COMBINING RUNTIME DIAGNOSIS AND AI-PLANNING IN A MOBILE AUTONOMOUS ROBOT TO ACHIEVE A GRACEFUL DEGRADATION AFTER SOFTWARE FAILURES

Jörg Weber and Franz Wotawa
Institute for Software Technology, Graz University of Technology, Inffeldgasse 16b/2, Graz, Austria

Keywords: Model-based reasoning, Autonomous robots, Diagnosis, Repair, AI planning, Plan monitoring, Capability model.

Abstract: Our past work deals with model-based runtime diagnosis in the software system of a mobile autonomous robot. Unfortunately, as an automated repair of failed software components at runtime is hardly possible, it may happen that failed components must be removed from the control system. In this case, those *capabilities* of the control system which depend on the removed components are lost. This paper focuses on the necessary adaptions of the high-level decision making in order to achieve a graceful degradation. Assuming that those decisions are made by an AI-planning system, we propose extensions which enable such a system to generate only plans which can be executed and monitored despite the lost capabilities. Among others, we propose an abstract model of software capabilities, and we show how to dynamically determine those capabilities which are required for monitoring a plan.

1 INTRODUCTION

Enabling mobile robots to operate autonomously in unknown, unpredictable and often harsh environments requires that the robots are equipped with onboard diagnosis and repair/reconfiguration abilities. Hardware may be damaged by unexpected interactions with a rough environment, or it may fail due to internal faults. Complex software systems, on the other hand, cannot be exhaustively tested and thus may contain bugs which may lead to runtime failures. The robot should be able to autonomously detect and locate failures of hardware or software components, and it should also be able to properly deal with runtime failures. This means that the robot should attempt to repair/reconfigure the system in order to restore the full capabilities of the system. If this is not possible, then the robot should try to achieve a graceful degradation, i.e., we want the robot to continue its mission, maybe at a lower performance.

Whereas most of the past research on runtime diagnosis and repair/reconfiguration in autonomous robots has focused on hardware aspects, our work has tackled the issue of software failures at runtime (Steinbauer and Wotawa, 2005; Weber, 2009). Typical failures are software crashes or deadlocks. We assume that the control system is composed of basically independent components. We proposed to detect failures by monitoring *properties* of the control system at runtime, and we apply model-based diagnosis techniques (Reiter, 1987; de Kleer and Williams, 1987) to locate the failed components.

We proposed to restart failed software components, hoping that they work correctly afterwards. However, as this is not a real repair which fixes the root cause of the failure, it may happen that the restarted component immediately fails again. In this case, the failed component is aborted and permanently removed from the control system. This leads to a degradation of the *capabilities* (functionalities) of the control system.

In this paper we focus on the adaption of the high-level decision making after the loss of software capabilities. Of course, if the lost capabilities are vital for the operation of the robot, then the robot is no longer able to do something meaningful. However, if non-vital components are affected, the robot may still be able to perform useful tasks, maybe in a degraded mode. It may happen that the planning system can find an alternative plan which achieves the same goal, or that the original goal can no longer be accomplished, but there are other (maybe less useful) goals which can still be achieved.

We assume that the high-level decision making

This research was partially funded by the Austrian Science Fund (FWF) under grant P20199-N15.

Weber J. and Wotawa F. (2010).
COMBINING RUNTIME DIAGNOSIS AND AI-PLANNING IN A MOBILE AUTONOMOUS ROBOT TO ACHIEVE A GRACEFUL DEGRADATION AFTER SOFTWARE FAILURES.
In *Proceedings of the 2nd International Conference on Agents and Artificial Intelligence - Artificial Intelligence*, pages 127-134
DOI: 10.5220/0002752101270134

of the robot is done by a classical AI-planning system which performs a closed-loop control of the robot. The planning is based on a representation language similar to the well-known STRIPS representation (Fikes and Nilsson, 1972). We propose to augment the planning system with an abstract model of the control system's capabilities. This model is utilized to derive the currently available capabilities, based on the diagnostic results provided by the runtime diagnosis and repair system. Moreover, we propose to enhance the action specifications with *capability preconditions*, stating which capabilities are required for the low-level execution of the action. Hence, based on the knowledge about available capabilities, the planning system is now able to generate plans whose actions can still be executed.

In Sec. 2 we give some background information concerning our past work on model-based runtime diagnosis and repair in a robot control system. We also outline a control system which serves as running example throughout this paper. Thereafter, in Sec. 3 we propose an approach which ensures that the AI-planning system computes only plans whose actions are supported by the currently available capabilities. In Sec. 4 we present a case study, executed on a real robot control system, which shows that our proposal can achieve a graceful degradation of the robot's behavior after component failures. Afterwards, in Sec. 5 we discuss the issue that the *plan monitoring* also relies on capabilities of the control system. In Sec. 5 and Sec. 6 we propose two alternative solutions to this problem with the aim that the generated plans are also monitorable. Sec. 7 contains a discussion of related research and open issues.

The utilization of diagnostic results in the deliberative layer of mobile autonomous systems has gained little attention in the past. In particular, we are not aware of any work from other researchers which addresses AI-planning with degraded capabilities of the software system.

2 BACKGROUND: DIAGNOSIS AND REPAIR IN A ROBOT CONTROL SYSTEM

In this section we briefly outline our past work on model-based runtime diagnosis and repair in the software system of mobile autonomous robots (Steinbauer and Wotawa, 2005; Weber, 2009). The goal of this work is to detect and locate failed software components at runtime without any human intervention. Our approach mainly aims at severe failures like software crashes, "freezes" (e.g., non-terminating loops or deadlocks), or major computing errors. We define components as (largely) independent binary modules

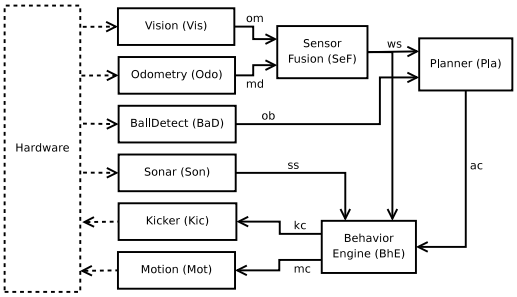

Figure 1: Architecture of a control system for an autonomous soccer robot. Connections depict data flows.

which have no shared memory and which communicate by exchanging events over communication channels which, in the ideal case, completely decouple the components.

Figure 1 depicts an architectural view on a subsystem of a control system of an autonomous soccer robot. This example system will be used throughout this paper. The components *Vision*, *Odometry*, *BallDetect*, and *Sonar* process sensor data. *SensorFusion* computes a continuous world model, containing (among others) the estimated positions of environment objects. *Sonar* supplies data which allows for the detection of obstacles, and *BallDetect*, relying on an infrared sensor, states whether or not the robot possesses the ball (this holds when the ball is between the grabbers of the robot s.t. the robot can kick the ball). The *Planner* is an AI-planning system which represents the deliberative layer of the system. It abstracts its continuous inputs using landmarks, and the resulting qualitative world state is stored as a set of atomic sentences in an internal knowledge base. The output of the planning system are high-level actions whose low-level execution is performed by component *BehaviorEngine*. Finally, *Kicker* and *Motion* directly access the actuators of the robot, namely the kicking device and the driving unit, respectively.

For detecting software failures at runtime, we proposed to assign *properties* to connections. Properties capture invariant conditions which must hold in the system. Properties relate to sequences of events on one or more connections, and at runtime they are continuously evaluated by executable entities called *monitoring rules*. Monitoring rules are responsible for detecting property violations.

When a failure is detected by monitoring rules, we employ the *consistency-based diagnosis* paradigm (Reiter, 1987; de Kleer and Williams, 1987) for the localization of the failed components. I.e., a logical system description *SD*, a set of observations *OBS*, and a set of components *COMP* are used for the computation of diagnoses. *SD* is a set of first-order sentences which is intended to describe the correct system be-

havior using the literal $\neg AB(c)$ (with $c \in COMP$) which denotes "not abnormal". We rely on Reiter's definition of a diagnosis (Reiter, 1987):

Definition 1 *A diagnosis Δ is a set of components ($\Delta \subseteq COMP$) s.t.*

$$SD \cup OBS \cup \{\neg AB(c) | c \in COMP \setminus \Delta\} \cup \{AB(c) | c \in \Delta\}$$

is consistent. A diagnosis is (subset-)minimal iff no proper subset of it is a diagnosis.

In our approach, *SD* only captures the logical dependencies between properties, whereas the property conditions are not represented in *SD*. We illustrate this in the following intuitive example, which considers only the subsystem with the components $\{Vis, Odo, SeF\}$:

Both components *Vis* and *Odo* are supposed to periodically produce new events at their output connections *om* and *md*, respectively, and *SeF* must produce a new output for every incoming event. Hence, we introduced the property type $\xi.eo$, where ξ is a connection and *eo* the name of the property (*eo*.."event occurs"), which means that at least one event must occur within a certain timespan at ξ. We assigned instances of this property type to the connections *om*, *md* and *ws*. *SD* expresses the dependencies between those three properties, using the atomic sentence $ok(\varphi)$ to state that property φ holds:

$$\neg AB(Vis) \rightarrow ok(om.eo)$$
$$\neg AB(Odo) \rightarrow ok(md.eo)$$
$$\neg AB(SeF) \wedge \big(ok(om.eo) \vee ok(md.eo)\big) \rightarrow ok(ws.eo)$$

We assume here that only the property *ws.eo* can be monitored at runtime. Now suppose that *ws.eo* is violated, i.e., the observations are $OBS = \{\neg ok(ws.eo)\}$. By applying a diagnosis algorithm like Reiter's Hitting Set algorithm (Reiter, 1987) we obtain two minimal diagnoses: $\Delta_1 = \{SeF\}$, $\Delta_2 = \{Vis, Odo\}$. I.e., either *SeF* has failed, or both *Vis* and *Odo* have failed (multiple failure).

As a real repair, which fixes the root causes of failures by correcting the bugs in the source code, is hardly possible at runtime, we proposed to restart failed components, hoping that they do not fail again. In cases when the diagnostic results are not unique, i.e., when there are several minimal diagnoses, we restart every component which occurs in one of the minimal diagnoses. Although this method often works in practice, it is not always successful: restarted components may fail immediately again due to the same root cause which has not been fixed.

When the repair is not successful, our diagnosis and repair system removes this component from the control system; i.e., the component is aborted and not restarted afterwards, but the robot continuous to operate, if possible. The runtime diagnosis system then continuous to monitor the system, after certain adaptions which we do not discuss here (see (Weber, 2009)). Note that software components can only be removed if they are loosely coupled with the rest of the system.

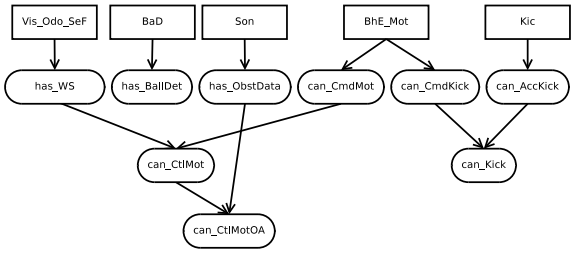

Figure 2: A capability graph. Rectangles are components, and rounded rectangles are capabilities (6 basic and 3 composed capabilities).

3 DEGRADED CAPABILITIES, PLANNING

Let $COMP_{fail}$ be the set of all those components which have so far been removed from the original control system. Then we use

$$COMP_{act} \stackrel{\text{def}}{=} COMP \setminus COMP_{fail}$$

to denote those components which are currently active (alive) and, according to the diagnosis system, correctly working. Our approach demands that the diagnosis and repair system reports this set to the planning system at least after each repair session which has removed components. Note that the handling of ambiguous diagnoses is done by the diagnosis system; i.e., if the diagnosis system is not able to come up with a unique diagnosis, then it might be possible that components of multiple diagnoses are removed from the control system. The planning system always receives a single set $COMP_{fail}$, which may contain several components which have been removed.

The planning system then employs a model of the abstract capabilities of the control system to infer the remaining capabilities. The proposed model is visualized by the *capability graph*. A capability graph for the system in Fig. 1 is depicted in Fig. 2. Note that, for brevity, we introduced a component *Vis_Odo_SeF* which subsumes the three components *Vis*, *Odo* and *SeF* in Fig. 2, and the same applies analogously to *BhE_Mot*. Moreover, the component *Pla* is not included, since we assume that the planning system has not failed.

A capability graph *CG* is a directed acyclic graph (DAG) with the following attributes:

- For each component $c \in COMP$ there is exactly one source node (a node without incoming edges) in *CG*. It is called *component node*.

- All other nodes represent either *basic capabilities* or *composed capabilities*.
- A basic capability has exactly one direct predecessor in *CG* which must be a component node. The involved edges indicate which capabilities are provided by the components. E.g., both *can_CmdMot* and *can_CmdKick* are provided by *BhE_Mot*.
- A composed capability has one or more direct predecessors which represent (basic or composed) capabilities. E.g., *can_CtlMot* is composed of *has_WS* and *can_CmdMot*, meaning that *can_CtlMot* is available if both *has_WS* and *can_CmdMot* are available.

Note that we distinguish between *sensing capabilities* (prefix *has_X*) and *acting capabilities* (prefix *can_X*). The component *Vis_Odo_SeF* provides a continuous world state (*has_WS*), *BaD* provides ball detection, *Son* provides obstacle data, *BhE_Mot* provides the capabilities to command motions as well as kicks, and *Kic* can access the kicking device. Capability *can_CtlMot* captures the following fact: in order to *control* a motion (in the sense of control theory), the system needs not only to be able to command motions (*can_CmdMot*), but also to receive sensor feedback indicating how the world changes (*has_WS*). Moreover, if the system also has obstacle data, then the motion control is augmented with a reactive obstacle avoidance (*can_CtlMotOA*).

The semantics of a capability graph *CG* is captured by the *software capabilities description (SCD)*. It is a set of Horn clauses using the predicate $active(c)$ which states that $c \in COMP$ is active and correctly working, and $av(b)$ indicates that capability b is currently available. Moreover, let *CAP* denote the set of all capabilities used in *CG*. *SCD* is automatically generated from *CG* as follows:

For every capability $b \in CAP$:

- if b is a basic capability provided by component c: add $active(c) \rightarrow av(b)$ to *SCD*.
- otherwise, b is composed of the capabilities $b_1, \ldots, b_m$ ($m \geq 1$):
  add $av(b_1) \wedge \ldots \wedge av(b_m) \rightarrow av(b)$ to *SCD*.

In our example, *SCD* contains the following sentences:

$$active(Vis\_Odo\_SeF) \rightarrow av(has\_WS)$$
$$active(BhE\_Mot) \rightarrow av(can\_CmdMot)$$
$$active(BhE\_Mot) \rightarrow av(can\_CmdKick)$$
$$av(has\_WS) \wedge av(can\_CmdMot) \rightarrow av(can\_CtlMot)$$
$$\ldots$$

With *AVCAP* we denote the set of currently available capabilities, and $\overline{AVCAP}$ comprises those capabilities which are presumably not available. Based on the set of remaining components $COMP_{act}$, these sets are computed as follows:

$$ACTCOMP \stackrel{\text{def}}{=} \{active(c) \mid c \in COMP_{act}\}$$
$$AVCAP \stackrel{\text{def}}{=} \{av(b) \mid b \in CAP \text{ and } ACTCOMP \cup SCD \models av(b)\}$$
$$\overline{AVCAP} \stackrel{\text{def}}{=} \{\neg av(b) \mid b \in CAP \text{ and } ACTCOMP \cup SCD \not\models av(b)\}$$

Clearly, *AVCAP* and $\overline{AVCAP}$ can be efficiently computed by applying a simple Horn clause forward chaining algorithm.

The important point is that the robot's actions have certain requirements concerning the available capabilities: the low-level behaviors, which correspond to a specific high-level action $\mathcal{A}$, can only be executed if certain sensing and acting capabilities are available. In STRIPS (Fikes and Nilsson, 1972) and in similar representation languages for classical AI-planning, each action $\mathcal{A}$ is specified by a precondition $pre(\mathcal{A})$ and an effect $eff(\mathcal{A})$. Note that we the planner regards actions as deterministic; the handling of unexpected action outcomes and environment changes is left to the plan executor which monitors the plans.

By integrating the required capabilities into the action precondition we can ensure that a plan comprises only actions which are supported by the currently available capabilities. For this purpose we propose to split every action precondition into two parts:

$$pre(\mathcal{A}) \stackrel{\text{def}}{=} pre_E(\mathcal{A}) \wedge pre_C(\mathcal{A})$$

where $pre_E(\mathcal{A})$ is the *environment precondition*, which mainly contains conditions related to the physical environment, and $pre_C(\mathcal{A})$ is the *capability precondition* which relates to the state of health of the robot. $pre_C(\mathcal{A})$ is a conjunction of (positive or negative) literals of the form $(\neg)av(b)$.

An action $\mathcal{A}$ can only be executed when $pre_C(\mathcal{A})$ is fulfilled, i.e., when the following holds:

$$AVCAP \cup \overline{AVCAP} \models pre_C(\mathcal{A})$$

Examples are presented in the subsequent section. Note that if $AVCAP \cup \overline{AVCAP}$ is added to the logical state descriptions, a classical STRIPS-like planner can be used, as the splitting of the precondition into two parts is irrelevant for the planning algorithm.

4 CASE STUDY: FINDING ALTERNATIVE PLANS AFTER SOFTWARE FAILURES

In this section we present a case study which we conducted on a real robot control system whose (simplified) architecture was depicted in Fig. 1. The control system was running on a PC and interacted with a

Block(obj$_1$, obj$_2$):
pre$_E$: $-$
pre$_C$: $av(can\_CtlMotOA)$
eff: $blocking(obj_1, obj_2) \wedge \neg possBall$
Goto(obj):
pre$_E$: $\neg inReach(obj)$
pre$_C$: $av(can\_CtlMotOA)$
eff: $inReach(obj) \wedge \neg possBall$
GrabBall:
pre$_E$: $\neg possBall \wedge inReach(Ball)$
pre$_C$: $av(can\_CtlMotOA)$
eff: $possBall$
DribbleTo(obj):
pre$_E$: $possBall$
pre$_C$: $av(can\_CtlMotOA)$
eff: $inKickPos(obj)$
KickBallTo(obj):
pre$_E$: $possBall \wedge inKickPos(obj)$
pre$_C$: $av(can\_CtlMotOA) \wedge av(can\_Kick)$
eff: $\neg possBall \wedge isAt(Ball, obj)$

Figure 3: STRIPS-like action schemas. $Block(obj_1, obj_2)$ lets the robot move in-between those two objects. The other actions have the expected meanings.

soccer robot simulator which simulates the hardware and the physical environment. We implemented the runtime diagnosis and repair system which we briefly described in Section 2.

The planning system uses a planning algorithm which supports a STRIPS-like language. The diagnosis system periodically transmits the set $COMP_{act}$ to the AI-planning system. Moreover, each time the diagnosis system detects a failure, it notifies the planning system, which then aborts the current plan and the robot becomes idle (i.e., it enters a safe state). After the repair is finished, the planning system is notified again, and re-planning is performed. The diagnosis system continuous to monitor the control system after certain adaptions to the system model in *SD* which we do not describe here. More details concerning this particular robot control system can be found in (Weber, 2009).

In this case study we simulated the crashes of three different components at three different times; i.e., there were three consecutive single failures, all of which led to a correct and unique single-fault diagnosis. In all of these three cases, the injected fault also prevented a successful restart of the component, hence the restarted components failed again during startup and thus were automatically removed from the control system. I.e., each failure led to a further degradation of the control system's capabilities.

Figure 3 depicts those actions which are relevant for this case study. An interesting insight is that it may be desirable to introduce additional actions with lower capability requirements in order to achieve a graceful degradation. All actions in Fig. 3 (indirectly) require the capability *has_ObstData*; i.e., their execution relies on the existance of an obstacle avoidance (OA). Hence, if the component *Son* fails, then none of those actions can be executed anymore. We tackled this problem by adding new actions which achieve a similar effect as the already existing ones, albeit with lower efficiency: for each of the actions in Fig. 3 we introduced corresponding actions which are performed at a lower motion speed and thus can be executed without obstacle avoidance, as the likelihood of physical damage in case of a collision is much smaller. These new actions have the suffix *_slow*, e.g:

Block_slow(obj$_1$, obj$_2$):
pre$_E$: ... [as in Fig. 3]
pre$_C$: $av(can\_ExecMotBeh) \wedge \neg av(can\_ExecMotBehOA)$
eff: ... [as in Fig. 3]

In this case study, initially all components were active and thus all capabilities were availble. The planning system selected the planning goal

$$G_1 = isAt(Ball, OppGoal)$$

, i.e., the ball should be in the opponent goal. The following plan was generated:

$$P_1 = \langle \texttt{Goto}(\texttt{Ball}), \texttt{GrabBall}, \texttt{DribbleTo}(\texttt{OppGoal}), \texttt{KickBallTo}(\texttt{OppGoal}) \rangle$$

During the execution of the first action `Goto(Ball)` we triggered the failure of component *Son*. After a failed restart, *Son* was removed from the system: $COMP_{fail} = \{Son\}$ and so $\overline{AVCAP} = \{\neg av(has\_ObstData), \neg av(can\_CtlMotOA)\}$.

The planning system performed re-planning, and an alternative plan was found for the same goal G_1:

$$P_2 = \langle \texttt{Goto\_slow}(\texttt{Ball}), \texttt{GrabBall\_slow}, \texttt{DribbleTo\_slow}(\texttt{OppGoal}), \texttt{KickBallTo\_slow}(\texttt{OppGoal}) \rangle$$

During the execution of `Goto_slow(Ball)`, component *Kic* failed, leading to $COMP_{fail} = \{Son, Kic\}$ and $\overline{AVCAP} = \{\neg av(has\_ObstData), \neg av(can\_CtlMotOA), \neg av(can\_AccKick), \neg av(can\_Kick)\}$.

Again, the planning system selected the planning goal G_1, but this time no plan was found which could achieve G_1, because the loss of the capability *can_Kick* prevents that the ball can be kicked into the opponent goal. Hence, another planning goal (which has a lower utility) was selected:

$$G_2 = blocking(Ball, OwnGoal)$$

and the corresponding plan was

$$P_3 = \langle \texttt{Block\_slow}(\texttt{Ball}, \texttt{OwnGoal}) \rangle$$

I.e., the robot should act as a defender by blocking the area between the ball and the own goal. During the execution of this action, component

SeF failed, and so $\overline{AVCAP} = \{\ldots, \neg av(can\_CtlMot), \neg av(can\_CtlMotOA)\}$. It can be seen that vital capabilities had been lost, and so none of the predefined planning goals could be accomplished anymore.

Nevertheless, this case study shows that it may be possible to achieve the same planning goals despite the loss of capabilities, although the performance (e.g., the motion speed) may decline. Moreover, if it is no longer possible to pursue the originally selected goal, then it can be useful to choose other goals, which can still be accomplished, in order to achieve a graceful degradation.

5 PLAN MONITORING

A capability precondition $pre_C(\mathcal{A})$ captures the capabilities which are required for the execution of the low-level behaviors which correspond to an action $\mathcal{A}$. However, an important observation is that even if $pre_C(\mathcal{A})$ holds, this does not necessarily imply that this plan can also be monitored by the plan executor, because the monitoring of an action $\mathcal{A}$ may require sensing capabilities which are not needed for the execution of the low-level behaviors.

As example we consider action `GrabBall` (Fig. 3). This action moves the robot to the ball with the intended effect that the robot possesses the ball afterwards (i.e., *possBall* should become true, which happens when an infrared sensor detects that the ball is between the grabbers of the robot). The low-level execution of this action does *not* require a ball detection capability (*has_WS* is sufficient); however, the truth value of the effect *possBall* can only be evaluated if *has_BallDet* is available. In other words, without this capability the plan executor can never detect when this action is finished.

It follows that we should seek plans which can also be monitored wrt the currently available capabilities. This particularly concerns those predicates used by the planning system which capture perceptions about the physical environment and whose (truth value) evaluation thus relies on sensing capabilities. We introduce a function $\gamma(p)$ which returns for each predicate p the (possibly empty) set of those sensing capabilities which are needed for the evaluation of p (i.e., $\gamma(p) \subseteq CAP$). Furthermore, for any first-order sentence Φ, let $\mathcal{P}(\Phi)$ denote the set of all predicates occurring in Φ, and we define:

$$\Gamma(\Phi) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathcal{P}(\Phi)} \left[\bigwedge_{b \in \gamma(p)} av(b) \right]$$

I.e., $\Gamma(\Phi)$ states all sensing capabilities required for the evaluation of the predicates in Φ. Our aim to create only plans which can also be monitored wrt the available capabilities can be achieved by augmenting each action precondition $pre(\mathcal{A})$ with a *monitoring precondition* $pre_M(\mathcal{A})$:

$$pre(\mathcal{A}) \stackrel{\text{def}}{=} pre_E(\mathcal{A}) \wedge pre_C(\mathcal{A}) \wedge pre_M(\mathcal{A})$$

$$\text{with:} \quad pre_M(\mathcal{A}) \stackrel{\text{def}}{=} \Gamma\big(pre_E(\mathcal{A})\big) \wedge \Gamma\big(eff(\mathcal{A})\big)$$

For example:

$\gamma(possBall) = \{has\_BallDet\}$
$\gamma(inReach) = \{has\_WS\}$
$pre_M(GrabBall) = av(has\_BallDet) \wedge av(has\_WS)$

This approach is simple and also efficient in the sense that it has a low impact on the planning complexity. However, it also has the severe drawback that it may be too restrictive, because it takes every effect of an action into account, although some effects may be irrelevant in the current context and thus do not need to be monitored:

E.g., let us consider the action `Goto(obj)`. One effect of this action is $\neg possBall$; i.e., even if the robot possesses the ball when this action is started, the ball will usually be lost during the movement (in contrast to `DribbleTo`). The monitoring precondition is $pre_M\big(\texttt{Goto(obj)}\big) = av(has\_BallDet) \wedge av(has\_WS)$. Now suppose that component *BaD* fails and so *has_BallDet* is lost. This means that a planning goal $G = inReach(X)$ can no longer be achieved, since $pre_M\big(\texttt{Goto(obj)}\big)$ is not fulfilled. Clearly, this is an undesired result, as `Goto(X)` does certainly not need a ball detection to achieve $inReach(X)$, and the action effect $\neg possBall$ is irrelevant here.

One attempt to resolve this issue would be to define $\neg possBall$ as a *side-effect* of `Goto(obj)`, meaning that this part of the effect does not need to be monitored. However, a static separation into primary-effects and side-effects is problematic in general, because the relevance of effects often depends on the context, i.e., it depends on the plan which contains the action and on the goal to achieve. E.g., it may well be the case that a subsequent action in a plan requires that $\neg possBall$ holds. Notice that side-effects have been used before in AI-planning, but mainly for reducing the search costs; e.g., see (Fink and Yang, 1993).

In the following section we propose an alternative approach which relies on a plan execution technique to ensure the monitorability of created plans.

6 DYNAMICALLY DETERMINE THE MONITORABILITY OF A PLAN

We propose to dynamically check the monitorability of entire plans rather than only considering single actions regardless of the context. This has the advantage that action effects, which are irrelevant in a given

Table 1: Kernels for the plan $\langle Goto(X)\rangle$ which achieves the planning goal $G = inReach(X)$.

	Kernel / Action:	$\Gamma(K_i)$**:**
K_1	$\neg inReach(X) \wedge av(can\_CtlMotOA)$	$av(has\_WS)$
$\mathcal{A}_1$	`Goto(X)`	
K_2	$inReach(X)$	$av(has\_WS)$

plan, are ignored. For this purpose we employ *kernel models*, an approved plan execution method that was introduced for the monitoring of STRIPS plans (Fikes et al., 1972). In the following we provide a brief introduction to kernels.

For a STRIPS plan $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, the corresponding kernels are $K_1, \ldots, K_{n+1}$. Table 1 depicts the kernel for a very simple plan containing a single action. A kernel K_i occurs immediately before $\mathcal{A}_i$ and K_{i+1} immediately after $\mathcal{A}_i$. A kernel K_i is a conjunction of literals, and it has the property that, if K_i holds in the current world state, then the sequence of actions $\langle \mathcal{A}_i, \ldots, \mathcal{A}_n \rangle$ can be executed and will achieve the planning goal, provided that the action executions lead to the desired results as indicated in the effects. The kernels are usually computed by the plan executor before the execution of the plan starts. The computation of kernels is done backwards, i.e., it starts with the last kernel and then moves forward:

1. K_{n+1} is equal to the STRIPS goal condition G.
2. For every other kernel K_i with $1 \leq i \leq n$: K_i is a conjunction of those literals which are contained in the precondition of $\mathcal{A}_i$ plus those literals in K_{i+1} which are not part of the effect of $\mathcal{A}_i$.

The plan execution based on the kernels works as follows: At each execution step, the plan executor iterates backwards through the kernels (i.e., in the order $K_{n+1}, \ldots, K_1$) and checks for each kernel K_i if it is satisfied in the current world state. As soon as such a kernel K_i is found, the action $\mathcal{A}_i$ is executed. If no kernel is satisfied, then replanning is necessary. A plan is finished when K_{n+1} is true in the current world state.

An important property of this monitoring approach is that those effects of an action $\mathcal{A}_i$ which are required neither for preconditions of subsequent actions $(\mathcal{A}_{i+1}, \ldots, \mathcal{A}_n)$ nor for the planning goal G are *not* added to the kernels. E.g., in Tbl. 1 we can see that the effect $\neg possBall$ of `Goto(obj)` does not occur in any kernel, but it would be added if a subsequent action required $\neg possBall$ in its precondition.

Hence, by relying on kernels we can resolve the issue that only relevant action effects should be considered when determining the monitorability of a plan. For a given plan $P = \langle \mathcal{A}_1, \ldots, \mathcal{A}_n \rangle$, which achieves a planning goal G, and the corresponding kernels $K_1, \ldots, K_{n+1}$, we say that P is *monitorable wrt the available software capabilities* iff the following holds:

$$AVCAP \models \bigwedge_{i=1,..,n+1} \Gamma(K_i)$$

For example, the plan $\langle Goto(X)\rangle$ for the goal $G = inReach(X)$ (Tbl. 1) is monitorable iff has_WS is available.

The monitorability can be checked after the entire plan has been computed. However, if we want to achieve that only monitorable plans are computed, then a better approach is to check the monitorability of (sub-)plans already *during* the planning. I.e., the computation of the kernels can be interleaved with the planning, and plans which are not monitorable according to our definition above can already be discarded during planning.

As the generation of kernels is done backwards (i.e., starting with the planning goal), it will often be straightforward to efficiently integrate the kernel generation with planning algorithms which perform a backwards search through the state-space. In fact, it can be easily shown that the well-known *regression planner* as described in (Weld, 1994) already computes the kernels as a by-product of the search.

7 DISCUSSION AND RELATED RESEARCH

The main goal behind our work is to enable an autonomous robot to perform meaningful tasks despite the failures of software components which cannot be repaired. The contribution of this paper is threefold:

First, we proposed an approach which ensures that the AI-planning system of the robot computes only plans whose actions can still be executed despite the degraded *capabilities* of the partly failed software system. For this purpose we introduced an abstract model of the capabilities of the control system, and we formally described how to derive the currently available capabilities. This self-awareness is utilized by enhancing action specifications with *capability preconditions*. The main advantages of our approach are that the available capabilities can be efficiently computed, and that classical planning algorithms can be applied without modifications.

Second, we presented a case study which we executed on a real robot control system. It shows that our approach can achieve a graceful degradation of the robot's behavior after the loss of capabilities. Sometimes the same planning goals can be accomplished by alternative actions, which may have a lower performance. In other cases the original goals can no longer be accomplished, but the robot may continue to operate after the selection of alternative goals, even though the new goals may have a lower utility.

Third, we discussed the issue that the *plan monitoring* also requires the availability of certain ca-

pabilities. We proposed two approaches with the aim to generate only plans which are also *monitorable*. The first approach enhances action specifications with *monitoring preconditions*. The second approach, which provides more flexibility, is to dynamically determine the monitorability of entire plans. We have also proposed to integrate the monitorability checks into the planning algorithm. Such an interleaved approach achieves that plans which are not monitorable are already discarded at an early stage during the planning process.

So far, the issue of utilizing diagnostic results in high-level planning in autonomous systems has gained little attention among researchers. In particular, we are not aware of any work from other researchers which has addressed the issue of AI-planning with degraded software capabilities in autonomous systems. The Remote Agent architecture (Williams et al., 1998) employs model-based diagnosis methods for the detection and localization of hardware failures. If such a failure cannot be handled locally, the degraded capabilities are reported to the planning system and replanning is performed. However, the scope of that work is quite different from ours: it deals with hardware rather than software, and it does not specifically address the modelling of capabilities or plan monitoring issues.

Model-based diagnosis techniques can also be employed for the execution monitoring of plans, see, e.g., (Roos and Witteveen, 2005). The authors of (Micalizio and Torasso, 2007) use model-based diagnosis techniques to monitor the execution of multi-agent plans. One aim of this work is to provide the global planner/scheduler with the assessed status of robots and the explanations of failures. However, a deeper discussion of the planning and plan monitoring issues in this context is not provided.

The practical applicability of our approach to existing robot control systems is limited by the fact that most components in such systems are vital, i.e., when a vital component fails, then the robot is not able to operate anyway. In our example system in Fig. 1, only 3 out of 9 components are non-vital, namely *BaD*, *Son* and *Kic*. This indicates that the architectural design of robot control systems should accomodate runtime reconfiguration; in particular, the number of components should be higher, and non-vital functionality should be encapsulated in separate components in order to achieve that there are many components with non-vital functionality.

An interesting direction for future research is the question of how to extend our approach to hardware failures. The modelling of hardware capabilities could be challenging, in particular because hardware components may degrade gradually. E.g., the driving unit may no longer be able to perform specific movements after the breakdown of a single wheel.

Another assumption behind our work is that the selected set of leading diagnoses comprises the real fault, hence removing all components in those diagnoses switches the system to a safe state where all remaining components work correctly. This may not be the case in practice, thus a careful selection of the leading diagnoses is necessary.

REFERENCES

de Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130.

Fikes, R. E., Hart, P. E., and Nilsson, N. J. (1972). Learning and executing generalized robot plans. *Artificial Intelligence*, 3(4):251–288.

Fikes, R. E. and Nilsson, N. J. (1972). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3-4):189–208.

Fink, E. and Yang, Q. (1993). Characterizing and automatically finding primary effects in planning. In *IJCAI*, pages 1374–1379.

Micalizio, R. and Torasso, P. (2007). On-line monitoring of plan execution: A distributed approach. *Knowledge-Based Systems*, 20(2):134–142.

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

Roos, N. and Witteveen, C. (2005). Diagnosis of plans and agents. In *Proceedings of the 4th International Central and Eastern European Conference on Multi-Agent Systems (CEEMAS)*.

Steinbauer, G. and Wotawa, F. (2005). Detecting and locating faults in the control software of autonomous mobile robots. In *Proc. IJCAI*, pages 1742–1743, Edinburgh, UK.

Weber, J. (2009). *Model-based Runtime Diagnosis of the Control Software of Mobile Autonomous Robots*. PhD thesis, Institute for Software Technology, Graz University of Technology, Austria.

Weld, D. S. (1994). An introduction to least commitment planning. *AI Magazine*, 15(4):27–61.

Williams, B. C., Nayak, P., and Muscettola, N. (1998). Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48.