects of the actions of the diagnosis candidates onto the initial world model. Depending on the length of the diagnosed history, the number of valid diagnosis candidates and the size of the world model, this can have a huge impact on the overall time needed by the diagnosis process. In our scenario, we diagnosed plans with between 5 and 17 actions in a world model that typically consists of 140 - 170 facts. With up to 100 valid diagnosis candidates, setting up the active diagnosis took up to 30 seconds.

One factor that has a negative influence on the time needed to set up the active diagnosis, is the number of diagnosis candidates that result in the same world model. These diagnosis candidates do not contribute to the result of the diagnosis. Instead, as soon as two diagnosis candidates that result in the same world state are detected, one of them gets deleted. Most of the diagnosis candidates only differ by the place in which an exogenous state-change action was inserted. Therefore, by restricting the insertion of this kind of actions further, we can reduce the number of unnecessarily generated diagnosis candidates and therefore speed up the active diagnosis.

6 Conclusion

We summarize our approach to integrate hardware models into a history-based diagnosis process. Also, possible improvements are presented considering the performance and restrictions.

6.1 Summary

In this work, we integrated a model of the hardware platform an agent operates on into a *history-based diagnosis* process. A component of the hardware platform is modelled as a finite state machine, where the states represent the state of the component and edges represent exogenous or executable state transitions. Integrating the component states and state transitions into a *PDDL* domain, we can then define action variations based on faulty component states. After a failure was detected or a conflicting observation was made, we can diagnose a sequence of executed actions by searching for alternations of the executed history. An alternation will be generated by inserting exogenous actions, that change hardware component states, and replacing subsequent actions with faulty variants, based on the changed hardware states. By integrating *order actions* into the *PDDL* domain to enforce the diagnosis candidate to be an alternated history, we can use the top-k planner K^* to generate k diagnosis candidates. The generated diagnosis candidates consist of one or more hardware components that changed their states, causing subsequent actions to behave differently than expected.

In order to identify the correct diagnosis we used *active diagnosis*, which is the task of selecting suitable sensing actions to exclude diagnosis candidates. To narrow the set of diagnosis candidates in an efficient manner, we select the sensing action with the highest mutual information. This is done by calculating the impact a sensing action has onto the set of diagnosis candidates. Here, we can integrate the probabilities of component state transitions, as modelled in the hardware models, to determine the probabilities of the diagnosis candidates. Based on these probabilities, the *active diagnosis* selects sensing actions that narrow the set of diagnosis candidates in an efficient manner. After excluding as many diagnosis candidates as possible and updating the world model, the agent has information about possible hardware failures, that explain the observation. The agent can then utilize this information to

try to repair the hardware components or to decide to stop operating and wait for human intervention.

We integrated our diagnosis approach into the goal reasoning framework CLIPS Executive and investigated the performance in the RCLL domain. The diagnosis approach gets integrated into the goal lifecycle as the evaluation step of a failed goal and consists of the *PDDL problem generation*, *active diagnosis* and *repair*. Using the Gazebo simulation of RCLL games, we compared the performance of agents with diagnosis with integrated hardware models to agents without diagnosis and to agents with diagnosis but lacking information about the hardware platform.

Integrating hardware models into the history-based diagnosis process led to the correct identification of the underlying hardware failure in most of the cases. This enabled the agent to repair the faulty hardware component or to decide to exclude itself from the production in order to make room for the other, still functional robots. Thus, a team of robots equipped with history-based diagnosis with integrated hardware models was able to achieve significantly more points than a team without diagnosis or with history-based diagnosis but missing information about the hardware.

All in all, this thesis presents a promising approach of diagnosing faults and contradicting observations while taking information about the underlying hardware platform into account, enabling a high-level control agent to attempt to recover or calling for human intervention. There is still room for improvement, especially regarding the time needed for the diagnosis process and the integration of the knowledge gained by the diagnosis process into the decision process of the agent.

6.2 Future Work

It was shown, that the upper bound of the cost and the length of the diagnosed history have a huge influence on the runtime of the planner. Therefore, in other, more complicated scenarios, the runtime of the planner could increase significantly. In order to decrease the time needed to generate diagnosis candidates, we can further restrict the insertion of actions, that changes component states, e.g. by allowing only insertions of component state changes right in front of actions that depend on this component. We can restrict the number of generated diagnosis candidates even further by using diagnosis templates, as presented in Iwan (2002).

Diagnosing action failures by searching for hardware component state changes that can explain misbehaving actions is based on the assumption, that action failures are caused by hardware failures. However, in reality actions may fail randomly sometimes. With integrated hardware models, diagnosing such random action failures may result in wrong accusation of hardware faults. In the worst case, the agent suspect a

hardware failure from which he can not recover and stop operating. For example, the action executor may have additional information about the reason of the action failure. Therefore, integrating further information about action failures in the diagnosis process may enable the agent to deal with this problem.

In this thesis, upon detecting an inconsistency, only the currently executing or last executed plan gets diagnosed. A failure, that was undetected during the execution of a previous plan and results in a failure during the execution of a later plan, cannot be properly diagnosed with a goal reasoning framework, like the CLIPS Executive, where only the current goal gets diagnosed. Extending the diagnosis framework to take all executed plans to this point into account, by appending all executed actions to an overall history, is straightforward. However, this will have a huge impact on the runtime of diagnosis generation and the calculation of the hypothetical world models, since the length of the history to be diagnosed will increase over time. Also, this extension is only applicable for a single agent. As soon as more than one agent operates in the same environment, a failure of one agent may cause the plan of another agent to fail. Extending the history-based diagnosis approach to multi-agent scenarios has to be subject of future work.

The integration into the CLIPS Executive represents uncertainty about the state of an object by reasoning about the non-existence of facts, that describe the property of an object. By that, disjunctive knowledge about the property of an object is lost. Integrating a three-valued logic into the CLIPS Executive will create new possibilities of reasoning about the results of the diagnosis. This could be done by introducing a method to reason about incomplete knowledge into the CLIPS Executive.

Bibliography

D. W. Aha. Goal reasoning: Foundations, emerging applications, and prospects. *AI Magazine*, 39(2), 2018.

H. Aljazzar and S. Leue. K∗: A heuristic search algorithm for finding the k shortest paths. *Artificial Intelligence*, 175(18):2129–2154, 2011.

J. A. Baier and S. A. McIlraith. Planning with temporally extended goals using heuristic search. In *International Conference on Automated Planning and Scheduling*, pages 342–345, 2006.

J. A. Baier, F. Bacchus, and S. A. McIlraith. A heuristic search approach to planning with temporally extended preferences. *Artificial Intelligence*, 173(5):593, 2009.

C. Baral, S. McIlraith, and T. C. Son. Formulating diagnostic problem solving using an action language with narratives and sensing. In *KR*, pages 311–322, 2000.

W. Bridewell and P. Langley. A computational account of everyday abductive inference. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 33, 2011.

F. de Jonge and N. Roos. Plan-execution health repair in a multi-agent system. *The 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group*, 2004.

E. H. Durfee, C. L. Ortiz Jr, M. J. Wolverton, et al. A survey of research in distributed, continual planning. *AI magazine*, 20(4):13–13, 1999.

T. Eiter, E. Erdem, and W. Faber. Diagnosing plan execution discrepancies in a logic-based action framework. Technical report, Citeseer, 2004.

R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

M. Fox and D. Long. Pddl+: Modeling continuous time dependent effects. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, volume 4, page 34, 2002.

M. Fox and D. Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

S. Gspandl, I. Pill, M. Reip, G. Steinbauer, and A. Ferrein. Belief management for high-level robot programs. In *Twenty-Second International Joint Conference on Artificial Intelligence*, volume 22, page 900, 2011.

M. Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

T. Hofmann, N. Limpert, V. Mataré, S. Schönitz, T. Niemueller, A. Ferrein, and G. Lakemeyer. The carologistics robocup logistics team 2018. *RWTH Aachen University and Aachen University of Applied Sciences*, 2018a.

T. Hofmann, V. Mataré, S. Schiffer, A. Ferrein, and G. Lakemeyer. Constraint-based online transformation of abstract plans into executable robot actions. 2018b.

T. Hofmann, N. Limpert, V. Mataré, A. Ferrein, and G. Lakemeyer. The carologistics robocup logistics team 2019. *RWTH Aachen University and Aachen University of Applied Sciences*, 2019.

G. Iwan. History-based diagnosis templates in the framework of the situation calculus. *AI Communications*, 15(1):31–45, 2002.

M. Katz, S. Sohrabi, O. Udrea, and D. Winterer. A novel iterative approach to top-k planning. In *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, pages 340–347. ACM, 1997.

N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2149–2154. IEEE, 2004.

L. Kuhn, B. Price, J. De Kleer, M. Do, and R. Zhou. Pervasive diagnosis: the integration of active diagnosis into production plans. In *International Workshop on Principles of Diagnosis (DX)*. Citeseer, 2008.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL-the planning domain definition language. 1998.

S. A. Mcllraith. Explanatory diagnosis: Conjecturing actions to explain observations. In *Logical Foundations for Cognitive Agents*, pages 155–172. Springer, 1999.

M. Molineaux and D. W. Aha. Continuous explanation generation in a multi-agent domain. Technical report, NAVAL RESEARCH LAB WASHINGTON DC, 2015.

C. Mühlbacher and G. Steinbauer. Active diagnosis for agents with belief management. In *International Workshop on Principles of Diagnosis (DX), Graz, Austria*, 2014.

C. Mühlbacher and G. Steinbauer. Belief management using the action history and consistency-based-diagnosis. In *International Workshop on Principles of Diagnosis (DX), Denver, CO, USA*, 2016a.

C. Mühlbacher and G. Steinbauer. Diagnosis makes the difference for a successful execution of high-level robot control programs. In *Intelligent Autonomous Systems 13*, pages 1119–1132. Springer, 2016b.

T. Niemueller, A. Ferrein, and G. Lakemeyer. A lua-based behavior engine for controlling the humanoid robot nao. In *Robot Soccer World Cup*, pages 240–251. Springer, 2009.

T. Niemueller, A. Ferrein, D. Beck, and G. Lakemeyer. Design principles of the component-based robot software framework fawkes. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 300–311. Springer, 2010.

T. Niemueller, G. Lakemeyer, and S. S. Srinivasa. A generic robot database and its application in fault analysis and performance evaluation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 364–369. IEEE, 2012.

T. Niemueller, G. Lakemeyer, and A. Ferrein. Incremental task-level reasoning in a competitive factory automation scenario. In *AAAI Spring Symposium: Designing Intelligent Robots*, 2013.

T. Niemueller, G. Lakemeyer, and A. Ferrein. The robocup logistics league as a benchmark for planning in robotics. In *WS on planning and robotics (PlanRob) at International Conference on Automated Planning and Scheduling*, 2015.

T. Niemueller, D. Ewert, S. Reuter, A. Ferrein, S. Jeschke, and G. Lakemeyer. Robocup logistics league sponsored by festo: a competitive factory automation testbed. In *Automation, Communication and Cybernetics in Science and Engineering 2015/2016*, pages 605–618. Springer, 2016a.

T. Niemueller, D. Ewert, S. Reuter, A. Ferrein, S. Jeschke, and G. Lakemeyer. Robocup logistics league sponsored by festo: a competitive factory automation testbed. In *Automation, Communication and Cybernetics in Science and Engineering 2015/2016*, pages 605–618. Springer, 2016b.

T. Niemueller, T. Neumann, C. Henke, S. Schönitz, S. Reuter, A. Ferrein, S. Jeschke, and G. Lakemeyer. International harting open source award 2016: Fawkes for the RoboCup Logistics League. In *Robot World Cup*, pages 634–642. Springer, 2016c.

T. Niemueller, T. Hofmann, and G. Lakemeyer. Clips-based execution for PDDL planners. In *Proceedings of the 2nd Workshop on Integrated Planning, Acting, and Execution (ICAPS IntEx)*, Delft, Netherlands, 2018. URL `https://kbsg.rwth-aachen.de/~hofmann/papers/clips-exec-pddl.pdf`.

T. Niemueller, T. Hofmann, and G. Lakemeyer. Goal reasoning in the clips executive for integrated planning and execution. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 754–763, 2019.

D. Poole. Normality and faults in logic-based diagnosis. In *International Joint Conference on Artificial Intelligence*, volume 89, pages 1304–1310. Detroit, MI, 1989a.

D. Poole. Explanation and prediction: an architecture for default and abductive reasoning. *Computational Intelligence*, 5(2):97–110, 1989b.

R. Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1): 57–95, 1987.

S. Richter and M. Westphal. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

M. Roberts, S. Vattam, R. Alford, B. Auslander, J. Karneeb, M. Molineaux, T. Apker, M. Wilson, J. McMahon, and D. W. Aha. Iterative goal refinement for robotics. Technical report, KNEXUS RESEARCH CORP SPRINGFIELD VA, 2014.

N. Roos. Learning-based diagnosis and repair. In B. Verheij and M. Wiering, editors, *Artificial Intelligence*, pages 1–15, Cham, 2018. Springer International Publishing. ISBN 978-3-319-76892-2.

S. Sohrabi, J. A. Baier, and S. A. McIlraith. Diagnosis as planning revisited. In *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*, 2010.

S. Sohrabi, A. V. Riabov, O. Udrea, and O. Hassanzadeh. Finding diverse high-quality plans for hypothesis generation. In *European Conference on Artificial Intelligence*, pages 1581–1582, 2016.

C. Witteveen, N. Roos, R. van der Krogt, and M. de Weerdt. Diagnosis of single and multi-agent plans. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 805–812. ACM, 2005.

R. M. Wygant. Clips—a powerful development and delivery expert system tool. volume 17, pages 546–549. Elsevier, 1989.