DRESDEN UNIVERSITY OF TECHNOLOGY
Department of Computer Science

Master's Thesis / Diplomarbeit

on the subject / zum Thema

A Layered Architecture for Robot Control using the Fluent Calculus

to receive the academic degree / zum Erreichen des akademischen Grades
Master of Science / Diplom-Informatiker

by / von

Andreas Henschel
born on the 20th of September 1974 in Leipzig

submitted the 18th of October 2002

Supervisor: Prof. Dr. Michael Thielscher
Second examiner: Dr. Axel Großmann

Contents

1 Introduction

1.1 Challenges of Robotic Environments

Mobile robotics is a very active and fascinating field of research for many reasons. It poses the challenge to transform a computer on wheels that is merely able to sense some physical properties into an intelligent agent that can identify features, recognize patterns, localize, construct maps, learn from experience and navigate. Thus, robotics virtually engages every subfield of artificial intelligence and demands lateral thinking. Computer science and engineering are core elements, but cognitive science, psychology and philosophy provide outlines, when it comes to intelligent behavior. In order to devise a comprehensive framework, an adequate robot architecture is needed, that allows for a synergetic cooperation in a scalable, modular fashion.

Cognitive Robotics, in turn, focuses on the ability to maintain a mental model of the world and drawing the right conclusions from it. The aim is to adopt insights from human-like perception and apply it to agent programming. Given the processed percepts from sensors in the form of logical sentences, the semantics of rational deliberation — commonly formalized within action theories — constitute the foundation for true autonomy: they enable knowledge-based agents to reason about general causalities and the effects of their own actions and thus depart from the paradigm of fixed, pre-encoded execution schemes (successfully applied examples for action theories are the Situation Calculus ([McC58, McC63]), the Fluent Calculus [Thi98], the Event Calculus [Sha97] and TAL [DGKea98]). The agent then is — to some extend — able to predict future states, and as a result ought to produce goal directed plans.

The need for action planning comes from the fact that it is impossible to program suitable action sequences for all possible tasks and situations in a typical robotic setup. Unforeseen events might happen, action execution might fail and robots cannot be assumed to have full control over the environment. In particular, autonomous robots hardly ever have complete information about the state of their environment. The executability of actions in the real world can never be predicted with absolute certainty. Reasoning robots need to constantly monitor their actions and explain and react to unexpected action failure.

On the level of formal logic, the so-called Frame Problem, the Qualification Problem and the Ramification Problem (dealing with the proliferation of frame axioms, exhausting preconditions and implicit consequences, resp.)[1] pose major challenges to action theories.

In the context of agents that act in a physical environment, the challenge for applying high level action theories becomes twofold: in addition to theoretical issues, further difficulties are introduced when the assumed level of abstraction clashes with properties inherent to the real world.

[1]These issues are examined in more detail in Section 3.3.

Traditionally, robot controllers were built in a top-down manner (see [NR95, Chapter 25.4]): a high level planner used to eject an abstract plan, neglecting the tedious work of real execution and assuming error-free interaction with a real world environment. Various experiments proofed this assumption to be too strong (see e.g. reports of Nilsson's Shakey [Nil84] or McCarthy's attempt to build a TV assembling robot) and led to the conclusion to take sensor and actuator errors more seriously. Apparently, much work has been done in the introduction and refinement of intermediate and low level layers.

1.2 Terminology

Terms, frequently used throughout this thesis, are defined below, although one should bear in mind a comment by Marvin Minsky (from "The Society Of Mind", [Min86]) pointing out the difficulty to grasp conceptions central to Artificial Intelligence:

> "It often does more harm than good to force definitions on things we don't understand. Besides, only in logic and mathematics do definitions ever capture concepts perfectly. The things we deal with in practical life are usually too complicated to be represented by neat, compact expressions. Especially when it comes to understanding minds, we still know so little that we can't be sure our ideas about psychology are even aimed in the right directions. In any case, one must not mistake defining things for knowing what they are."

The remark is of significant nature for the present work in many ways: reality induces a vast complexity that makes it hard to capture concepts in terms of mathematical descriptions.

Intelligence. Intelligence refers to behavior, and the measure is dependent on our own education, understanding and point of view. Humans are considered to be more or less intelligent, and consequently, machines that can do, what we do has to be considered intelligent as well.

Agents. The word is used for an entity that produces an effect intentionally.

Robots. The word stems from the Czech "robota", which means "forced labor". From the variety of definitions for robots, the one given in [NR95, Chapter 25] serves the purpose in the current setting:

> *"... active, artificial agents, whose environment is the physical world."* *Active*, as to rule out rocks, the *artificial* part rules out animals and the *physical* part rules out pure software agents or softbots.

Autonomy. Autonomous agents are those that make decisions on their own, guided by the feedback they get from their own physical sensors.

Real World. From a standpoint of environment classifications in Artificial Intelligence, the real world belongs to the hardest category of environments. It is characterized by some criteria, that distinguish it from other agents environments:

Inaccessible: Sensors allow only for partial surveillance (usually within a close neighborhood) and the input in general is imperfectly perceived. Chapters 3.2.1 and 3.3 are dedicated to the problem of incomplete knowledge.

Nondeterministic: The next state of the physical environment is in general not completely determined by the current state and the actions selected by the robot. It turns out that most real situations are so complex that whether they are really deterministic is a moot point; for practical purposes they should be considered as nondeterministic: wheels slip, batteries run down, and parts break, so you can not know for sure if an action is going to work.

Non-episodic: The effects of an action might change over time. So a robot has to handle sequential decision problems and learning.

Dynamic: The environment can change while the robot is performing an action or deliberating. Obstacles might appear and disappear, the state of doors, people, light conditions — in fact any perceivable aspect of the world is subject to change.

Continuous: In principle states and actions are drawn from a continuum of physical world configurations and motions. The Problem of continuity is central in the topic on representational matters, Chapter 2.

Execution Monitoring. Execution Monitoring is the robot's process of observing the world for discrepancies between the actual world and its internal representation of it, and recovering from such perceived discrepancies.[2]

Action Monitoring. Action Monitoring is similar to Execution Monitoring, but monitors are tied to the scope of single actions only.

State. A state represents a configuration (or a set of configurations) of the environment.

Actions. By means of actions the robot causes a transformation from one state into another.

[2] This definition is given in [GRS98].

1.3 Problem Definition

The goal is to analyze, develop and implement/simulate a distributed system for robust robot control, where the major concern is to *bridge the gap between cognitive robotics theory and real robot applications.*

In accordance to the initially mentioned broad range over several subfields within robotics, emphasis is drawn on a number of key points:

- The incorporation of the Fluent Calculus
- Execution Monitoring
- Definition of adequate module interfaces
- Organization of a tailor-made runtime-system.

These topics, however, stand in close interrelation to each other and are scrutinized in more detail below:

1.3.1 The Fluent Calculus

It will be demonstrated, why and how the Fluent Calculus can be seen as a solution to the problem of applying programming and reasoning methodologies for robots, which allow to abstract as far as possible from concrete physical platforms and specific environments, meanwhile still meeting the abovementioned requirements (wrt. incomplete knowledge, representation compatibility, the need for monitors, etc.)

1.3.2 Execution Monitoring

Monitors are organized in a hierarchical fashion throughout the layers of the robot architecture. Moreover, the Fluent Calculus will be used to abductively provide explanations for recognized action failures together with recovery planning, which is of particular relevance for execution monitoring.

1.3.3 Data compatibility

The architecture is organized in several layers, ie. sequentally arranged blocks of modules. These layers operate in state representations that differ in the degree of abstraction and hence are in principle incompatible to each other. Percepts are to be symbolized in order to make them comprehensible and in turn, high level commands need to be converted to code that is processible by the underlying machine.

1.3.4 Runtime system

Execution monitoring brings up the subject of temporal regulation for all kind of data flow. A communication protocol is to be defined that guarantees smooth perception conveyance throughout the layers.

1.3.5 Assumptions

Here the taken assumptions are specified in order to clearly restrict the scope of the task to the key points mentioned above.

It might be desirable in some robotic settings to take the so-called *tabula rasa* approach, i.e. to have a knowledge state that initially starts from scratch and increases gradually by learning the environment. Although learning tasks play an important role for robots, it is inadequate not to allow any kind of apriori knowledge in the current setting. For example, map data or names for people's offices are in principle learnable but will cause tedious work that deviates from the key points of this thesis. Hence, map data is assumed to be provided initially. Furthermore, the acquisition of statical knowledge about the domain is mocked up by means of first order logic clauses, which is a realistic assumption as it is obvious that the presence of a suitable information acquisition module would remedy this deficiency.

1.3.6 Sample Application: Delivery Robot

Under the given circumstances (existing hardware, available modules), a specific robot application will be implemented and is described in detail in Chapter 5. The utilization of the Fluent Calculus within a complex robot framework will be exemplified.

1.3.7 Evaluation Criteria

As pointed out by Adele Howe in [How99], modern robot results are difficult to compare, in fact, the comparison only intensifies the already difficult task of evaluation of AI systems: The nondeterminism of actions and the environment argue for repeated experiments, yet the difficulty to reproduce conditions make it hard to draw general conclusions. Comparative evaluation as in robot competitions is based on previously agreed on, measurable and usually objective criteria, but can reward hacks and persistence. In this thesis, I will argue for the system's performance in terms of

Scalability and Modularity - to what degree is the system extendable? Can the architecture cope with newly introduced modules?

Reliability - How robust is the robot?

Flexibility - what environment conditions can be changed without loss of reliability?

Efficiency - How fast and how costly are executions?

1.3.8 Overview

The structure of this thesis is organized as follows:

Chapter 2 (Representational matters). This chapter is dedicated to the representation of action space and the robots environment. A very general formalization is provided that serves as a reference throughout all representational attempts, that are discussed in later chapters.

Chapter 3 (Preliminary Work). Related work is investigated. Planning techniques, their advantages/disadvantages and their preferred task domains are discussed. Furthermore, one subsection contains a summary of previous work on execution monitoring. Finally, the Fluent Calculus (along with some relevant augmentations) is briefly outlined.

Chapter 4 (The General Framework). A generic robot control architecture and a communication structure is derived on the base of conclusions from Chapter 3. No assumption about any particular hardware configuration or any particular application is done. A general approach to execution monitoring is presented.

Chapter 5 (Implementation). This chapter sheds light on various implemental issues. In so doing, it follows a top down manner by first describing high level issues (axiomatization etc.) and then descending to lower layers of the architecture. The chapter ends up with sample runs in the extended delivery scenario.

Chapter 6 (Conclusions and Future Work). This chapter is a summary of the presented work in a broader scope. Performance according to some performance measures is discussed and promising paths for future work will be outlined .

2 Representational matters of State and Action Space

The intention in this section is to construct a formal framework without any commitment to a particular approach. The focus is put on abstract concepts for actions and the surrounding environment. Various derivations from the generic framework will be discussed in the chapter on planning techniques (Chapter 3.1).

In general, the environment of a robot can be formally described as a tuple $\langle \mathcal{S}, \mathcal{A} \rangle$, where

- $\mathcal{S}$ is a set of states of the world
- $\mathcal{A}$ is a set of actions

The state space describes all the possible configurations of the environment. The appropriate design of the state space is one of the major challenges in robotics and effects the choice of the corresponding planner. Due to the innate difficulty of continuous environments it is impossible to enumerate the set of possible actions. Both the configuration of the robots body and the locations of objects in physical space are defined by real-valued coordinates.

As a consequence, many of the classical logical planning algorithms are not applicable without modification. Furthermore, many decision theoretic approaches approximate the world by conversion to a discrete state space. In fact, much of the work in robot planning has dealt with methods to tame these problems intrinsic to continuity. Common techniques are:

Discretization of world states. The state space $\mathcal{S}$ is to be partitioned into a grid of world states, either in crisp or fuzzy cells (also referred to as cell decomposition). This approach proved to be successful in particular for robot navigation, where states consist basically of spatial information. A famous example for this representation is used for localization and navigation of the museum-guide robot Rhino, cf. [BFL+98]. The design of $\mathcal{S}$ is mainly determined by a granularity trade-off: the size of state space vs. accuracy, where the former is bounded by computational complexity.

Discretization of actions. Here, the choice of a set of basic parameterized operations is the central design question. Imagine, e.g. a robot only being able to turn multiples of 90° and step forward on a regularly tiled ground. The best known example of this kind is the "LMT-Approach", cf. [LMT84].

Coercion. Uncertainty of the world can be reduced by forcing the agent into a known state. If, for example, the robot is not sure about the exact position within a room, it navigates to an easy to reach reference point (like a corner, for example by means of wall following behaviors). Coercion is described in more detail in [NR95, Chapter 13.4].

Abstraction. Abstraction is a well suited tool for handling complexity. As an extension to discretization, it also allows the agent to neglect details of a problem which it may not have complete knowledge about. In the context of *complex* delivery tasks, certain properties of the environment can preferably be modeled as structured, discrete knowledge: the robot being loaded with an item, abstract spatial information (what room, what floor, the state of a door), dialogue information, etc. Still afflicted with uncertainty, the range of these values are naturally discrete.

Meanwhile the first two methods are adequate for robot motion, the latter two are used in classical logical planning systems, relying on the assumption that lower levels take care of details (see the discussion in [NR95, Chapter 13.4]). They allow for a compact representation within formal logic.

3 Preliminary Work

Instead of building a robot control system from scratch, state-of-the-art techniques for planning and execution monitoring are examined. A thorough comparison of planning strategies, the problem attached to each strategy and an account for the taken choices of the current approach will be presented in this section.

3.1 Survey of Planning Strategies

In the remainder, various planning techniques, relevant for gofer robot applications, are critically examined. The objective of this investigation is to find out, what planning system (including its appendant representation) best accomplishes which robot tasks. Remarks and final conclusions will justify the choice of employed planners. In addition, possible synergy effects (due to planner combination) along with the induced compatibility problems will be pointed out. Key criteria in this survey are:

Applicability: The application domain of planning strategies depends strongly on the type of problem to be solved (i.e., what is best to solve what task).

Scalability: One main requirement for robot planners is the ability to scale to complex real world tasks.

Performance: Attention is drawn on computational complexity in conjunction with accuracy, cost minimization and maximization success probability.

3.1.1 Representational tools

Kalman Filters

Kalman filtering assumes a continuous state space $\mathcal{S}$, which is described by real-valued parameters, that are usually distributed according to Gaussian distribution. Further assumptions are:

- each sensor suffers from unbiased noise
- each action can be described as a function of real values
- the new state is a linear function of the previous state and the action.

Successful applications were e.g. reported by Dietl et al. in [DGN01], who excessively used Kalman filtering for controlling a *Pioneer 2* robot geared to play robot soccer (RoboCup, middle-size league). It is of particular interest here because of the comparable hardware configuration.

Remarks. It is important to notice, that uncertainty also might range over discrete parameters. The robot's position could be likely to be, e.g, in front of the third office in the second floor or in front of the fifth office in the fourth floor. Critically viewed, the Kalman approach is problematic for these cases. In general, Kalman filters lack the expressive power of probability grids, see below.

Probabilistic methods

Bayesian methods are often used for state estimation in mobile robot navigation. A probability grid is attached to the state $\mathcal{S}$ and updated by means of Bayes' rule: Under a given history of sensor readings $s_1 \ldots s_n$, the probability of some position grid L depends on the apriori probability $p(L)$ and the probability that $s_1 \ldots s_n$ were read given L:

$$p(L|s_1 \ldots s_n) = \frac{p(s_1 \ldots s_n|L)\ p(L)}{\sum\limits_{L' \in \mathcal{S}} p(s_1 \ldots s_n|L')\ p(L')} \tag{3.1}$$

Formula 3.1 can be (computationally) simplified and forms the base for Markov localization as given in [FBTC98] and elaborated in Großmann and Poli [GP] for robots with sparse and noisy sensors. A similar technique is Monte Carlo localization (see [TFBD00]).

Remarks. The advantages of these approaches are, that they can handle ambiguities and express degree of belief. World states commonly range over spatial information only. On the other hand, probabilistic approaches lack expressive power: It is impossible, within the language of probability theory, to formulate beliefs such as "if any person is in his/her office, the computer of that person will be turned on with a probability of 0.7". Probability theory has no quantifiers ("any person") and no functions ("the computer of the person").

3.1.2 Behavior Based Planning

This approach is advocated by Rodney Brooks, who emphasizes the use of minimal internal state. His subsumption architecture [Bro86] uses a layered finite state machine. The idea is, that the overall design can be decomposed into small behaviors like wall-following, obstacle avoidance, single leg coordination and exploration. Every independent subprocess reacts on (its local copy of) the current sensor data and contributes to the emerging overall behavior.

Remarks. For tasks, in which robot actions mainly depend on current sensor input, the assumption "the world is its own representation" (Brooks) is appropriate. However, it remains to be proved how this methodology can scale up to more complex tasks other than demonstrating basic competences in the physical world. A delivery agent, that needs to keep track of a number of (possibly complicated) user requests, would perform poorly if it had no internal knowledge.

3.1.3 Decision Theoretic approaches

A Markov decision process (MDP) is a model of an agent interacting synchronously with the world. The agent takes as input the state of the world and generates as output actions, which themselves affect the state of the world. In the MDP framework, it is assumed that, although there may be a great deal

of uncertainty about the effects of an agent's actions, there is never any uncertainty about the agent's current state.
A Markov decision process extends the general framework (cf. Section 2) in the following way: It can be described as a tuple $\langle \mathcal{S}, \mathcal{A}, T, R \rangle$ where

- $\mathcal{S}$ is a *finite* set of states of the world;
- $\mathcal{A}$ is a *finite* set of actions;
- T: $\mathcal{S} \times \mathcal{A} \rightarrow \Pi(S)$ is the state transition function, giving for each world state and agent action a probability distribution over world states ;
- $R : \mathcal{S} \times \mathcal{A} \rightarrow R$ is the reward function, giving the expected immediate reward gained by the agent for taking each action in each state. A policy π is a mapping from $\mathcal{S}$ to $\mathcal{A}$ specifying an action to be taken in each state.

In this model, the next state and the expected reward depend only on the previous state and the action taken; even if we were to condition on additional previous states, the transition probabilities and the expected rewards would remain the same. This is known as the *Markov property* — the state and reward at time $t+1$ is dependent only on the state at time t and the action at time t

Markov decision processes serve as a basis for solving the more complex partially observable problems: In a realistic robot environment not only action effects introduce uncertainty about the successor state, but already the current state might be uncertain. The POMDP framework provides a systematic method for using memory of previous actions and observations to resolve the disambiguation. To this end, the formal notation of MDPs is extended by the robots perception, which is defined as a tuple $\langle \Omega, O \rangle$, where

- Ω is a finite set of the agents observations
- $O : \mathcal{S} \times \mathcal{A} \rightarrow \Pi(\Omega)$ is the observation function, yielding a probability distribution over possible observations, given an action and the resulting state.

In the (assumed) presence of a state estimator (that resolves the ambiguity about the state) the agent is aware of the most likely current state. Given a policy π, we attach to each state $s \in \mathcal{S}$ a value or utility V, expressed by the expected sum of rewards to be received at each future time step [3] :

$$V_{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right] \tag{3.2}$$

where r_t is the reward received on time step t of executing policy π in s and γ is the discount factor, that controls the reward in the future. The value V can be given in a recursive fashion

[3] below only MDPs with infinite horizon are shown, for details on finite horizon MDPs see Kaelbling ([KLC96]).

$$V_\pi(s) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')V_\pi(s') \tag{3.3}$$

In order to act optimally, the agent should perform the action a that maximizes $V_\pi(s)$, which is called the optimal policy (π^*):

$$\pi^*(s) = \underset{a \in A}{arg\ max}\, R(s,a) + \gamma \sum_{s' \in \mathcal{S}} T(s,a,s')V_\pi(s') \tag{3.4}$$

A simple iteration through state space and action space determines the optimal policy repeatedly for each time step, until the relative value gain is less then some ϵ (value iteration).

Remarks. The complexity of the value iteration algorithm is $\mathcal{O}(k \cdot |\mathcal{S}|^2 \cdot |\mathcal{A}|)$ (where k is the horizon of the MDP), which has several consequences. The complexity depends on the size of $\mathcal{S}$ and can thus be regarded as a counterpart to the frame problem. For performance reasons it is necessary to either prune the action search space (by modifying the algorithm) or make use of a more compact state space representation. Many applications of MDPs and POMDPs within mobile robotics focus on mere hallway navigation ([KLC96, SK95]). Large problems as in transportational logistic or manufacturing domains including various objects with multiple properties, where state spaces (represented in un-factored ways) are highly dimensional, easily turn to be intractable with standard MDP-techniques.

The logical representations most often used in planning have a number of advantages over the flat state space representations associated with MDPs and POMDPs. The main advantage comes from their compactness — just as with operator schemata (see STRIPS). Logical representations can be exponentially more concise than a fully expanded (grounded) statebased transition matrix for an MDP. Algorithms for manipulating compact (or factored) POMDP have begun to appear— this is a promising area for future research. At present, however, there is no evidence that these algorithms result in improved planning time significantly over the use of a “flat” representation of the state space.

The philosophy of MDPs is based on a somewhat pessimistic underlying assumption, that distinguishes it from classical AI planning: action effects are in principle *unpredictable*[4] but observable. Moreover, actions in the MDP context usually have a very restricted amount of effects.

3.1.4 Classical AI Planning

Symbolic logical reasoning is regarded as the classical attempt for AI Planning. It equips the agent with the ability to consider abstract plans using goal-directed search, thus providing a massive increase in problem solving power. The Fluent

[4] Unpredictability in the sense that an action can cause a state transition to virtually any state with a certain probability (possibly 0).

Calculus as one main component in our robot architecture, stands in the tradition of classical action theories and thus inherits its basic characteristics from there.

STRIPS

In the STRIPS language ([FN71]), states are represented by conjunctions of first order logic terms. State transitions are denoted with an operator name, a set of preconditions and the action effects. The use of STRIPS (together with theorem provers) pioneered robot development as it was implemented on Shakey ([Nil84]), the first general-purpose mobile robot.

Situation Calculus

The Situation Calculus is the most prominent representative, and has been applied to robotics and extended in various ways. World changes are described in first order logic. Dynamic relations and properties (fluents) are expressed with predicates, augmented with a situation argument. The representational frame problem is solved with successor-state axioms: A fluent holds in a situation, roughly speaking, if it was true before and the current action preserved it, or it was one of the positive effects. Based on the Situation Calculus, high-level robot programming is accomplished by means of the language GOLOG ([LRL$^+$97]).

In order to bridge the gap between the abstract language of GOLOG and a real robot, GOLEX ([HBL]) was developed and applied in the mentioned museum guide. It demonstrated the successful use of logic to control a robot in real world domains.

Remarks. For planning tasks, situation calculus applications often employ the heavy apparatus of theorem provers (see blocks world examples, [FPR00]). The inferential frame problem remains unsolved within the situation calculus, which leads to inefficiency in state estimation as soon as the problem scales up in number of fluents and action sequence length, as demonstrated in the blocks world example of [Thi98]. For a thorough efficiency analysis, the reader is referred to [Thi02b].

Fluent Calculus

The simple Fluent Calculus [Thi99] combines, in pure classical logic, the Situation Calculus with STRIPS-like action operator specifications. The Fluent Calculus (see Chapter 3.3 for a comprehensive description) can cope with the Situation Calculus in terms of expressive power, moreover it solves (along with its augmentations) both the representational and the inferential frame problem, the ramification problem and the qualification problem. While benefitting from experiences made with the Situation Calculus, the application of the Fluent Calculus promises to exceed the success due to the mentioned advantages. Various extensions for the Fluent Calculus exist. Those related to the current application are dealing with the qualification problem, ramifications, knowledge and sensing, conditional planning and continuous change (again, see Chapter 3.3)

3.1.5 Conglomerate Strategies

At present it is a hot research topic to construct a synthesis of logic planning and probabilistic reasoning. MDP-style planning has been at the core of a range of most real robot applications. Search-based approaches for solving MDPs can use heuristics, learning, sampling and pruning to improve their efficiency ([BBS95, GB98]). Declarative search control knowledge, used successfully in classical planning, might also be used for pruning the search space.

Several mixed planning techniques have been suggested, e.g. a combination of STRIPS and MDPs in [BDH99]. Boutilier et al. present in [BRST00] an extension to GOLOG where MDP-planning is intermingled with high level programming. GOLOG's control structures are merely used for programming purposes, though.

3.1.6 Conclusions for the Planning System

Motivated by the discussion above, we here outline a planner design that is well-suited for complex gofer robots in general and for the delivery robot at hand in particular.

Abstract agent design assumes that the actions returned by the planner are directly executable in the environment. The discussion of abstraction automatically leads to the issue of the connection between plans and physical actions. At the moment, there is no general theory for mapping abstract logical descriptions into actual robot behavior. High level planning methods are severely limited by its inability to model stochastic domains for applications within robotics. We here attempt to overcome these flaws by use of a distributed planning system, which is arranged in a hierarchical structure, as depicted in Figure 1. Subsystems can take an action from the plan and generate a sequence of primitive actions to be carried out. The subsystem might, for example, generate a speech signal from an utterance description returned by the planner. Note, that the subsystem itself might be a planner of some sort; its goal is to find a sequence of lower-level actions that materialize the effects of the higher-level action.

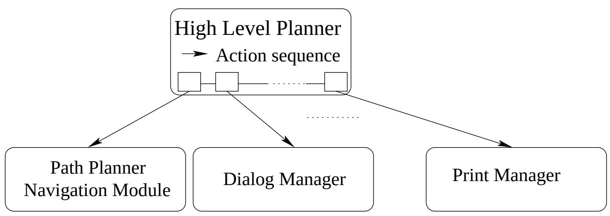

Figure 1: Hierarchical planning system

The modular planning subsystems maintain state representations, that are adequate for their range of planning problems. In our delivery scenario, lower levels keep track of precise positions, speed, elapsed time, movement characteristics, etc ($\mathcal{S}_{spatial}$). The states of fluents, on the other hand, are registered in upper layers ($\mathcal{S}_{fluent}$). Nevertheless, the split also implies the challenge of information exchange between the layers.

Reactive motion control

Reactive motion control is best realized by several parallel behaviors (which bases in the ideas outlined in Chapter 3.1.2), each one governed by a set of fuzzy logic rules. Internal knowledge is kept minimal.

Localization

Probabilistic state representations as presented in Chapter 3.1.3 serve as a base for localization and navigation methods, thus overcoming the problems inherent to stochastic domains. On the other hand, state space explosion is avoided by separating it from world knowledge about high level request solving.

3.2 Execution Monitoring

Execution Monitoring has been studied by AI researchers (see e. g. [FS97] and [AIS88]). However, robot system design is a very active field of research and no well-approved standard theory has been established, yet.

Execution Monitoring can be defined as an agents process of identifying discrepancies between observations of the actual world and the prediction and expectations derived from its representation of the world, classifying such discrepancies and recover from them.[5]

The goal is to devise a method that is able to cover for as many types of unexpected situations as possible, while at the same time equipping the robot with tools that allow for successful recovery.

In the sequel, various existing approaches for Execution Monitoring are investigated. Some key ideas are critically reviewed and adapted.

3.2.1 Planning vs. Replanning

The following discussion is inspired by [NR95, Chapter 13]) and [AIS88]. It is of great relevance for the presented approach to robot control.

The assumptions for flawless planning and execution are that the world is accessible, static and deterministic. Clearly, in a real world domain these assumptions are usually violated, which leads to the problem of handling incorrect and incomplete information of the surrounding environment. In general there are two ways to deal with this:

[5] Definition according to [Bjä98]

Conditional Planning: Which deals with incomplete information by constructing a conditional plan that accounts for each possible situation that could arise. By including sensing actions the agent decides at runtime what branch of the plan to take.

Execution Monitoring: Instead of executing a constructed plan blindly, the agent can monitor what is happening while it executes the plan. Upon failure detection, it can then do replanning to find a way to achieve its goals from the new state. This topic is discussed in the subsequent Section 4.3.

The two approaches are related in the sense that the former takes into account the possible conditions that constitute execution errors at planning time. The latter approach, on the other hand, is deferring the job of dealing with these cases until they actually arise (at execution time).

The successful combination of the two approaches should remedy for the drawbacks of the respectively other strategy:
Conditional planning at the one extreme of the continuous spectrum – involving nondeterministic actions and sensing – leads to exponential state space explosion in the number of steps of the developed plan. Given that only one state will occur, this seems rather wasteful as well as impractical. Many of the events being planned for have only an infinitesimal chance of occurring. Besides, it seems tedious, if not impossible, to cover for all contingencies of plan execution in advance, even if all of them were predictable. Obviously, a conditional planner can not cope with so called *unforeseen* situations either.

Postponing failure recovering completely to an execution monitoring and replanning module is disadvantageous in cases, where recovery is very difficult. E.g., imagine an agent that is to drive a car to a certain goal. The whole conception of a spare tire in order to fix a puncture during the trip is due to cautious conditional planning. A pure replanning agent who did not plan for puncture initially, would — troublesome enough — be faced with a long walk, since it did not see the need for the spare tire. Thus, starting to troubleshoot when the damage is already caused can be cumbersome and expensive.

Also, consider nondeterministic actions with different postconditions of approximately equal probability, as it is the case in the elevator example of [Thi00b] (after pressing the call-button, one out of n elevators appears). A conditional plan neatly and provably leads to the goal state (getting into one of the elevators to appear) by means of sensing and an according conditional action. An agent with a purely sequential action execution together with some recovery strategy could easily get confused; as well as it seems unnatural to regard a closed elevator door as a "failure". Furthermore, in the case of unknown door states in the roaming robot scenario, it would be a rather inelegant (although feasible) approach to let the robot try to pass through the door, risking to bump into a closed door and then to start replanning. Imagine, sensing the state of the door was possible from any location of the room/environment (for example by assistance of auxiliary surveillance agents). In contrast to the trial-and-error agent, the "sensible" Look-ahead agent could predict failures earlier and thus save time and cost.

The way of designing an *intermediate* agent is to specify disjunctive state update axioms for all *reasonably likely* outcomes. Those eventualities, that are unlikely or easy to deal with, are left to the responsibility of the execution monitoring module.[6]

In addition, the recovery module has access to the complete plan structure and notices when unexpected events remove preconditions of a future plan step. In this manner, planning and execution monitoring can be interleaved, thus giving rise to a powerful control strategy: integrated planning, execution and monitoring. It has been argued that this method could considerably bring down the time lost in planning, especially in dynamic domains.

3.2.2 Hierarchical Structure of Monitors

Simmons et al. ([FS97]) suggest an architecture that employs an execution monitoring scheme which is distributed over several layers. Monitors are organized in a hierarchical manner, structured in layers of increasing specificity.

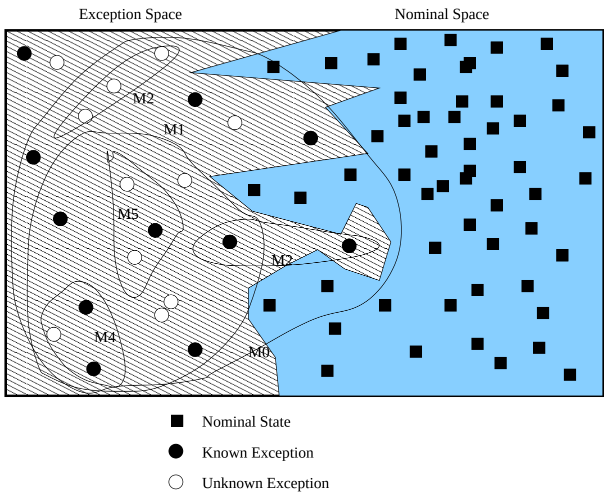

Figure 2: Exceptional and nominal states. Monitors range over subspaces of the exceptional state space

For example, timeout-monitors are of very general applicability. As demonstrated in the extended delivery scenario, a wide range of activities can be monitored with timeout. However, general monitors have its drawbacks too: they might range wrongly over nominal states, as it is the case for timeouts, when

[6]The distinction between bounded and unbounded indeterminacy is subject to domain comprehension and depends on likelihoods and costs of various outcomes, domain complexity and domain dynamics. Planning should furthermore minimize costs and maximize executability likelihood.

an action takes longer then estimated due to unexpected difficulties (dynamic obstacles on the robots path).

Certain stall-recognition techniques, on the other hand, are examples of higher specificity in the sense that they fire only in rare occasions. Usually, they respond faster and are more precise.

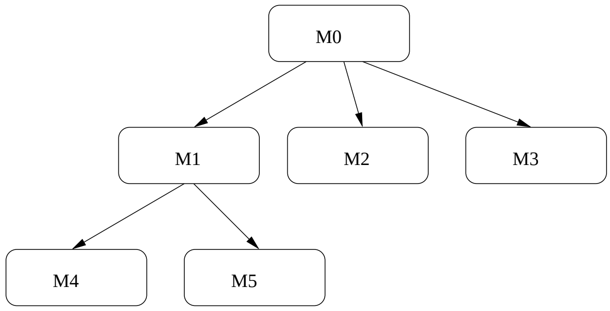

Figure 3: Monitor hierarchy according to Figure 3.2.2

Remarks Notice, that there are several problems associated with this approach: One disadvantage is the fact that both monitors and recovery procedures are *hand-coded*. Ideally, robots develop monitors autonomously, e.g. by learning. This topic is not pursued here, see instead [GP98]. Secondly, the approach does not make use of symbolic world models. Hence, high level execution monitoring, as described in the subsequent chapter, is not applicable in a straightforward way.

3.2.3 High Level Execution Monitoring

It has been argued, that some parts of execution monitoring can be performed on a cognitive level. A symbolic world representation and causalities of actions in general and the presented work on explanation provision by means of default logic in particular promise to support recovery from failures.

Giacomo et al. demonstrate the utilization of high level execution monitoring in ([GRS98]):

"The robot is executing a program on-line. By this, we mean that it is physically performing actions in sequence, as these are specified by the program. After each execution of a primitive action or of a program test action, the execution monitor observes whether an exogenous actions has occurred. If so, the monitor determines whether the exogenous action can affect the successful outcome of its on-line execution. If not, it simply continues with this execution. Otherwise, there is a serious discrepancy between what the robot sensed and its internal world model. Because this discrepancy will interfere with the further execution of the robot's program, the monitor needs to determine corrective action in the form of another program that the robot should continue executing on-line instead of its original program. So we will understand an execution monitor as a mechanism that gets output from sensors, compares sensor measurements with its internal world model and, if necessary, produces a new program whose on-line execution will make things right again."

In this approach, programs are specified in GOLOG (see [LRL+97]), a logic programming language that is based on the Situation Calculus ([MH69]) and whose primitive actions are those of a background domain theory. Logical definitions (together with straightforward lifts to a prolog implementation) for state transitions, monitors and recovery predicates are provided. On that base, Giacomo et al. developed a cautious online interpreter for GOLOG programs, where "cautious" is used in the sense of an offline check for termination of the remaining action sequence.
Monitors have the logical definition:

$$\begin{gathered} \mathit{Monitor}(\delta, s, \delta', s') \equiv \exists\ \mathit{exo}.\mathit{Do}(\mathit{exo}, s, s') \wedge \\ [\exists\ s''.\mathit{Do}(\delta, s', s'') \wedge \delta = \delta' \vee \neg\exists s''.\mathit{Do}(\delta, s', s'') \wedge \mathit{Recovery}(\delta, \mathit{exo}, s', \delta')] \end{gathered} \tag{3.5}$$

Monitor checks for the existence of an exogenous program *exo*, determines the situation s' reached by this program, and if the monitored program δ terminates offline, the monitor returns δ, else it invokes a recovery mechanism to determine a new program δ' (second part of the disjunction).

Critical Review

In order to be able to prove the *Do*-predicate with the existentially quantified variable *exo* in Equation (3.5), an answer substitution for *exo* could hardly be deduced, if s and s' were given. The approach (therefore) builds on the strong assumption, that *exo*, i.e. the program of exogenous events, is given. As Giacomo et al. state themselves:

> "On the face of it, this idealization seems dubious in practice. One can argue, that agents never observe action occurrences - Fido ate the sandwich - only their effects - the sandwich is no longer on the table. One can reconcile both points of view by posing the observation of action occurrences as an abduction problem: Given an observation of one or more effects, hypothesize an exogenous action occurrence to account for the observation. This is the implicit stance we take in this paper when we suppose that the robot has "observed" an exogenous action occurrence. By doing so, we are glossing over all the obvious technical problems associated with realizing this abductive point of view."

The problems do not remain of technical nature, if one realistically assumes, that only a subset of effects (of exogenous events) are perceivable. The approach neglects the source of nondeterminism, introduced by abductive reasoning: there could be many reasons for not sensing the sandwich. Moreover, robot actions are unrealistically assumed to be perfect. In the light of these assumptions, one should notice that the implementation is kept on a rather abstract level: a softbot blocks-world domain combined with user interaction (for specifying all exogenous events). Porting to real robots will require an approach, that takes noisy effectors more serious.

Conceptually speaking, the approach has the disadvantage that one monolithic execution monitor has to cover all levels of computation and thus is not compatible with a hierarchical modular structure for robot control systems. It shifts the complexity from robot programs to a complex execution monitor.

3.3 The Fluent Calculus, Augmentations and FLUX

3.3.1 Fundamental Definitions

The Fluent calculus is a second-order logic specification language with sorts, it is used for axiomatizing dynamic environments and reasoning about actions. It roots in the logic formalism of [HS90], but was developed to solve the well known representational and the inferential frame problem.
The proliferation of frame axioms, i.e. axioms, that describe how the world stays the same (as opposed to how it changes) is called the *representational* frame problem. Originally, it centered on the apparently unavoidable need for a large number of frame axioms that made for a very inelegant and inefficient description of actions. The *inferential* frame problem concerns the way, conclusions are drawn about the world, which essentially means not to apply separate inference steps for each uneffected piece of knowledge. In the Fluent Calculus these non-effects are computed efficiently by only specifying the effects of an action in terms of how this action modifies the state of the environment. This way, the Fluent Calculus both overcomes the representational and the inferential frame problem.

The Fluent Calculus allows to solve planning problems where we have incomplete state knowledge. Its basic ingredients are *Fluents, States, Situations* and *Actions*, where the latter two are directly inherited from the Situation Calculus:

- *Fluents*
  A fluent represents a single atomic property of the physical world which may be manipulated by the agent. Fluents are terms in the language with the reserved sort FLUENT.
  In a delivery scenario, the fluent $Request(P_1, O, P_2)$ where *Request* is a ternary function symbol, denotes that there is a request to bring object O from person P_1 to person P_2.

- *States*
  A state is a snapshot of the environment at a certain moment, they are of the reserved sort STATE. States are identified with the fluents that hold. A special function $\circ$ is used to compose state terms, i.e. this function maps two states into a new state, in which the fluents in both arguments holds. In the delivery scenario the state

  $$Request(P_1, O, P_2) \circ Carries(O) \tag{3.6}$$

  denotes the state where the robot carries object O which is supposed to be brought from person P_1 to person P_2.

  There is also the predefined state constant $\emptyset$ which denotesthe empty state, i.e. no fluent is true. A fundamental notion is that of a fluent to *hold* in a state. Fluent f is said to hold in state z can be decomposed in two states one of which is the singleton f. Conversely, f does not hold in z if the latter cannot be decomposed in this way. For notational convenience, the macro $Holds(f, z)$ is used as an abbreviation for the corresponding equality formula [7]

  $$Holds(f, z) \stackrel{\text{def}}{=} (\exists z')z = f \circ z' \tag{3.7}$$

  This fundamental notion of truth and falsity of fluents in states requires a special theory of equality of state terms. The foundational axioms for associativity, commutativity, idempotency and the unit element (ACI1-unification) are given in [Thi00a] and shall be referred to as $\sum_{state}$.

- *Actions*
  An action is a first order term of the reserved sort ACTION consisting of an action function symbol and its arguments.
  In the delivery scenario, the action of receiving object O from person P is denoted by the action term $Receive(O, P)$.

  Actions can have preconditions, the Fluent Calculus provides the predefined predicate *Poss*: ACTION $\times$ STATE. The first argument is an action which is possible in the state denoted by the second argument.

[7]Free variables in formulae are assumed universally quantified. Variables of sorts ACTION, FLUENT, SIT and STATE shall be denoted by the letters a, f, s and z, respectively.

Let A be a function symbol with range ACTION. A simple action precondition axiom for A is of the form

$$Poss(A(\vec{x}), z) = \Pi_A(\vec{x}, z) \tag{3.8}$$

where $\Pi_A(\vec{x}, z)$ is a pure state formula in z.

- *Situations*
  A situation is a first order term of the reserved sort SIT and they are identified with the sequence of the actions performed so far. These sequences are represented using a binary function symbol Do, the term $Do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. In the delivery scenario the situation term[8]

$$Do(Deliver(O, P_2), Do(Goto(P_2), Do(Receive(O, P_1), S_0))) \tag{3.9}$$

denotes the sequence of actions

$$[Receive(O, P_1), Goto(P_2), Deliver(O, P_2)] \tag{3.10}$$

The initial situation is usually denoted by the special constant S_0.

3.3.2 Axiomatizing the domain theory

The domain theory is axiomatized in the Fluent Calculus with the following five classes of axioms

- *State Constraint* is a formula

$$(\forall s)\Gamma\{Z/State(s)\} \tag{3.11}$$

where $\Gamma(Z)$ is a first order formula with free variable Z and without occurrence of SIT terms.

- *Precondition Axioms.* For an action A($\vec{x}$) is a formula

$$(\forall \vec{x})(\forall\, s)(Poss(A(\vec{x}), s) \equiv \Pi\{State(s)\} \tag{3.12}$$

where $\Pi(z)$ is a first order formula with free variable z and without occurrence of SIT terms.

[8]The actions *Deliver* and *Goto* should be self-explanatory, however, see Section 5.3 for a brief description

- *State Update Axioms.* Reasoning about actions essentially means reasoning about the effects of performing them. A crucial advantage when specifying effects lies in the fact that actions almost affect very few fluents and thus leave most of a state unchanged. Actions should be describable by saying which fluents under which circumstances are changed by each action, while most of the other fluents are not explicitly mentioned and assumed to remain unchanged. On the other hand, it is precisely this desire for "focused" action specification that brings about the famous Frame Problem, which is recognized as one of the most important modeling problems in computer science. The Frame Problem is to find a representation formalism which allows for specifying actions solely in terms of effects in a way that all unchanged knowledge about a state still follows about the successor state. This representational aspect of the Frame Problem is linked with an inferential perspective concerned with the efficient computation of non-changes, which essentially means to apply separate inferences steps for each unaffected piece of knowledge. The Fluent Calculus approach to the Frame Problem exploits the explicit notion of states. Change is modeled by specifying the difference between two states. The simple Fluent Calculus is restricted to deterministic actions which are performed in isolation and which have a bounded number of direct and no indirect effects. Positive effects are modeled by adding them to a state, negative effects are modeled by removing them. For a (nondeterministic) action A($\vec{x}$) a state update axiom is a formula:

  $$\begin{aligned}&Poss(A(\vec{x}), s) \supset \\ &\quad (\exists \vec{y_1})(\Delta_1\{z/State(s)\} \wedge State(Do(A(\vec{x}), s)) = (State(s) \circ {\vartheta_1}^+) - {\vartheta_1}^-) \\ &\qquad \vee \ \ldots \ \vee \\ &\quad (\exists \vec{y_k})(\Delta_k\{z/State(s)\} \wedge State(Do(A(\vec{x}), s)) = (State(s) \circ {\vartheta_k}^+) - {\vartheta_k}^-)\end{aligned}$$

  where k $\geq$ 1 $and{\vartheta_i}^+$ and ${\vartheta_i}^-$ are respectively, the positive and negative effects of A under condition Δ_i .

  each $\Delta_i(\vec{x},\ \vec{y_i},$ z) is a first order formula with free variables among $\vec{x}$, $\vec{y_i}$, z and without occurrence of SIT terms.

  ${\vartheta_i}^+$ and ${\vartheta_i}^-$ are STATE terms of the form $F_1(\vec{\tau_1}) \circ \ldots \circ F_m\ (\vec{\tau_m})$ with free variable among $\vec{x}, \vec{y_i}$ $(m \geq 0)$.

- *Auxiliary axioms.* Including uniqueness of names and domain closure

  - **Uniqueness of names**

    $$UNA[h_1, \ldots, h_n] \stackrel{\text{def}}{=} \bigwedge_{i<j} h_i(\vec{x}) \neq h_j(\vec{y}) \wedge \bigwedge_{i} [h_i(\vec{x}) = h_i(\vec{y}) \supset \vec{x} = \vec{y}] \tag{3.13}$$

- **Domain closure**

$$(\forall\, a : \textsc{Action})(\exists\, g, h)a = \mathit{Receive}(g, h) \vee (\exists\, f)a = \dots \tag{3.14}$$

The axiom guarantees minimization of the actions of the given domain.

- *Foundational axioms.* They mainly consist of
  - Initial state

$$\neg s < S_0 \tag{3.15}$$

  - Successor situation

$$s < \mathit{Do}(a, s') \equiv \mathit{Poss}(a, s') \wedge s \leq s' \tag{3.16}$$

Various extensions of the Fluent Calculus exist ([Thi00c, Thi01a, MT01]). Relevant for the current application are:

- *Ramifications* (handled in Section 3.3.3).
- *The solution to the Qualification Problem* (which is the central topic of Section 3.3.4).
- *Continuous Change.* As demonstrated in the conveyer belt example of [Thi01a] or for mobile agents in the traffic world [HT99], continuously changing fluents (and consequently natural actions) can be modeled within the Fluent Calculus under certain circumstances. The robots real coordinates are subject to continuous change and could – in theory – be described with continuous fluents as a function of time. Grosskreutz follows this approach using ConGolog, see [Gro99]. However, in the presence of a behavior-controlled navigation subsystem, the evolving trajectory is influenced by many runtime events (dynamic obstacles), as several parallel low level behaviors all compete for control. Fluents for robot position are hence unpredictable and tedious, if not impossible to describe in a mathematical way in advance. This kind of state information is conveniently handled with probability techniques, while at high level less detailed position information (rooms, areas) is sufficient for reasoning about (high level) actions.
- *Knowledge and Sensing* is described in detail in [Thi00c]. A structural change to the standard approach is done and is justified in Section 5.4.

3.3.3 Ramification

Ramification is the ability to reason about indirect effects. A specification of update by means of explicit collections of positive and negative effects, as in simple (disjunctive) state update axioms, always formalizes the entire effect of an action. It is however important to distinguish between direct changes and further indirect effects:

- Firstly, axiomatizations of large domains can be structured much better, hence be made more concise and elaboration tolerant.
- Secondly, actions may cause unbounded chains of effects, which are generally difficult to summarize in a direct formalization of update.

As an example, consider the possibility that objects are somehow attached to other objects, like a cable being plugged into an adapter which is in turn connected to an Ethernet card. The effect of picking up, say, the card is then not only to carry the card itself but also to carry the adapter, hence also the cable. Likewise dropping one of them means to get rid of the other at the same time as indirect effect, which may in turn cause the completion of a request concerning the attached objects. Since attachment between objects may vary freely, the effect of moving things cannot be specified as a fixed effect of an action.

The challenge of extending solutions to the Frame Problem so as to cope with indirect effects is known as the Ramification Problem. The approach of the Fluent Calculus is to infer indirect effects with the help of causal chains: Starting with the direct effects of an action, further changes are successively determined on the basis of a causal relation. The various causal relations of a domain are

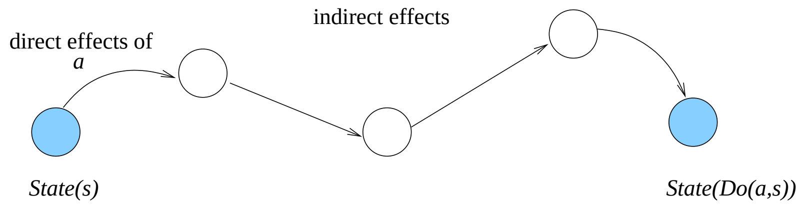

Figure 4: Ramification: initial and successor state are depicted as filled circles, intermediate states are simple circles

axiomatized with the help of the expression $Causes(\varepsilon, \varrho, z)$ [9]

where ε (the triggering effect) and ϱ (the ramification) are possibly negated atomic fluent formulas and z is a state. The intuitive meaning is that in state z, indirect effect ϱ is triggered by the preceding effect ε. As an example, consider the fluents *Attached*: OBJECT × OBJECT → FLUENT and *Carries* (see above) along with the constraints

$$\begin{aligned} Holds(Attached(x,y),s) &\equiv Holds(Attached(y,x),s) \\ Holds(Attached(x,y),s) &\supset [Holds(Carries(x),s) \equiv Holds(Carries(y),s)] \end{aligned}$$

[9] $Causes/3$ represents the reified version of $Causes/4$ which keeps track of the corresponding situation term. The latter is used in the remainder of this section.

Some dynamic aspects of a delivery scenario are expressed by the exemplary ramification axioms given below:

$$
\begin{aligned}
& \mathit{Holds}(\mathit{Attached}(x,y),z) \supset \\
& \quad \mathit{Causes}(\mathit{Carries}(x), \mathit{Carries}(y), z) \\
& \mathit{Holds}(\mathit{Attached}(x,y),z) \supset \\
& \quad \mathit{Causes}(\neg \mathit{Carries}(x), \neg \mathit{Carries}(y), z) \\
& \mathit{Holds}(\mathit{Request}(r_1,x,r_2),z) \supset \\
& \quad \mathit{Causes}(\neg \mathit{Carries}(x), \neg \mathit{Request}(r_1,x,r_2), z) \\
& \mathit{Holds}(\mathit{InRoom}(r),z) \wedge r_1 \neq r_2 \supset \\
& \quad \mathit{Causes}(\neg \mathit{Request}(r_1,x,r_2), z), \mathit{Request}(r,x,r_2), z)
\end{aligned}
$$

If x and y are attached, then the positive effect $\mathit{Carries}(x)$ causes the indirect effect $\mathit{Carries}(y)$. Conversely, if again x and y are attached, then the negative effect of $Carries(x)$ becoming false causes $\mathit{Carries}(y)$ to not hold too. Dropping an object incidentally cancels a possible request concerning this object. If the object is dropped in the wrong office then a new request is caused, which asks for taking the object from its current location to the desired one.
The distinction between a context of causal relation and a triggering effect (which represent a kind of momentum) is essential when considering intermediate states. To appreciate this, consider a state z in which two objects a and b are attached and neither is carried. If then $\mathit{Carries}(a)$ occurs as a direct or indirect effect, we expect $\mathit{Carries}(b)$ to be additionally caused, that is,

$$z \longrightarrow z \circ \mathit{Carries}(a) \longrightarrow z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b) \tag{3.17}$$

On the other hand, if a and b are attached in $z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b)$ and $\mathit{Carries}(b)$ becomes false, then $\mathit{Carries}(a)$ is expected no longer to hold, that is,

$$z \circ \mathit{Carries}(a) \circ \mathit{Carries}(b) \longrightarrow z \circ \mathit{Carries}(a) \longrightarrow z \tag{3.18}$$

Since the intermediate states in (3.17) and (3.18) are identical, getting the indirect effect right relies on knowing the preceding effect. The expression *Causes* is just syntactic sugar to allow for convenient specifications of cause-effect pairs.

A tuple $S \cup \langle \mathit{Ramify}, \mathit{Causes} \rangle$ is a *fluent calculus signature with ramifications* if S is a fluent calculus signature including the following predicates

$\mathit{Causes} : \textsc{State} \times \textsc{State} \times \textsc{State} \times \textsc{State} \times \textsc{State} \times \textsc{State} \times \textsc{Sit}$
$\mathit{Ramify} : \textsc{State} \times \textsc{State} \times \textsc{State} \times \textsc{State}$

An instance $\mathit{Causes}(z_1, e_1^+, e_1^-, z_2, e_2^+, e_2^-, s)$ means that if state z_1 is the result of positive effects e_1^+ and negative effects e_1^-, then an additional effect is caused which leads to state z_2 (now the result of positive and negative effects e_2^+, e_2^-, resp.) in situation s. Collections of effects are terms of sort State, they should not be viewed as corresponding to an actual complete state of the world. The *Causes* predicate is used to define the macro $Causes(\varepsilon, \varrho, z, s)$ as follows

$$
\begin{aligned}
& Causes(f, f', z_1, s) = \\
& \quad (\forall e^+, e^-) Causes(z_1, e^+ \circ f, e^-, z_1 \circ f', e^+ \circ f \circ f', e^- - f', s) \\
& Causes(f, \neg f', z_1, s) = \\
& \quad (\forall e^+, e^-) Causes(z_1, e^+ \circ f, e^-, z_1 - f', e^+ \circ f - f', e^- \circ f', s) \\
& Causes(\neg f, f', z_1, s) = \\
& \quad (\forall e^+, e^-) Causes(z_1, e^+, e^- \circ f, z_1 \circ f', e^+ \circ f', e^- \circ f - f', s) \\
& Causes(\neg f, \neg f', z_1, s) = \\
& \quad (\forall e^+, e^-) Causes(z_1, e^+, e^- \circ f, z_1 - f', e^+ - f', e^- \circ f \circ f', s)
\end{aligned}
\tag{3.19}
$$

which has the intended meaning that in situation s and state z, effect ε triggers indirect effect ϱ.

The second predicate introduced, $Ramify(z, e^+, e^-, s)$, shall mean that state s can be reached by iterated application of the underlying causal relation, starting in state z with momentum e^+, e^-.

The *foundational axioms of the Fluent Calculus with ramifications* are $\sum_{state} \cup \sum_{ramify}$ where $\sum_{ramify}$ contains the following foundational axiom which defines *Ramify* as fix-points of *Causes*

$$
\begin{aligned}
Ramify(z, e^+, e^-, s) \equiv (\exists z_1, e_1^+, e_1^-)(State(s) = z_1 \wedge \\
(z, e^+, e^-, z_1, e_1^+, e_1^-, s) \in \mu[Causes])
\end{aligned}
\tag{3.20}
$$

where $(\vec{x}, \vec{y}, s) \in \mu[P]$ abbreviates the following formula, which is a standard second-order schema to axiomatize that $(\vec{x}, \vec{y})$ belongs to the reflexive and transitive closure of predicate P and that $\vec{y}$ is a fix-point

$$
(\forall \Phi) \left\{ \begin{array}{c} (\forall \vec{u}) \Phi(\vec{u}, \vec{u}, s) \wedge \\ {[\Phi(\vec{u}, \vec{v}, s \wedge P(\vec{v}, \vec{w}, s) \supset \Phi(\vec{u}, \vec{w}, s)] \supset \Phi(\vec{x}, \vec{y}, s)} \end{array} \right\}
$$

On the basis of causal relation and its closure, the following generalization of simple state update axioms accounts for actions with ramification.

Consider a fluent calculus signature with action function A, and let $A \in \mathcal{A}$.

A *ramification update axiom* for A is a formula

$$\begin{aligned}
&Poss(A(\vec{x}), s) \supset \\
&(\exists \vec{y}, z_1)(\Delta_1\{z/State(s)\} \wedge z_1 = \\
&\qquad State(s) \circ \vartheta_1^+ - \vartheta_1^- \wedge Ramify(z_1, \vartheta_1^+, \vartheta_1^-, Do(A(\vec{x}), s))) \\
&\vee \cdots \vee \\
&(\exists \vec{y}, z_1)(\Delta_k\{z/State(s)\} \wedge z_1 = \\
&\qquad State(s) \circ \vartheta_k^+ - \vartheta_k^- \wedge Ramify(z_1, \vartheta_k^+, \vartheta_k^-, Do(A(\vec{x}), s)))
\end{aligned} \tag{3.21}$$

where $\Delta_i(\vec{x}, z)$ is a pure state formula in z and ϑ_i^+ and ϑ_i^- are fluent collections.

3.3.4 Qualification Problem within the Fluent Calculus

The qualification problem arises because it is difficult in the real world to describe all circumstances under which an action is guaranteed to work. The remainder of this chapter is adapted from [MT01]. The theoretical account of the Qualification Problem uses the binary function $\mathbf{Ab}(p(\vec{x}), y)$ whose range is of sort FLUENT. The first argument, $p(\vec{x})$, denotes a property like $Movable(u)$ or $Functioning(Gripper(r))$. The second argument, y, indicates the cause for the abnormality. For convenience, we use the macros $\mathbf{Ab}(p(\vec{x}), z)$ and $\mathbf{Ab}(p(\vec{x}), s)$ to represent that for some y, $\mathbf{Ab}(p(\vec{x}), y)$ holds in state z and situation s, respectively:

$$\begin{aligned}
\mathbf{Ab}(p(\vec{x}), z) &\stackrel{\text{def}}{=} (\exists y) Holds(\mathbf{Ab}(p(\vec{x}), y), z) \\
\mathbf{Ab}(p(\vec{x}), s) &\stackrel{\text{def}}{=} \mathbf{Ab}(p(\vec{x}), State(s))
\end{aligned}$$

Instances of the generic 'abnormality' fluent are used to summarize the abnormal qualifications of actions, i.e., obstacles which are *a priori* unlikely to happen and therefore need to be assumed away by default in order to jump to the conclusion that the action is possible under normal circumstances. E.g., in the light of abnormal qualifications, the preconditions for an extended *Move* action in the blocks world are specified by

$$\begin{aligned}
&Poss(Move(r, u, v, w), s) \equiv \\
&\quad u \neq w \ \wedge \ v \neq w \ \wedge \ Holds(On(u, v), s) \ \wedge \\
&\quad (\forall y)(\neg Holds(On(y, u), s) \ \wedge \ \neg Holds(On(y, w), s)) \ \wedge \\
&\quad \neg \mathbf{Ab}(Movable(u), s) \ \wedge \ \mathbf{Ab}(Functioning(Gripper(r)), s)
\end{aligned} \tag{3.22}$$

The general scheme for preconditions with abnormalities is given in Equation (3.23):

$$\begin{aligned} &Poss(a(\vec{x}), s) \equiv \\ &\quad \Delta(s) \ \wedge \\ &\qquad \neg\mathbf{Ab}(p_1(\vec{x}), s) \ \wedge \ \ldots \ \wedge \ \neg\mathbf{Ab}(p_n(\vec{x}), s) \end{aligned} \tag{3.23}$$

Abnormal qualifications that have been caused by the agent himself are commonly accounted for by suitable causal relationships, which is how the general problem of anomalous models is overcome; see [Thi01b].

To account for abnormal qualifications other than those caused by the agent, instances of **Ab** are allowed to become true during any situation transition as a side effect of the mere fact that the very transition takes place. So doing requires additional causal relationships which describe exogenously caused abnormalities. These are modeled using the predicates $\mathit{ExogCaused}(f, s)$ and $\mathit{ExogUncaused}(f, s)$, indicating that in situation s fluent f arises (resp. vanishes) due to an exogenous cause. The effect of exogenous causes is specified by causal relationships.

Up to this point the treatment of qualifications did not affect the monotonicity of the solution to the Frame and Ramification Problem. A non-monotonic component, however, is required to *minimize* abnormal qualifications whenever they are not caused by an action that has been performed. This is achieved by adding default rules in the sense of [Rei80], by which the Fluent Calculus gets embedded into a default theory. Formally, exogenous influence on abnormalities is minimized by default rules of the following form:

$$\delta_{\mathit{Exog}}(\vec{x}, s) = \frac{: \neg\mathit{ExogCaused}(\mathbf{Ab}(p(\vec{x}), \mathit{Exog}), s)}{\neg\mathit{ExogCaused}(\mathbf{Ab}(p(\vec{x}), \mathit{Exog}), s)} \tag{3.24}$$

and

$$\delta_{\mathit{Exog}}(\vec{x}, s) = \frac{: \neg\mathit{ExogUncaused}(\mathbf{Ab}(p(\vec{x}), \mathit{Exog}), s)}{\neg\mathit{ExogUncaused}(\mathbf{Ab}(p(\vec{x}), \mathit{Exog}), s)} \tag{3.25}$$

An accompanying default assumption concerns abnormalities of any kind in the *initial* situation. Their minimization is carried out by defaults of the following form:

$$\delta_{S_0}(\vec{x}, y) = \frac{: \neg\mathit{Holds}(\mathbf{Ab}(p(\vec{x}), y), S_0)}{\neg\mathit{Holds}(\mathbf{Ab}(p(\vec{x}), y), S_0)} \tag{3.26}$$

If, e.g., the observations suggest no abnormalities initially, then the underlying default theory has a unique extension (in the sense of [Rei80]), which includes $(\forall x)\neg\mathbf{Ab}(p(\vec{x}), S_0)$. (Recall that $\neg\mathbf{Ab}(p(\vec{x}), s)$ means $\neg\mathit{Holds}(\mathbf{Ab}(p(\vec{x}), y), s)$ for any y.) Assume an initial state S_0 with the following (incomplete) description:

$$\begin{aligned} &\mathit{Holds}(\mathit{On}(A, \mathit{Table}), S_0) \ \wedge \ \mathit{Holds}(\mathit{On}(C, \mathit{Table}), S_0) \ \wedge \ \mathit{Holds}(\mathit{On}(B, C), S_0) \ \wedge \\ &(\forall y)(\neg\mathit{Holds}(\mathit{On}(y, A), S_0) \ \wedge \ \neg\mathit{Holds}(\mathit{On}(y, B), S_0)) \end{aligned}$$

Hence, (3.22) implies, that $\mathit{Move}(\mathit{Robbie}, A, \mathit{Table}, B)$ is possible in S_0. If this action nonetheless fails, i.e. the observation $\neg\mathit{Poss}(\mathit{Move}(\mathit{Robbie}, A, \mathit{Table}, B))$

is added, then the default theory admits different extensions. These are obtained by applying all defaults except for one instance of (3.26) with either $x/Movable(A)$ or $x/Functioning(Gripper(Robbie))$. This approach can also be used to account for so-called weak qualifications. Fluents denoting weak qualifications strengthen the antecedents of state update axioms. Thus the generic shape of state update axioms with possible weak qualifications is given below:

$$\begin{aligned}
&Poss(a(\vec{x}), s) \supset \\
&\quad \neg\mathbf{Ab}(p(\vec{x}), s) \supset \\
&\qquad \Delta_1 \ \wedge \ State(Do(a(\vec{x}), s)) = (State(s) \circ \vartheta_1^+) - \vartheta_1^- \ \wedge \\
&\quad \mathbf{Ab}(p(\vec{x}), s) \supset \\
&\qquad \Delta_2 \ \wedge \ State(Do(a(\vec{x}), s)) = (State(s) \circ \vartheta_2^+) - \vartheta_2^-
\end{aligned} \tag{3.27}$$

3.3.5 FLUX - A Fluent Calculus Implementation

FLUX, the Fluent Calculus Executer, is a programming language based on the ideas of the Fluent Calculus. It allows to write concise and modular programs for agents which base their decisions on an explicit world model. In order to keep this model up to date during the execution of the program, every agent program requires an encoding of the underlying domain axiomatization. Due to the declarative nature of FLUX, domain axioms in the Fluent Calculus translate directly into executable program statements.

FLUX agents are implemented on top of a kernel, which endows agents with general reasoning capabilities for maintaining and using their internal model. As a *logic program*, the kernel can easily be verified against the semantics of the Fluent Calculus. In a similar fashion, formal methods can be applied to the agent programs themselves to prove their correctness, which is of particular interest if agents handle secure information or are used for safety-critical control tasks.

The general architecture for FLUX is in Figure 5. The agent program itself implements the behavior of the agent with the help of two basic commands defined in the kernel program: The expression $Holds(f, z)$ is used to condition the behavior of the agent on the current state environment. The predicate means exactly what it means in the Fluent Calculus: It is true if and only if condition (i.e., fluent) f is true in the current world model (i.e., state z). Secondly, agent programs use the statement $Execute(a, z_1, z_2)$ to trigger the agent to actually perform an action a in the outside world. This command has the side effect that the current state z_1 is updated to z_2 according to the changes effected by action a.

The update computation itself relies on a specification of the elementary actions of the agent. As in the fluent calculus, the effects of actions are encoded

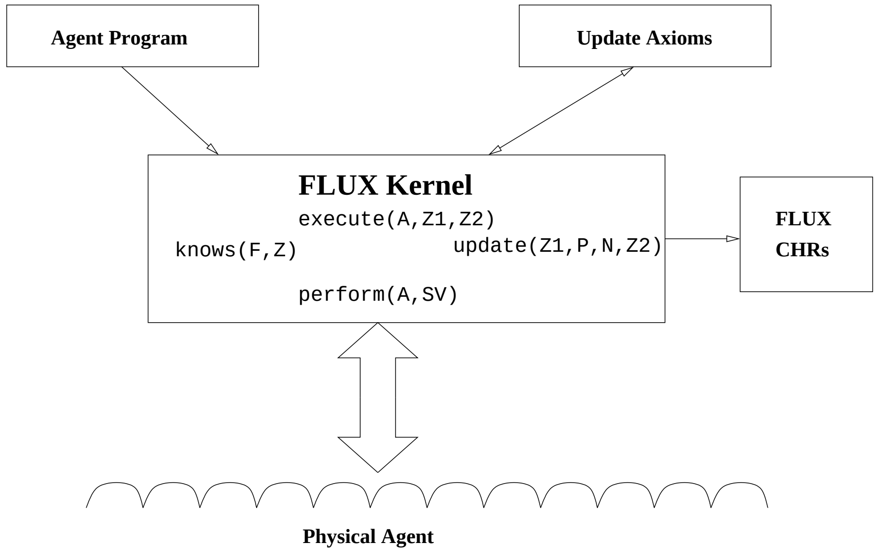

Figure 5: FLUX Architecture.

by state update axioms. To facilitate these specifications, the FLUX kernel provides a definition of the auxiliary predicate $Update(z_1, \vartheta^+, \vartheta^-, z_2)$. Its intuitive meaning is that the state z_2 is the result of positive and negative effects ϑ^+ and ϑ^-, respectively, in state z_1. In other words the predicate encodes the equation $z_2 = (z_1 - \vartheta^-) + \vartheta^+$.
The general version of FLUX (as opposed to "special FLUX", see [Thi02a]) is designed in such a way to endow agents with the general ability to reason about effects of their actions in the presence of incomplete states. It requires that the states in FLUX should be able to encode partial information by constraints. General FLUX appeals to the paradigm of *constraint logic programming* (CLP, for short). Certain atoms, the so-called constraints, in clauses maybe partially evaluated only and kept in store for later consideration.

The FLUX Kernel

General FLUX is based on a particular technique of a state representation. Incomplete states are encoded as lists with a *variable* tail. This is a very natural way of representing compound states in the fluent calculus that contain a state variable. For example the term $InRoom(R423) \circ Carries(Coffee) \circ z$ is represented by this list:

$$[InRoom(R423), Carries(Coffee) \mid z]$$

A state formula $\Phi(z)$ is FLUX-expressible if it is of the form

$$(\exists \vec{x}, z')(z = \varphi_1 \circ \cdots \circ \varphi_m \circ z' \wedge \Psi_1(\vec{x}, z') \wedge \cdots \wedge \Psi_n(\vec{x}, z')) \tag{3.28}$$

where $m, n \geq 0$, each $\varphi_i (1 \leq i \leq m)$ is a fluent with variables among $\vec{x}$, and each $\Psi_i(\vec{x}, z')(1 \leq i \leq n)$ is one of the following forms:

1. $\neg Holds(\varphi, z')$, where φ is a fluent with variables among $\vec{x}$;
2. $(\forall \vec{y})\ \neg Holds(\varphi, z')$, where φ is a fluent with variables $\vec{y}$;
3. quantifier-free arithmetic formula over natural or rational numbers with variables among $\vec{x}$.

Thus, FLUX-expressible are state specifications which consist of existentially quantified positive and negated fluents, universally quantified negated fluents, and auxiliary axioms expressing properties of the arguments of fluents. For example, the fact that a functional fluent F has a unique value in state z is FLUX-expressible by the formula

$$(\exists \vec{x}, z')(z = F(\vec{x}) \circ z' \wedge (\forall \vec{y}) \neg Holds(F(\vec{y}), z')).$$

FLUX-expressible formulae are encoded as FLUX *states*, which consist of an incomplete list of fluents along with a set of *constraints*. The constraints encode the sub-formulae Ψ_j of a state formula 3.28. An auxiliary constraint called $DuplicateFree(z)$ is used to ensure that the state list z is free of multiple occurrences of fluents.

Let $\langle F, \circ, \emptyset \rangle$ be a *Fluent Calculus state signature.* A FLUX *state* is a list

$$z = [f_1, \ldots, f_n \mid z']$$

of pairwise different fluents ($n \geq 0$) along with the auxiliary constraint $DuplicateFree(z)$ and a finite number of constraints of the form

1. $NotHolds(f, z')$;
2. $NotHoldsAll(f, z')$ or
3. arithmetic constraints.

The implementation of constraints and key predicates is based on constraint handling rules (CHRs). For a more detailed description of the use of CHRs within FLUX the reader is referred to [Thi02a].

4 The General Framework

In this chapter, a generic robot control architecture and a communication structure is derived on the base of conclusions from Chapter 3. No assumption about any particular hardware configuration or any particular application is done. A general approach to execution monitoring is presented.

4.1 A Generic Model of the Architecture

In this section we discuss a general outline for the robot architecture. The components and their relation to each other are depicted in Figure 6.

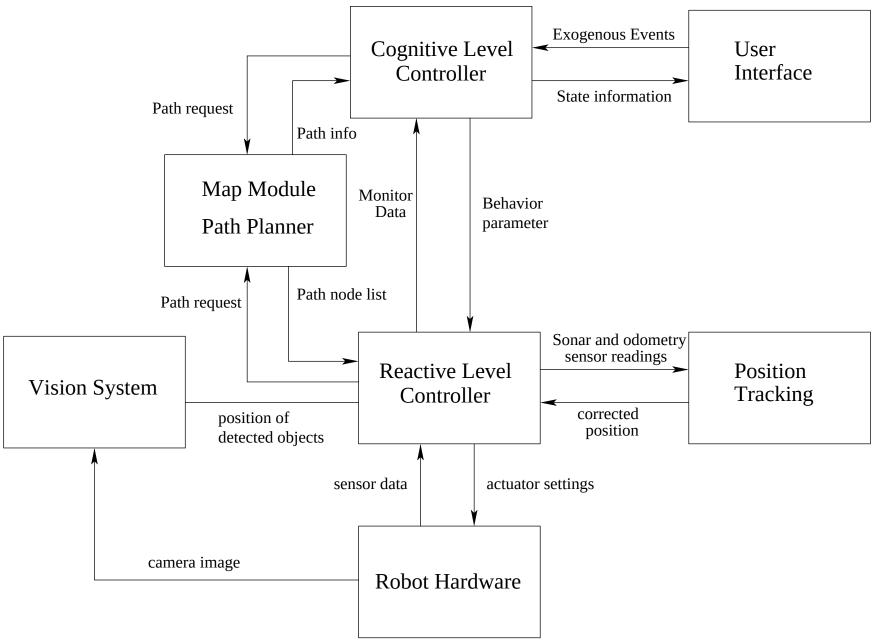

Figure 6: The generic architecture

All components are shortly described below[10]:

Cognitive-level controller

- Planning and reasoning system
- Execution monitoring at the cognitive level

[10]adapted from the Giovanna Project home page [Gro01]

Reactive-level controller

- Maintaining a model of the robot's environment using the readings of the robot's distance and odometry sensors
- Computing target values of the robot's actuators based on the output of a set of reactive behaviors
- Execution monitoring at the level of reactive behaviors

Vision system

- Model-based object detection

Position tracking system

- Implementation of some localization method
- Computing a probability distribution (belief) about the robot's position with respect to a map of the environment using sensor data

Map module and path planner

- Computing a topological graph for global navigation using Voronoi segmentation
- Translating symbols for locations into target coordinates

User Interface

- User Requests

The presented architecture is motivated by the ongoing discussion about reactive (or perceptual) robot controllers and controllers on a cognitive (or deliberative) level. This separation can be illustrated by an analogy to the psychological conceptions *consciousness* and *subconsciousness*: You are trying to enter a room (like your own office) as a single step in an action sequence. Almost unconsciously a human being would grasp for the key in his/her pocket, if the door was locked. Similarly, small adjustments while putting the key into the key hole are handled by lower layers of consciousness (reactive behavior, in the sense of Brooks). On the contrary, if one notices that the key is not in the pocket, then the high levels of consciousness are alerted. Replanning seems necessary. Inevitably the world knowledge must be invoked: information of whereabouts for the key or what person to ask need to be retrieved.

All peripheral modules are subject to the current hardware configuration. They can roughly be classified in sensors and actuators.

4.2 A Generic Model of the Communication System

In order to attach the cognitive level planner to neighbored layers, the communication is built up in the following way: The interface of the high level module is responsible for data exchange between these components:

Cognitive Controller - Reactive Robot Controller: Parameterized high level commands are passed to the lower level of control. Furthermore, if high level decides to interrupt the current action, the respective signal is sent. For positioning queries, high level can "tap" the reactive robot controller as well, as it is convenient in most settings to register the controller as a position tracking client. Finally, the results from some of the execution monitors are conveyed back and processed by the handlers for execution monitoring data (see Figure 7).

Cognitive Controller - User interface: Exogenous events in a very general sense are directed to the dedicated handler. User information will be used to simulate input from external sensors and is conveyed directly to the corresponding handler in the cognitive controller.

Cognitive Controller - Path planner: Path information are preferably communicated between the reactive controller and path planning module. For path cost calculation, however, it is useful to query the path planner directly.

4.3 Execution Monitoring - The General Scheme

In order to recover from failures, a robot needs to perform the following steps:

1. Being aware of a failure
2. Providing an explanation (abandoned in some approaches)
3. Launch a recovery procedure

each of which is handled in the subsequent subsections, respectively.

4.3.1 Discrepancy Detection

Catching exceptional states follows the general pattern of a diagnose: due to imperfect sensors and general non-omniscience the agent is limited to detect symptoms, i.e. supposedly crucial properties of the real world.

Consider Figure 8. The overall process of error detection is threefold. First, observation (by means of sensing, sensor data processing and abductive reasoning) is regarded as the process of mapping the real world to distributed *state representations*. In our delivery scenario, lower levels keep track of precise positions, speed, elapsed time, movement characteristics. The states of fluents, on the other hand, are registered in upper layers.

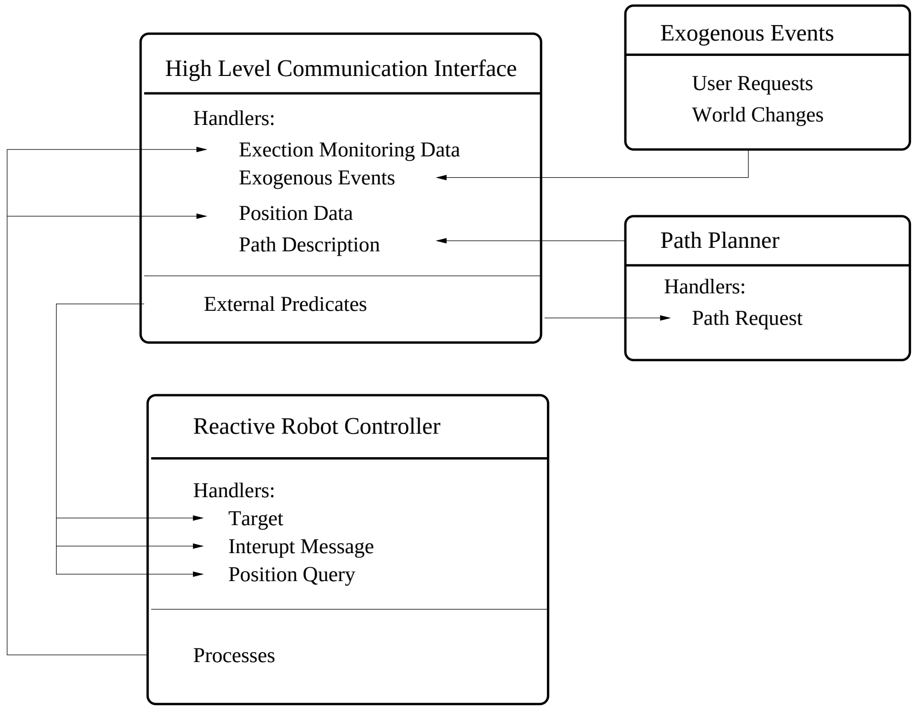

Figure 7: The generic communication scheme for the high level and related modules

Likewise, the expectations on the different levels are generated according to some method, e.g. on high level by means of an knowledge base and an underlying action theory. This requires means of *calculating* expectations in first place. The definition of execution monitoring inevitably requires the continuous maintenance of an internal world model.

Finally, discrepancies are detected, when internal and external state representation contradict each other. Possible metrics for measuring state distance can be

- time
- spatial difference
- symbolic state description

It is important to notice, that the generation of both internal and external model might be faulty. Sensors might report noisy data, and world models might not match the real world activities (wrong expectations, as discussed in [Bjä98]).

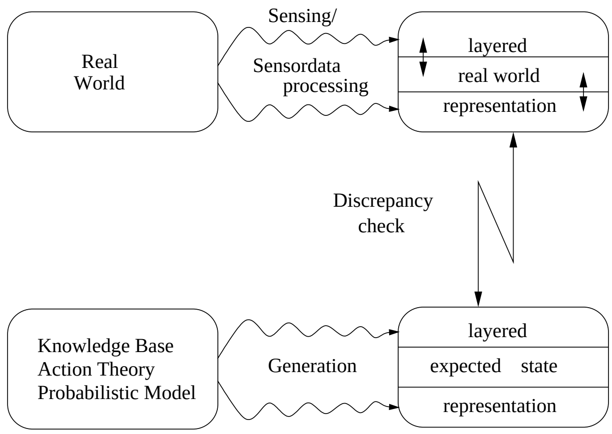

Figure 8: Discrepancy check from real world percepts and expected state

4.3.2 Explanations

Failures come in many flavors. A robot, for example, can get stuck because of many reasons. The symptoms are usually discrepancies in the actual and the expected position after some time. The reason is not always clear to the robot; all original causes like a crowded room, a closed door, table legs, low energy etc. might map to the same mentioned symptom.

The more circumstances we know about an action, the easier it is to conjecture about the possible failures. Hierarchical planning meets this policy: if, for example, a complex path-following task is broken down into smaller actions like door passings, corridor and room traverses, then failures can be substantiated with higher reliability.

By use of *default logic*, the planning module abstracts away from the sheer non-exhausting (but increasingly unlikely) set of preconditions, thus solving the Qualification Problem (McCarthy, [McC77]). In case of failures though, these default assumptions are double checked according to a partially ordered preference list (see also [Mar01].)

A similar abductive technique for sensor data explanation was successfully incorporated in Khepera robots using the Event Calculus (Shanahan, [Sha96]). Due to lack of powerful sensors, the robot could not keep track of its position. Entering a door was achieved by following the wall until side sonars signal an opening. If a door was closed, then the robot continued to the next corner and could then explain unexpected front sonar firing.

Alternatively, missing information could be acquired by *active sensing*. In the abovementioned example, consulting other sensors like camera or laser range finder (via the vision module) could give insight to the nature of an obstacle.

Realistically speaking, both techniques have its flaws. A preference list for failure explanation might be faulty and needs to be hand-coded in advance (though this seems learnable and is subject to future work, cf. Chapter 6.2). Abduction in general acts as a makeshift, when only indirect symptoms are sensible. The reconstruction of the implication chain from the original cause to the symptom might involve speculation. Powerful sensors, on the other hand, could cut short the abduction and thus lower the amount of speculation but brings as usual utter uncertainty due to noise.

Hence, the applicability of the mentioned techniques depends highly on the quality of sensors. For example, Shanahan's "half-blind" Khepera robot could be easily fooled by dynamic obstacles, but equipped with a position tracking system, direct door sensing would be possible.

4.3.3 Recovery

Once an explanation is found and the corrective state calculation is performed (leading to a state I'), some recovery strategy must remedy from the failure. Simmons et al. suggest to launch a predefined failure recovery plan. I.e., when the robot notices that it ran in a dead end, it develops a path, that brings it back on the original track and continues with the initial plan. In the presence of a strong planning tool, however, replanning is actually only a side product of planning. A plan P is generated given an initial situation I, the foundational axioms F and a goal G, such that

$$I \wedge F \wedge P \models G \tag{4.1}$$

In this fashion the recovery plan can be generated too, given that the new state I' is properly observed (by means of abduction or direct sensing) and used instead of I.

Furthermore, imagine, the robot did not run into a dead end but found a shortcut (generally speaking, a serendipitous event occurs). Now the planner provides an easier solution, if returning to the old track proves to be more costly.

5 Implementation

This chapter is committed to the generic solutions of the previous chapter. Sections 4.1 and 4.2 are in direct correspondence to sections 5.1 and 5.2, respectively. Implemental issues are described in a top down manner by first describing high level issues (axiomatization etc.) and then descending to lower layers of the architecture. The chapter ends up with sample runs in the extended delivery scenario.

The presented implementation can be seen as extension to the Giovanna Project ([Gro01]), where a cognitive level planning module is embedded into the existing structure.

5.1 Architecture Revisited

The system architecture presented here follows the general skeleton given in Section 4.1, see Figure 9.

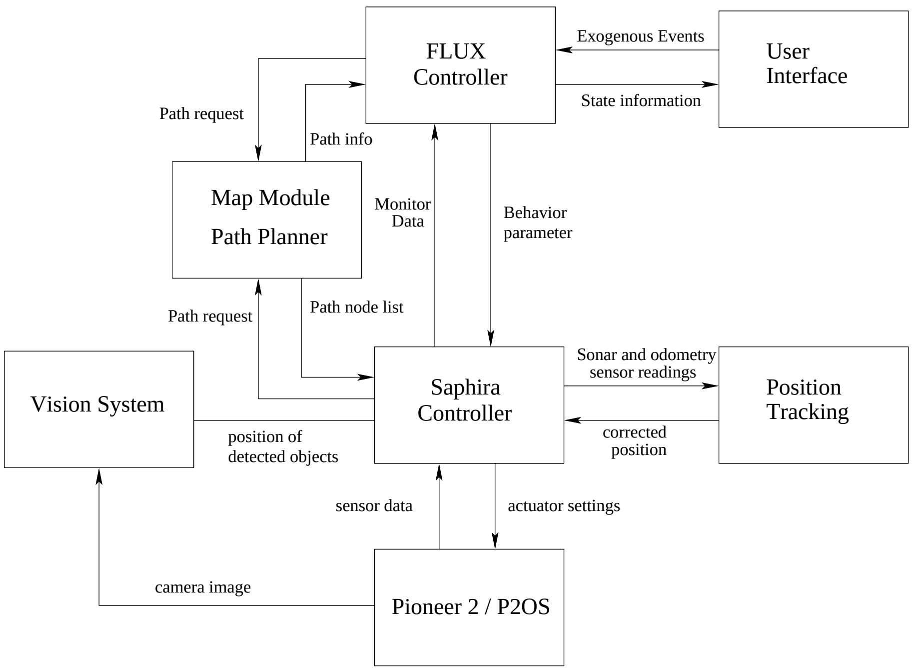

Figure 9: The system architecture for the sample application

The components are shortly described below:

Hardware:

- ActivMedia *Pioneer 2* robot equipped with odometry and sonar sensors, laser range finder, compass, and camera. The robot is depicted in Figure 10.

Figure 10: Bender, the *Pioneer 2* robot

Flux controller:

- Planning and reasoning system
- Symbolic world model using fluents (logical description of situations) and state update axioms.
- High level execution monitoring

Saphira controller:

- Local perceptual space
- Probabilistic world model
- Hierarchy of low-level behaviors. Each behavior is implemented as a set of fuzzy-logic rules and is equipped with a priority.
- Low-level execution monitoring
- Based on Saphira OS, supports multiple processes, process control handled by simple "Round-Robin" scheduler. Figure 11 shows Saphira's system architecture.[11]

Map module:

- Computation of a topological graph for global navigation from a given metrical map by Voronoi segmentation

[11] A comprehensive report on Saphira is given in [KMRS97].

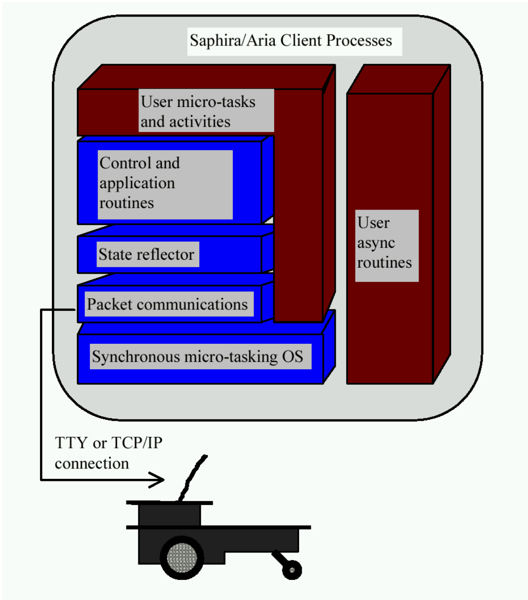

Figure 11: Saphira OS, courtesy of K. Konolige

Localization module:

- Implementation of Monte-Carlo localization using laser-range data

Communication server:

- Task Control Architecture (TCA) by R. Simmons ([SM89]) used for interprocess communication among all the modules

5.2 Inter-modular Communication

A major design question is, when to take possible world changes into account. Most approaches deal with newly obtained information at the end of the currently performed action ([GRS98],[Mar01],[Cam00]).

It is in principle possible, though, to interrupt the current action, because world changes give rise to immediate plan changes. For example the robot is about to travel to a far office. In that moment it receives an additional request for that office. Instead of walking twice, it could minimize costs by handling both requests at once.

Other events — possibly independent from the current action and registered asynchronously — are closed doors, canceled delivery requests or detected people.

These exogenous events might occur at any time and may influence the current or future actions. Discrepancy checks between internal world model and reality may therefore take place continuously, and indeed some monitors for the current action respond immediately to detected failures. In general, discrepancy checks are convenient at several phases:

1. Verification of an actions preconditions;
2. Verification of an actions postconditions;
3. Whenever underlying modules signal world changes, that should be taken into account in the fluent-symbolics
4. Deliberative verification of certain aspects, for example, crucial fluents that seemingly become unlikely.

In the presented approach, special attention is drawn on point 2 and point 3.

5.2.1 Communication Protocol

A sample communication protocol is given in Figure 12. It demonstrates the different cases

- action execution,
- uncritical world changes,
- critical world changes.

In the depicted schedule it is guaranteed, that world changes can not disturb running planning activities but are taken into account frequently.

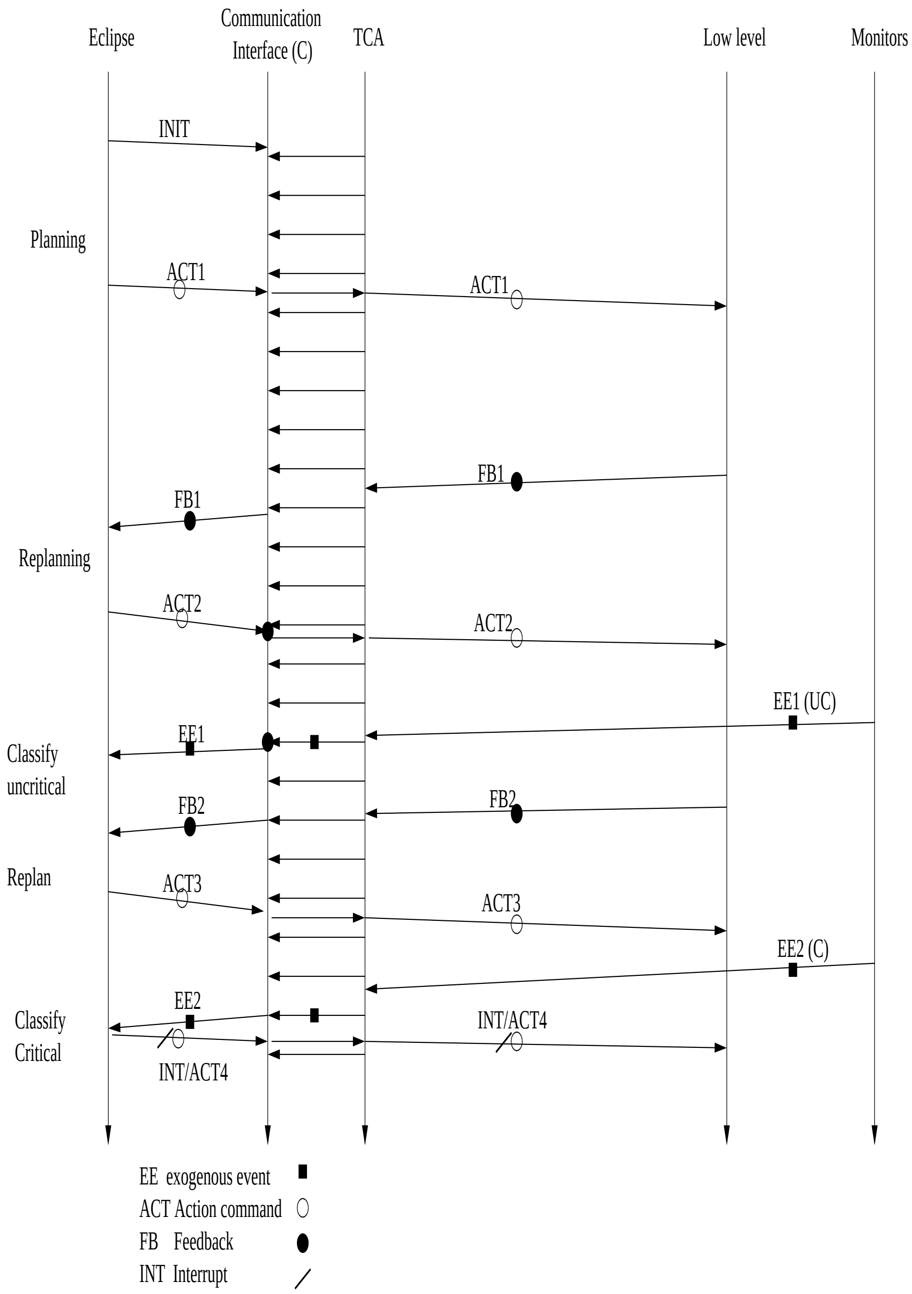

Figure 12: The Communication Protocol. For the sake of simplicity all monitors and all low level modules are projected to one execution thread. The mechanism, that frequently queries for new events on its dedicated TCA channels in order to update Eclipse is symbolized by the arrows between the threads for TCA and the communication interface.

5.2.2 Communication

The generic communication model was presented in Section 4.2. Here the specific model for the current application is pursued.

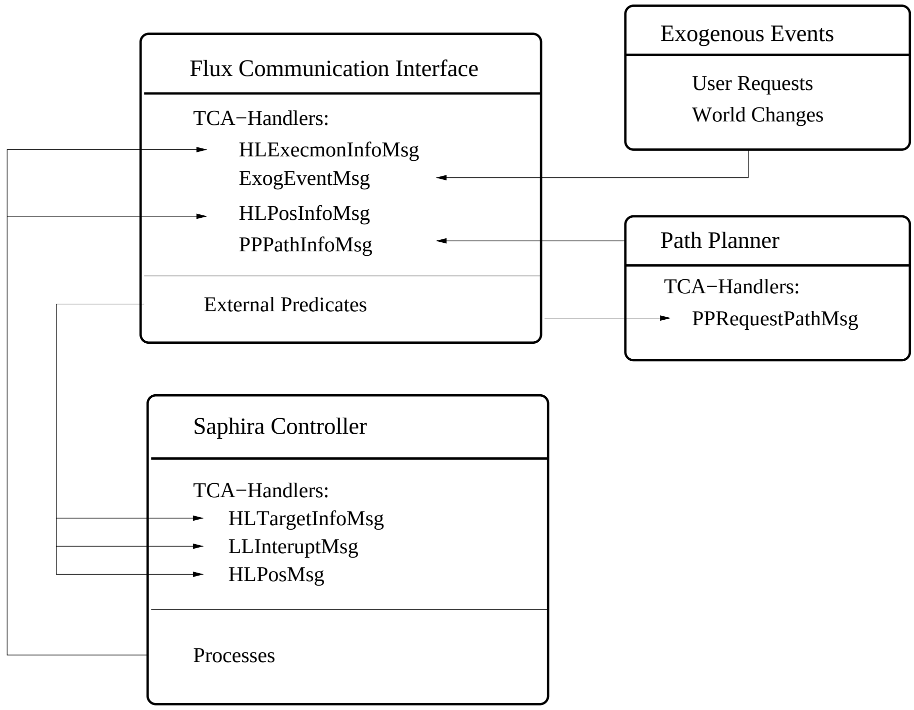

Figure 13: The specific message handlers relevant for communication with the FLUX Module

Figure 13 shows the communication of FLUX with neighbored modules. The figure can be regarded as an instantiation of the general pattern shown in Figure 7.

5.2.3 Integration of Planning and Execution Monitoring

The overall controlling mechanism is given by the following algorithm:

Algorithm *Integral Controller*(I)
Input: An initial state I
1. $\mathit{FindPlan}(+I, -P)$
2. $Z \leftarrow I$
3. **repeat**
4. Execute first action A from P
5. (* Now listen for feedback *)
6. $\mathit{CheckForWorldUpdates}(-FB, -WC)$

```
        (* Outputs: Action Feedback FB and additional World Changes WC *)
        if WC ≠ ∅
          then UpdateState(+Z, +WC, −Z')
                 (* Here inconsistencies are solved, in principle newly observed
                    fluents should be of greater reliability *)
          else Z' ← Z (* No further state update *)
        if ClassifyCritical(+WC) and FB is nil
                 (* A critical exogenous event occurred*)
          then FindPlan(+Z', −P'')
                 interrupt current action execution
          else
                 (* uncritical changes, still might cause later failures *)
                 if not executable(+Z', +P)
                        (* Double check plan executability *)
                   then FindPlan(+Z', −P')
                   else P'' ← P' (* keep old plan *)
        if FB = Success
          then P' ← P\A
                 Z'' follows from Z' according to State Update Axioms
        if FB = Failure
          then CheckDefaults(+Z', −Z'')
                 (* Register preferred abnormalities in Z', thus obtaining Z'' *)
                 FindPlan(+Z'', −P'') (* Replanning *)
        Z ← Z''
        P ← P''
    until Goal(Z) (* Goal state not reached *)
```

Note, that the pseudo-code differs syntactically from the actual Prolog-implementation given in Appendix A.

5.3 High Level Domain Description and Axiomatization

5.3.1 Fluents

First, all involved fluents are described. Their signature and their intended meaning are listed in the table below

Function	Type	Meaning
Carries	OBJECT $\mapsto$ FLUENT	The robot carries object x
InRoom	ROOM $\mapsto$ FLUENT	The robot is in room x
Request	PERSON$\times$OBJECT$\times$ PERSON$\mapsto$ FLUENT	There is a request to deliver object x_2 from person x_1 to person x_3.
At	NODE $\mapsto$ FLUENT	The robot is at node x
Here	PERSON $\mapsto$ FLUENT	Person x is close to the robot
Has	PERSON$\times$OBJECT $\mapsto$ FLUENT	Person x_1 is in possession of object x_2
Elevator	$\mapsto$ FLUENT	Robot in elevator mode
EleUp	$\mapsto$ FLUENT	Elevator direction: upwards
EleDown	$\mapsto$ FLUENT	Elevator direction: downwards
SRequest	PERSON::LIST$\times$OBJECT $\times$PERSON $\mapsto$ FLUENT	There is a search request for object x_2 among x_1 to person x_3.
MightHave	PERSON $\times$ OBJECT $\mapsto$ FLUENT	Person x_1 might have object x_2
OutOfReach	PERSON $\mapsto$ FLUENT	Person x is out of reach
Emailed	OBJECT $\times$ PERSON $\mapsto$ FLUENT	Person x_2 is informed (via email) about the failed search request for object x_1

5.3.2 Actions

Action	Type	Meaning
Receive	OBJECT × PERSON ↦ ACTION	Robot receives object x_1 from person x_2 (placed on top of the robot)
Deliver	OBJECT × PERSON ↦ ACTION	Robot delivers object to person
Enter	ROOM ↦ ACTION	Robot enters room x
GotoDoor	DOOR ↦ ACTION	Robot goes to a door of the room
Goto	NODE ↦ ACTION	Robot goes to a node
CallHelp	PERSON ↦ ACTION	The robot calls person x to come close
ChDir	↦ ACTION	Robot changes overall direction (in elevator mode)
Idle	↦ ACTION	Robot waits for exogenous events
Unreach	PERSON ↦ ACTION	Robot alerts that person x is unreachable
AddRequest	PERSON×OBJECT× PERSON ↦ ACTION	Request added by user
CancelRequest	PERSON×OBJECT× PERSON ↦ ACTION	Request canceled by user
Exog	FLUENT ↦ ACTION	Exogenous event occurs
UnExog	FLUENT ↦ ACTION	Exogenous event with negative effect occurs
WalkOn	↦ ACTION	Robot walks away from a person
Email	OBJECT × PERSON ↦ ACTION	Robot alerts (via email) to person x_2 that his/her search request for object x_1 was unsuccessful

5.3.3 Defaults

Default	Type	Meaning
Enterable	DOOR ↦ DEFAULT	Door x is enterable (open)
Mobility	↦ DEFAULT	Robot can move freely
Traceable	PERSON ↦ DEFAULT	Person can be detected
NotLost	OBJECT ↦ DEFAULT	Object did not get lost
Reachable	PERSON ↦ DEFAULT	Person is reachable (i.e., no obstacles block the way to the person)
Delivery	PERSON×OBJECT ↦ DEFAULT	Delivery of object x_2 to person x_1 succeeds

5.3.4 Static world knowledge

Predicate	Type	Meaning
Office	PERSON×ROOM	Room x_2 is known to be person x_1's office.

5.4 Axiomatization

Given the fluents and linguistic action descriptions from the previous chapter, we can now formalize action preconditions and state update axioms of the delivery scenario within the Fluent Calculus. The axioms are lifted to corresponding FLUX encodings in a straightforward fashion.

5.4.1 Action Preconditions

In the sequel, most action preconditions are presented. Actions with no precondition are omitted. Explanations are given, where precondition formulae might not be fully self-explanatory.

$$\begin{aligned}
&Poss(Receive(o,p),s) \equiv \\
&\quad \exists r\ Knows(InRoom(r),s) \land Office(r,p) \land \\
&\quad (Knows(At(p),s) \lor Knows(Here(p),s)) \land \\
&\quad (\exists r Knows(Request(p,o,r),s) \lor Knows(Has(p,o),s)) \land \\
&\quad \neg\mathbf{Ab}(Traceable(p),s)
\end{aligned}$$

Notice, that receiving is possible if the providing person p is either close to the robot ($Here(p)$) or the robot is at the persons place ($At(p)$). Moreover it is required that the person is in possession of some object o ($Has(p,o)$, in the case of search requests) or some request from that person exists.

$$\begin{aligned}
&Poss(Enter(r),s) \equiv \\
&\quad \exists d\ Knows(At(d),s) \land \exists r'\ Knows(InRoom(r'),s) \land \\
&\quad Connected(r',d,r) \land \\
&\quad \neg\mathbf{Ab}(Mobility,s) \land \neg\mathbf{Ab}(Enterable(d),s)
\end{aligned}$$

In this approach, doors are modeled to be enterable by default. This is due to the realistic assumption, that our robot's hardware does not allow for door opening.

$$\begin{aligned}
&Poss(Deliver(o,p),s) \equiv \\
&\quad Knows(Carries(o),s) \land \exists r\ Knows(InRoom(r),s) \land \\
&\quad Office(o,p) \land \neg\mathbf{Ab}(Traceable(p),s) \land \neg\mathbf{Ab}(NotLost(o),s)
\end{aligned}$$

$$\begin{aligned}
&Poss(GotoDoor(d),s) \equiv \\
&\quad \exists r,r'\ Knows(InRoom(r),s) \land Connected(r,d,r') \land \\
&\quad \neg\mathbf{Ab}(Mobility,s)
\end{aligned}$$

$$
\begin{aligned}
& Poss(Goto(l), s) \equiv \\
& \quad \exists r \; Knows(InRoom(r), s) \wedge In(l, r) \wedge \\
& \quad \neg \mathbf{Ab}(Mobility, s)
\end{aligned}
$$

$$
\begin{aligned}
& Poss(CallHelp(p), s) \equiv \\
& \quad Knows(\mathbf{Ab}(Reachable(p)), s) \wedge In(l, r) \wedge \\
& \quad \neg \mathbf{Ab}(Mobility, s)
\end{aligned}
$$

All other precondition (for action not mentioned) are empty, i.e. true.

5.4.2 State Update Axioms

ChDir **- Change Direction (in Elevator-mode)**

$$
\begin{aligned}
& Poss(ChDir, s) \supset \\
& \quad (Knows(EleUp, s) \supset \\
& \quad\; State(Do(ChDir, s)) = (State(s) - EleUp) + EleDown) \vee \\
& \quad (Knows(EleDown, s) \supset \\
& \quad\; State(Do(ChDir, s)) = (State(s) - EleDown) + EleUp)
\end{aligned}
$$

Idle **- Waiting for Exogenous Events**

The *Idle* action itself does not alter the world state.

$$Poss(Idle, s) \supset State(Do(Idle, s)) = State(s)$$

Goto **- Go to an arbitrary node**

Goto is designed in a flexible way. The robot might initially be at a persons spot or not and the *Goto*-action might bring the robot into another room (or not), hence the four cases:

$$
\begin{aligned}
&Poss(Goto(p_2), s) \supset \\
&\ \exists p_1, r_1, r_2\ (Knows(At(p_1), s) \wedge \\
&\quad (Knows(InRoom(r_1), s) \wedge Office(r_1, p_2) \supset \\
&\qquad State(Do(Goto, s)) = (State(s) - At(p_1)) + At(p_2)) \vee \\
&\quad (Knows(InRoom(r_1), s) \wedge Office(r_2, p_2) \wedge r_2 \neq r_1 \supset \\
&\qquad State(Do(Goto, s)) = (State(s) - At(p_1) \circ InRoom(r_1)) \\
&\qquad\qquad\qquad + At(p_2) \circ InRoom(r_2))) \vee \\
&\ \neg\exists p_1\ Knows(At(p_1), s) \wedge \exists r_1, r_2 \\
&\quad (Knows(InRoom(r_1), s) \wedge Office(r_1, p_2) \supset \\
&\qquad State(Do(Goto, s)) = State(s) + At(p_2)) \vee \\
&\quad (Knows(InRoom(r_1), s) \wedge Office(r_2, p_2) \wedge r_2 \neq r_1 \supset \\
&\qquad State(Do(Goto, s)) = (State(s) - InRoom(r_1)) + At(p_2) \circ InRoom(r_2))
\end{aligned}
$$

Receive **- Receiving an object**

$$
\begin{aligned}
&Poss(Receive(o, p), s) \supset \\
&\quad \exists r\, Knows(Request(p, o, r), s) \supset \\
&\qquad State(Do(Receive, s)) = (State(s) - Request(p, o, r)) + Carries(o, r) \vee \\
&\quad \exists l, r\, Knows(Has(p, o), s) \wedge Knows(SRequest(p, o, r), s) \wedge Member(p, l) \supset \\
&\qquad State(Do(Receive, s)) = (State(s) - SRequest(p, o, r) \circ Has(p, o)) \\
&\qquad\qquad + Carries(o, r)
\end{aligned}
$$

Deliver **- Delivering an object**

$$
\begin{aligned}
&Poss(Deliver(o, p), s) \supset \\
&\quad \neg\exists o'\, Knows(\mathbf{Ab}(Delivery(p, o')), s) \supset \\
&\qquad State(Do(Deliver(o, p), s)) = State(s) - Carries(o, p) \vee \\
&\quad \exists o', p'\, Knows(\mathbf{Ab}(Delivery(p, o')), s) \wedge \\
&\quad\ Knows(Carries(o', p')) \wedge o \neq o' \wedge p \neq p' \supset \\
&\qquad State(Do(Deliver(o, p), s)) = State(s) - Carries(o', p')
\end{aligned}
$$

GotoDoor **- Go to a door**

Note, that mutual distinctness of the disjunctive cases prevents from nondeterminism.

$$
\begin{aligned}
&Poss(GotoDoor(d), s) \supset \\
&\quad \exists r, d'\, Knows(InRoom(r), s) \wedge Knows(At(d'), s) \wedge \\
&\quad \neg Knows(\mathbf{Ab}(Reachable(d, r)), s) \supset \\
&\qquad State(Do(GotoDoor(d), s)) = (State(s) - At(d')) + At(d) \vee \\
&\quad \exists r\, Knows(InRoom(r), s) \wedge \neg\exists d'\, Knows(At(d'), s) \wedge \\
&\quad \neg Knows(\mathbf{Ab}(Reachable(d, r)), s) \supset \\
&\qquad State(Do(GotoDoor(d), s)) = State(s) + At(d) \vee \\
&\quad \exists r, d'\, Knows(InRoom(r), s) \wedge Knows(At(d'), s) \wedge \\
&\quad Knows(\mathbf{Ab}(Reachable(d, r)), s) \supset \\
&\qquad State(Do(GotoDoor(d), s)) = State(s) - At(d') \vee \\
&\quad \exists r\, Knows(InRoom(r), s) \wedge \neg\exists d'\, Knows(At(d'), s) \wedge \\
&\quad Knows(\mathbf{Ab}(Reachable(d, r)), s) \supset \\
&\qquad State(Do(GotoDoor(d), s)) = State(s)
\end{aligned}
$$

Enter - **Enter a room**

$$
\begin{aligned}
&Poss(Enter(r), s) \supset \\
&\quad \exists d, r'\, Knows(At(d), s) \wedge Knows(InRoom(r'), s) \wedge \\
&\quad \neg Knows(\mathbf{Ab}(Enterable(d))) \supset \\
&\qquad State(Do(Enter(r), s)) = (State(s) - InRoom(r')) + InRoom(r) \vee \\
&\quad \exists d, r'\, Knows(At(d), s) \wedge Knows(InRoom(r'), s) \wedge \\
&\quad Knows(\mathbf{Ab}(Enterable(d))) \supset \\
&\qquad State(Do(Enter(r), s)) = State(s)
\end{aligned}
$$

Unreach - **A person is registered unreachable**

$$
\begin{aligned}
&Poss(Unreach(p), s) \supset \\
&\qquad s' = State(s) + OutOfReach(p) \supset \\
&\qquad\quad Ramify(s', OutOfReach(p), \emptyset, State(Do(Unreach, s)))
\end{aligned}
$$

CancelRequest - **User cancels a request**

$$
\begin{aligned}
&Poss(CancelRequest(p, o, p'), s) \supset \\
&\qquad State(Do(CancelRequest(p, o, p'), s)) = State(s) - Request(p, o, p')
\end{aligned}
$$

AddRequest - **User adds a request**

$$\begin{aligned}&Poss(AddRequest(p,o,p'),s) \supset\\&\quad s' = State(s) + Request(p,o,p') \supset\\&\qquad Ramify(s', Request(p,o,p'), \emptyset, State(Do(AddRequest(p,o,p'),s)))\end{aligned}$$

Exog - **Exogenous event occurs**

Exogenous events are expected to be of fluent form, when given to the state update axiom. They might cause side effects.

$$\begin{aligned}&Poss(Exog(e),s) \supset\\&\quad s' = State(s) + e \supset\\&\qquad Ramify(s', e, \emptyset, State(Do(Exog(e),s)))\end{aligned}$$

UnExog - **Exogenous event with negative effect**

See above.

$$\begin{aligned}&Poss(UnExog(e),s) \supset\\&\quad s' = State(s) - e \supset\\&\qquad Ramify(s', \emptyset, e, State(Do(UnExog(e),s)))\end{aligned}$$

CallHelp - **Call nearby person for help**

$$\begin{aligned}&Poss(CallHelp(p),s) \supset\\&\quad \neg Knows(\mathbf{Ab}(Traceable(p)),s) \supset\\&\qquad State(Do(CallHelp(p),s)) = (State(s) - \mathbf{Ab}(Reachable(p))) + Here(p) \vee\\&\quad Knows(\mathbf{Ab}(Traceable(p)),s) \supset\\&\qquad State(Do(CallHelp(p),s)) = State(s)\end{aligned}$$

WalkOn - **Continue moving**

$$\begin{aligned}&Poss(WalkOn,s) \supset\\&\quad Knows(Here(p),s) \supset\\&\qquad State(Do(WalkOn,s)) = (State(s) - Here(p)) \vee\\&\quad \neg Knows(Here(p),s) \supset\\&\qquad State(Do(WalkOn,s)) = State(s)\end{aligned}$$

Email - **Email notification**

$$\begin{aligned}&Poss(Email(o,p),s) \supset\\&\quad State(Do(Email(o,p),s)) = State(s) + Emailed(o,p)\end{aligned}$$

Remark. Sensing actions are implemented slightly different in this approach. A sensing action causes the retrieval of a chunk of information. In that sense, some dialog action can be regarded as sensing, too.[12] Since the notion of exogenous events was designed in a very general fashion, that ranges over all kind of external information input, the incoming result of sensing actions coincides to an exogenous event. As a consequence, some dialogue actions don't have any effects but "provoke" exogenous events. The possibility to launch conditional plans upon the usually restricted number of sensing results is lost, but as will be justified in Section 5.5, no full-fledged plans are generated anyway. Alternatively, a different treatment of sensing actions in the sense of [Thi00b] would be possible.

5.4.3 Ramifications

Side effects are calculated with the causal relationships listed below. The macro *Causes*/3, defined in Section 3.3.3, is used:

$$\begin{aligned}&Knows(MightHave(p,i,r),s) \supset\\&\quad Causes(Has(p',i), \neg MightHave(p,i,r), s)\end{aligned} \tag{5.1}$$

$$\begin{aligned}&Member(p,plo), \neg Knows(MightHave(p,i,r),s) \supset\\&\quad Causes(SRequest(plo,i,r), MightHave(p,i,r), s)\end{aligned} \tag{5.2}$$

$$\begin{aligned}&Knows(Request(p,i,r),s) \wedge \neg Knows(DelayedReq(p,i,r),s) \supset\\&\quad Causes(OutOfReach(p), DelayedReq(p,i,r), s)\end{aligned} \tag{5.3}$$

$$\begin{aligned}&Knows(OutOfReach(p),s) \wedge \neg Knows(DelayedReq(p,i,r),s) \supset\\&\quad Causes(Request(p,i,r), DelayedReq(p,i,r), s)\end{aligned} \tag{5.4}$$

$$\begin{aligned}&Knows(Request(p,i,r),s) \supset\\&\quad Causes(DelayedReq(p,i,r), \neg Request(p,i,r), s)\end{aligned} \tag{5.5}$$

$$\begin{aligned}&Knows(DelayedReq(p,i,r),s) \wedge \neg Knows(Request(p,i,r),s) \supset\\&\quad Causes(\neg OutOfReach(p), Request(p,i,r), s)\end{aligned} \tag{5.6}$$

[12] Dialogues are well-suited for simulated sensing, such as tracing some person, which would require a cumbersome image processing routine.

$$Knows(DelayedReq(p,i,r),s) \supset \\ \quad Causes(Request(p,i,r), \neg DelayedReq(p,i,r), s) \tag{5.7}$$

$$Knows(Alerted(p),s) \supset \\ \quad Causes(\neg OutOfReach(p), \neg Alerted(p), s) \tag{5.8}$$

$$Knows(OutOfReach(p),s) \wedge \exists path\ PathPlan(p, path, s) \supset \\ \quad Causes(\neg(\mathbf{Ab}(Enterable(d))), \neg OutOfReach(p), s) \tag{5.9}$$

$$Knows(OutOfReach(p),s) \wedge \exists path\ PathPlan(p, path, s) \supset \\ \quad Causes(\neg(\mathbf{Ab}(Reachable(d,r))), \neg OutOfReach(p), s) \tag{5.10}$$

$$Knows(OutOfReach(p),s) \wedge \exists path\ PathPlan(p, path, s) \supset \\ \quad Causes(\neg(\mathbf{Ab}(Reachable(p))), \neg OutOfReach(p), s) \tag{5.11}$$

Remarks. Formula (5.1) virtually elaborates on the robots knowledge state: The gained information, that some person is in possession of some searched item, implies that none of the other candidates should have it. As a consequence, the planning engine will not choose to visit the person (unless there are other requests for that person). Formulae (5.3)-(5.7) handle automatic postponing of requests. Formulae (5.9)-(5.11) demonstrate the implicit use of the path planner. If during a state update it turns out, that a person renders reachable again for some reason (like an obstacle was removed, some door was opened, etc.), then the path planner is consulted in order to double check the person's reachability. In the confirming case, the fluent $OutOfReach(p)$ is retracted.

The ramification predicate calculates the transitive closure of possible world states, or more precisely the boundary of it. Intermediate states are commonly ruled out by means of state constraints. In the current setting, this would require a considerable amount of constraint handling rules and a heavily loaded constraint store during runtime. For efficiency reasons the ramification predicate of the FLUX kernel is slightly amended: the first clause of the $Ramify/4$ predicate succeeds (which terminates the ramification), if $Causes/5$ fails, i.e. no further causal relationships are applicable to the current state: This encoding turned out to result in a considerable speed up and is closer to the declarative semantics of ramification fix-points. Original Version:

```
ramify(Z1, _ , _ ,Z2):-  cons(Z1) , equal(Z1, Z2).
```

Amended version:

```
ramify(Z1, EP , EN ,Z2):-  \+ causes(_,_,Z1,EP,EN), equal(Z1, Z2).
```

5.5 Components of the Cognitive Planning Module

The FLUX planner comprises several parts. They are shown in Figure 14.

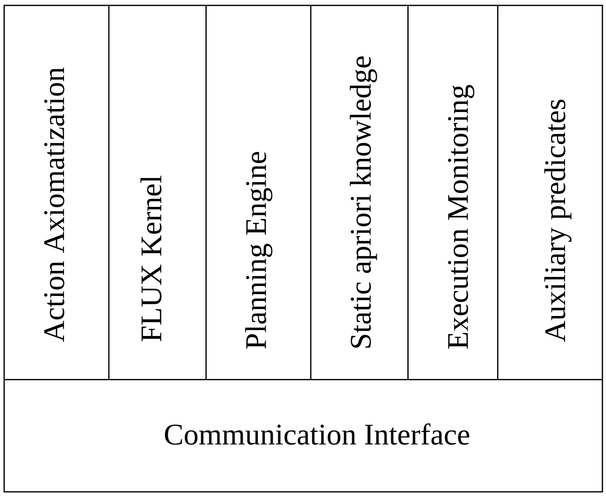

Figure 14: FLUX components

A brief description of the parts are given below:

5.5.1 Action Axiomatization

This part comprises all encoded axioms. The lifting from the state update axioms — as presented in Section 5.4 to the actual FLUX encoding is straightforward and is listed in Appendix A.

5.5.2 Flux Kernel

A General-FLUX kernel based on Constraint Handling Rules is used. A comprehensive FLUX kernel description can be found in [Thi02c]. However, small amendments are done for the ramification predicates (see Section 3.3.3).

5.5.3 Planning Engine

Traditional goal oriented planning using symbolic reasoning aims to produce a fullfledged action sequence ($A \in \mathcal{A}$) right from the initial situation ($I \in \mathcal{S}$) up to the goal ($G \in \mathcal{S}$):

$$I \wedge F \wedge A \models G \tag{5.12}$$

where F stands for all foundational background knowledge. While in certain (mathematical) settings this approach is appropriate, in the context of a real world environments it often renders infeasible. Experiments with Shakey revealed weaknesses mainly due to lack of reactivity. A high amount of interactivity must be taken into consideration.

As a result of various test runs with our robot, some changes of the abovementioned design were made: A controlling mechanism that monitors action execution inevitably requires a high amount of reactivity. On low level, complex 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 maintainance of a state S and an abstract state evaluation functions $E: \mathcal{S} \rightarrow \mathcal{A}$, that is consulted during every action-execution cycle, are the base for the approach taken here. Once a critical world change is realized, the current agenda is dropped, and the planner is invoked again.

```
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]
        ),
```

Figure 15: Eclipse Code for Planning algorithm

As shown in Figure 15, the planner investigates the current state according to state dependent criteria. The predicates on the left hand side of the arrows can be understood as diagnostic functions of the current state, ie. :

- *NotifyReach* investigates, whether people became out of reach recently. The output parameter *URL* is an action sequence of notifications about the unreachable 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 (ie. all the previous state diagnostic predicates failed), the *Continue* predicate is invoked. Depending on the

current state (position, elevetor mode, etc.), 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.

```
execute([A|L], S, Z):-
        write("[FL] State: "), writeln(Z),
        write("[FL] Agenda: "),writeln([A|L]),
        execute(A, Z, FB, Obs),
        administrate(Obs, S, S1, Z, Z1),
        state_update(Z1, A, Z2),
        wr_feedback(FB),
        (
          ll_success(FB) -> execute(L, [A|S1], Z2);
          ll_special(FB) -> loop([A|S1], Z2); % success+update, deletes old agenda
          ll_strn_ql(FB) -> br, strong_qual(A,S1, AA), recover(A, AA, ActAb),
                            res(ActAb,Zab), loop(ActAb, Zab);
          ll_weak_ql(FB) -> weak_qual(A, S1, Obs, ActAb),
                            res(ActAb,Zab), loop(ActAb, Zab);
          ll_intrupt(FB) -> state_detect(Z1,Z3,Ai), loop([Ai|S1], Z3);
          true
        ).

execute([],S,Z):-loop(S,Z).
```

Figure 16: Eclipse Code for Planning loop

5.5.4 Static apriori knowledge

Static apriori knowledge is easily specified in terms of Prolog facts. Data like topological information, that in principle could be learned starting in *tabula rasa*, is given:

- peoples whereabouts,
- room connectivity,
- room names,
- peoples names.

5.5.5 Execution Monitoring

The Execution Monitoring module contains all predicates that are relevant for extension generation in default logic. It furthermore has a hierarchy of abnormalities.

5.5.6 Communication Interface

The communication interface is twofold. On Eclipse side, the predicate *execute/4* is responsible for calling external predicates. Its first (input) argument is the action to be executed, the second (input) argument is the current state. As output variables, feedback and general observations are returned in argument three and four, resp.
External predicates are specified in C. Their generic pattern is:

- They convert Eclipse Prolog terms into C-readable data structures.
- Low level instruction messages are constructed and sent via TCA.
- A process suspending loop is entered, where the arrival of certain TCA messages serve as stop conditions.
- After constructing Eclipse terms for feedback and observation, control is again returned to high level.

Note, that the CPU work load caused by the Eclipse process is minimal while suspended.[13]

5.6 Qualifications

According to [Mar01], action failures can be explained by strong or weak qualifications. In the former case, the action is physically impossible, meanwhile in the latter case the action execution can be initiated but will not lead to the desired results.[14]

In comparison to [Mar01], the implementation of defaults is slightly changed with the perspective of enhanced code readability and improved efficiency. On the other hand, the occurrence of accidents and twice occurring failures is no longer supported for the sake of simplicity. Exogenous events, that cause abnormalities, are no longer understood as side effects of other actions. They are viewed as independent, singleton actions, as an exogenous event might occur at *any* time, even when no action is performed. This way, the current state can be reconstructed from the initial state and without a (partial) blocking of ramifications.

Strong vs. weak qualifications

If an action a is in principle possible in a situation s ($Poss(a(\vec{x}), s)$), but unexpected effects occur, it can be conjectured, which weak abnormality $\mathbf{Ab}(p(\vec{x}), s)$ most presumably occurred. In the given scenario it is reasonable to expect,

[13] The implementation is based on the C-procedure *mspause*, courtesy of A. Großmann.

[14] In reality, the border line between strong and weak qualifications is somewhat blurred. Many actions with strong qualifications can be regarded as weak, when precondition checks are omitted and actions are in some sense initiated. On the other hand, increased sensor quality will improve prediction of execution failures/success which would render weak qualifications as strong ones.

that the abnormality is caused by the most recent action. In contrast to the LEGO® Mindstorms™ robot in [Mar01], that was equipped with weak sensors only, it is not reasonable to assume, that a's preconditions can be affirmed by the powerful sensors of the *Pioneer* robot used here.[15] In this case, the materialization of some effects ϑ^+ and ϑ^- indicate an abnormality by "reverse engineering" of state update axioms with weak qualification (specified as in (3.27)):

$$
\begin{aligned}
& Poss(a(\vec{x}), s) \ \wedge \ \exists z \, State(Do(a(\vec{x}), s)) = z \ \wedge \\
& Holds(\vartheta^+, z) \ \wedge \ \neg Holds(\vartheta^-, z) \ \wedge \\
& (\exists e^+) \ \vartheta^+ \circ e^+ = \vartheta_2^+ \ \wedge \ (\exists e^-) \ \vartheta^- \circ e^- = \vartheta_2^- \ \wedge \\
& \neg(\exists e^+) \ \vartheta^+ \circ e^+ = \vartheta_2^+ \ \wedge \ \neg(\exists e^-) \ \vartheta^- \circ e^- = \vartheta_2^- \\
& \qquad\qquad \supset \mathbf{Ab}(p(\vec{x}), s)
\end{aligned}
$$

This follows easily by substituting $Poss(a(\vec{x}), s)$ with its equivalent (again see Formula (3.27)) and a counterfactual argument assuming $\neg\mathbf{Ab}(p(\vec{x}), s)$. The Eclipse-Prolog implementation is shown in Figure 17.

```
weak_qual(A, S, obs(_,CC), [A, exog(ab(Ab))|S]):-
        res(S,Z1),
        materialize(A, CC, Fluents),
        constrain(Fluents, Z2),
        state_update(Z1,A,Z2),
        res(S,Z1a),
        knows(ab(Ab),Z1), \+ knows(ab(Ab), Z1a).
```

Figure 17: Eclipse Code for weak qualifications

The future state `Z2` (a fresh variable) is constrained with the materialized fluents. The state update axiom with `Z2` being partly instantiated would fail for the default case but yield a solution that contains a newly introduced abnormality to `Z1`, which has not been there before (inferred out by the last line).

Qualification preferences are specified by the ordering of Eclipse-Prolog's clause choice will choose defaults in a top-down manner, and once a solution is found it will commit to it by means of the prolog's cut (see alternative solutions discussed in Chapter 6). That way, the suggested partial ordering of extensions (as defined in [Bre94] for prioritized default logic) is implemented in a very simple and efficient way. Although the ordering for defaults might be defined partially, the partial order can be flattened without loss of generality to a respective linearization (ie. total order). Note, that this way no extra numeric level description is necessary. Context dependency as in [Mar01] is neglected, though.

[15] I.e. Action Monitoring, as defined in Chapter 1.2, is applicable.

5.7 Low level vs. High level Execution Monitoring

Monitored information are communicated throughout the layers of a state representation, as indicated by the double-sided arrows in Figure 8.

A strategy for trapping discrepancies between the inner state model and the real world should act on *all* levels in a bottom-up manner: Before bothering high level control about obstacles, accidents or other failures, the corresponding behavior should be eager to fix a problem itself. If impossible, or a timeout occurs, the higher level or an adequate module is notified and handles the situation. It is then the responsibility of the high level module to explain the failure and to replan, where both processes reach beyond the scope of the current action (because of the planners ability to reason both backward and forward in time). This is essential, when action failures might be caused earlier then the current action.

At low level, exceptions of the following kind are supposed to be trapped:

Stalling: Overcome a situation, when the robot got stuck

Obstacle Avoidance: Small avoidable obstacle on the direct way to a goal should be circumnavigated (Saphira's *sfKeepOf* and *sfAvoidCollision*). If they however render impregnable obstacle (timeout), a message is triggered to higher level.

5.8 Low Level Execution Monitoring

The Execution Monitoring routine on low level is adapted from [Gro01].[16] It follows the structure of usual Saphira Processes, i.e. within a fixed time slice of 100 ms it is enabled to perform monitoring. The process cycles through the following states:

- **StateWaiting** waits for activation
- **StateActivate** initializes the process
- **StateWaitForGoal** waits until FLUX Planner sends goal coordinates
- **StateRequestPlan** queries the path planner
- **StateReceivePlan** waits for all packages
- **StatePlanReceived** starts Giovanna's *Gotopos* Behavior
- **StateExecutePlan** Restarts *Gotopos* Behavior with next subgoal node Monitors, whether final target node is reached, in which case **StateExecutionDone** is entered
- **StateMonitorExec** Monitors:
  - Single Subgoal Points reached (subsequent state: **StateExecutePlan**)

[16] modified with respect to activation and interrupt monitoring

  - Checks when wheels are stalled, flag (subsequent state: **StateStalled**)
  - Controls timeout (subsequent state: **StateTimedOut**)
  - Reacts on externally caused interrupts (subsequent state: **StateInterrupt**)

- **StateExecutionDone**
- **StateInterrupt**
- **StateStalled**
- **StateTimedOut**

The latter four states send corresponding notifications to the TCA communication server, deactivate the *GotoPos* behavior and explicitely activate Saphira's *sfStop* behavior [17].

Note, that further monitors are distributed in effector control modules, such as the audio dialog module.

5.9 Cooperating Modules

In this subsection, a brief description of the involved modules — mainly from the Giovanna project ([Gro01]) — and their functionality is given.

5.9.1 Giovanna Path Planning

Path-planning is based on Voronoi diagrams (cf. [For92]), that can be constructed from an artifact map. This method might not always yield the shortest way but maximizes the distance to static obstacles.

The graphical user interface, see screen-shot in Figure 18, offers the display of several graphs. In the implementation, a small amendment was taken by calculating the second last node of the LEDA node list in a way, that prohibits turning angles greater than 90° degrees.

5.9.2 Giovanna Simulator

The elaborated version of the simulator is depicted in Figure 19. Equipped with a "Change World" button, the user is able to simulate dynamic aspects such as appearance/disappearance of obstacles and door states. Furthermore, positional changes can be performed easily by mouse operations.

[17] to avoid slow creeping with a remaining speed from other behaviors during dialogues

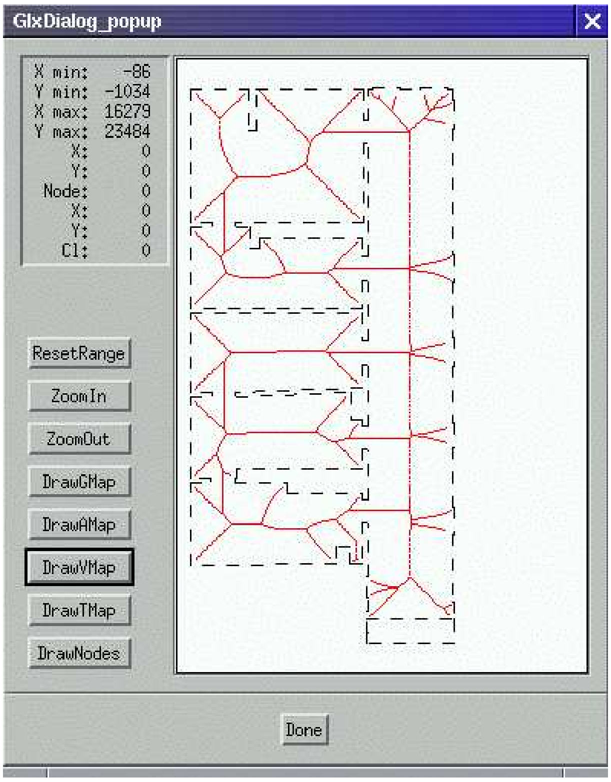

Figure 18: Screen-shot of the path planner

5.9.3 Saphira Controller

With the help of the Saphira controller a set of displaying functions are visualized. These range over

- current behaviors/processes and their respective states
- reckoned position
- auxiliary data such as battery
- visualized sonar data, including recognized artifacts and goal positions

See Figure 20 for a screen-shot.

5.9.4 User Interface

The user interface is displayed in Figure 21. The interaction is kept very simple, but in terms of functionality, a number of control possibilities are given:

- Request Handling

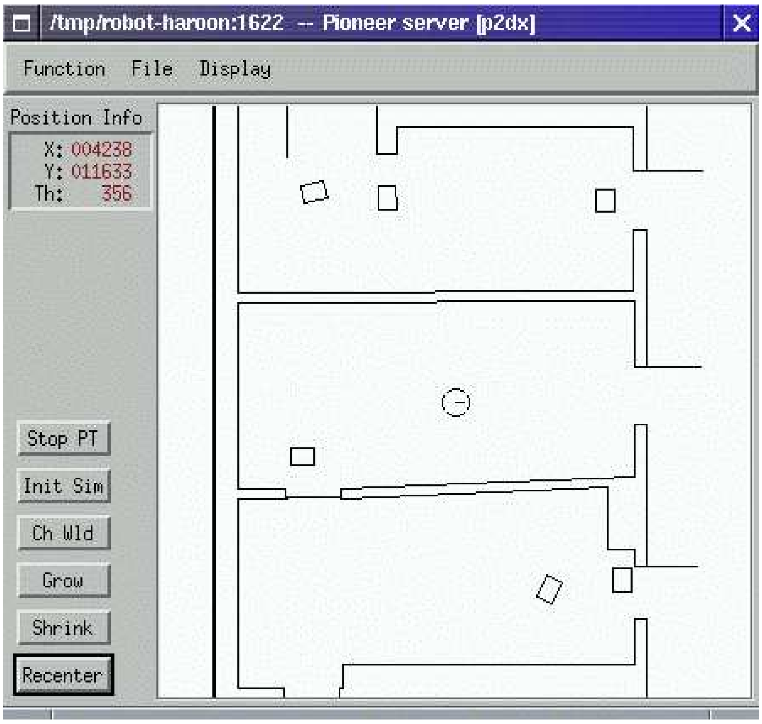

Figure 19: Screen-shot of the simulator

- Adding requests
- Canceling requests
- Adding search requests
- Canceling search requests

• World Changes

- Notification about peoples presence
- Removal of obstacles that formerly hindered the robot
- Opening of doors that formerly hindered the robot

5.10 The Office Delivery Robot

Our office delivery robot lives in an office environment, that has rooms, doors, obstacles, people and their work places. A sample environment is shown in Figure 22. User requests can be ordinary delivery tasks from one object provider to one receiver. Additionally, search tasks can be specified. The initiator specifies a set of people, amongst whom he/she is suspecting an object to be found.
All reported simulations are run with a setup, that at least involves

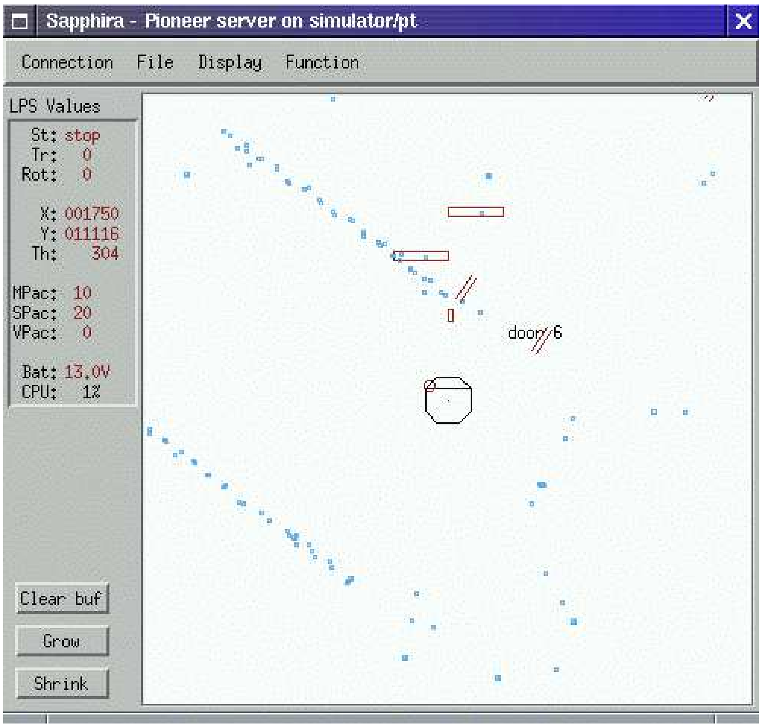

Figure 20: Saphira screen-shot with sonar data

- FLUX Controller
- Pioneer Simulator (with or without doors)
- Position Tracking Module
- Saphira Controller
- Path Planner

Optionally are :

- User interface
- Audio server

Notice, that the simulated version does not have a laser range finder at its disposal. Hence, position tracking is solely based on (simulated) sonar sensor data. Furthermore, no vision system is attached since no camera input is simulated/provided. A screen-shot of the complete scenery is shown in Figure 23.

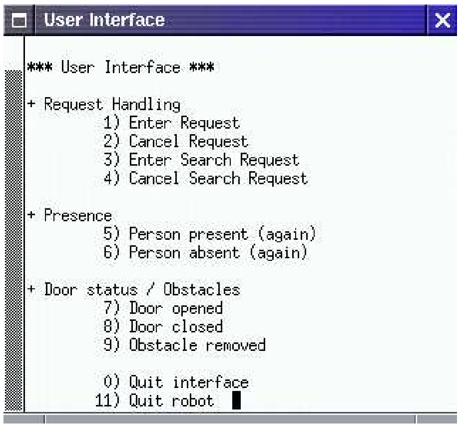

Figure 21: User Interface

5.11 Sample scenarios

In the sequel, parts of the systems functionality are demonstrated by sample scenarios. The presentation of each scenario comprises the FLUX output log together with some brief explanation. The intention is to exemplify crucial state changes on high level.

Examples are run on a *Pioneer 2*-simulator that is realistic in several respects:

- Sonar sensor readings underlie a realistic error distribution.
- Dynamic world changes can be simulated: door states can be changed.
- Obstacles are distributed according to some pre-configured probabilities and can change dynamically as well.

5.11.1 Case 1a): Search Request

The task is to search amongst Uschi, Pascal and Horst for a book, which is supposed to be brought to Andreas.

The robot, in elevator mode, decides to go to Pascal, as he is one of the candidates, where there is a task to fulfill. Within the right room, an obstacle hinders the robot (weak qualification). Knowing that it is in the right room, it is able to ask Pascal for help. On successful confirmation, which causes the fluent $Here(Pascal)$, the robot asks for the book. Pascal answers (hence success), but with "no", so the fluent $MightHave(Pascal, Book, Andreas)$ is no longer valid. The robot walks on (which has a negative effect: $Here(Pascal)$), asks Uschi and is more successful, i.e., it receives the book. Horst (and any other potential candidate for Andreas' book) are erased (from the list of candidates) due to the ramification rules for *Receive*-actions. Hence the book is brought directly to Andreas.

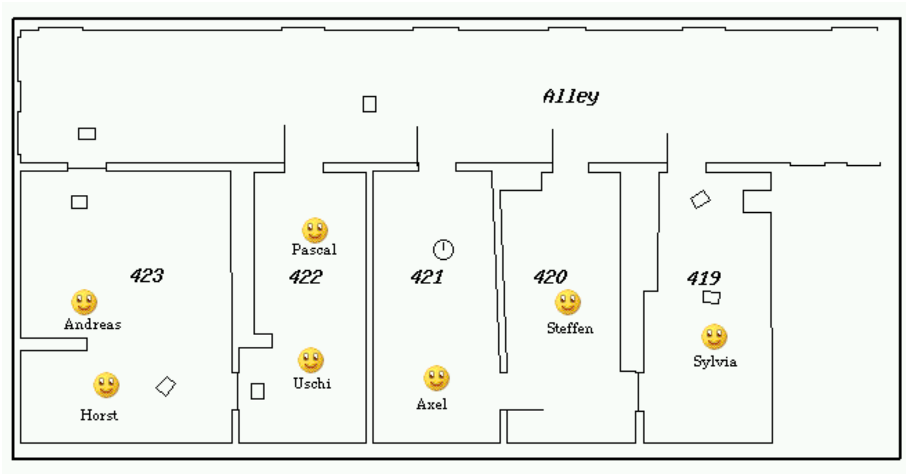

Figure 22: Sample environment with indicated spots for peoples whereabouts

The corresponding log of the FLUX module:

```
[FL] State: [in_room(alley), elevator, ele_up,  ...,
     s_request([uschi, pascal, horst], book, andreas), might_have
     (uschi,book, andreas), might_have(horst, book, andreas),
     might_have(pascal, book, andreas)|_1658]
...
[FL] Agenda: [goto(pascal)]
[FL]  *** Weak qual. occured ***
[FL] State: [ab(reachable(pascal)), in_room(r422), ... |_63195]
[FL] Agenda: [call_help(pascal)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [here(pascal), ... |_63195]
[FL] Agenda: [ask_for(pascal, book, andreas)]
[FL]  *** SUCCESS ***
[FL] State: [here(pascal), s_request([uschi, pascal, horst], book,
     andreas), might_have(uschi, book, andreas), might_have(horst,
     book, andreas)|_81508]
[FL] Agenda: [goto(uschi)]
[FL]  *** SUCCESS ***
[FL] State: [at(uschi), in_room(r422), ...|_81508]
[FL] Agenda: [ask_for(uschi, book, andreas)]
[FL]  *** SUCCESS ***
[FL] State: [has(uschi, book), at(uschi), in_room(r422),
     s_request([uschi, pascal, horst], book, andreas)|_108665]
[FL] Agenda: [receive(book, uschi)]
```

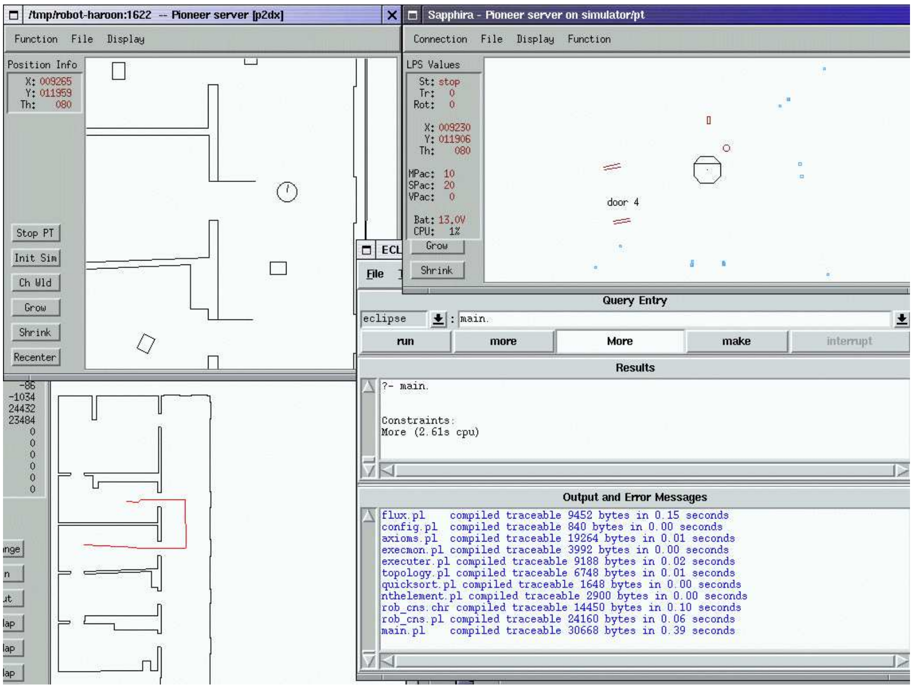

Figure 23: General overview of the system

```
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [carries(book, andreas), at(uschi), in_room(r422),
     ...|_108665]
[FL] Agenda: [goto(andreas)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r423), at(andreas), carries(book, andreas),
     ...|_108665]
[FL] Agenda: [deliver(book, andreas)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [in_room(r423), at(andreas),...|_108665]
[FL] Agenda: [walk_on, chdir, goto(axel)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r423), at(andreas),...|_108665]
[FL] Agenda: [chdir, goto(axel)]
[FL] Changing general direction
[FL]  *** SUCCESS ***
[FL] State: [request(axel, book_1, sylvia), ...|_108665]
[FL] Agenda: [goto(axel)]
```

```
[FL]  *** INTERRUPT ***
[FL] Bailing out peacefully...
```

5.11.2 Case 1b): Search Request (unsuccessful)

If none of the possible candidates has the book, the search request is canceled and the initiator is notified by email (simulated).

The corresponding log of the FLUX module:

```
[FL] Agenda: [goto(horst)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r423), at(horst),..., s_request([uschi,
     pascal, horst], book, andreas), might_have(horst, book,
     andreas)|_47501]
[FL] Agenda: [ask_for(horst, book, andreas)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r423), at(horst),..., s_request([uschi,
     pascal, horst], book, andreas)|_74664]
[FL] Agenda: [email(book, andreas)]
[FL] andreas, your search for the book was unsuccessful, sorry!
[FL]  *** SUCCESS ***
[FL] State: [emailed(book, andreas), in_room(r423), at(horst),...]
[FL] Agenda: [walk_on, chdir, goto(axel)]
[FL]  *** SUCCESS ***
[FL] State: [emailed(book, andreas), in_room(r423), at(horst),...]
...
```

5.11.3 Case 2) The Closest Candidate

This scenario demonstrates the functionality of the high level path planning procedure when in “Closest-Candidate”-mode.

The robot is to deliver a book from Axel to Sylvia, plus the search request as in the previous case. After having received the book, the robot is on the way to Sylvia. In office 420, neither the attempt to reach office 419 directly nor the attempt via the alley work out (depicted in the left part of Figure 24). Being faced with a detour back to office 421, the robot is tempted to postpone the delivery of Sylvia's book and instead heads for Pascal. However, after failing on the way to Pascal on some obstacle in the alley as well, the path to Sylvia again renders to be shortest, so the robot decides to deliver the book to Sylvia, this time successfully.

The corresponding log of the FLUX module:

```
[FL] State: [in_room(alley), request(axel, book, sylvia), request
     (axel, book_2, steffen),  ...|_1654]
```

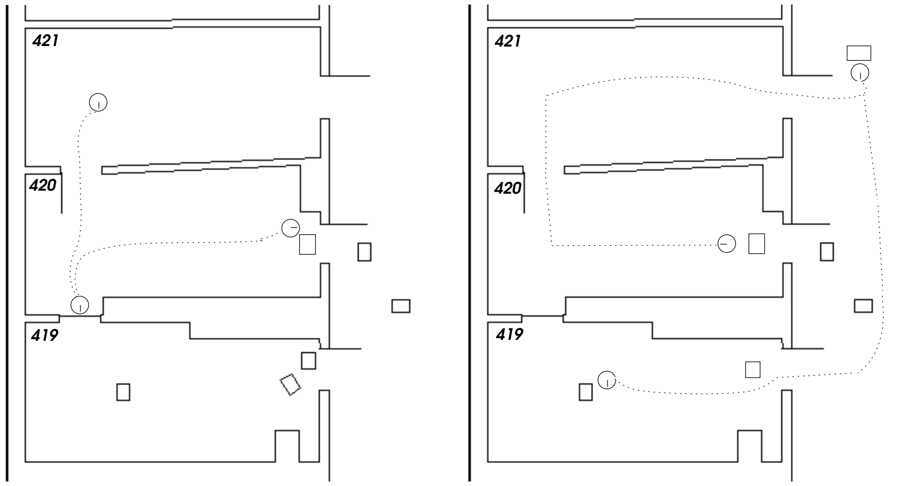

Figure 24: Left: In room 420, some navigation actions fail. The only possible way is back to room 421. Right: On the way to Pascal, failing and attempting to Sylvia again

```
...
[FL] Agenda: [goto(axel)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r421), at(axel), request(axel, book, sylvia),
     request(axel, book_2, steffen),   ...|_1654]
[FL] Agenda: [receive(book, axel)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [carries(book, sylvia), in_room(r421), at(axel),
     request(axel, book_2, steffen),   ...|_1654]
...
[FL] Agenda: [goto(sylvia)]
[FL]  *** Weak qual. occured ***
[FL] State: [ab(reachable(d6, r420)), in_room(r420), carries(book,
     sylvia),   ...|_10751]
[FL] Agenda: [gotodoor(d2), enter(alley), gotodoor(d1),
     enter(r419), r_goto(sylvia)]
[FL]  *** Weak qual. occured ***
[FL] State: [ab(reachable(d2, r420)), ab(reachable(d6, r420)),
     in_room(r420), carries(book, sylvia),   ...|_208047]
[FL] Agenda: [gotodoor(d7), enter(r421), ...  r_goto(pascal)]
[FL]  *** SUCCESS ***
...
[FL] State: [in_room(alley), at(d3), ab(reachable(d2, r420)),
     ab(reachable(d6, r420)), carries(book, sylvia),   ...]
```

```
[FL] Agenda: [gotodoor(d4), enter(r422), r_goto(pascal)]
[FL]  *** Weak qual. occured ***
[FL] State: [ab(reachable(d4, alley)), in_room(alley),
     carries(book, sylvia),   ...|_77913]
[FL] Agenda: [goto(sylvia)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r419), at(sylvia),   ...|_77913]
[FL] Agenda: [deliver(book, sylvia)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [in_room(r419), at(sylvia), ab(reachable(d4, alley)),
     ab(reachable(d2, r420)), ab(reachable(d6, r420)),   ...]
```

5.11.4 Case 3a) Delivery Failure

This case study demonstrates obstacle-circumnavigation on high level, the hierarchy of explanations and the recalculation of the current state after detected abnormalities. An action failure is detected several action steps later, which is an example to true execution monitoring in contrast to simple action monitoring.

The task is to deliver two books from Axel, *Book1* to Sylvia and *Book2* to Steffen. Apart from some detours, execution apparently goes smoothly up to point where the delivery of *Book*1 is supposed to take place. Sylvia complains, that delivery is impossible, which might have several explanations:

- A previous delivery failed (someone took the book that was allotted to Sylvia, i.e. Steffen took *Book*1)
- The book simply got lost

Since these explanations are ordered in a respective preference list, the robot believes in the uppermost case. 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. Having calculated this state, the recovery actions emerge automatically. Notice, that although further requests in other offices exists, the resulting behavior is, that recovery actions are taken immediately. If, e.g. the robot was in elevator mode or other candidates were closer, then the recovery actions where accomplished at a later point of time due to efficiency reasons.

The corresponding log of the FLUX module:

```
[FL] State: [in_room(alley), request(axel, book_1, sylvia),
     request(axel, book_2, steffen),  ... |_1654]
[FL] Agenda: [goto(axel)]
```

```
...
[FL] State: [here(axel), in_room(r421), request(axel, book_1, sylvia),
     request(axel, book_2, steffen),  ... |_152432]
[FL] Agenda: [receive(book_1, axel)]
[FL]  *** SUCCESS ***
[FL] State: [carries(book_1, sylvia), here(axel), in_room(r421),
     request(axel, book_2, steffen),  ... |_152432]
[FL] Agenda: [receive(book_2, axel)]
[FL]  *** SUCCESS ***
[FL] State: [carries(book_2, steffen), carries(book_1, sylvia),
     here(axel), in_room(r421),  ... |_152432]
[FL] Agenda: [goto(steffen)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r420), at(steffen), carries(book_2, steffen),
     carries(book_1, sylvia),  ... |_152432]
[FL] Agenda: [deliver(book_2, steffen)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [in_room(r420), at(steffen), carries(book_1, sylvia),
     ... |_152432]
[FL] Agenda: [walk_on, goto(sylvia)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r420), at(steffen), carries(book_1, sylvia),
     ... |_152432]
[FL] Agenda: [goto(sylvia)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r419), at(sylvia), carries(book_1, sylvia),
     ... |_152432]
[FL] Agenda: [deliver(book_1, sylvia)]
[FL] Enter Action result:  1
[FL]  *** Strong qual. occured ***
[FL] Abnormality occured: [delivery(steffen, book_1)]
[FL] State: [request(steffen, book_1, sylvia), in_room(r419),
     at(sylvia), carries(book_2, steffen),  ... |_465630]
...
[FL] Agenda: [goto(steffen)]
[FL]  *** Weak qual. occured ***
[FL] State: [ab(enterable(d6)), at(d6), carries(mail, steffen),
     ... |_137446]
[FL] Agenda: [gotodoor(d1), enter(alley), gotodoor(d2),
     enter(r420), r_goto(steffen)]
...
[FL] State: [at(steffen), in_room(r420), request(steffen,
     book_1, sylvia), carries(book_2, steffen),  ... |_137446]
[FL] Agenda: [deliver(book_2, steffen)]
[FL] Enter Action result:  0
```

```
[FL]  *** SUCCESS ***
[FL] State: [at(steffen), in_room(r420), ab(enterable(d6)),
     request(steffen, book_1, sylvia),  ... |_137446]
[FL] Agenda: [receive(book_1, steffen)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [carries(book_1, sylvia), at(steffen), in_room(r420),
      ab(enterable(d6)),  ... |_137446]
[FL] Agenda: [walk_on, gotodoor(d2), enter(alley), gotodoor(d1),
      enter(r419), r_goto(sylvia)]
[FL]  *** SUCCESS ***
...
[FL] State: [at(sylvia), in_room(r419), carries(book_1, sylvia),
     ab(enterable(d6)),  ... |_137446]
[FL] Agenda: [deliver(book_1, sylvia)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
```

5.11.5 Case 3b) Delivery Failure - Alternative

If, however, no delivery happened after the missing object of interest was put on the robot's tray, the first explanation does not do and the second explanation is chosen.

The corresponding log of the FLUX module:

```
[FL] State: [in_room(alley), request(axel, book_1, sylvia),
     request(axel, book_2, steffen), ... |_1654]
[FL] Agenda: [goto(axel)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r421), at(axel), request(axel, book_1,
     sylvia), request(axel, book_2, steffen), ... |_1654]
[FL] Agenda: [receive(book_1, axel)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [carries(book_1, sylvia), in_room(r421), at(axel),
     request(axel, book_2, steffen), ... |_1654]
[FL] Agenda: [receive(book_2, axel)]
[FL] Enter Action result:  0
[FL]  *** SUCCESS ***
[FL] State: [carries(book_2, steffen), carries(book_1, sylvia),
     in_room(r421), at(axel), ... |_1654]
[FL] Agenda: [goto(steffen)]
[FL]  *** SUCCESS ***
[FL] State: [in_room(r420), at(steffen), carries(book_2,
     steffen), carries(book_1, sylvia), ... |_1654]
```

```
[FL] Agenda: [deliver(book_2, steffen)]
[FL] Enter Action result:  1
[FL]  *** Strong qual. occured ***
[FL] Alert: book_2 got lost!
[FL] State: [ab(lost(book_2)), in_room(r420), at(steffen),
     carries(book_1, sylvia), ... |_274146]
[FL] Agenda: [goto(sylvia)]
[FL]  *** INTERRUPT ***
```

6 Conclusion

The presented work demonstrates the successful combination of various subfields of robotics. Moreover, it can be seen as an investigation, to what extent logic in general and the Fluent Calculus in particular can contribute to real world applications within robotics.

Promising results could be documented in several respects: Firstly, high level domain description could be utilized for action planning in the extended delivery domain, where multiple and possibly complex delivery tasks were to be solved. Secondly, the employment of the Fluent Calculus along with its augmentations assisted during execution monitoring by means of explanation provision and corrective action planning in the case of execution failures.

The combination of various different modules required smooth integration in an adequate architecture with a comprehensive communication system. The suggested solutions in Chapter 4 are dedicated to this issue in general and were exemplified in the mentioned sample scenario in Chapter 5.

6.1 Revision of evaluation criteria

Initially, key criteria for performance were mentioned and are revised below:

Scalability and modularity: The system can easily be extended with additional modules. In fact, information about some world changes — by now provided via the user interface — can come from a surveillance system without change.[18]

Reliability: Due to execution monitoring on several levels reliability was enhanced considerably, in the sense that a high number of user requests could be fulfilled successfully or at least an appropriate failure message could be generated.

Flexibility: As mentioned initially, certain assumptions about the knowledge of the environment were made. Nonetheless, it has been argued, that with little effort the robot can adapt to other task specifications or major changes in the environmental setup.

Efficiency: "Multiple-request" planning in two different modes together with the possibility for immediate user interrupts provided a high standard of efficiency.

Remark. Reliability and efficiency are sometimes conflicting goals, which requires a convenient trade-off. For example, the path planner does not deliver the shortest route for the sake of obstacle avoidance.

[18] It suffices to register the new module and obey the original message format.

6.2 Future Work

Preference lists for action failures involve a great deal of speculation and need to be preprogrammed in advance. An alternative method would be "hypothesis testing"[19]: 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 within the field of Active Vision. The application of a cognitive planner for sensing actions in the above outlined fashion could help 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.

6.2.1 Fluent Calculus and MDPs

It is a promising attempt to combine the Fluent Calculus and stochastic approaches.

As seen in Equation 3.4, *all* elements of $\mathcal{S}$ and $\mathcal{A}$ are taken into consideration, leading to exponential complexity in the number of state variables.

Given a (possibly disjunctive and incomplete) description of the current state, the reachability structure is computed on the base of disjunctive state update axioms. This idea is based on the assumption that the transition function T maps only on a restricted subspace of $\mathcal{S}$. Reachability analysis allows one to restrict value and policy computation to states reachable by some action sequence (or action tree). Moreover, the set of considerable actions does not need to range over $\mathcal{A}$, but can be restricted by means of state dependent strategies.

This form of *directed value iteration* can be implemented in the Fluent Calculus in a straightforward way: As opposed to the Situation Calculus, a single *action* plays the central role of each state update axiom within the Fluent Calculus. Hence, the calculation of successor states and the utility function $V$are tightly coupled: given a state $s \in \mathcal{S}$, the state transition function T should be constrained to all executable actions and reachable states (with nonzero probability) by means of the state update axiom.

6.3 Acknowledgements

I owe great thanks to my two supervisors Michael Thielscher and Axel Großmann for their guidance on all aspects of this thesis. I would also like to thank Yves Martin for the fruitful discussions. Furthermore, I am grateful to Matthias Fichtner, Sebastian Bader and Jan Eichhorn for the valuable contributions in both substance and style. Last but not least, Muna and Gihan deserve a big "bossa" for their moral support.

[19]This term has been coined differently in several contexts—an appropriate definition from statistics declares hypothesis testing to be "rules for the inspection/examination of conjectures against reality"

A Source Code Digest

A.1 High Level

A.1.1 Flux Module

The file `$(GIOVANNA)/highlevel/eclipse/HomuncuFlux/main.pl`

```
:- [flux].
:- [config].
:- [axioms].
:- [execmon].
:- [executer].
:- [topology].
:- [quicksort].
:- [nthelement].


:- chr2pl(rob_cns),[rob_cns].
:- dynamic(initial_room/1).
:- assert(initial_room("alley")). %default

cons(Z) :- duplicate_free(Z),
        %possessive(Z),
        %request_del(Z),
        %\+ undelay(Z),
        %\+ reach_incons(Z),
        %lost_item(Z),
        holds(in_room(_), Z, Z1),
        not_holds_all(R, in_room(R), Z1),
        at_unique(Z).

/* Inconsistent: a person being reachable but marked as
unreachable */
reach_incons(Z):-
        knows(unreach(P),Z), detailed_pp(P, Z, _).

undelay(Z):- knows(delayed_req(P,_,_),Z),
        detailed_pp(P, Z, _).

init_room:-
        initial_room(R),
        retract(initial_room(R)),
        (sim_mode(on)->R1="alley"; curpos(N), node2room(N,R1)),
        assert(initial_room(R1)).

init2(Z):-Z=[in_room("r421"), at("axel"), request("axel","book",
"steffen")|Z1], cons(Z1).

init(Z0) :- initial_room(R),
        Z0 = [in_room(R),
```

```
              %elevator, ele_down,
              % uncomment the above line for elevator mode
              %s_request(["andreas", "axel", "uschi"],
"book", "horst"),
                  %carries(coffee, axel),
                  %ab(reachable(d3, alley)),
                  %request(horst, scrue, pascal),
                  %request(horst, tea, uschi),
                  %request("axel", "file", "steffen"),
                  %request(horst, book, steffen),
                  %carries(coffee_milk, sylvia),
                  %carries(mail, steffen),
                  %unreach(horst),
                  %carries(book, andreas),
              %ab(traceable("axel")),
              request("axel","book_1","sylvia"),
              request("axel","book_2","steffen"),
              request("sylvia","mail","steffen"),
              s_request(["uschi","pascal","horst"],"book",
"andreas"),
              might_have("uschi", "book", "andreas"),
              might_have("horst", "book", "andreas"),
              might_have("pascal", "book", "andreas")
                                  %has(pascal, book)
                                  %ab(enterable(d5)),
                                  %ab(enterable(d3))
             | Z],
        not_holds_all([O,P], carries(O,P), Z),
        not_holds_all([R1,Y,R2], request(R1,Y,R2), Z),
        cons(Z0).

next_up(Z, PN):-
        curpos(node(_,Y)),
        people_up(Y, PL),
        candidates(Cs, Z),
        try_first(PL, Cs, PN),!.

next_down(Z, PN):-
        curpos(node(_,Y)),
        people_dn(Y, PL), reverse(PL, PL1),
        candidates(Cs, Z),
        try_first(PL1, Cs, PN),!.

people_up(Y,PU):- findall(P, (p_coords(P,_,Yp),Yp>Y), PU).
people_dn(Y,PU):- findall(P, (p_coords(P,_,Yp),Yp<Y), PU).

try_first([P|_],Cs, P):-member(P,Cs),!.
try_first([_|Ps],Cs, P):-try_first(Ps,Cs,P).

% elevator mode:
```

```
continue(GoAct, Z):-
        knows(elevator, Z),
        knows(ele_up, Z),
        next_up(Z, PN),!,
        hierarch_pp(PN, Z, Go, _),
        reverse(Go, GoAct).

continue([chdir|GoAct], Z):-
        knows(elevator, Z),
        knows(ele_up, Z), % failed above -> thus go down
        next_down(Z, PN),!,
        hierarch_pp(PN, Z, Go, _),
        reverse(Go, GoAct).


continue(GoAct, Z):-
        knows(elevator, Z),
        knows(ele_down, Z),
        next_down(Z, PN),!,
        hierarch_pp(PN, Z, Go, _),
        reverse(Go, GoAct).

continue([chdir|GoAct], Z):-
        knows(elevator, Z),
        knows(ele_down, Z), % failed above -> thus go up
        next_up(Z, PN),!,
        hierarch_pp(PN, Z, Go, _),
        reverse(Go, GoAct).

% goto-closest mode:

continue(GoAct, Z):-
        closest(CR, Z), hierarch_pp(CR, Z, Go, _),
        reverse(Go, GoAct).

notify_reach(Z, UR):- reachability(Z,UR), UR\=[].

reachability(Z, UR):-
        candidates(LoP1, Z),
        reach_check(LoP1,Z,UR).

reach_check([P|T], Z, [unreach(P)|UP]):-
        \+ knows(at(P), Z),
        \+ knows(alerted(P), Z),
        \+ (knows(in_room(R),Z), office(R,P),
              \+ knows(ab(traceable(P)),Z)),
        \+ detailed_pp(P, Z, _),!,
        reach_check(T,Z,UP).

reach_check([_|T], Z, T1):-
```

```
        reach_check(T,Z,T1).
reach_check([],_,[]).

remove_duplicates([A|T], [A|TwoA1]):-
        remove(A, T, TwoA),
        remove_duplicates(TwoA,TwoA1).
remove_duplicates([], []).
remove(El, [El|R], R1):-
        remove(El,R,R1),!.
remove(El, [H|T], [H|TwoEl]):-remove(El, T, TwoEl).
remove(_, [], []).

closest(CR, Z):-
        candidates(LoP1, Z),
        find_closest(LoP1, Z, CR).

candidates(LoP1, Z):-
        findall(P,(knows(request(P,_,_),Z),
           \+ knows(unreach(P),Z),
           \+ knows(ab(traceable(P)),Z)), LoR),
        findall(Q,(knows(carries(I,Q),Z),
            \+ knows(ab(lost(I)),Z),
            \+ knows(unreach(Q),Z),
            \+ knows(ab(traceable(P)),Z)), LoQ),
        findall(SL,(knows(might_have(SL,_,_),Z),
            \+ knows(unreach(SL),Z),
            \+ knows(ab(traceable(SL)),Z) ), LoS),
        flatlist([LoR, LoQ, LoS], LoP),
        remove_duplicates(LoP, LoP1).

find_closest(LoR, Z, CR):- find_closest(LoR, Z, CR, _).
find_closest([R0|T], Z, ClosR, MinC):-
        find_closest(T, Z, R1, Cost1),
        hierarch_pp(R0, Z, _, Cost0),
        (Cost1<Cost0
        ->  MinC=Cost1, ClosR = R1;
            MinC=Cost0, ClosR = R0).

find_closest([R],Z,R,Cost):- hierarch_pp(R, Z, _, Cost).

hierarch_pp(P, Z, [goto(P)], Cost):-
        sim_mode(off),
        ask_for_route(P, Route),
        succ_split(P, Route, AS),
        executable(AS, Z),
        route_length(Route, Cost),!.

hierarch_pp(P, Z, GoAct, Cost):-
        detailed_pp(P, Z, GoAct),
        cost(GoAct, Z, _, Cost).
```

```
ask_for_route(P, Route):-
        curpos(CP),
        p_coords(P, X, Y),
        ask_pp(CP, node(X,Y), Route).

cost([A|GoAct], Z0, Z2, Cost):-
        cost(GoAct, Z0, Z1, C1),
        where(Z1, N1),
        state_update(Z1, A, Z2),
        where(Z2, N2),
        route_length([N1,N2], RouteL),
        Cost is RouteL+C1.

cost([A], Z1, Z2, RouteL):-
        (sim_mode(on)->curpos(CP,Z1);curpos(CP)),
        state_update(Z1, A, Z2),
        where(Z2, N),
        route_length([CP,N], RouteL).
cost([],_,_,0).

where(Z, node(X,Y)):- knows(at(P),Z),
        person(P), p_coords(P,X,Y),!.
where(Z, node(X,Y)):- knows(at(D),Z),
        door(D),!,
        knows(in_room(R),Z),
        gotopos(gtd, R, D, X, Y, _).
where(Z, node(X,Y)):-
        knows(in_room(R), Z),
        office(R,P),!,
        p_coords(P, X,Y).

curpos(CP,Z):-where(Z, CP).     % mocked up

detailed_pp(P, Z, GoPlan):-
        knows(in_room(R1), Z),
        office(R2, P),
        setof(R, route(R1,R2,[],R), SoR),
        quicksort(SoR,SR),
        find_executable(SR, P, Z, GoPlan),!.

find_executable(SR, P, Z, AL1):-
        member(L,SR),
        convert(L,Z,AL),
        (knows(at(P),Z) -> AL1=AL; AL1=[r_goto(P)|AL]),
        executable(AL1, Z).

convert([R1,R2|T],Z, AL):-
        connects(D, R1, R2),
        convert([R2|T],Z, CT),
```

```
        (knows(at(D), Z) -> AL = [enter(R1)|CT];
                 AL =[enter(R1),gotodoor(D)|CT]).
convert([_],_,[]).

executable([],_).
executable([A|T], Z):-executable([A|T], Z, _).
executable([A|T], Z, Z2):-
        executable(T,Z,Z1),
        poss_succ(A,Z1),!,
        state_update(Z1,A,Z2).
executable([], Z, Z).

route_length([node(X1,Y1),node(X2,Y2)|T],C):-
        route_length([node(X2,Y2)|T],RL),
        XYS is (X1-X2)*(X1-X2)+(Y1-Y2)*(Y1-Y2),
        sqrt(XYS, XYSR),        % Pythagoras
        C is RL+XYSR.
route_length([_],0).

route(R,R, RL, [R|RL]):- !.

route(R1, R2, RL,FRL):-
        connects(_,R1,R3),
        /* check abnorm... */
        \+ member(R3, RL),
        route(R3, R2, [R1|RL], FRL).

poss_succ(enter(R), Z):-
        poss(enter(R), Z),      % no strong abn
        state_update(Z, enter(R), Z1),
        knows(in_room(R), Z1).  % no weak abn

poss_succ(gotodoor(D), Z):-
        poss(gotodoor(D), Z),   % no strong abn
        state_update(Z, gotodoor(D), Z1),
        knows(at(D), Z1). % no weak abn

poss_succ(r_goto(P), Z):-
        poss(r_goto(P), Z),     % no strong abn
        state_update(Z, r_goto(P), Z1),
        knows(at(P),Z1). % no weak abn

search([ask_for(P,I,R)], Z):-
        knows(might_have(P,I,R),Z),
        (knows(at(P), Z),!; knows(here(P), Z)).

call_help(P,Z):-poss(call_help(P), Z), %    poss
        candidates(C,Z), member(P, C). %  necessary

delivery(O, P, Z) :- poss(deliver(O,P),Z),
```

```
        \+ exception(deliver(O,P),Z).

receipt(O, P, Z) :- poss(receive(O,P),Z),
        \+ exception(receive(O,P),Z).

search_fail(O,P,Z):-
        knows(s_request(_,O,P),Z),
        \+ knows(has(_,O), Z),
        \+ knows(might_have(_,O,P), Z),
        \+ knows(emailed(O,P),Z).

main :- init_room,init(Z),loop([],Z).

loop(_,Z):- knows(quit,Z),
            writeln("[FL] Bailing out peacefully..."), !.

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).

execute([A|L], S, Z):-
        write("[FL] State: "), writeln(Z),
        write("[FL] Agenda: "),writeln([A|L]),
        execute(A, Z, FB, Obs),
        administrate(Obs, S, S1, Z, Z1),
        state_update(Z1, A, Z2),
        wr_feedback(FB),
        (
          ll_success(FB) -> execute(L, [A|S1], Z2);
          ll_special(FB) -> loop([A|S1], Z2);
          % success+update, deletes old agenda
          ll_strn_ql(FB) -> strong_qual(A,S1, AA), recover(A,
                                                    AA, ActAb),
                            res(ActAb,Zab), loop(ActAb, Zab);
          ll_weak_ql(FB) -> weak_qual(A, S1, Obs, ActAb),
                            res(ActAb,Zab), loop(ActAb, Zab);
          ll_intrupt(FB) -> br, state_detect(Z1,Z3,Ai),
                            loop([Ai|S1], Z3);
          true
```

```
        ).

execute([],S,Z):-loop(S,Z).

/* Incorporates Observations like async. requests, sensed values
   Performs according sua's (in res)
*/

wr_feedback(0):- writeln("[FL]  *** SUCCESS ***").
wr_feedback(1):- writeln("[FL]  *** Strong qual. occured ***").
wr_feedback(2):- writeln("[FL]  *** Weak qual. occured ***").
wr_feedback(3):- writeln("[FL]  *** SUCCESS + UPDATE ***").
wr_feedback(4):- writeln("[FL]  *** INTERRUPT ***").

state_detect(Z1, Z2, go_room(R)):-
        curpos(N),
        node2room(N,R),
        state_update(Z1, go_room(R), Z2).

administrate(obs([],_), S, S, Z, Z):- !.
administrate(obs(Fluents,_), S, S1, _, Z1):-
        fluent_list(Fluents, FL),
        append(FL, S, S1),
        res(S1, Z1).

fluent_list([F|FL], [exog(F)|FL1]):-
        F \=  -(_), fluent_list(FL,FL1).
fluent_list([-F|FL], [unexog(F)|FL1]):-
        fluent_list(FL,FL1).
fluent_list([],[]).

recover(deliver(O,P), AA,
        [exog(request(P1,O,P)), unexog(ab(delivery(P1,O)))|AA]):-
        res(AA,Z), knows(ab(delivery(P1,O)), Z),!.
recover(_,S,S).

strong_qual(A, S, ActAb):-
        exogcaused([A|S], [A|ActAb], Ab),
        res(ActAb, Z),
        (\+poss(A,Z); exception(A,Z)),!,
        alert(Ab).

weak_qual(goto(Spot), S, Obs, ActAb):-!,
        split(goto(Spot), Obs, [FA|As]),
        append(As,S,Act),
        weak_qual(FA, Act, Obs, ActAb).

weak_qual(A, S, obs(_,CC), [A, exog(ab(Ab))|S]):-
        res(S,Z1),
        materialize(A, CC, Fluents),
```

```
        constrain(Fluents, Z2),
        state_update(Z1,A,Z2),
        res(S,Z1a),
        knows(ab(Ab),Z1), \+ knows(ab(Ab), Z1a).

split(goto(S), obs(_,cc(RN,AN)), [FPA|SPA]) :- !,
        reverse(RN, RC),
        reverse(AN, AC),
        succ_path_actions(RC, AC, SPA),
        failed_path_action(RN, AC, S, FPA).

split(A,_,[A]).

succ_split(P, R, SA):-
        p_coords(P,X,Y),
        split(goto(P), obs(_,cc(R,[node(X,Y)])) , SA).

materialize(gotodoor(D), cc([_|_], _), [-at(D)]):-!.

materialize(enter(R), _, [-in_room(R)]):- !.
materialize(r_goto(P), _, [-at(P)]):- !.
materialize(call_help(P), _, [-here(P)]):- !.
materialize(_,_,[]).

exogcaused(Act, AbIAct, Ab):- default(Ab),
                              insert_ab(Ab, Act, AbIAct).

insert_ab(Ab, [A|AL], [A, exog(Ab)|AL]):- \+ irrelevant(Ab, A).
insert_ab(Ab, [A|AL], [A|AL1]):-
        insert_ab(Ab, AL, AL1).

irrelevant(ab(delivery(_,_)), A) :-  A \= deliver(_,_).


flatlist([H|T], L):-
        flatlist(T,T1),
        append(H,T1, L).
flatlist([],[]).


alert(ab(lost(I))):- write("[FL] Alert: "), write(I),
                     writeln(" got lost!"),!.
alert(ab(Ab)):- write("[FL]: Abnormality occured: "),
                writeln([Ab]).

% Strong Qualifications:

default(ab(delivery(_,_))).
default(ab(lost(_))).
default(ab(traceable(_))).
```

```
default(ab(mobility)).
```

A.1.2 Axiomatization

The file `$(GIOVANNA)/highlevel/eclipse/HomuncuFlux/axioms.pl`

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                 Preconditions                               %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

poss(call_help(P), S):-
        knows(ab(reachable(P)), S),
        knows(in_room(R), S), office(R,P),
        \+ knows(here(P), S),
        \+ knows(ab(traceable(P)),S).

poss(goto(P), S) :-
        (knows(at(P1),S) -> P1 \= P; true),
        \+ exception(goto(P), S).

poss(r_goto(P), S) :-
        office(R,P),
        knows(in_room(R), S),!,
        \+ knows(ab(mobility), S),
        \+ knows(ab(stalled), S).

poss(gotodoor(D), S) :-
        knows(in_room(R), S),!,
        connects(D, R, _),
        \+ knows(at(D), S),
        \+ knows(ab(mobility), S).

poss(enter(R), S) :-
        knows(at(D), S),
        knows(in_room(R1), S),
        connects(D, R1, R).

poss(receive(O,P),S):-
        knows(in_room(R),S), office(R,P),
        (knows(at(P), S); knows(here(P), S)),
        (knows(request(P,O,_),S); knows(has(P,O),S)).

poss(deliver(O,P),S):-
        knows(carries(O,P), S),
        (knows(at(P), S),
            knows(in_room(R), S), office(R,P)
        ;
            knows(here(P), S)).

poss(go_room(_),_).
```

```
poss(idle, _).
poss(chdir, _).
poss(email(_,_), _).
poss(exog(_),_).
poss(unexog(_),_).
poss(add_request(_,_,_),_).
poss(ask_for(_,_,_),_).
poss(walk_on,_).

exception(goto(_), S):-
        knows(ab(mobility), S).

exception(receive(_,P), S):-    knows(ab(traceable(P)), S).

exception(enter(_), S):- knows(ab(mobility), S).

exception(deliver(O,P), S):-
        knows(ab(lost(O)),S),!;
        knows(ab(traceable(P)), S).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                 State Update Axioms                         %%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% abstract fc-path-planning!
% goes from an arbitrary spot to another, neglecting all details

state_update(Z1, chdir, Z2):-
        knows(ele_up, Z1),!,
        update(Z1,[ele_down],[ele_up], Z2).
state_update(Z1, chdir, Z2):-
        knows(ele_down, Z1),
        update(Z1,[ele_up],[ele_down], Z2).

state_update(Z1, idle, Z1).
% within same office !
state_update(Z1,  goto(Spot2), Z2):-
        knows(at(Spot1), Z1),
        knows(in_room(R1), Z1),
        office(R1,Spot2),
        update(Z1, [at(Spot2)], [at(Spot1)], Z2),!.

state_update(Z1,  goto(Spot2), Z2):-
        knows(in_room(R1), Z1),
        office(R1,Spot2),
        update(Z1, [at(Spot2)], [], Z2),!.

state_update(Z1,  goto(Spot2), Z2):-
        knows(at(Spot1), Z1),
        knows(in_room(R1), Z1),
```

```
        office(R2,Spot2),
        update(Z1, [at(Spot2), in_room(R2)],
                   [at(Spot1), in_room(R1)], Z2),!.

state_update(Z1,  goto(Spot2), Z2):-
        knows(in_room(R1), Z1),
        office(R2,Spot2),
        update(Z1, [at(Spot2), in_room(R2)], [in_room(R1)], Z2).


state_update(Z1, go_room(R), Z2):-
        knows(in_room(R), Z1), knows(at(Spot), Z1),
        update(Z1,[],[at(Spot)],Z2),!.

state_update(Z1, go_room(R), Z1):-
        knows(in_room(R), Z1),!.

state_update(Z1, go_room(R), Z2):-
        knows(in_room(R1), Z1), knows(at(Spot), Z1),
        update(Z1,[in_room(R)],[in_room(R1),at(Spot)],Z2),!.

state_update(Z1, go_room(R), Z2):-
        knows(in_room(R1), Z1),
        update(Z1,[in_room(R)],[in_room(R1)],Z2),!.


state_update(Z1,  receive(O,P), Z2) :-
        knows(request(P,O,R),Z1),!,
        update(Z1, [carries(O,R)],[request(P,O,R)], Z2).

state_update(Z1, receive(O,P), Z2) :-
        knows(has(P,O),Z1), knows(s_request(LP,O,D),Z1),
        member(P,LP),
        update(Z1, [carries(O,D)],[s_request(LP,O,D),
                                   has(P,O)], Z2).


state_update(Z1,  deliver(O,P), Z2) :-
        \+ knows(ab(delivery(P,_)), Z1),
        update(Z1, [], [carries(O,P)], Z2),!.

/* TODO */
state_update(Z1,  deliver(O,P), Z2) :-
        holds(ab(delivery(P,O1)), Z1),
        knows(carries(O1, P1), Z1),
        O1 \= O, P\=P1,
        update(Z1, [], [carries(O1,P1)], Z2).

state_update(Z1, add_request(P1,O,P2), Z2):-
        update(Z1, [request(P1, O, P2)], [], Z2).
```

```
%-------------------------------------------------------------------
%  detailed fc-path-planning ...

% gotodoor with weak qualification!

state_update(Z1,  gotodoor(D), Z2) :-
        knows(in_room(R), Z1),
        not_holds(ab(reachable(D,R)), Z1),
        knows(at(D1), Z1), update(Z1, [at(D)], [at(D1)], Z2),!.

state_update(Z1,  gotodoor(D), Z2) :-
        knows(in_room(R), Z1),
        not_holds(ab(reachable(D,R)), Z1),
        not_holds_all(D1, at(D1), Z1),
        update(Z1, [at(D)], [], Z2),!.

state_update(Z1,  gotodoor(D), Z2) :-
        knows(in_room(R), Z1),
        holds(ab(reachable(D,R)), Z1),
        holds(at(D1), Z1),!,
        update(Z1, [], [at(D1)], Z2).

state_update(Z1,  gotodoor(D), Z2) :-
        knows(in_room(R), Z1),
        holds(ab(reachable(D,R)), Z1),
        not_holds_all(D1, at(D1), Z1),!,
        update(Z1, [], [], Z2).


%  enter with weak qualification

state_update(Z1,  enter(R), Z2) :-
        knows(at(D), Z1),
        not_holds(ab(enterable(D)), Z1),
        holds(in_room(R1), Z1),
        update(Z1, [in_room(R)], [in_room(R1)], Z2),!.

state_update(Z1,  enter(_), Z1) :-
        knows(at(D), Z1),
        holds(ab(enterable(D)), Z1),!.

state_update(Z1,  r_goto(Spot), Z1) :- knows(at(Spot), Z1),!.

state_update(Z1,  r_goto(Spot), Z2) :-
        not_holds(ab(reachable(Spot)), Z1),
        knows(at(S), Z1),
        update(Z1,[at(Spot)],[at(S)],Z2),!.

state_update(Z1,  r_goto(Spot), Z2) :-
```

```
        not_holds(ab(reachable(Spot)), Z1),
        not_holds_all(S1, at(S1), Z1),
        update(Z1,[at(Spot)],[],Z2),!.

state_update(Z1,  r_goto(Spot), Z2) :-
        holds(ab(reachable(Spot)), Z1),
        knows(at(S), Z1),
        update(Z1,[],[at(S)],Z2),!.

state_update(Z1,  r_goto(Spot), Z2) :-
        holds(ab(reachable(Spot)), Z1),
        not_holds_all(S1, at(S1), Z1),
        update(Z1,[],[],Z2),!.

state_update(Z1, unreach(P), Z2):-
        update(Z1, [alerted(P), unreach(P)], [], Z),
        ramify(Z, [unreach(P)], [], Z2).

state_update(Z1, cancel_req(P1,O,P2), Z2):-
        update(Z1, [], [request(P1,O,P2)], Z2).

state_update(Z1, exog(E), Z2):-
        update(Z1, [E], [],Z), ramify(Z, [E], [], Z2).

state_update(Z1, unexog(E), Z2):-
        update(Z1, [], [E], Z), ramify(Z, [], [E], Z2).


state_update(Z1, ask_for(_,_,_), Z1) :-
        true.% handled by exog

% call for help - weak (called person might not come)

state_update(Z1, call_help(P), Z2) :-
        not_holds(ab(traceable(P)), Z1),
        update(Z1, [here(P)], [ab(reachable(P))], Z2),!.

state_update(Z1, call_help(P), Z1) :-
        holds(ab(traceable(P)), Z1),!.

state_update(Z1, walk_on, Z2) :-
        knows(here(P), Z1), update(Z1, [], [here(P)], Z2),!.

state_update(Z1, walk_on, Z1).

state_update(Z1, email(O,P), Z2):-
        update(Z1, [emailed(O,P)], [], Z2).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%                 Ramifications                           %%%%
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

causes(has(_,I), -might_have(P,I,R), Z, EP, _):-
        holds(has(_,I), EP), knows(might_have(P,I,R),Z).

causes(unreach(P), delayed_req(P,I,R), Z, EP, _):-
        holds(unreach(P), EP),
        knows(request(P,I,R),Z), \+ knows(delayed_req(P,I,R),Z),
        nonvar(P).

causes(request(P,I,R), delayed_req(P,I,R), Z, EP, _):-
        holds(request(P,I,R), EP),
        knows(unreach(P),Z), \+ knows(delayed_req(P,I,R),Z),
        nonvar(P).

causes(delayed_req(P,I,R), -request(P,I,R), Z, EP, _):-
        holds(delayed_req(P,I,R), EP),
        knows(request(P,I,R),Z).

causes(-unreach(P), request(P,I,R), Z, _, EN):-
        holds(unreach(P), EN),
        knows(delayed_req(P,I,R),Z), \+ knows(request(P,I,R),Z).


causes(request(P,I,R), -delayed_req(P,I,R), Z, EP, _):-
        holds(request(P,I,R), EP),
        knows(delayed_req(P,I,R),Z).

causes(-unreach(P), -alerted(P), Z, _, EN):-
        holds(unreach(P), EN),
        knows(alerted(P), Z).

causes(-(EF),-unreach(P),Z,_, EN):-
        member(EF, [ab(enterable(_)),ab(reachable(_,_)),
        ab(reachable(P))]),
        holds(EF,EN),!,
        knows(unreach(P),Z), detailed_pp(P, Z, _).


causes(s_request(PLO,I,R), might_have(P,I,R),Z,EP,_):-
        holds(s_request(PLO,I,R), EP),
        member(P,PLO), \+ knows(might_have(P,I,R),Z).
```

A.1.3 Topology

The file `$(GIOVANNA)/highlevel/eclipse/HomuncuFlux/topology.pl`

```
/* Simple Position Corespondence Table
```

```
   between fluents and actual positions in world map

   According To Map worlds/GRU_rooms-closedup.wld

   Author: Andreas Henschel

   Usage:
   given * action index (gtd - go to door
   etr - enter room)
   * the current room (gtd)/ the goal room(etr)
   * the door to be approached
   yield goal position
   * xpos
   * ypos
   * angle
*/

% d1
gotopos(gtd, "alley", d1,7900, 5400,180).
gotopos(gtd, "r419",  d1,7000, 5400,  0).
% d2
gotopos(gtd, "alley", d2,7900, 8300,180).
gotopos(gtd, "r420",  d2,7000, 8300,  0).
% d3
gotopos(gtd, "alley", d3,7900,11700,180).
gotopos(gtd, "r421",  d3,7000,11700,  0).
% d4
gotopos(gtd, "alley", d4,7900,15200,180).
gotopos(gtd, "r422",  d4,7000,15200,  0).
% d5
gotopos(gtd, "alley", d5,7900,20700,180).       %
gotopos(gtd, "r423",  d5,7000,20700,  0).%
% d6
gotopos(gtd, "r419",  d6,1800, 6100,270).
gotopos(gtd, "r420",  d6,1600, 7100, 80).
% d7
gotopos(gtd, "r420",  d7,1800, 9500,270).
gotopos(gtd, "r421",  d7,1800,10500, 90).
% d8
gotopos(gtd, "r422",  d8,1800,16300, 225).%
gotopos(gtd, "r423",  d8,1800,17300,  90).%


gotopos(etr, R, D, X, Y, Theta):-
        gotopos(gtd, R, D, X, Y, T),
        Theta is (T+180) mod 360. % facing opposite dir.


connects(D, X, Y) :-
        gotopos(gtd, X, D, _, _, _),
```

```
        gotopos(gtd, Y, D, _, _, _), X\=Y.

/* adjusts autom. to abovementioned topology marks */

office("r419", "sylvia").
office("r420", "steffen").
office("r421", "axel").
office("r422", "pascal").
office("r422", "uschi").
office("r423", "horst").
office("r423", "andreas").
office("alley", "bogus").

/* Personal Coordinates */

p_coords( "sylvia",  3200,  5200).
p_coords( "steffen", 4300,  8700).
p_coords( "axel",    2000, 11100).
p_coords( "pascal",  5100, 15000).
p_coords( "uschi",   2300, 15200).
p_coords( "horst",   1700, 20200).
p_coords( "andreas", 4800, 20100).
p_coords( "bogus",   8800, 13100).

pos2room(X, Y, "r423"):-
        X>   455,
        X<  7440,
        Y> 16841,
        Y< 22258,!.

pos2room(X, Y, "r422"):-
        X>   455,
        X<  7440,
        Y> 13391,
        Y< 16842,!.

pos2room(X, Y, "r421"):-
        X>   455,
        X<  7440,
        Y> 10084,
        Y< 13391,!.

pos2room(X, Y, "r420"):-
        X>   455,
        X<  7440,
        Y>  6577,
        Y< 10084,!.

pos2room(X, Y, "r419"):-
        X>   455,
```

```
        X<  7440,
        Y>  3200,
        Y<  6577,!.

pos2room(X, Y, "alley"):-
        X>  7439,
        X< 10947,
        Y>  1075,
        Y< 22344,!.

pos2room(_,_,_):-
        sfmessage("No match for this node !").

node2room(node(X,Y),R):-        % syntactic sugar
        pos2room(X,Y,R).

sfmessage(S):-
        write("[FL] "),writeln(S).
```

A.1.4 Execution, Eclipse predicates

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%      EXECUTION  (Call of Low Level)        %%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Command Conversion

execute(idle, Z, 0, Obs):-
        idle(ObsList),
        ui_refine(ObsList, Z, ObsFl),
        Obs = obs(ObsFl,_).

execute(A, S, FB, Obs) :-
        (A=goto(P);A=r_goto(P)),
        p_coords(P, X, Y),
        call_ll(gotopos(X, Y, 0), FB, Obs, S).

execute(gotodoor(D), S, FB, Ob):-
        knows(in_room(R), S),
        gotopos(gtd, R, D, X, Y, Th),
        call_ll(gotopos(X, Y, Th), FB, Ob, S).

execute(enter(R1), S, FB, Ob):-
        knows(in_room(R), S),
        connects(D, R, R1),
        gotopos(etr, R1, D, X, Y, Th),
        call_ll(gotopos(X, Y, Th), FB, Ob, S).

execute(ask_for(P,I,R), _, 0, Obs):-
        read(FB),
```

```
        (FB = 0 ->
            Obs = obs([has(P,I)],_);
            Obs = obs([-might_have(P,I,R)],_), br).

execute(deliver(O,R), S, X, Obs):-
        knows(carries(O,R),S),
        (sim_audio(off)
        -> dialog(deliver(O,R,bogus));true), % --> LL
        sim_feedback(X),
        (sim_mode(off)-> my_listening(Obs); Obs=obs([],_)).


execute(receive(O,P), Z, X, Obs):-
        (
          knows(request(P,O,R),Z);
          knows(s_request(_,O,R),Z)
        ),!,
        (sim_audio(off)
        -> dialog(receive(O,R,P));true),        % --> LL
        sim_feedback(X),
        (sim_mode(off)-> my_listening(Obs); Obs=obs([],_)).


execute(unreach(P), _, 0, obs([],_)):-
        write("[FL] Alert! "), write(P),
        writeln(" is out of reach!"). % Cancel request?

execute(walk_on, _, 0,  obs([],_)).

execute(chdir, _, 0,  obs([],_)):-
        writeln("[FL] Changing general direction").

execute(email(O,P), _, 0,  obs([],_)):-
        write("[FL] "), write(P), write(", your search for the "),
        write(O), writeln(" was unsuccessful, sorry!"). % sim. email notification

execute(call_help(P), _, FB, Obs ):-
        sim_feedback(FB),
        (FB=1 -> Obs = obs([unreach(P), ab(traceable(P))],_);
                 Obs = obs([],_)).

execute(idle, _, 0, Obs ) :- idle(Obs).

call_ll(A, FB, OB, S):-
        transport(A, FB, obs(Obs,CC)),
        ui_refine(Obs,S,ObsList1),
        OB=obs(ObsList1,CC).


sim_feedback(FB):-
```

```
        write("[FL] Enter Action result:  "),
        read(FB),writeln(FB).

br.

ui_refine([UIF|T], Z, [F|T1]):-
        ui_fluent(UIF, Z, F),
        ui_refine(T, Z, T1).

ui_refine([],_,[]).


ui_fluent(cancel_s_req(O,P), Z, -s_request(PL,O,P)):-
        knows(s_request(PL,O,P), Z),!.
ui_fluent(present(P), _, -ab(traceable(P)) ):-!.
ui_fluent(-present(P), _, ab(traceable(P)) ):-!.
ui_fluent(open(R1,R2),_,  -ab(enterable(D))) :-
        connects(D, R1, R2),!.
ui_fluent(-open(R1,R2),_,  ab(enterable(D))) :-
        connects(D, R1, R2),!.
ui_fluent(obstacle(P), _,  -ab(reachable(P))):-!.
ui_fluent(obstacle(R1,R2), _,  -ab(reachable(D, R1))):-
        connects(D, R1, R2),!.

/* Otherwise no conversion */
ui_fluent(F, _, F).
```

A.2 Interface (in C)

The file `$(GIOVANNA)/highlevel/eclipse/FluxTCA/interface.cc`

```
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <string.h>
#include <signal.h>
#include <tca.h>

#include "eclipseclass.h"

#include "hlcentraldata.h"
#include "ppcentraldata.h"
#include "uicentraldata.h"
#include "interface.h"


HL_TARGET_INFO_TYPE    pnt;
HL_EXECMON_INFO_TYPE   flux_fb;
LL_INTERRUPT_TYPE      dat;
DELIVER_TYPE           del;
```

```
RECEIVE_TYPE           rec;

//REPORT_PTR             robotReport;
TCA_REF_PTR            killref;

const char MaxMsgs = 3;

char InterruptLL = 0;
char FeedbackReceived = 0;
char ExogEventOcc = 0;
char setEndOfPlan = 0;

float robot_x;
float robot_y;

EC_functor    f_cc( "cc", 2 );
EC_functor    f_obs( "obs", 2 );
EC_functor    f_node( "node", 2 );

EC_functor    f_minus( "-", 1);
EC_functor    f_req( "request", 3);
EC_functor    f_s_req( "s_request", 3);
EC_functor    f_present( "present", 1);
EC_functor    f_cancel_s_req( "cancel_s_req", 2);
EC_functor    f_door( "open", 2);
EC_functor    f_obstacle( "obstacle", 2);
EC_functor    f_obstacle_person( "obstacle", 1);
EC_atom       f_quit( "quit");

EC_word route = nil();
EC_word remaining_route = nil();
EC_word absolved_route = nil();

EC_word observation = nil();

static void
ms_pause( int milliseconds )
{
    struct timeval delay;
    int readfds, writefds, execfds;

    delay.tv_sec = milliseconds / 1000;
    delay.tv_usec = 1 + ( milliseconds % 1000 ) * 1000;
    select( 0, (fd_set*) &readfds, (fd_set*) &writefds,
            (fd_set*) &execfds, &delay );
}


extern"C" void
HLExecmonInfoMsgHnd(TCA_REF_PTR ref1, HL_EXECMON_INFO_PTR fb)
```

```
{
  flux_fb = *fb; // copy whole package
  // construct EC List of nodes
  if (flux_fb.last_node == 2)
    absolved_route  = list( term(f_node, (int)flux_fb.posX,
                                 (int)flux_fb.posY),
                            absolved_route);
  else
      remaining_route = list( term(f_node, (int)flux_fb.posX,
                                   (int)flux_fb.posY),
                              remaining_route);
  if (flux_fb.last_node == 0) FeedbackReceived = 1;
  tcaSuccess(ref1);
  tcaFreeData(tcaReferenceName(ref1), fb);
}


extern"C" void
PPPathInfoMsgHnd( TCA_REF_PTR ref, PP_TARGET_INFO_PTR data )
{
  if ( data->status == 0 ) setEndOfPlan = 1;
  else
    route = list( term(f_node, (int)data->target_x,
                       (int)data->target_y),
                  route);
  tcaFreeData( tcaReferenceName( ref ), data );
}

extern"C" void
HLPosInfoHnd( TCA_REF_PTR ref, HL_POS_MSG_PTR data )
{
    robot_x = data->target_x;
    robot_y = data->target_y;
    FeedbackReceived = 1;
}

extern"C" void
ExogEventMsgHnd( TCA_REF_PTR ref, EXOG_EVENT_MSG_PTR event )
{
  EC_word people_list = nil();
  int i;
  printf("Event Message received\n");

  switch(event->id) {
  case REQ:
    observation  = list( term(f_req, people[event->person1],
                              event->object,
                              people[event->person2]),
                         observation);
    break;
```

```
case CANCEL_REQ:
  observation  = list( term(f_minus,
                            term(f_req, people[event->person1],
                                 event->object,
                                 people[event->person2]) ),
                       observation);
  break;
case S_REQ:
  people_list  = nil();
  for (i = 0; i < event->NoCandidates; i++)
    people_list = list( people[ event->people[i] ],
                        people_list);
  observation  = list( term(f_s_req, people_list,
                            event->object,
                            people[ event->person1]),
                       observation);
  break;
case CANCEL_S_REQ:
  observation  = list( term(f_cancel_s_req, event->object ,
                            people[ event->person1] ) ,
                       observation);
  break;

case PRESENT:
  observation  = list( term(f_present, people[event->person1]),
                       observation);
  break;
case NOT_PRESENT:
  observation  = list( term(f_minus, term(f_present,
                                          people[event->person1]) ),
                       observation);
  break;

case OPEN_DOOR:
    observation  = list( term(f_door, rooms[ event->room1 ] ,
                              rooms[ event->room2 ]),
                       observation);
  break;
case CLOSED_DOOR:
    observation  = list( term(f_minus,
                              term(f_door,
                                   rooms[ event->room1 ] ,
                                   rooms[ event->room2 ])),
                       observation);
  break;
case RM_OBSTACLE:
    observation  = list( term(f_obstacle_person, people[event->person1]),
                         observation);
  break;
case RM_OBSTACLE_T:
```

```
    observation  = list( term(f_obstacle, rooms[event->room1] ,
                              rooms[event->room2]),
                         observation);
    break;
  case QUIT_ROBOT:
    observation  = list( EC_word(f_quit),
                         observation);
    break;
  }
  printf("Eclipse Term Constructed from Event Message\n");
  ExogEventOcc = 1; // Causing idle to finish
}


int init_tca()
{

  // Register the modules

  char  hostname [50] = "localhost";
  if ( (char*)getenv( "CENTRALHOST" ) != (char*)NULL )
    strcpy( hostname, getenv( "CENTRALHOST" ) );

  // Register the module/listener

  tcaConnectModule( "Giovanna Flux Module", hostname );

  // Register Handlers, mainly feedback from low level
  tcaRegisterCommandMessage("HLExecmonInfoMsg",
                            HL_EXECMON_INFO_FORMAT);
  tcaRegisterHandler("HLExecmonInfoMsg",
                     "HLExecmonInfoMsgHnd",
                     HLExecmonInfoMsgHnd);

  tcaRegisterInformMessage("PPPathInfoMsg",
                           PP_TARGET_INFO_FORMAT);
  tcaRegisterHandler("PPPathInfoMsg",
                     "PPPathInfoMsgHnd",
                     PPPathInfoMsgHnd);

  tcaRegisterInformMessage("HLPosInfo",
                           HL_POS_MSG_FORMAT);

  tcaRegisterHandler("HLPosInfo","HLPosInfo", HLPosInfoHnd);

  tcaRegisterInformMessage("ExogEventMsg",
                           EXOG_EVENT_MSG_FORMAT);
  tcaRegisterHandler("ExogEventMsg",
                     "ExogEventMsg",
                     ExogEventMsgHnd);
```

```
  // Kill all pending messages
  killref = tcaCreateReference( "HLExecmonInfoMsg" );
  tcaKillTaskTree( killref );

  tcaWaitUntilReady();
  return EC_succeed;
}


int idle(void)
{
  observation = nil();
  ExogEventOcc = 0;

  while (!ExogEventOcc)  {
    tcaHandleMessage(1);
    ms_pause(300); // wait a bit
  }
  unify( EC_word( EC_arg(1)), observation);
  return EC_succeed;
}

int my_listening(void)
{
  char i;
  observation = nil();
  ExogEventOcc = 0;

  for (i=0; i<MaxMsgs; i++)
    tcaHandleMessage(1); // Dont wait for pending messages

  unify( EC_word( EC_arg(1)), term(f_obs, observation, nil()) );
  return EC_succeed;
}


int transport(void)
{
  int     res;
  double  d1, d2, d3;
  char    act[50];
  // Parameteruebergabe
  EC_word  action(EC_arg(1));
  // EC_word  fluents(EC_arg(2)); //World State Description
  // EC_word  result(EC_arg(3));  //Result from LL


  EC_word arg1, arg2, arg3;
  EC_word Critic_Coords;
```

```
EC_functor f;
EC_atom a,v1,v2;


if (action.functor(&f) == EC_succeed) {
  cout << "ci: Action: " << f.Name() << "\n";
  strcpy(act, f.Name() );

  // gotopos/3
  if (strcasecmp(act, "gotopos") == 0) {
    if (action.arity() != 3) {
      cout << "ci: Wrong number of arguments!\n";
      return EC_fail;
    }

    if ((action.arg(1, arg1) == EC_fail) ||
        (action.arg(2, arg2) == EC_fail) ||
        (action.arg(3, arg3) == EC_fail)) {
      cout << "ci: Wrong argument type!\n";
      return EC_fail;
    }
    if ((arg1.is_double(&d1) == EC_fail) ||
        (arg2.is_double(&d2) == EC_fail) ||
        (arg3.is_double(&d3) == EC_fail)) {
      cout << "ci: Wrong argument type!\n";
      return EC_fail;
    }

    pnt.target_x  = d1;
    pnt.target_y  = d2;
    //pnt.th = d3;

    tcaInform("HLTargetInfoMsg", &pnt);
    tcaWaitUntilReady();

    remaining_route = nil();
    absolved_route = nil();
    observation = nil();

    FeedbackReceived = 0;
    ExogEventOcc = 0;
    InterruptLL = 0;

    while ( !FeedbackReceived ) {
      tcaHandleMessage(1);
      ms_pause(300); // wait a bit
      if ((ExogEventOcc) && (!InterruptLL))
        {
          InterruptLL = 1;
          tcaInform("LLInterruptMsg", &dat);
```

```
        }
      }
      // HUSK: AFBRYD LL, NAAR EVENT ER SKET!!
      if (ExogEventOcc)
        unify( EC_word( EC_arg(2) ), EXOG_EVENT);
      else
        unify( EC_word( EC_arg(2) ), flux_fb.status);

      switch (flux_fb.status)
        {
        case SUCCESS:
          unify( EC_word( EC_arg(3)), term(f_obs,
                                      observation, nil()) );
          break;
        case STALLED: // TIMEOUT handled here as well
          unify( EC_word( EC_arg(3)),
                 term ( f_obs, observation,
                        term (f_cc, absolved_route, remaining_route ) ));
          break;
        }
      return EC_succeed;


    }
    else {
      cout << "ci: wrong action name:" << act << " \n";
      return EC_fail;
    }
  }
  if (action.is_atom(&a) == EC_succeed)
    cout << a.Name() << "\n";

  return EC_succeed;
}


int ask_pp(void)
{
  long t_x, t_y;

  EC_word snode(EC_arg(1));
  EC_word gnode(EC_arg(2));
  EC_word  arg1, arg2;
  PP_REQUEST_PATH_TYPE goal;

  // Simple conversion
  snode.arg(1, arg1);
  snode.arg(2, arg2);
  arg1.is_long(&t_x);
  arg2.is_long(&t_y);
```

```
  goal.start_x = t_x;
  goal.start_y = t_y;

  gnode.arg(1, arg1);
  gnode.arg(2, arg2);
  arg1.is_long(&t_x);
  arg2.is_long(&t_y);

  goal.target_x = t_x;
  goal.target_y = t_y;
  goal.target = 0;

  tcaInform("HLRequestPathMsg", &goal);

  route = nil();

  setEndOfPlan = 0;
  while (!setEndOfPlan) {
    tcaHandleMessage(0);
    ms_pause(100); // wait a bit
  }
  unify( EC_word( EC_arg(3) ), route  );

  return EC_succeed;
}



int curpos(void)
{
  HL_POS_MSG_TYPE bogus;
  tcaInform("HLPosMsg", &bogus);
  FeedbackReceived = 0;
  while (!FeedbackReceived) {
    tcaHandleMessage(0);
    ms_pause(50);
  }
  unify( EC_word( EC_arg(1)), term (f_node, (int)robot_x, (int)robot_y));
}



int dialog(void) //handles deliver/3 and receive/3
{
  int     res;
  //char*   act;
  char    act[20];

  // Parameteruebergabe
```

```
EC_word  action(EC_arg(1));
// needed for parsing of Eclipse term:
EC_word  arg1, arg2, arg3;
EC_functor f;
EC_atom a1,a2,a3;

DIALOG_TYPE            dlg;

if (action.functor(&f) == EC_succeed) {
  strcpy(act, f.Name() );

  if (action.arity() != 3) {
    cout << "ci: Wrong number of arguments!\n";
    return EC_fail;
  }

  if ((action.arg(1, arg1) == EC_fail) ||
      (action.arg(2, arg2) == EC_fail) ||
      (action.arg(3, arg3) == EC_fail)) {
    cout << "ci: Wrong argument type!\n";
    return EC_fail;
  }
  if ((arg1.is_atom(&a1) == EC_fail) ||
      (arg2.is_atom(&a2) == EC_fail) ||
      (arg3.is_atom(&a3) == EC_fail)) {
    cout << "ci: Wrong argument type!\n";
    return EC_fail;
  }

  // deliver/3 (3rd arg. = provider)
  if (strcasecmp(act, "deliver") == 0)
    dlg.action_id = DELIVER;

  // receive/3 (3rd arg. = receiver)
  if (strcasecmp(act, "receive") == 0)
    dlg.action_id = RECEIVE;

  if (strcasecmp(a1.Name(),"coffee") == 0) dlg.object = COFFEE;
  if (strcasecmp(a1.Name(),"mail") == 0) dlg.object = MAIL;
  if (strcasecmp(a1.Name(),"item") == 0) dlg.object = ITEM;

  if (strcasecmp(a2.Name(),"horst") == 0) dlg.receiver = HORST;
  if (strcasecmp(a2.Name(),"axel") == 0) dlg.receiver = AXEL;
  if (strcasecmp(a2.Name(),"pascal") == 0) dlg.receiver = PASCAL;
  if (strcasecmp(a2.Name(),"andreas") == 0) dlg.receiver = ANDREAS;

  if (strcasecmp(a3.Name(),"horst") == 0) dlg.provider = HORST;
  if (strcasecmp(a3.Name(),"axel") == 0) dlg.provider = AXEL;
  if (strcasecmp(a3.Name(),"pascal") == 0) dlg.provider = PASCAL;
  if (strcasecmp(a3.Name(),"andreas") == 0) dlg.provider = ANDREAS;
```

```
    cout << "Action ID: " << dlg.action_id << "\n";
    cout << "Receiver: " << dlg.receiver << "\n";
    cout << "Provider: " << dlg.provider << "\n";
    cout << "Object: " << dlg.object << "\n";

    tcaExecuteCommand("dialogMsg", &dlg);
    tcaWaitUntilReady();
  }
  return EC_succeed;
}
```

The file `$(GIOVANNA)/highlevel/eclipse/FluxTCA/interface.h`

```
typedef struct{
  int  call_status;
  float  data;
}REPORT_TYPE, *REPORT_PTR;

typedef struct{
  float x, y, th;
}POINT_TYPE, *POINT_PTR;

typedef struct{
  int status;
  float x, y;
}OBSTACLE_TYPE, *OBSTACLE_PTR;

typedef struct{
  int   x;
  int   y;
}FEEDBACK_TYPE, *FEEDBACK_PTR;

typedef struct{
  char*   obj;
  char*   supplier;
}RECEIVE_TYPE, *RECEIVE_PTR;

typedef struct{
  char*   obj;
  char*   receiver;
}DELIVER_TYPE, *DELIVER_PTR;

typedef struct{
  int  action_id;
  int  receiver;
  int  provider;
  int  object;
}DIALOG_TYPE, *DIALOG_PTR;
```

```
#define DIALOG_FORMAT "{int, int, int, int}"
#define ACTION_FORMAT "{int, int}"
#define POINT_FORMAT  "{float, float, float}"
#define REPORT_FORMAT "{int, float}"
#define OBSTACLE_FORMAT "{int, float, float}"

#define HORST   0
#define AXEL    1
#define PASCAL  2
#define ANDREAS 3

#define RECEIVE 0
#define DELIVER 1

#define COFFEE  0
#define MAIL    1
#define ITEM    2

extern "C"{
  int   transport(void);
  int   idle(void);
  int   my_listening(void);
  int   init_tca(void);
  int   dialog(void);
  int   ask_pp(void);
  int   curpos(void);
  int   tca_set_hostname( void );
  int   get_name( void );
  int   tca_connect_to_server( void );
  int   tca_disconnect_from_server( void );
}
```

A.3 Low level execution monitoring

The file `$(GIOVANNA)/learning/dtw/ctrl/gotopos/ProcExecMon.cc`

```
#include "sapphira.h"
#include "SfProcess.hh"
#include "ProcExecMon.hh"

#include "ProcRecData.hh"

#include "ProcBhrCtrl.hh"

#include "ppcentraldata.h"
#include "hlcentraldata.h"
#include "tca.h"

#include "toplevel.h"
```

```
ProcExecMon     ExecMonProc( "execmon_proc", 100 );


ProcExecMon::ProcExecMon( const char* name, int max_num_targets ):
  SfProcess( name ),
  maxNumTargets( max_num_targets )
{
  waitCycles = 10;
  waitCounter = 0;

  planTargetX = new float [maxNumTargets];
  planTargetY = new float [maxNumTargets];
  numTargets = 0;

  receiveTargetPositions = 0;

  notifyStalledFlag = 0;
  notifyUnstalledFlag = 0;

  ExogInterrupt = 0;

  timerCount = 0;
  timeoutCount = 0;
}


ProcExecMon::~ProcExecMon( void )
{
  delete [] planTargetX;
  delete [] planTargetY;
}


enum    ProcessStates
{
  // User-defined state values greater than 10
  StateWaiting = 20,
  StateActivate,
  StateWaitForGoal,
  StateRequestPlan,
  StateReceivePlan,
  StatePlanReceived,
  StateExecutePlan,
  StateMonitorExec,
  StateInterrupt,
  StateExecutionDone,
  StateStalled,
  StateTimedOut
```

```
};


void
ProcExecMon::activate( void )
{
  SFPROCESS     p;

  // Reset process
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, StateActivate );
  else
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
}


void
ProcExecMon::deactivate( void )
{
  SFPROCESS     p;

  // Reset process
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, sfINIT );
  else
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
}


void
ProcExecMon::resetProc( void )
{
  SFPROCESS     p;

  // Reset process
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, sfINIT );
  else
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
}


void
ProcExecMon::process( void )
{
  PP_REQUEST_PATH_TYPE  request;
  HL_EXECMON_INFO_TYPE  feedback;
```

```
switch ( process_state )
  {
  case sfINIT:
  case sfRESUME:

    receiveTargetPositions = 0;

    numTargets = 0;
    numTargetsDone = 0;

    break;

  case StateWaiting:

    if ( ++waitCounter > waitCycles )
      {
        waitCounter = 0;
        process_state = waitProcessState;
      }
    break;

  case StateActivate:

    // Get rid of pending target point packets by waiting

    waitProcessState = StateWaitForGoal;
    process_state = StateWaiting;

    break;

  case StateWaitForGoal:
    // Empty (Triggered by Flux-Goal Msg)
    break;

  case StateRequestPlan:

    //tcaInform( "DRSChangeWorldMsg", NULL );
    empty_sbuf( 0, 0 );
    empty_sbuf( 1, 0 );
    RecDataProc.resetHistoryBuffer();

    request.start_x = sfRobot.ax;
    request.start_y = sfRobot.ay;
    request.target = 1;
    request.target_x = goalX;
    request.target_y = goalY;
    tcaInform( "PPRequestPathMsg", &request );

    receiveTargetPositions = 1;
```

```
      numTargets = 0;

      process_state = StateReceivePlan;
      break;

    case StateReceivePlan:
      // Empty
      break;

    case StatePlanReceived:

      sfSMessage(
          "New path plan consisting of %i target points received",
                  numTargets );

      sfMessage( "Deactivate Stop behaviour" );
      BhrCtrlProc.deactivateStopBhr();

      sfMessage( "Activate GoToPos behaviour" );
      BhrCtrlProc.activateGoToPosBhr();
      RecDataProc.startTrial();

      process_state = StateExecutePlan;
      break;

    case StateExecutePlan:

      // Check whether the whole plan has been completed
      if ( numTargetsDone == numTargets )
        {
          process_state = StateExecutionDone;
          break;
        }

      // Set new goal position for GoToPos behaviour
      GTPBhrGoalPoint->ax  = planTargetX[numTargetsDone];
      GTPBhrGoalPoint->ay  = planTargetY[numTargetsDone];
      GTPBhrGoalPoint->ath = 0.0;
      sfSetLocalCoords( GTPBhrGoalPoint );

      setTimeoutCount( GTPBhrGoalPoint->ax, GTPBhrGoalPoint->ay );

      RecDataProc.setTargetCoordinates( GTPBhrGoalPoint->ax,
                                        GTPBhrGoalPoint->ay );

      sfSMessage( "New goal position (%.0f, %.0f) for GoToPos
behaviour",
                  GTPBhrGoalPoint->ax, GTPBhrGoalPoint->ay );

      process_state = StateMonitorExec;
```

```
  break;

case StateMonitorExec:

  if ( notifyStalledFlag )
    {
      // Robot has stalled

      sfMessage( "Stalling detected" );
      sfMessage( "Deactivate GoToPos behaviour" );
      BhrCtrlProc.deactivateGoToPosBhr();
      BhrCtrlProc.activateStopBhr();

      RecDataProc.setOutcome( 0 );
      RecDataProc.finishTrial();

      notifyStalledFlag = 0;

      process_state = StateStalled;
      break;
    }

  if ( timerCount > timeoutCount )
    {
      // Timeout possibly due to deadlock situation

      sfMessage( "Timeout detected" );

      sfMessage( "Deactivate GoToPos behaviour" );
      BhrCtrlProc.deactivateGoToPosBhr();
      BhrCtrlProc.activateStopBhr();

      RecDataProc.setOutcome( 2 );
      RecDataProc.finishTrial();

      process_state = StateTimedOut;
      break;
    }

  if ( GTPBhrGoalPointDone ) //Single target position reached
    {
      numTargetsDone++;
      process_state = StateExecutePlan;
    }

  if ( ExogInterrupt )
    {
      sfMessage( "Exogenous Interrupt realized" );

      sfMessage( "Deactivate GoToPos behaviour" );
```

```
        BhrCtrlProc.deactivateGoToPosBhr();
        BhrCtrlProc.activateStopBhr();

        RecDataProc.setOutcome( 2 );
        RecDataProc.finishTrial();

        ExogInterrupt = 0;
        process_state = StateInterrupt;
      }
    break;

  case StateInterrupt:

    notifyRouteState(STALLED);
    process_state = StateWaitForGoal;
    break;

  case StateExecutionDone:

    sfMessage( "Deactivate GoToPos behaviour" );
    BhrCtrlProc.deactivateGoToPosBhr();
    BhrCtrlProc.activateStopBhr();


    RecDataProc.setOutcome( 1 );
    RecDataProc.finishTrial();

    feedback.status = SUCCESS;
    // rest is dummy;
    feedback.posX = 0;
    feedback.posY = 0;
    feedback.last_node = 0;


    tcaExecuteCommand("HLExecmonInfoMsg", &feedback);

    process_state = StateWaitForGoal;
    break;



  case StateStalled:
    if ( notifyUnstalledFlag )
      {
        notifyUnstalledFlag = 0;

        // Prepare feedback for High level
        // report rest of remaining path nodes

        notifyRouteState(STALLED);
```

```
          process_state = StateWaitForGoal;
          break;
        }

      break;
    case StateTimedOut:

      // Prepare feedback for High level

      notifyRouteState(TIMEOUT);
      process_state = StateWaitForGoal;
      break;
    }


  if ( sfBehaviorControl == 1 )
    timerCount++;
}


void
ProcExecMon::addTargetPoint( float pos_x, float pos_y )
{
  if ( !receiveTargetPositions )
    return;

  if ( numTargets < maxNumTargets )
    {
      planTargetX[numTargets] = pos_x;
      planTargetY[numTargets] = pos_y;
      numTargets++;
    }
  else
    sfMessage( "Maximum number of target points exceeded" );
}


void
ProcExecMon::setEndOfPlan( void )
{
  SFPROCESS     p;

  // Reset process
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, StatePlanReceived );
  else
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
```

```
  numTargetsDone = 1;
}

void
ProcExecMon::setFluxGoalPoint( float pos_x, float pos_y ) {
  goalX = pos_x;
  goalY = pos_y;
}

void
ProcExecMon::stopWaitForGoal( void ) {
  SFPROCESS     p;

  // set process to StateRequestPlan
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, StateRequestPlan );
  else // fis
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
}

void
ProcExecMon::interruptState( void ) {
  SFPROCESS     p;

  ExogInterrupt = 1;
  // set process to StateMonitorExec
  if ( ( p = sfFindProcess( procName ) ) != NULL )
    sfSetProcessState( p, StateMonitorExec );
  else // fis
    sfSMessage( "%s: Cannot find process %s", procName,
                procName );
}


void
ProcExecMon::setTimeoutCount( float pt_x, float pt_y )
{
  const float   min_speed = 50.0;       // [mm/sec]

  float dist;

  dist = sqrt( ( pt_x - sfRobot.ax ) * ( pt_x - sfRobot.ax ) +
               ( pt_y - sfRobot.ay ) * ( pt_y - sfRobot.ay ) );

  // 100 millisec cycles
  timeoutCount = timerCount + (int)( dist / min_speed * 10.0 );
}

void
```

```
ProcExecMon::notifyRouteState(int state)
{
  HL_EXECMON_INFO_TYPE  feedback;

  int i = 0;

  // Reached nodes
  for (; i<numTargetsDone; i++) {
    feedback.last_node = 2;
    feedback.status = state;
    feedback.posX = planTargetX[i];
    feedback.posY = planTargetY[i];
    tcaExecuteCommand("HLExecmonInfoMsg", &feedback);
  }
  // curent node
  feedback.status = STALLED;
  feedback.last_node = 2;
  feedback.posX = sfRobot.ax;
  feedback.posY = sfRobot.ay;
  // attempted nodes
  for (; i<numTargets-1; i++) {
    feedback.last_node = 1;
    feedback.status = STALLED;
    feedback.posX = planTargetX[i];
    feedback.posY = planTargetY[i];
    tcaExecuteCommand("HLExecmonInfoMsg", &feedback);
  }

  feedback.status = STALLED;
  feedback.last_node = 0;
  feedback.posX = planTargetX[numTargets-1];
  feedback.posY = planTargetY[numTargets-1];
  tcaExecuteCommand("HLExecmonInfoMsg", &feedback);
  sfSMessage("last HLExecMon sent, node(%d,%d)",
             feedback.posX,feedback.posY);
 }

// End of file
```

References

[AIS88] J. Ambros-Ingerson and S. Steel. Integrating planning execution and monitoring, 1988.

[BBS95] Andrew G. Barto, Steven J. Bradtke, and Satinder P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence*, 72(1–2):81–138, 1995.

[BDH99] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999. URL: `citeseer.nj.nec.com/boutilier98decision.html`.

[BFL+98] W. Burgard, Dieter Fox, G. Lakemeyer, D. Haehnel, D. Schulz, W. Steiner, Sebastian Thrun, and A. Cremers. Real robots for the real world — the rhino museum tour-guide project. In *Proceedings of the 1998 AAAI Spring Symposium*, 1998.

[Bjä98] Marcus Bjäreland. Recovering from modeling faults in GOLOG, 1998. URL: `citeseer.nj.nec.com/239352.html`.

[Bre94] Gerhard Brewka. Adding priorities and specificity to default logic. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *Proceedings of the European Workshop on Logics in AI (JELIA)*, volume 838 of *LNAI*, pages 50–65. Springer, September 1994.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2:14–23, 1986.

[BRST00] Craig Boutilier, Raymond Reiter, Mikhail Soutchanski, and Sebastian Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *AAAI/IAAI*, pages 355–362, 2000.

[Cam00] Tulio Camminati. Caporal. Master's thesis, Technische Universität Dresden, 2000.

[DGKea98] Patrick Doherty, Joakim Gustafsson, Lars Karlsson, and et al. Temporal action logics (tal): Language specification and tutorial, 1998. URL: `citeseer.nj.nec.com/doherty98temporal.html`.

[DGN01] M. Dietl, J. Gutmann, and B. Nebel. Cooperative sensing in dynamic environments, 2001. URL: `citeseer.nj.nec.com/dietl01cooperative.html`.

[FBTC98] Dieter Fox, Wolfram Burgard, Sebastian Thrun, and Armin B. Cremers. Position estimation for mobile robots in dynamic environments. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 983–988, Menlo Park, July 26–30 1998. AAAI Press.

[FN71] R. E. Fikes and N. Nilsson. STRIPS: A New Approach to the Application Theorem Proving to Problem Solving. *Artificial Intelligence*, 5(2):189–208, 1971.

[For92] Fortune. Voronoi diagrams and delaunay triangulations. In *Computing in Euclidean Geometry, Edited by Ding-Zhu Du and Frank Hwang, World Scientific, Lecture Notes Series on Computing – Vol. 1.* 1992.

[FPR00] Alberto Finzi, Fiora Pirri, and Ray Reiter. Open world planning in the situation calculus. In *Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00)*, pages 754–760, Menlo Park, CA, July 30– 3 2000. AAAI Press.

[FS97] Joaquin L. Fernandez and Reid G. Simmons. Robust execution monitoring for navigation plans, 1997. URL: `citeseer.nj.nec.com/3359.html`.

[GB98] H. Geffner and B. Bonet. High-level planning and control with incomplete information using POMDPs, 1998. URL: `citeseer.nj.nec.com/geffner98highlevel.html`.

[GP] Axel Großmann and Riccardo Poli. Robust mobile robot localisation from sparse and noisy proximity readings. URL: `citeseer.nj.nec.com/336905.html`.

[GP98] Axel Großmann and Riccardo Poli. A constructive connectionist approach towards continual robot learning, 1998. URL: `citeseer.nj.nec.com/28353.html`.

[Gro99] H. Grosskreutz. Probabilistic temporal projections in congolog, 1999. URL: `citeseer.nj.nec.com/grosskreutz99probabilistic.html`.

[Gro01] Axel Großmann, Giovanna robotics project. 2001. URL: `www.wv.inf.tu-dresden.de/Research/Robotics/Projects/Giovanna`.

[GRS98] Giuseppe De Giacomo, Raymond Reiter, and Mikhail Soutchanski. Execution monitoring of high-level robot plans. In Anthony G. Cohn, Lenhart Schubert, and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 453–464. Morgan Kaufmann, San Francisco, California, 1998.

[HBL] Dirk Hähnel, Wolfram Burgard, and Gerhard Lakemeyer. GOLEX - Bridging the Gap between Logic (GOLOG) and a Real Robot. URL: `citeseer.nj.nec.com/51266.html`.

[How99] Adele Howe. Evaluating robot planning systems: Some issues and successes. In *Online Proceedings of the Workshop on Robot Action Planning at the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[HS90] Steffen Hölldobler and Josef Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8(3):225–244, 1990.

[HT99] Andreas Henschel and Michael Thielscher. The LMW traffic world in the Fluent Calculus. *Linköping Electronic Articles in Computer and Information Science*, 1999. URL: `www.ida.liu.se/ext/etai/lmw/TRAFFIC/001/index.html`.

[KLC96] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. Technical Report CS-96-08, 1996.

[KMRS97] K. Konolige, K. L. Myers, E. H. Ruspini, and A. Saffiotti. The Saphira architecture: A design for autonomy. *Journal of experimental & theoretical artificial intelligence: JETAI*, 9(1):215–235, 1997. URL: `citeseer.nj.nec.com/konolige97saphira.html`.

[LMT84] T. Lozano-Pérez, M. Mason, and R. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, 3(1):3–24, 1984.

[LRL+97] Hector J. Levesque, Raymond Reiter, Ives Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1–3):59–83, April–June 1997.

[Mar01] Yves Martin. Qualification problem in FLUX. Master's thesis, Technische Universität Dresden, 2001.

[McC58] John McCarthy. Programs with Common Sense. In *Proceedings of the Symposium on the Mechanization of Thought Processes*, volume 1, pages 77–84, London, November 1958.

[McC63] John McCarthy. *Situations and Actions and Causal Laws*. Stanford Artificial Intelligence Project, Memo 2, 1963.

[McC77] John McCarthy. Epistemological problems of artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1038–1044, Cambridge, MA, 1977. MIT Press.

[MH69] John McCarthy and P. J. Hayes. Some philosphical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*. Edinburgh University Press, Edinburgh, 1969.

[Min86] M. Minsky. *The Society of Mind.* Simon and Schuster, New York, 1986.
A loosley-coupled theory of intelligence based on the model of a distributed society of interacting "experts".

[MT01] Yves Martin and Michael Thielscher. Addressing the qualification problem in FLUX. *Lecture Notes in Computer Science*, 2174:290–??, 2001. URL: `citeseer.nj.nec.com/453589.html`.

[Nil84] Nils J. Nilsson. Shakey the robot. Technical Note 323, SRI International, Menlo Park, California, April 1984.

[NR95] P. Norvig and S. Russell. *Artificial Intelligence - A Modern Approach.* Prentince Hall, 1995.

[Rei80] Raymond Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, 1980.

[Sha96] Murray 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. AAAI Press, Menlo Park, California, 1996.

[Sha97] Murray Shanahan. *Solving the Frame Problem: A Mathematical Investigation of the Common Sence Law of Inertia.* Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, USA, 1997.

[SK95] Reid Simmons and Sven Koenig. Probabilistic Navigation in Partially Observable Environments. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1080–1087, 1995.

[SM89] R. Simmons and T. Mitchell. A task control architecture for mobile robots. In *AAAI Spring Symposium on Robot Navigation*, pages 85–89, Stanford CA, 1989.

[TFBD00] Sebastian Thrun, Dieter Fox, Wolfram Burgard, and Frank Dellaert. Robust Monte Carlo localization for mobile robots. *Artificial Intelligence*, 128(1–2):99–141, 2000.

[Thi98] Michael Thielscher. Introduction to the Fluent Calculus. *Electronic Transactions on Artificial Intelligence*, 2(3–4):179–192, 1998.

[Thi99] Michael Thielscher. From Situation Calculus to Fluent Calculus: State update axioms as a solution to the inferential frame problem. *Artificial Intelligence*, 111(1–2):277–299, 1999.

[Thi00a] Michael Thielscher. The Fluent Calculus: A Specification Language for Robots with Sensors in Nondeterministic, Concurrent, and Ramifying Environments. Technical Report CL-2000-01, Computational

Logic Group, Department of Computer Science, Dresden University of Technology, October 2000.

[Thi00b] Michael Thielscher. Representing the knowledge of a robot. In A. Cohn, F. Giunchiglia, and B. Selman, editors, *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 109–120, Breckenridge, CO, April 2000. Morgan Kaufmann.

[Thi00c] Michael Thielscher. Representing the knowledge of a robot. In Anthony G. Cohn, Fausto Giunchiglia, and Bart Selman, editors, *KR2000: Principles of Knowledge Representation and Reasoning*, pages 109–120, San Francisco, 2000. Morgan Kaufmann.

[Thi01a] Michael Thielscher. The concurrent, continuous Fluent Calculus. *Studia Logica*, 67(3):315–331, 2001.

[Thi01b] Michael Thielscher. The Qualification Problem: A solution to the problem of anomalous models. *Artificial Intelligence*, 131(1–2):1–37, 2001.

[Thi02a] M. Thielscher. *The Art and Science of Programming Reasoning Agents*. Draft, 2002.

[Thi02b] Michael Thielscher. Pushing the envelop: Programming reasoning agents, 2002. URL: `www.fluxagent.org`.

[Thi02c] Michael Thielscher. Reasoning about actions with CHRs and finite domain constraints. In *Proceedings of the International Conference on Logic Programming (ICLP)*, Copenhagen, Danmark, 2002.

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Dresden, den 14. Oktober 2002

Andreas Henschel