A Robot Control System Integrating Reactive Control, Reasoning, and Execution Monitoring

Axel Großmann   Andreas Henschel   Michael Thielscher

Artificial Intelligence Institute
Department of Computer Science
Technische Universität Dresden

Technical Report WV-03-02/CL-2003-01

Abstract

We present a robot control system that integrates robust reactive control with efficient reasoning about actions and execution monitoring. On the reactive level, the robot is controlled using a hierarchy of low-level behaviors. On the high level, a logical representation of the world enables the robot to reason about the state of the world and to plan action sequences. If the execution of an action fails, probabilistic information on the state of the world based on the robot's sensors are used to update the logic-based representation of the world. High-level reasoning then allows to infer possible explanations and to recover from the failure situation. The proposed system is evaluated using a set of realistic office-delivery scenarios.

1 Introduction

In recent years, robotics has been subject to promising advances in sensor and actuator hardware, sensory processing techniques, and low-level control methods. Yet, the area has not been benefited to the full amount from the availability of powerful knowledge representation tools and action calculi. Imagine a mobile robot that is to accomplish complex delivery tasks in a typical office environment. The ability to plan would allow the robot to divide the overall task into executable steps. Moreover, if a delivery action fails, a smart recovery procedure could possibly explain the failure and, based on this explanation, calculate alternative solutions: The current item could be delivered to an alternative recipient, for example, or the item may be scheduled for delivery at a later time.

The capabilities described above require the robot to maintain information on its own state, usually obtained by processing the sensory data, as well as knowledge about the operating environment and the task at hand, commonly referred to as world model. To deal with the uncertainty in the robot's perceptions, it is common practice to represent state information such as the robot's location in the environment or the position of objects of interest using state enumeration and probabilities. Popular approaches include position probability grids [4] and particle sets [15]. On the other hand, we would like to avoid state enumeration as a representation of the world model and prefer a logical representation instead [1, 6]. We feel that up to now there is still a large gap between state-of-the-art reactive control mechanisms using probabilistic representations and symbolic reasoning and action planning methods for mobile robots.

Existing symbolic reasoning theories serve the purpose of reasoning and planning on an abstract level. Only a scarce number of serious attempts to implement them on real robots exist at present [12, 8, 7]. In the field of cognitive robotics, many approaches focus on the generation of high-level plans and never leave the ground of pure soft-bot applications, neglecting the tedious work of realistic action execution and assuming an error-free interaction with the environment. Experiences as in [10] proved this assumption to be too strong and led to the conclusion to take sensor and actuator errors more seriously.

The aim of our work is to enable a mobile robot to reason and to perform action planning and thus to gain problem solving power. We present a hierarchical, modular control architecture that integrates low-level behaviors and symbolic reasoning, thus combining the robustness of reactive control with abstract thinking. The system includes a layered scheme of execution monitoring. It is evaluated using realistic scenarios for an office-delivery robot.

The paper is organized as follows. In Section 2, we introduce the notion and concepts of execution monitoring. In Section 3, we describe the main components of the proposed architecture and the representations and techniques used at the reactive and the abstract level. In Section 4, we demonstrate the functionality of the system using example scenarios. We conclude in Section 5.

2 Execution Monitoring

There is no generally accepted definition of execution monitoring. [5] defined execution monitoring as 'the robot's process of observing the world for discrepancies between the actual world and the robot's internal representation of it, and recovering from such discrepancies'. In this work, we extend this notion to the extent that the robot should come up with explanations for the detected discrepancies as well.

The process of execution monitoring is divided into three steps: detecting discrepancies, explaining the situation, and launching a recovery procedure. The individual steps are discussed subsequently.

2.1 Detecting Discrepancies

The execution of complex actions generally requires the robot to have a representation of its current state – such as the robot's position in the environment, the distance to obstacles, and the state of the gripper – as well as a model of the environment and a description of the task to be solved. By comparing the current state with the model of the world and the task, it should be possible to detect erroneous situations. However, we do not expect such a comparison to be straightforward as the models and representations are likely to be complex and incompatible.

In general, the representation of the robot's state will be layered and distributed. The control architectures usually include specialized modules for sensor data processing, e.g., to compute the robot's position of the state of doors from distance information, or to detect objects of interest from image data. At the low levels of control, we mostly use probabilistic representations. For example, the robot's position in the environment can be encoded as a probability distribution over all possible locations, and the state of a door as a probability distribution over the set of possible door configurations. At the highest level of control, in contrast, we prefer symbolic, logic-based representations. For instance, the robot's position and the state of doors can be described using states of fluents. The robot's state at all levels is updated at regular intervals.

Suppose the robot is to execute a sequence of actions. At the beginning, we usually have an expectation of the intended effects, i.e., the change of state caused by each action. These expectations are going to be layered and distributed as well. The anticipated effect of a go-to action at the low level, for example, is a change of the robot's position within a certain amount of time while maintaining a minimum distance to obstacles. At the highest level of control, expectations can be inferred using a knowledge base and an underlying action theory.

Given the availability of state information and expectations, it should be possible to detect discrepancies between the current state and the anticipated effects of actions. The means of detection will depend on the individual levels of control. These could be time or confidence thresholds, spatial distances, or symbolic state discrepancies.

2.2 Providing Explanations

Once an action had not the intended effect, we would like to know the reason, i.e., find an explanation for the encountered discrepancy between state information and expectation. In general, this will require reasoning. This goes beyond the capabilities of reactive control and has to be performed at the highest level of control.

The robot can generally only observe symptoms of the current situation. For instance, a robot getting stuck could have been caused by a variety of reasons: a localization failure, a visible obstacle (a person), an invisibly obstacle (a chair leg), an unexpected change of the environment (a door being closed), etc. On the other hand, the more relevant features of the environment are included in the state information and the smaller the granularity of the world model and task description, the easier it is to make conjectures about the possible reasons of failure.

Hierarchical planning meets the requirements mentioned above. If, for example, a complex navigation task is broken down into smaller actions like door passings, corridor and room traverses, then failures can be substantiated with higher reliability. By using default logic [11], the planning and reasoning module can abstract away from the sheer non-exhausting, but increasingly unlikely, set of preconditions, thus solving the qualification problem. In case of unexpected situations though, these default assumptions must be double checked according to an ordered preference list [9], thus providing the most likely explanation for action failure.

2.3 Recovering

Once an explanation of the current situation is found and the state information and world model are corrected, some recovery strategy is expected to remedy the failure. In approaches such as [3], it is suggested to launch a predefined recovery plan. For example, when the robot notices that it ran into a dead end, it computes a path that brings it back on the original track and continues with the initial plan. Given a strong planning tool, instead of just correcting the mistake, we are able to find an optimal plan for the current situation (provided that the state and world model are correct). Suppose the robot did not run into a dead end, but found a shortcut. Now, the planner can provide a better solution if returning to the old track proves to be more costly.

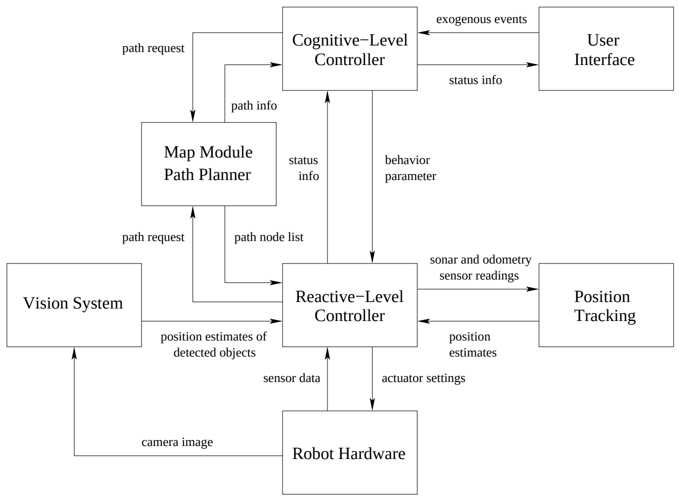

Fig. 1: Control architecture of the robot.

3 Building the System

To put the functionality described above into practice, we added a high-level planning and reasoning component to a fairly standard hierarchical robot control system. In the following, we describe the parts of the system that are relevant specifically to execution monitoring.

3.1 System Architecture

The control architecture of the robot, as depicted in Figure 1, consists of several modules. The hardware controller talking directly to the robot's sensors (odometry, sonars, laser) and actuators (drive motors, gripper) is considered the lowest level of control. The basic perceptual and behavioral functions of the robot are implemented by the reactive-level controller. We have used a behavior-based approach. That is, the reactive controller includes several interacting, task-specific programs that are referred to as low-level behaviors. Each behavior program takes the current sensor readings and the state information and computes target values of the robot's actuators. Individual behaviors can overwrite the output of other behaviors. There are specialized sensor-processing modules for visual object detection and laser-based position tracking. These components maintain a probabilistic representation of the detected objects and robot poses, respectively.

The robot's goal-oriented behavior is directed by the cognitive-level controller. This is done by setting task specific parameters of the low-level behaviors such as the activation context and target coordinates. To navigate the office environment, both the reactive and the cognitive controller obtains information from the map and path planning module.

3.2 At the Lower Levels

Independent of the task to be performed, the safety of the robot has to be maintained at all times. Therefore, the reactive controller includes a set of low-level safety behaviors, e.g., for obstacle avoidance and velocity control, that cannot be switched off by higher levels of control. The other low-level behaviors are designed to achieve specific (parameterized) goals such as to travel to a target position or to pick up an object. Suppose the robot is to execute a sequence of high-level actions. Then for each action, there is a designated process that supervises the execution of that action. This execution monitoring process invokes the appropriate low-level behaviors and deals with exceptional situations. In the following, we illustrate this concept for the action of traveling to a given office.

The monitoring processes are implemented as finite state machines. Some states are common to all actions, others are specific to the task. In the example, we have used the following states:

Init	Initialize the process
Deactivated	Wait for activation
WaitForGoal	Wait for target from cognitive controller
RequestPlan	Query the path planner
ReceivePlan	Receive path node list
PlanReceived	Activate low-level behavior *GoToPos*
ExecutePlan	Set new intermediate target position and react on status info from *GoToPos*

For each monitoring process, there is a predefined set of exceptions, represented by status information. There are exceptions that are passed on by the low-level behaviors and there are exceptions that were detected by the sensor processing systems. The low-level behavior *GoToPos* was implemented as finite state machine, too. The common interface to low-level behaviors consists of three states: *Init*, *Deactivated*, and *Running.* The status information used in the example above are:

InProgress	Execution in progress
Success	Execution terminated successfully
FailureStalled	Drive motors stalled
FailureObstacle	Path blocked by obstacle
FailureDoor	Path blocked by door
Timeout	Execution timeout
Interrupt	Execution interrupted

The monitoring process passes this status information on to the cognitive controller.

3.3 At the Highest Level

The high-level controller maintains a symbolic world model. Reasoning about actions is used at this level to plan complex tasks and to generate expectations as to the effects of actions. When a discrepancy arises between the expectations and the actual situation, the high-level control uses its reasoning facilities to come up with suitable explanations and a recovery plan. As the underlying action theory we use the fluent calculus with its solution to the classical frame, ramification, and qualification problems. Our system builds on the inference engine FLUX for the fluent calculus [14].

Specifying actions

The cognitive controller requires precondition and effect specifications of each high-level action. To account for unexpected action failure, we make the distinction between normal and abnormal preconditions. The former need to be ascertained before an action can be planned while the latter are assumed away by default but serve as possible explanations in case the action surprisingly fails. For example, the following axiom specifies the preconditions of the action $\mathit{Deliver}(o,p)$ of delivering object o to person p:

$$\begin{aligned}
&\mathit{Poss}(\mathit{Deliver}(o,p),s) \equiv \\
&\quad \mathit{Holds}(\mathit{Carries}(o,p),s) \wedge \exists r\ \mathit{Holds}(\mathit{InRoom}(r),s) \wedge \\
&\quad \mathit{Office}(r,p) \wedge \\
&\quad \neg \mathrm{Ab}(\mathit{Traceable}(p),s) \wedge \neg \mathrm{Ab}(\mathit{NotLost}(o),s)
\end{aligned}$$

Here, the standard predicates $\mathit{Poss}(a,s)$ and $\mathit{Holds}(f,s)$ denote that in situation s, action a is possible and property f is known to hold, respectively. An instance of $\mathrm{Ab}(f,s)$ indicates the presence of abnormal condition f in situation s. Hence, the precondition axiom says that normally a delivery is possible if the robot carries the object in question and happens to be in the office of the recipient. However, the action fails under the unusual circumstances that the respective person is not traceable or the object has been lost.

Effects of high-level actions are specified by state update axioms, which provide a solution to the frame problem. For example, the action $\mathit{Receive}(o,p)$ of receiving object o from person p is specified by:

$$\begin{array}{l} \mathit{Poss}(\mathit{Receive}(o,p),s) \supset \\ \quad \exists r\, \mathit{Holds}(\mathit{Request}(p,o,p'),s) \supset \\ \qquad \mathit{State}(\mathit{Do}(\mathit{Receive},s)) = (\mathit{State}(s) - \mathit{Request}(p,o,p')) \\ \qquad\qquad\qquad\qquad\qquad\qquad + \mathit{Carries}(o,p') \end{array}$$

Here, the standard functions $\mathit{State}(s)$ and $\mathit{Do}(a,s)$ denote, respectively, the state in situation s and the situation reached after performing action a in situation s. Hence, the axiom describes the subsequent state in terms of an update of the current state by the negative effect $\mathit{Request}(p,o,p')$ and the positive effect $\mathit{Carries}(o,p')$. That is, upon receiving o from p addressed to person p', the robot carries the object and the corresponding delivery request is canceled.

Actions sometimes fail to produce the intended effect. For example, in exceptional cases a delivery may leave the recipient with the wrong item:

$$\begin{array}{l} \mathit{Poss}(\mathit{Deliver}(o,p),s) \supset \\ \quad \neg\exists o' \mathrm{Ab}(\mathit{Delivery}(p,o'),s) \supset \\ \qquad \mathit{State}(\mathit{Do}(\mathit{Deliver}(o,p),s)) = \mathit{State}(s) - \mathit{Carries}(o,p) \vee \\ \quad \exists o',p' \mathrm{Ab}(\mathit{Delivery}(p,o'),s) \wedge \\ \quad\; \mathit{Holds}(\mathit{Carries}(o',p')) \wedge o \neq o' \wedge p \neq p' \supset \\ \qquad \mathit{State}(\mathit{Do}(\mathit{Deliver}(o,p),s)) = \mathit{State}(s) - \mathit{Carries}(o',p') \end{array}$$

The condition $\mathrm{Ab}(\mathit{Delivery}(p,o),s)$ represents the abnormal case of delivering the wrong item o to person p in situation s. Abnormal conditions in state update axioms, too, are assumed away by default but may serve as explanation for observed discrepancies between the expected and the actual outcome.

Possible indirect effects of actions are specified by causal relationships, which solves the ramification problem. For example, suppose the robot searches for an object o among the group of people in some room r. Whenever $\mathit{Has}(p',o)$ becomes true, stating that person p' is in possession of the object, then all other previously considered possibilities of people having o are ruled out:

$$\begin{array}{l} \mathit{Holds}(\mathit{MightHave}(p,o,r),s) \supset \\ \qquad \mathit{Causes}(\mathit{Has}(p',o), \neg \mathit{MightHave}(p,o,r),s) \end{array}$$

Here, the standard macro definition $\mathit{Causes}(e,r,s)$ means that effect e causes indirect effect r in situation s.

Explaining action failures

Unless there is evidence to the contrary, abnormal conditions $\mathrm{Ab}(f, s)$ are assumed away by default. We use non-monotonic default theory to this end, which provides a solution to the qualification problem [13]. Whenever the observations suggest a discrepancy between the default expectations and the actual world, the default theory entails that one or more default assumptions no longer hold. In this way, the high-level controller generates suitable explanations for the encountered failure [9]. By appealing to prioritized default logic [2], one can specify qualitative knowledge of the relative likelihood of the various explanations for abnormal qualifications. The accompanying concept of preferred extensions then helps selecting the most likely explanations.

Planning Engine

A controlling mechanism that monitors action execution inevitably requires a high amount of reactivity. On the low level, a set of interacting behaviors seem to meet this requirement. In an analogous manner, an abstract task planner should be able to react on events, that might effect the current agenda. Action failures and general world changes require replanning for both successful accomplishment and efficiency reasons.

The maintenance of a state S and a set of abstract state evaluation functions $E : \mathcal{S} \rightarrow \mathcal{A}$ that are consulted during every action-execution cycle, are the base for the planning loop. Once a critical world change is realized, the current agenda is dropped, and the planner is invoked again. The planner investigates the current state according to state dependent criteria:

```
loop(S,Z):-
  ( /* Stationary */
    notify_reach(Z, URL)    -> As = URL;
    call_help(P,Z)          -> As = [call_help(P)];
    search_fail(O,P,Z)      -> As = [email(O,P)];
    delivery(O, P, Z)       -> As = [deliver(O,P)];
    receipt(O, P, Z)        -> As = [receive(O,P)];
    /* Knowledge Acquisition */
    search(SearchDialog,Z)  -> As = SearchDialog;
    /* Navigation */
    continue(GoAct, Z)      -> As = [walk_on|GoAct];
    /* Otherwise */             As = [idle]
  ),
  execute(As, S, Z).
```

The predicates on the left hand side of the arrows can be understood as diagnostic functions of the current state. This is, the predicate *NotifyReach* investigates whether people became out of reach recently. The output parameter *URL* is an action sequence of notifications about the unreachable

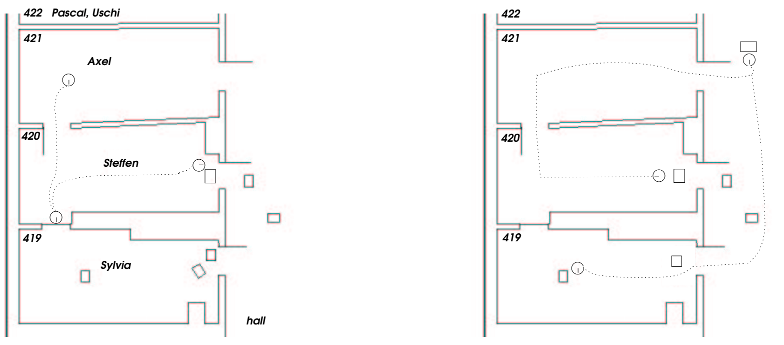

Fig. 2: Office environment consisting of four rooms (419, 420, 421, 423) and a hallway. *Left*: In room 420, the low-level *GoToPos*-actions fail as the way is blocked by a door and an obstacle. The only possible way now is to enter room 421. *Right*: Starting in room 420, on the way to room 422, the robot fails and attempts to go to 419 again.

people. *CallHelp* succeeds if the preconditions for the action *CallHelp* are met and if it is necessary in the current state to call help. *SearchFail* launches an email notification to the originator of a search request if none of the possible candidates has the desired item. If delivery or receipt is possible, the according actions are performed. If no stationary action is launched, i.e., all the previous state diagnostic predicates failed, the *Continue* predicate is invoked. Depending on the current state (e.g. the position), a rather complex path planning procedure is invoked and returns, if possible, a navigation action sequence (consisting of *Goto*, or several *Enter* and *GotoDoor* actions). If nothing works, the robot goes *Idle*.

If stationary actions are possible, then these are executed. The plan can either be a single action or an action sequence.

4 Example Scenarios

The robot control system has been used on a Pioneer 2 mobile robot. In the following, we present execution traces taken from office delivery tasks. The examples were recorded in a simulator only. However, the system's functionality has been tested on the real robot as well. We have taken great care to make the simulation environment as realistic as possible. We have used realistic sensor models for odometry, sonars, and laser. The operating environment is changed dynamically by opening and closing doors and by introducing obstacles that block the way of the robot.

Keeping track of multiple requests

In the following example, we want to demonstrate the functionality of the high-level planning system, in particular, its ability to keep track of multiple requests. In the usual mode of operation, the robot attempts to go to the nearest location first where a delivery or dialog action is supposed to be performed.

Take the office environment as depicted in the left-hand part of Figure 2. The robot is asked to deliver a book from Axel (room 421) to Sylvia (419). In addition, it received a request to search for another book amongst Pascal and Uschi (422), in the following denoted as `s_request` fluent. Initially, the robot is located in room 421. After having received the book, the robot is on its way to Sylvia. The corresponding execution trace of the high-level controller (FLUX) is printed below:

```
>State: [carries(book, sylvia), in_room(r421), at(axel),
 s_request([pascal, uschi], book_2, andreas), ...|Z1]
>Agenda: [goto(sylvia)]
```

In room 420, neither the attempt to reach room 419 directly nor the attempt to enter through the hallway are successful.

```
> *** Weak qual. occurred ***
>State: [ab(reachable(d6, r420)), in_room(r420), carries(
 book, sylvia), ...|Z2]
>Agenda: [gotodoor(d2), enter(hall), gotodoor(d1),
 enter(r419), r_goto(sylvia)]
> *** Weak qual. occurred ***
>State: [ab(reachable(d2, r420)), ab(reachable(d6, r420)),
 in_room(r420), carries(book, sylvia), ...|Z3]
```

Having to detour back to room 421, the robot postpones the delivery of Sylvia's book and heads for Pascal's room instead.

```
>Agenda: [gotodoor(d7), enter(r421), ... r_goto(pascal)]
...
>State: [in_room(hall), at(d3), ab(reachable(d2, r420)),
 ab(reachable(d6, r420)), carries(book, sylvia), ...]
>Agenda: [gotodoor(d4), enter(r422), r_goto(pascal)]
```

The robot fails on the way to Pascal because of another obstacle in the hall:

```
> *** Weak qual. occurred ***
>State: [ab(reachable(d4, hall)), in_room(hall),
 carries(book, sylvia), ...|Z4]
```

The path to Sylvia again seems to be shortest, so the robot decides to deliver the book to Sylvia, this time successfully.

```
>Agenda: [goto(sylvia)]
> *** SUCCESS ***
>State: [in_room(r419), at(sylvia), ...|Z2]
>Agenda: [deliver(book, sylvia)]
...
```

Global path planning is performed in collaboration between the high-level task-planner and the path planning module. FLUX determines the set of locations where the delivery, receive, or dialog actions are supposed to take place. The closest of all these locations is then found by repeatedly invoking the path planner. Afterward, the goal locations are passed on the low-level controller, where they are translated into target coordinates of the low-level *GoToPos* behavior. Please note the division of labor between the high and lower levels of control. While traveling to the requested goal position, the low-level controller attempts to avoid obstacles on its own. It will notify the high level only about exceptional situations if its own attempts to recover the situation have eventually failed.

Recovering from failure by finding explanations

The next example demonstrates how the robot recovers from an unsuccessful attempt to deliver an item. During a delivery, objects can get mixed up which comes to light when the original recipient does not find the desired item on the robot's tray. Usually, such a situation is detected only several action steps after the failure actually occurred, requiring the robot to reconsider all encountered states since the time of the failure.

Take the office environment shown in right-hand part of Figure 2. The robot's task is to deliver two books from Axel: *Book1* to Sylvia and *Book2* to Steffen. The path planner decides to go to Steffen first because it is closest according to the information on known obstacles and closed doors currently available. While traveling, small obstacles are successfully avoided by the low level behaviors; the high-level remains unnotified.

The first delivery apparently goes smoothly because the robots itself cannot check using its sensors whether the correct item was taken from the tray. The corresponding execution trace is shown below:

```
>State: [in_room(r420), at(steffen),
 carries(book_2, steffen), carries(book_1, sylvia),...|Z2]
>Agenda: [deliver(book_2, steffen)]
> *** SUCCESS ***
```

Execution is continued in the belief that *Book1* is still on the robots tray:

```
>State: [in_room(r420), at(steffen),
 carries(book_1, sylvia), ...|Z2]
>Agenda: [goto(sylvia)]
> *** SUCCESS ***
```

up to the point where the delivery of *Book1* is supposed to take place. Sylvia signals that delivery is impossible because her book is not in the robot's tray:

```
>State: [in_room(r419), at(sylvia),
 carries(book_1, sylvia), ...|Z2]
>Agenda: [deliver(book_1, sylvia)]
> *** Strong qual. occurred ***
```

a fact which might have several explanations: (1) a previous delivery failed (someone else took Sylvia's book, i.e., Steffen took *Book1* instead of *Book2*) or (2) the book simply got lost from the tray. Since these explanations are ordered in a respective preference list, the robot assumes (1). Hence, for recovery, the current state is calculated by assuming that Steffen took the wrong book (failure at the preceding delivery action), which means that Steffen's book is still on the robot's tray and Sylvia's book is at Steffen's.

```
>Abnormality occurred: [delivery(steffen, book_1)]
>State: [request(steffen, book_1, sylvia), in_room(r419),
 at(sylvia), carries(book_2, steffen), ...|Z3]
```

Having calculated this state, the recovery actions emerge automatically:

```
>Agenda: [goto(steffen)]
...
>State: [at(steffen), in_room(r420), request(steffen,
 book_1, sylvia), carries(book_2, steffen), ...|Z4]
>Agenda: [deliver(book_2, steffen)]
> *** SUCCESS ***
>State: [at(steffen), in_room(r420), ab(enterable(d6)),
 request(steffen, book_1, sylvia), ...|Z4]
>Agenda: [receive(book_1, steffen)]
> *** SUCCESS ***
>State: [carries(book_1, sylvia), at(steffen),
 in_room(r420), ab(enterable(d6)), ...|Z4]
>Agenda: [walk_on, gotodoor(d2), enter(hall),
 gotodoor(d1), enter(r419), r_goto(sylvia)]
> *** SUCCESS ***
...
>State: [at(sylvia), in_room(r419),
 carries(book_1, sylvia), ab(enterable(d6)), ...|Z4]
>Agenda: [deliver(book_1, sylvia)]
> *** SUCCESS ***
```

Note that recovery actions are not necessarily launched immediately but possibly at a later point in time due to efficiency reasons. If, however, no delivery happened after the missing object of interest was put back in the robot's tray, the first explanation is found to be false and the second explanation is chosen for further processing.

5 Conclusions and Future Work

The presented work demonstrates a successful combination of techniques from various subfields of robotics. In particular, we were able to integrate reactive control with high-level planning and reasoning for the sake of an increased reliability in performing complex delivery tasks. It can also be seen as an investigation to what extent logic in general and the fluent calculus in particular can contribute to real-world robotic applications.

Promising results could be documented in several aspects. First, high-level domain descriptions could be utilized for action planning in a domain where multiple and possibly complex delivery tasked are performed. Second, the use of the fluent calculus along with its augmentations assist during execution monitoring by means of providing explanations and corrective action planning in the case of execution failures.

The planning and reasoning component is embedded in a modular control architecture using an interface for interprocess communication. The robot control system can easily be extended with additional modules, e.g., for perceptual processing or user interaction.

Our approach is similar to the systems by [12, 8, 7] in the sense that all use high-level planning on real robots. Shanahan's Khepera robot based on the event calculus makes heavy use of abductive planning and explanations of failures. However, these failures are mainly due to the sensor limitations of this fairly simple robot. In contrast to our work, the high-level planner is not embedded in a complex modular architecture with clearly defined low-level behaviors such as obstacle avoidance and sophisticated localization techniques.

The control architecture used in our work is more comparable to the ones proposed in [8] and [7]. These, in turn, provide only hand-coded recovery procedures for failures of the current action. Undetected failures of earlier actions are not considered. Furthermore, the use of the fluent calculus allows us to model side effects of actions.

Future Work

Preference lists for action failures involve a great deal of speculation and need to be specified in advance. An alternative method would be a form of hypothesis testing: being left with a set of possible explanations for some action failure, each of them could be double checked by additional sensing/state verification. This topic is closely related to central questions in the field of active perception, for example, active vision. The application of a cognitive planner for sensing actions in the fashion outlined above could help in minimizing action effort and maximizing knowledge gain. Thus, it seems promising to formalize active vision domains within the fluent calculus. Furthermore, the static nature of the preference list could be overcome by learning from experiences. Please note, no general theory about the connection between high and low level has been established yet.

References

[1] C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proc. of the 17th Int. Joint Conf. on Artificial Intelligence (IJCAI-01)*, pages 690–700, 2001.

[2] G. Brewka. Adding priorities and specificity to default logic. In *Proc. of the European Workshop on Logics in AI (JELIA-94)*, volume 838 of *LNAI*, pages 247–260. Springer, 1994.

[3] J. L. Fernández and R. G. Simmons. Robust execution monitoring for navigation plans. In *Proc. of the Conf. on Intelligent Robots and Systems (IROS-98)*, 1998.

[4] D. Fox, W. Burgard, and S. Thrun. Markov localization for mobile robots in dynamic environments. *Artificial Intelligence Research*, 11:391–427, 1999.

[5] G. D. Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning*, pages 453–465, 1998.

[6] A. Großmann, S. Hölldobler, and O. Skvortsova. Symbolic dynamic programming with the Fluent Calculus. In *Proc. of the IASTED Int. Conf. on Artificial and Computational Intelligence (ACI-2002)*, pages 378–383, 2002.

[7] D. Hähnel, W. Burgard, and G. Lakemeyer. GOLEX - Bridging the gap between logic (GOLOG) and a real robot. In *Proc. of the 22nd German Conf. on Artificial Intelligence (KI-98)*, 1998.

[8] K. Z. Haigh and M. M. Veloso. High-level planning and low-level execution: Towards a complete robotic agent. In *Proc. of First Int. Conf. on Autonomous Agents*, pages 363–370, 1997.

[9] Y. Martin and M. Thielscher. Addressing the qualification problem in FLUX. In *Proc. of the German Annual Conf. on Artificial Intelligence (KI-2001)*, pages 290–304, 2001.

[10] N. J. Nilsson. Shakey the robot. Technical Note 323, SRI International, Menlo Park, CA, USA, 1984.

[11] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1-2):81–132, 1980.

[12] M. Shanahan. Robotics and the common sense informatics situation. In *Planning with Incomplete Information for Robot Problems: Papers from the 1996 AAAI Spring Symposium*, pages 95–106, 1996.

[13] M. Thielscher. The qualification problem: A solution to the problem of anomalous models. *Artificial Intelligence*, 131(1-2):1–37, 2001.

[14] M. Thielscher. Programming of reasoning and planning agents with FLUX. In *Proc. of the Int. Conf. on Principles of Knowledge Representation and Reasoning (KR-2002)*, pages 435–446, 2002.

[15] S. Thrun, D. Fox, W. Burgard, and F. Dellart. Robust Monte Carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2):99–141, 2000.