AI-Planning in a Mobile Autonomous Robot with Degraded Software Capabilities [*]

Jörg Weber and **Franz Wotawa**
Technische Universität Graz
Institute for Software Technology
8010 Graz, Inffeldgasse 16b/2, Austria
Tel: +43 316 873 5723, Fax: +43 316 873 5706
{jweber,wotawa}@ist.tugraz.at

Abstract

Experience has shown that a major obstacle towards real autonomy of mobile robots is the occurrence of unexpected faults at runtime. While past research has mainly focused on the hardware, we have developed methods for the localization and repair of faulty software components at runtime. However, as it is often not possible to autonomously repair failed components, the deliberative layer of the control system should be aware of the lost capabilities of the system and adapt its decision-making. In this paper we present an AI-planning system for an autonomous soccer robot. We model the abstract capabilities of the control system, and we show how the planning system infers the available capabilities from the results obtained by the runtime diagnosis. We augment action preconditions with capability requirements, and we propose a method which allows to dynamically determine the sensing capabilities required for monitoring of plans.

Introduction

Enabling intelligent mobile systems to act autonomously in unknown and uncertain environments requires not only sensing and planning abilities, but also the awareness of the system's capabilities. The system capabilities may degrade during a mission due to the failure of hardware or software components. Hardware may suffer damages from unexpected interactions with a rough environment, or it may fail due to internal faults. Software failures are caused by bugs.

Hence, a crucial step towards real autonomy is the development of runtime diagnosis and repair/reconfiguration techniques. While past research has mainly focused on hardware aspects, in particular the diagnosis of sensor and actuator failures in mobile autonomous robots, we have tackled the issue of diagnosing software failures at runtime (Peischl, Weber, & Wotawa 2006; Weber & Wotawa 2007). We assume that the control system is decomposed into independent components (modules), which fail independently and which can be (re-)started separately. Components which have failed, e.g. due to a crash or a deadlock, are restarted in the hope that they work correctly afterwards. However, since a real repair of software is not possible at runtime, it often happens that components fail permanently, which leads to a persistent degradation of the capabilities of the control system.

It is highly desirable that an autonomous system has a certain degree of robustness against the loss of hardware and software capabilities. Obviously, this includes that the deliberative layer of the control system should be able to adapt its decision-making. It must be aware of the remaining capabilities, and it must be able to determine which actions and plans can still be executed. While some hardware and software components are vital, i.e., their failure prevents the system from doing anything meaningful, other components are only required for specific actions. For example, when a soccer robot loses the capability to kick the ball, it is still able to interfere with an opponent robot by blocking the line between the own goal and the opponent.

In this paper we describe how an AI-planning system can be enhanced with the ability to adapt its decision-making to degraded software capabilities. We integrate the output from a runtime diagnosis and repair engine into a planning system which is responsible for the closed-loop control of an autonomous soccer robot. The planning system uses a representation language similar to STRIPS (Fikes & Nilsson 1972). Relying on diagnostic data indicating which components are available and work correctly, it utilizes a model of abstract capabilities of the control system to infer the remaining capabilities. Furthermore, we augment action preconditions with capability requirements. This allows the planning system to determine which actions can be safely executed wrt the remaining capabilities. Finally, we discuss the issue that the monitoring and execution of plans require certain sensing capabilities, and we propose an extension to the PLANEX (Fikes, Hart, & Nilsson 1972) monitoring technique which enables the executor to dynamically detect whether or not the required sensing capabilities are available.

The integration of diagnosis results in the deliberative layer of mobile autonomous systems has gained little attention in past research. In particular, we are not aware of any previous work which addresses AI-planning with degraded capabilities of the control system. Even though we use robotic soccer as example domain, the proposed concepts are general and thus can be applied in other contexts as well. Furthermore, as we will discuss in the last section, our work can also be extended to hardware failures.

[*]This research has been funded in part by the Austrian Science Fund (FWF) under grant P17963-N04. Authors are listed in alphabetical order.

Background: Runtime Diagnosis and Repair in the SW System of an Autonomous Robot

Figure 1 depicts the most important software components of the control system for our autonomous soccer robot. This system is an example of a hybrid architecture which combines the advantages of goal-directed reasoning with the reactivity of the *Sense-Act* control paradigm.

The components *Vision*, *Odometry*, *Sonar*, and *Kicker* process sensor data. The results of the former two are fused by *SensorFusion* into a continuous world model, containing, among others, the estimated positions of environment objects. *Sonar* supplies data which allows for the detection of obstacles, and *Kicker*, relying on an infrared sensor in the kicking device, states whether or not the robot possesses the ball (this holds when the ball is between the grabbers s.t. the robot can kick it). In addition, *Kicker* may receive a "kick now" command from *BehaviorEngine*, which causes *Kicker* to simply pass the command to the kicking device.

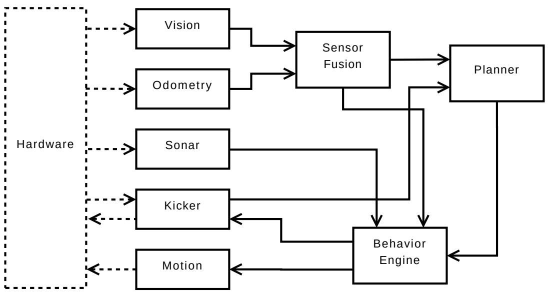

Figure 1: Part of the control system of an autonomous soccer robot. Connections depict data flows between components.

Planner is an AI-planning system representing the deliberative layer of the system. The output of the the planning system are high-level actions which are executed by *BehaviorEngine*. The latter, following the paradigm of (Brooks 1986), executes high-level actions by decomposing them into reactive low-level behaviors. Finally, *Motion* transforms drive-independent commands into low-level commands which are sent to the drive.

The components of our system are independent services which employ CORBA communication mechanisms for exchanging data. One advantages of this architectural design is the fact that software failures, e.g. crashes or deadlocks, are confined to single components. Therefore, we seek to detect failures at runtime and to locate the failed components in order to "repair" them (see below).

In (Peischl, Weber, & Wotawa 2006; Weber & Wotawa 2007) we propose a model-based approach to the runtime diagnosis in a robot control system. We assign *properties* to components and their connections which capture invariants of the system behavior. E.g., a property could express that a service must spawn a certain (minimum) number of processes (threads), or that every n milliseconds there must be a new event (carrying data) on a certain connection. The properties are monitored at runtime. When faulty behavior is detected, i.e., one or more property violations occur, then the failed component(s) can be located by employing the system model which, in our case, represents the dependencies between the properties.

The output of the fault localization algorithm is a set of *minimal diagnoses* (de Kleer, Mackworth, & Reiter 1992). Intuitively, a minimal diagnosis is a subset-minimal set of components which can be reasonably assumed to have failed. E.g., the diagnostic process could return the set $\{\{WorldModel\}, \{Vision, Odometry\}\}$, indicating that either *WorldModel* or both *Vision* and *Odometry* have failed. The latter is a multiple fault, which often occurs in practice when a component fails in dependence of another failed component.

Our runtime diagnosis approach mainly aims at severe faults like crashes or deadlocks. The authors of (Steinbauer & Wotawa 2005) propose to "repair" failed software components by simply restarting them. Although this is not a real repair, practice has shown that it is often the case that the faulty components work correctly after a restart. If there are multiple diagnoses, then we consider all components in the diagnoses as failed. All components which are assumed as failed are immediately aborted and restarted afterwards. After the detection of a fault, the robot is switched to a safe standby mode, i.e., it is idle. It remains in this state until the repair process terminates. Note that the robot may be able to operate during the repair; this issue is discussed at the end of this paper.

However, a restart may not succeed due to the internal fault, or a component may fail multiple times within a short period of time. In these cases, the component is removed forever, and so the capabilities of the system are permanently degraded. The goal of our work is to enable the planning system to determine the remaining capabilities and to create plans which can be executed in spite of the lacking capabilities.

AI-Planning in a Mobile Autonomous Robot

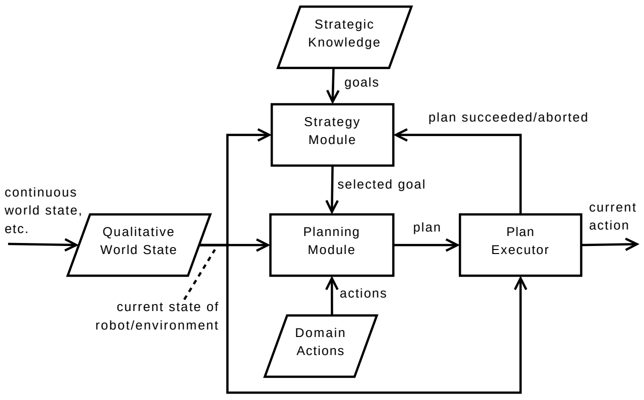

Figure 2: Architecture of the planning system. Arrows represent data flows.

Figure 2 depicts an overview of our planning system. The inputs to the planning system, e.g., the continuous world states generated by component *SensorFusion*, are abstracted, and the results are stored as atomic sentences in the *Qualitative World State (QWS)*. Formally, QWS is a conjunction of atoms. The closed-world assumption holds, and so each atom is either true or false. Table 1 introduces some symbols which we use in the example domain.

Fact	**Is true iff.:**
$inReach(Ball)$	the ball is close, e.g. within 1m
$closer(Ball)$	this robot is closer to the ball than the opponent robot
$isAt(Ball, OppGl)$	the ball is in the opponent's goal
$blocking(Ball, OwnGl)$	the robot blocks line between the ball and the own goal
$possBall$	the robot possesses the ball
$inKickPos(OppGl)$	the robot's position and orientation is opportune for a kick to *OppGl*
$perc(Ball)$	the robot's vision perceives the ball

Table 1: Symbols used for the qualitative world model

$perc(o)$ is particularly important as a mobile robot's vision is imperfect. When the robot cannot perceive o for some reason, the truth value of atoms like $inReach(o)$ is spurious, but $perc(o)$ is false. Hence, wherever such an atom occurs, $perc(o)$ should be used in the same context in order to ensure that the value of, e.g., $inReach(o)$ is valid. By contrast, the evaluation of the predicate $possBall$ relies on data from an infrared sensor which is processed by component *Kicker*. This data is usually reliable.

The task of the *strategy module* is, relying on the predefined *strategic knowledge*, to select a goal which can be passed to the planning algorithm. The strategic knowledge is a sequence $\langle \Gamma_1, \ldots, \Gamma_n \rangle$ of *goal specifications*, where each goal specification Γ_i is a tuple (pre, inv, g). The goal precondition pre is a (first-order) sentence which must be true before this goal can be selected, g is the STRIPS goal condition, and inv is the goal invariant. The latter is relevant only during the execution of a plan: when the invariant of the current goal is violated, then the ongoing plan which is supposed to achieve the goal is aborted, and a new goal is selected.

The goal selection process is simple: the strategy module selects a goal specification Γ_i s.t. $pre(\Gamma_i)$ is satisfied in QWS, but QWS does not entail $g(\Gamma_i)$. If there are two goal specifications Γ_i and Γ_j, $i < j$, which could be selected, then Γ_i is preferred. If no goal can be selected, then the planning system outputs an *Idle* action. Notice that it may also happen that Γ_i is selected but the planning algorithm cannot find a plan which achieves $g(\Gamma_i)$. In this case, the strategy module tries to select another goal.

Table 2 depicts a simple strategy: either the robot aims to score a goal (g_0), or it moves to a position between the ball and the own goal in order to "defend" the own goal (g_1).

The *planning module* contains a Graphplan implementation (Blum & Furst 1997). Based on the QWS as initial state and the selected goal, the planning algorithm computes a parallel plan which is subsequently linearized by randomly establishing order relations between parallel actions.

goal:	**precondition:**	**goal condition:**
g_0	$closer(Ball)$	$isAt(Ball, OppGl)$
g_1	$\neg closer(Ball)$	$blocking(Ball, OppGl)$

Table 2: Simple strategy for a soccer robot. The goal invariants are equal to the goal preconditions.

At present, our planning system is not able to execute parallel plans, but the *BehaviorEngine* can execute behaviors in parallel. E.g., the action *Score* corresponds to several concurrent low-level behaviors: for dribbling, for kicking at the "right" moment, for obstacle avoidance, etc.

Figure 3 introduces some action schemas for our domain. It can be seen that we use very coarse-grained actions. This has several reasons. First, we want the computed plans to be short in order to ensure quick (re-)planning in this highly dynamic domain. Second, the execution details are better left to reactive behaviors. Third, coarse-grained plans are more robust against environment changes than finer-grained ones, and so fewer plan abortions and subsequent re-planning activities are necessary.

Block(obj$_1$, obj$_2$):
`pre`: $perc(obj_1) \wedge perc(obj_2)$
`eff`: $blocking(obj_1, obj_2) \wedge \neg possBall$
Goto(obj):
`pre`: $perc(obj)$
`eff`: $inReach(obj) \wedge \neg possBall$
GrabBall:
`pre`: $\neg possBall \wedge perc(Ball) \wedge inReach(Ball)$
`eff`: $possBall$
Dribble(obj):
`pre`: $possBall \wedge perc(obj)$
`eff`: $inKickPos(obj)$
Score(obj):
`pre`: $possBall \wedge perc(obj) \wedge inKickPos(obj)$
`eff`: $\neg possBall \wedge isAt(Ball, obj)$

Figure 3: STRIPS-like action schemas

E.g., a valid plan for achieving the goal g_0 (Tbl. 2) could be $\langle Goto(Ball), GrabBall, Dribble(OppGl), Score(OppGl) \rangle$. Notice the usage of $perc(obj)$ in the action preconditions. It ensures that an action is only executed when the environment object(s), which the action relates to, are currently perceived by the vision system. Moreover, note the $\neg possBall$ facts in the effects of *Block* and *Goto*: these actions will lose the ball, even if the robot possesses the ball before.

The *plan executor* is responsible for executing those actions which are necessary for achieving the goal, for re-executing parts of the plan which have failed to achieve the desired result, and for aborting plans. When a plan is aborted, we do not attempt to perform any kind of "plan repair", because the plans in our domain are usually small; instead, our planning system discards the plan, selects a new goal, and invokes the planning algorithm. Apart from this, our plan execution is similar to PLANEX (Fikes, Hart, & Nilsson 1972). Remember that also a violation of the goal invariant leads to the abortion of a plan (see above).

Adding Capability Requirements to Actions

As already explained, the control system capabilities are permanently degraded when one or more SW components could not be restarted after failures or when components have failed multiple times within short periods of time and thus were removed from the system. Nevertheless, the robot may be able to perform certain actions which do not depend on the lost capabilities. In order to allow for the creation and execution of meaningful plans, the planning system needs to know which components are available and work correctly. Hence, the diagnosis/repair engine notifies the planning system about the states of all components, and these informations are encoded using the ok predicate: for the point of view of the planning system, each component c is either available and works correctly, i.e., $ok(c)$ holds, or the component is unavailable ($\neg ok(c)$ holds).

In order to determine which actions are executable we model the abstract capabilities of the control system. As we shall see later, this allows us to add capability requirements to action preconditions. For inferring the available capabilities at runtime, we employ a *capability graph*, which is depicted in Fig. 4. Note that, for brevity, we introduce a component *WorldModel* which subsumes the components *Vision*, *Odometry*, and *SensorFusion* in Fig. 1, and *BehaviorExec* subsumes *BehaviorEngine* and *Motion*.

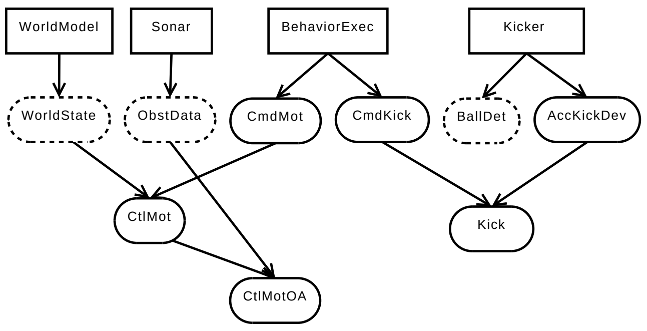

Figure 4: A capability graph. Rectangles are components, solid rounded rectangles are *acting capabilities*, and dotted rounded rectangles are *sensing capabilities*.

The capability graph is a directed acyclic graph (DAG). The sources (nodes without incoming edges) are the components. Each component provides one or more basic capabilities. E.g., *Sonar* provides `ObstData` (obstacle data), *BehaviorExec* can command motions and kicks, and *Kicker* provides the ball detection capability and it also accesses the kicking device. We distinguish between *sensing* and *acting* capabilities.

Basic capabilities, which are provided by single components, can be combined to derive higher-level capabilities. E.g., the capabilities `WorldState` and `CmdMot` are combined to `CtlMot`: in order to *control* a motion (in the sense of control theory), the system needs not only to be able to command motions, but also to receive sensor feedback indicating how the world changes. Moreover, if the system also has obstacle data, then the motion control is augmented with a reactive obstacle avoidance (capability `CtlMotOA`).

The semantics of the cabability graph can be captured by a set of logical sentences. We use the predicate has for sensing capabilities and can for acting capabilites:

$$\begin{aligned}&ok(WorldModel) \Leftrightarrow has(WorldState)\\&ok(BehaviorExec) \Leftrightarrow can(CmdMot)\\&ok(BehaviorExec) \Leftrightarrow can(CmdKick)\\&has(WorldState) \wedge can(CmdMot) \Leftrightarrow can(CtlMot)\\&\ldots\end{aligned}$$

These sentences are added to the knowledge base in the form of rules. We employ a simple forward-chaining algorithm which, starting with the given $ok(c)/\neg ok(c)$ facts, infers the capabilities.

We use the capability predicates in action preconditions in order to state those capabilities required for the execution of the action by the control system. More precisely, the precondition of a high-level action should include the acting capabilities as well as the sensing capabilities required by those low-level behaviors which correspond to this action. E.g., the action *GrabBall* moves to the ball s.t. the ball is between the grabbers of the robot. The acting capability required by the low-level behaviors is `CtlMotOA`, the required sensing capability is `WorldState`. In this example, the latter is redundant since $has(WorldState)$ always holds when $can(CtlMotOA)$ is true.

The augmented action definitions are given in Fig. 5. For the sake of conceptual clarity, we split the precondition $pre(\mathcal{A})$ of an action $\mathcal{A}$ into two parts: $pre(\mathcal{A}) = pre_E(\mathcal{A}) \wedge pre_C(\mathcal{A})$, where $pre_E(\mathcal{A})$ is the *environment precondition* and $pre_C(\mathcal{A})$ the *capability precondition*. The environment preconditions in Fig. 5 conform to the preconditions in Fig. 3. Furthermore, notice that all of actions in Fig. 5 require an obstacle avoidance in order to avoid physical damage caused by collisions.

Block(obj$_1$, obj$_2$):
- `pre`$_E$: $perc(obj_1) \wedge perc(obj_2)$
- `pre`$_C$: $can(CtlMotOA)\ [\wedge\, has(WorldState)]$
- `eff`: $blocking(obj_1, obj_2) \wedge \neg possBall$

Goto(obj):
- `pre`$_E$: $perc(obj)$
- `pre`$_C$: $can(CtlMotOA)\ [\wedge\, has(WorldState)]$
- `eff`: $inReach(obj) \wedge \neg possBall$

GrabBall:
- `pre`$_E$: $\neg possBall \wedge perc(Ball) \wedge inReach(Ball)$
- `pre`$_C$: $can(CtlMotOA)\ [\wedge\, has(WorldState)]$
- `eff`: $possBall$

Dribble(obj):
- `pre`$_E$: $possBall \wedge perc(obj)$
- `pre`$_C$: $can(CtlMotOA)\ [\wedge\, has(WorldState)]$
- `eff`: $inKickPos(obj)$

Score(obj):
- `pre`$_E$: $possBall \wedge perc(obj) \wedge inKickPos(obj)$
- `pre`$_C$: $can(CtlMotOA) \wedge can(Kick)\ [\wedge\, has(WorldState)]$
- `eff`: $\neg possBall \wedge isAt(Ball, obj)$

Figure 5: Action schemas with environment and capability preconditions (`pre`$_E$ and `pre`$_C$, resp.). Redundant capability atoms are enclosed by brackets [].

For example, suppose the component *Kicker* has failed. Then the goal g_0 in Tbl. 2 can no longer be achieved, as the action $Score(OppGl)$ requires $can(Kick)$. However, for g_1 the plan $\langle Block(Ball, OwnGl)\rangle$ is valid, since this action does not depend on any capability provided by *Kicker*.

These action definitions also illustrate *why* we have chosen to model capabilities rather than simply enumerating the required components in action preconditions. First, enumerating all required components would be tedious and error-prone in systems with a large number of components. Second, finding the capability requirements of actions is more intuitive since capabilities capture higher-level concepts. Third, the action definitions are easier to maintain when the design of the control system changes, because the system capabilities may be unaffected by modifications in the component structure.

Finding Actions and Goals with Lower Capability Requirements

So far, in order to enable our system to make rational decisions after software failures, we have added capability requirements to existing actions. This guarantees that generated plans only contain acions which can actually be executed by the control system. However, in practice it is often desirable to add new actions which require fewer capabilities but which achieve the same (or at least similar) effects as already existing actions, albeit with lower efficiency. For example, suppose the *Sonar* component fails. Then the system lacks the capability `CtlMotOA` which is contained in all preconditions in Fig. 5. Hence, we add similar actions which, however, do not rely on an obstacle avoidance. These actions are performed at a lower motion speed and thus are less likely to cause physical damage in case of a collision, e.g.:

Goto_slow(obj):
`pre`$_E$: $perc(obj)$
`pre`$_C$: $can(CtlMot) \wedge \neg can(CtlMotOA)$
`eff`: $inReach(obj) \wedge \neg possBall$

Note that $\neg can(CtlMotOA)$ ensures that slow actions are selected only if there is no obstacle avoidance. The effect remains unchanged. Hence, after a failure of *Sonar* the plan $\langle Goto\_slow(Ball)$, $GrabBall\_slow$, $Dribble\_slow(OppGl)$, $Score\_slow(OppGl)\rangle$ could be computed for goal g_0 in Tbl. 2. Obviously, this plan is less likely to succeed: e.g., it may happen now that the opponent robot approaches the ball faster than this robot is able to. In this case, the goal invariant $closer(Ball)$ of g_0 may be violated, leading to an abortion of the plan. Afterwards the goal g_1 would be selected.

Furthermore, suppose the component *Kicker* fails. If $\neg closer(Ball)$ holds in the current world state, then the goal g_1 is selected, and the plan $\langle Block(Ball, OwnGl)\rangle$ is computed. However, if $closer(Ball)$ holds, then no plan can be found which achieves g_0, and the robot becomes idle. This example shows that we should also seek to provide additional goal specifications having lower capability demands and thus allowing the robot to do something meaningful in spite of software failures. In this example, by removing the precondition of goal g_1 in Tbl. 2, we could accomplish that g_1 is also selected if $closer(Ball)$ holds but no plan can be found for g_0 due to a lack of capabilities.

Capabilities Required for Plan Monitoring

If the capability precondition of an action is fulfilled, then the control system has the capabilities to execute the low-level behaviors which correspond to this action. Unfortunately, even if the capability preconditions of all actions in a plan are true, this does not necessarily imply that this plan can also be monitored by the plan executor. Suppose, for example, a plan which contains the action $GrabBall$. The reactive behaviors which implement this action can be executed if $pre_C(GrabBall) = can(CtlMotOA)$ holds; the capability `BallDet` is not required. However, the plan executor needs this capability for the monitoring of this action: the effect of $GrabBall$ is $possBall$, and the truth value of this fact is only valid if the ball detection works, otherwise it is spurious (remember that, due to the closed-world assumption, each atom is either true of false).

A crucial point is the observation that the evaluation (of the truth value) of each predicate p requires certain sensing capabilities. Hence, we introduce a function $\pi(p)$ which returns the set of sensing capabilities needed for the evaluation of p. E.g., $\pi(possBall) = \{BallDet\}$, whereas $\pi(p) = \{WorldState\}$ for all other predicates in Tbl. 1. For predicates which do not depend on sensor perceptions, π would return an empty set. Furthermore, for any first-order sentence φ, let $\mathcal{P}(\varphi)$ denote the set of all predicates occurring in φ, and we define:

$$\Pi(\varphi) = \bigwedge\nolimits_{p\in\mathcal{P}(\varphi)} \Big[\bigwedge\nolimits_{\gamma\in\pi(p)} has(\gamma)\Big]$$

I.e., $\Pi(\varphi)$ is a conjunction of has-atoms indicating all sensing capabilities required for the evaluation of the predicates in φ. A quick solution to the problem explained above could be to add all those sensing capabilities to the precondition of an action $\mathcal{A}$ which are required for the evaluation of the predicates occurring in the precondition and the effect of $\mathcal{A}$. I.e., we could define a *monitoring precondition* $pre_M(\mathcal{A})$, which is part of $pre(\mathcal{A})$:

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

with

$$pre_M(\mathcal{A}) = \Pi\big(pre_E(\mathcal{A})\big) \;\wedge\; \Pi\big(eff(\mathcal{A})\big)$$

As all actions in Fig. 5 contain the predicate $possBall$ in the environment precondition or the effect, $has(BallDet)$ would be included in all monitoring preconditions (in addition to $has(WorldState)$). This solution would guarantee that actions are only executed if all sensing capabilities required for the evaluation of the predicates in the precondition and the effect are available. However, it can be easily seen that this method is too restrictive. Suppose the component *Kicker* has failed and, for some reason, we simply want the robot to be at a certain position X. The goal $inReach(X)$ could not be achieved, because $pre_M(Goto)$ would contain $has(BallDet)$, as $\neg possBall$ is an effect of $Goto$. Obviously, this is an undesired result, as $Goto(X)$ does certainly not require a ball detection for achieving $inReach(X)$, and the action effect $\neg possBall$ is irrelevant in this context.

One could try to resolve this issue by defining $\neg possBall$ as a *side-effect* of *Goto*, meaning that this part of the effect is not relevant and thus does not need to be monitored. However, it may well be the case that a subsequent action in a plan requires that $\neg possBall$ holds. The relevance of the effect $\neg possBall$ depends on the context, i.e., on the plan which contains $Goto$ and on the goal to achieve. Notice that side-effects have been used before in AI-planning, but mainly for reducing the search costs; e.g., see (Fink & Yang 1993) and the references therein.

We propose to use the plan executor to determine the sensing capabilities required for the monitoring of a plan. Our approach is based on *kernel models*, which were introduced for the monitoring of STRIPS plans (Fikes, Hart, & Nilsson 1972). Many real-world planning systems still use plan execution methods similar to this one. In the following we provide a brief introduction to kernels.

	Kernel / Action:	$\mathbf{\Pi(K_i)}$**:**
K_1	$perc(obj) \wedge can(CtlMotOA)$	$has(WorldState)$
$\mathcal{A}_1$	$\mathbf{Goto(X)}$	
K_2	$inReach(X)$	$has(WorldState)$

Table 3: Kernels for the plan $\langle Goto(X)\rangle$ and the goal $inReach(X)$.

From a STRIPS plan, a kernel can be extracted for before and after each action. For a plan $\langle \mathcal{A}_1, \ldots, \mathcal{A}_n\rangle$, the corresponding kernels are $K_1, \ldots, K_{n+1}$. Table 3 and 4 depict the kernels for two example plans. 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 actions $\mathcal{A}_i, \ldots, \mathcal{A}_n$ can be executed and they will achieve the goal, provided that the executions lead to the desired results as indicated in the action effects. K_{n+1} is the STRIPS goal condition. For $1 \leq i \leq n$, K_i consists of those literals contained in the precondition of $\mathcal{A}_i$ plus those literals in K_{i+1} which are not part of the effect of $\mathcal{A}_i$.

At each execution step, the plan executor performs a backwards search 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. When such a kernel K_i is found, then 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.

For a kernel K_i, $\Pi(K_i)$ states the sensing capabilities required for evaluating the truth value of the facts in K_i. It can be seen that the kernels in Tbl. 3 do not include the atom $possBall$, i.e., it is irrelevant for the monitoring of this plan, even though the effect of $Goto$ contains $\neg possBall$. Consequently, this plan can be executed (and monitored) in spite of a failure of component *Kicker*. By contrast, several kernels in Tbl. 4 require $has(BallDet)$ for their evaluation.

We propose to adapt the execution policy, which we have outlined above, as follows. Before checking if a kernel K_i is satisfied in the current world state, the plan executor checks if $\Pi(K_i)$ holds. If not, then we know that K_i contains facts whose truth value is spurious. Hence, if the plan execution would be continued, then it might happen, e.g., that actions would be started whose precondition do not hold in the real environment, or that the execution of an action continous forever, although its effect has already been achieved in the real world. Therefore, the plan executor should abort the plan, and either another goal should be selected or an alternative plan be computed.

Discussion and Related Research

A general problem is the fact that most components of real control systems are vital, i.e., the robot cannot pursue any task when one of these components fail. For example, only two components in Fig. 1 are non-vital, namely *Kicker* and *Sonar*. This issue can only be resolved by decomposing control systems into a larger number of independent components. E.g., if there is a fault in *BehaviorEngine*, then the robot's motion cannot be controlled anymore. However, if each reactive behavior is represented by a separate component, then some of these components are not vital: a behavior `BEH_KICK`, for example, would not be required for actions like $Goto$.

An interesting question is how our approach could be extended to hardware failures. The qualitative modelling of hardware capabilities is more difficult. Regarding software systems, the planning system only needs to know which components are available (and working correctly) in order to infer the capabilities of the SW system. By contrast, hardware components may degrade gradually. E.g., the precision of sensors may decline, or the driving unit is no longer able to perform specific movements after the breakdown of a single wheel (Hofbaur *et al.* 2007).

Hence, a hardware diagnosis (and reconfiguration) engine would need to provide more specific informations to the planning system. E.g., a vision diagnoser could directly supply facts like $has(precision\_high)$, $has(precision\_low)$, etc., which can be utilized in capability preconditions of actions: $pre_C(GrabBall) = has(precision\_high) \wedge \ldots$ and $pre_C(Block) = has(precision\_medium) \wedge \ldots$ (grabbing a small object like a ball requires high sensing precision).

Another issue is that the robot could continue to pursue tasks while failed software components are restarted. This is particularly important as restarts may take a long time. Although our current approach enables the planning system to create and execute plans during repair, it would be advantageous to employ partial-order plans which integrate abstract repair actions. For example, suppose the component *Kicker* has failed. In our current approach, at the beginning of the repair process no plan could be computed for the goal $isAt(Ball, OppGl)$. We would prefer that a plan as depicted in Fig. 6 is found. Relying on this plan, *Kicker* may be repaired concurrently to the action $Goto(Ball)$. When the repair is finished, the remaining actions can be executed.

For generating partial-order plans one can use, e.g., the UCPOP planner (Weld 1994). It should be noted that two actions which are unordered relative to one another in a partial-order plan can not always be executed in parallel, see (Knoblock 1994). Moreover, although the Graphplan

	Kernel / Action:	$\mathbf{\Pi(K_i)}$**:**
K_1	$perc(OppGl) \wedge perc(Ball) \wedge can(CtlMotOA) \wedge can(Kick)$	$has(WS)$
$\mathcal{A}_1$	$\mathbf{Goto(Ball)}$	
K_2	$perc(OppGl) \wedge \neg possBall \wedge perc(Ball) \wedge InReach(Ball) \wedge can(CtlMotOA) \wedge can(Kick)$	$has(WS) \wedge has(BallDet)$
$\mathcal{A}_2$	$\mathbf{GrabBall}$	
K_3	$possBall \wedge perc(OppGl) \wedge can(CtlMotOA) \wedge can(Kick)$	$has(WS) \wedge has(BallDet)$
$\mathcal{A}_3$	$\mathbf{Dribble(OppGl)}$	
K_4	$possBall \wedge perc(OppGl) \wedge inKickPos(OppGl) \wedge can(CtlMotOA) \wedge can(Kick)$	$has(WS) \wedge has(BallDet)$
$\mathcal{A}_4$	$\mathbf{Score(OppGl)}$	
K_5	$IsAt(Ball, OppGl)$	$has(WS)$

Table 4: Kernels for a plan with four actions and the goal $IsAt(Ball, OppGl)$. WS abbreviates $WorldState$.

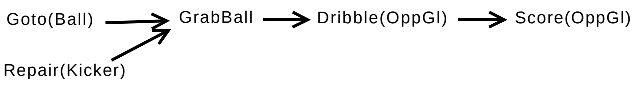

Figure 6: Partial-order plan containing a repair action.

algorithm produces parallel plans, it is unclear if Graphplan is appropriate for our purpose, because its plans are often too restrictive wrt execution flexibility, i.e., it imposes too many precedence relations between actions (Nguyen & Kambhampati 2001).

So far, the issue of integrating the results obtained by runtime diagnosis in AI-planning systems of autonomous systems has gained little attention among researchers. The Remote Agent architecture (Williams, Nayak, & Muscettola 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 authors do not explain this in detail; neither the related modelling issues nor the importance of sensing capabilities for plan monitoring are discussed.

The authors of (Micalizio, Torta, & Torasso 2004; Micalizio & Torasso 2007) use model-based diagnosis techniques to monitor the execution of multi-agent plans by a team of service robots. 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.

Model-based diagnosis techniques can also be employed for the execution monitoring of plans, see, e.g., (Roos & Witteveen 2005) and the references therein.

A related issue, which we have not addressed in our work, is the repair of plans after software or hardware failures. For example, (Gallien, Ingrand, & Lemai 2005) propose an approach to temporal planning and execution control which includes plan repair and replanning. The context of this work are autonomous exploration missions.

We are not aware of any past research which has addressed the issue of AI-planning with degraded software capabilities in autonomous systems. We have presented an AI-planning system for an autonomous soccer robot. This system is able to adapt its decision-making after software faults which are detected and located by a model-based diagnosis system. Relying on the output of the diagnostic process, our planning system infers the available capabilities of the control system. In order to achieve that plans contain only actions which can also be executed by the (degraded) control system, we augment action preconditions with capability requirements. Furthermore, we show that also the monitoring of plans requires certain sensing capabilities, and we propose an extension to the PLANEX plan execution technique which allows the executor to detect whether or not the required sensing capabilities are available.

References

Blum, A. L., and Furst, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1–2):279–298.

Brooks, R. A. 1986. A robust layered control system for a mobile robot. *IEEE Jornal of Robotics and Automation* RA-2(1):14–23.

de Kleer, J.; Mackworth, A. K.; and Reiter, R. 1992. Characterizing diagnoses and systems. *Artificial Intelligence* 56(2–3):197–222.

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.

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

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

Gallien, M.; Ingrand, F.; and Lemai, S. 2005. Robot Actions Planning And Execution Control For Autonomous Exploration Rovers. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*.

Hofbaur, M.; Köb, J.; Steinbauer, G.; and Wotawa, F. 2007. Improving robustness of mobile robots using model-based reasoning. *Journal of Intelligent and Robotic Systems* 48(1):37–54.

Knoblock, C. A. 1994. Generating parallel execution plans with a partial-order planner. In *Proceedings of the 2nd In-*

ternational Conference on Artificial Intelligence Planning Systems, 98–103.

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

Micalizio, R.; Torta, G.; and Torasso, P. 2004. Monitoring and diagnosis as support tools for planning and scheduling in robocare. In *3rd Italian Workshop on Planning and Scheduling*.

Nguyen, X., and Kambhampati, S. 2001. Reviving partial order planning. In *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, 459–466. San Francisco, CA: Morgan Kaufmann Publishers, Inc.

Peischl, B.; Weber, J.; and Wotawa, F. 2006. Runtime fault detection and localization in component-oriented software systems. In *Proceedings of the 17th International Workshop on Principles of Diagnosis (DX-06)*.

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)*, volume 3690 of *Lecture Notes in Computer Science*, 357–366.

Steinbauer, G., and Wotawa, F. 2005. Detecting and locating faults in the control software of autonomous mobile robots. In *Proceedings of the* 19^{th} *International Joint Conf. on Artificial Intelligence*, 1742–1743.

Weber, J., and Wotawa, F. 2007. Diagnosing dependent failures in the hardware and software of mobile autonomous robots. In *Proceedings of IEA/AIE 2007*. To appear.

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.