Dissertation

Dependable Belief Management for High-Level Robot Programs

Michael Reip

Graz, 2016

Institute for Software Technology
Graz University of Technology

Supervisor/First reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Franz Wotawa
Second reviewer: Univ.-Prof. Dipl.-Ing. Dr. techn. Gerhard Friedrich

Abstract (English)

Robots play an increasingly important role in our daily lives. They are omnipresent not only in household items such as vacuum cleaners but also robot arms or automated guided vehicles in factories. A vacuum cleaner that moves randomly around in an apartment and an automated guided vehicle that simply follows a line are certainly not the best examples of an intelligent robot, but they are affordable and do operate reliably. On top of these mass products there are several solutions with higher cognitive capabilities such as robots that are able to prepare pancakes. Typically, an intelligent robot has a belief system about its environment and about itself, e.g. a map of the robot's area, with the help of which the robot can determine its position and find a given destination. While moving around, it may pick up further information about the configuration and position of other robots, humans or objects: The robot may infer additional knowledge from those observations. For instance, it may learn that when detecting a coffee machine, there is a good chance to encounter a group of people around it as well. Hence, the belief system of a robot can be comprehensive with respect to the application domain. However, similar to the human brain, parts of the robot's belief can be contradictory to the real world. One reason for that could be the robot's overestimation. It might think it drove 10 kilometers, whereas in fact it drove only 5 km because of a problem in the robot's engine and end up far from the destination. Based on such incorrect knowledge, the robot might set incorrect and even dangerous consecutive actions.

This thesis presents a system that detects inconsistencies within the robot's belief and, if possible, corrects them. The presented system provides the robot with a set of rules that continuously check the belief's consistency. Moreover, all sensing information retrieved by the robot is compared with the expected value based on past experiences. If an inconsistency is detected, the robot tries to correct the internal state. In order to have best knowledge of the environment, the correction is always driven by the question "What happened". This is why the robot maintains a pool of possible explanations which are always kept updated. The current favorite out of this explanation pool forms the belief of the robot.

This thesis shows that a robot's performance benefits from such a pool of explanations ("hypothesis pool"). A formal description of the system, which includes proofs of several attributes, is also provided in this thesis. Having started with a simulation, the system is then integrated into a real world robot's architecture. Further results show that an automatic domain pattern analysis significantly improves the performance and the runtime of the belief system. These results are strengthened by a comprehensive evaluation.

Abstract (German)

Roboter nehmen eine immer wichtigere Rolle in unserer Gesellschaft ein. Dies gilt füer den Heimgebrauch ebenso wie füer das industrielle Umfeld. Im Heimgebrauch zeigt sich das z.B. in Form von Haushaltshelfern wie Staubsaugerroboter oder Rasenmäher. Das industrielle Pendant sind Roboterarme oder fahrerlose Transportsysteme. Auch wenn z.B. scheinbar planloses Hin- und Herfahren eines Staubsaugerroberts oft befremdlich wirken mag, so ist der Einsatz dieser genannten Systeme doch wirtschaftlich und bewährt sich im Alltag. Neben solch bedingt intelligenten Massenprodukten gibt es jedoch auch eine beachtliche Anzahl von Robotersystemen mit ausgeprägten cognitiven Fähigkeiten, wie z.B. Roboter, die in der Lage sind Palatschinken zuzubereiten. In der Regel besitzt ein Roboter eine Wissensbasis, die seine Umgebung und ihn selbst repräsentiert. Dazu zählt z.B. eine Karte seines Einsatzgebiets, mit deren Hilfe er seine Position ermitteln und zu Zielen finden kann. Während sich Roboter bewegen, können sie weitere Informationen über die Konfiguration und Position anderer Roboter, Menschen und Objekte sammeln. Dies kann benutzt werden um daraus auf weiteres Wissen zu schlieszen. Beispielsweise kann ein Roboter lernen, dass im Falle eines Kaffeeautomaten die Chancen gut stehen, in der Nähe des Automaten eine Gruppe von Personen zu entdecken. Je nach Einsatzgebiet kann daher die Wissensbasis eines Roboters sehr umfangreich werden. Ähnlich wie im menschlichen Gehirn können auch Teile dieser Wissensbasis der realen Welt widersprechen. Ein Grund dafür kann die Selbstüberschätzung des Roboters sein. Folgedessen kann es aufgrund eines Motorfehlers vorkommen, dass der Roboter denkt, 10 Kilometer gefahren zu sein, obwohl er tatsächlich nur 5 Kilometer zurückgelegt hat. Solch falsches Wissen kann dazu führen, dass der Roboter falsche oder gar gefährliche Aktionen setzt.

Diese Dissertation präsentiert ein Wissensmanagement-System zur Erkennung und wenn möglich auch zur Reparatur von Fehlern innerhalb einer Roboterwissensbasis. Dieses neue System stattet einen Roboter mit einer Menge von Regeln aus, die kontinuierlich die Konsistenz und somit die Korrektheit der Wissensbasis prüfen. Zusätzlich wird jede eingehende Sensorinformation mit dem internen, erwarteten Wert verglichen, der sich auf bisher gesammeltes Wissen stützt. Wird eine Inkonsistenz entdeckt, versucht der Roboter den internen Wissenszustand zu korrigieren. Ziel dieser Korrektur ist es ein möglichst umfangreiches Wissen von der Umwelt zu erlangen. Dazu geht die Arbeit der Frage nach "Was ist passiert?". Aus einem Pool von möglichen Hypothesen, der regelmäszig aktualisiert wird, selektiert das System einen Favoriten, der die aktuelle Wissensbasis des Roboters definiert.

Diese Dissertation zeigt weiters, wie Roboter durch ein Wissensmangement-System robuster und autonomer agieren können. Dieses System wird sowohl formal beschrieben als auch wichtige Eigenschaften formal bewiesen. Im ersten Schritt wird dieses System in einer Simulation umgesetzt. Im

nächsten Schritt wird das System in die Architektur eines realen Roboters eingepflegt. Weiters wird gezeigt, dass eine automatisierte Domänenanalyse die Leistungsfähigkeit und die Laufzeit des Systems drastisch verbessern kann. Alle Resultate werden durch eine umfangreiche Evaluierung untermauert.

Acknowledgement

I would like to thank Gerald Steinbauer for leading the project that enabled this thesis. He was always eager to provide and create the best environment that allows creative but also efficient work with robots. Besides organisational issues he always came up with fruitful ideas that guided my work. Furthermore, he initiated the idea of working on robots, by founding and leading the first RoboCup-Team at Graz University of Technology.

Furthermore, I would like to thank my supervisors Franz Wotawa and Gerhard Friedrich. Franz Wotowa's presenting style of how to automatize systems fascinated me already in the first semester. Therefore, he was the obvious first option for doing my bachelor, master and PhD thesis. Besides scientific input, I would like to thank Gerald Steinbauer and him for collecting and providing a financial basis, that allowed the RoboCup-Team to grow and to travel to competitions around the globe.

Additionally, I would like to thank Alexander Ferrein for his essential input. Though, discussing via skype often leads to different views, he was patient enough to discuss every problem in detail over hours and also days.

Moreover, I would like to thank Ingo Pill for his help in proof-reading papers and rephrasing paragraphs. Special thanks goes to the RoboCup-Team KickOffTUG that accompanied me most part of the studies. It was a great opportunity to meet outstanding people, and to start a company in a further step. Out of this group I would like to thank especially a few people. MÃątÃl' Wolfram built the basis for the real world robot tests within his master thesis. Christof Hoppe's way of scrutinizing things opened many new perspectives. I was lucky to share nearly every work day with Stephan Gspandl since the first semester. We took most of the courses together and worked close together in our bachelor and master thesis and shared the office during our PhD time at University. Special thanks goes to my groomsman Christoph Zehentner.

Additionally, I would like to thank my friends (especially Klemens, Bernhard, Andreas, Georg and Christian) for motivating each other, doing sports and for having a good time together. Furthermore I would like to thank my family for all the support during school and University that enabled me to finish this PhD. Especially I would like to thank my wife Verena-Maria Barbara for subtilizing my life, buying my time and encouraging me to finish my PhD. And of course I would like to thank my daughter Valentina Viktoria who gave me a very good reason to spend time at home and when sleeping use this time to finish this thesis.

Michael Reip in Graz, 2016

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Place, Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Ort, Datum Unterschrift

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Robots play an increasing role in our society. In order to fulfill society's expectations selecting the best action in every situation is crucial. This thesis aims to provide robots a consistent representation of the world, so that they can indeed select the best action in every situation. Section 1.1 further motivates this aim and introduces some background information. This is followed by investigating robotic frameworks in the surrounding of the presented work. Finally, Section 1.5 presents the structure of the remaining part of this thesis.

1.1. Background and Motivation

Robots are becoming more and more important in our daily lives. Vacuum cleaners whiz around apartments, or lawn-mowers cut huge areas of lawn independently. Though their cognitive capabilities are relatively modest, their features can not be neglected. Both type of robots are able to operate within boundaries but the environment is typically unknown and no map is used. They cruise the area uniformly, and they find back to their charging stations. Moreover they are very robust by means of avoiding falling down stairs, or colliding with obstacles safely. Of course there are many more elaborated examples aside from the user market. For instance, Minerva a tour-guide robot (see Thrun et al. (1999)) or Google's self driving car (Thrun, 2010) that covered a distance of 500,000 (Muller, 2013) miles without human interference. The amount of sensors on and in the car is enormous which makes the price of the vehicle far too high to spread quickly, but the project was able to overcome legal issues and launch a car that is allowed to drive autonomously in a few U.S. states such as Nevada or California. While Google's self driving car or similar cars from the DARPA Grand Challenge (see Iagnemma and Buehler (2006)) can be seen as classical robots with its sensors, actuators and a corresponding decision making system, boundaries blur when it comes to subsystems like lane-departure warning, collision avoidance or active park assist.
Another example for elaborated robot systems are robot driven warehouses or manufacturing plants. For example Kiva systems is a goods-to-man order picking and fulfillment system (Wyland, 2008), where persons operate at designated workstations and goods are delivered and retrieved by a fleet of robots (Wurman et al., 2007). Goods are stored at mobile inventory shelves. Robots pass under, lift them and move them to the designated location. For ease of coordination robot movements are executed on a grid and persons are not allowed to enter the workspace of the robots. More flexible

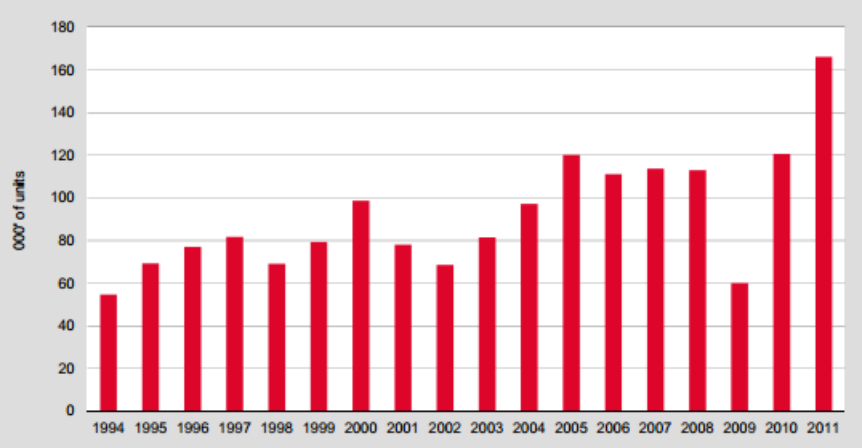

Figure 1.1.: Estimated worldwide annual shipments of industrial robots (IFR Statistical Department, 2012)

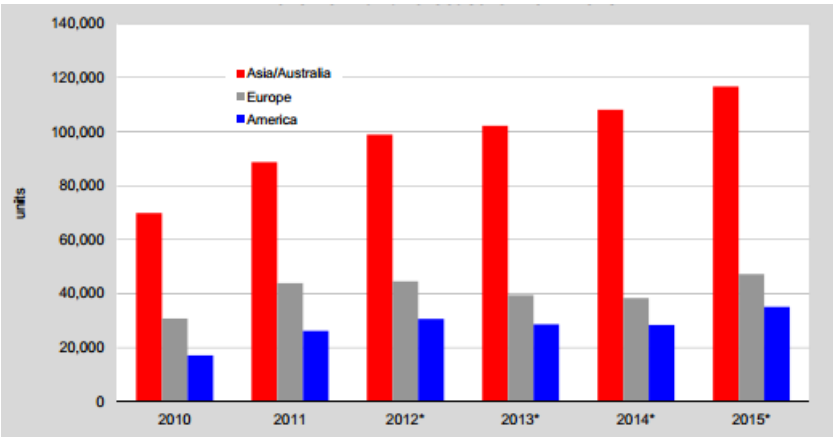

Figure 1.2.: Annual supply of industrial robots 2010-2011 and forecast for 2012-2015 (IFR Statistical Department, 2012)

solutions make it possible to share space with human employees and drive around obstacles, but are far from 4-digit numbers of robots per installation like Kiva. According to past sales volumes (see Fig. 1.1) and recent trends the number of robot units sold is expected to grow continually in Asia and respectively stay constant at this level or grow in 2015 in Europe and America (see Fig. 1.2).

A third application example for sophisticated robots is the exploration of outer space, especially in the Mars Exploration Rover Mission (Trebi-Ollennu, 2006) or the recent landing of Curiosity (Wall, 2012). Due to the communication delay of 14 minutes and 6 seconds over the 567 million kilometers tele-operated navigation with a 1-ton robot becomes complicated. Contingent upon the availability of enough images from the previous day, NASA navigation staff plans specific navigation commands. If there is too little information available Curiosity can switch to its autonomous mode and navigate to a predefined goal. Daily missions typically cover 100-200 meters with a maximum speed of 90 meters per hour.

All examples have in common that robots decide actively. They lack a global view of their envi-

ronment and dependability is crucial, though difficult to maintain. Dependability refers to the ability to perform its mission without causing catastrophes. This means that faults are inevitable and their consequences are negligible. Thus a dependable robot is safe to humans and to the environment. Dependability can be measured by the number of goals achieved and the amount of resources needed (e.g. time and distance) (Lussier et al., 2005).

Robots typically encounter a number of different faults during their missions. So let us step back to our manufacturing plant example where we inspect the work of a delivery robot. Imagine a manufacturing plant with some production machines, goods receipt where trucks supply raw materials and an on-site shop, where customers can pick up items on their own after reserving them online. The robot's task is to express deliver the blue box from the conveyor near the production machines to the shop (see Fig. 1.3).

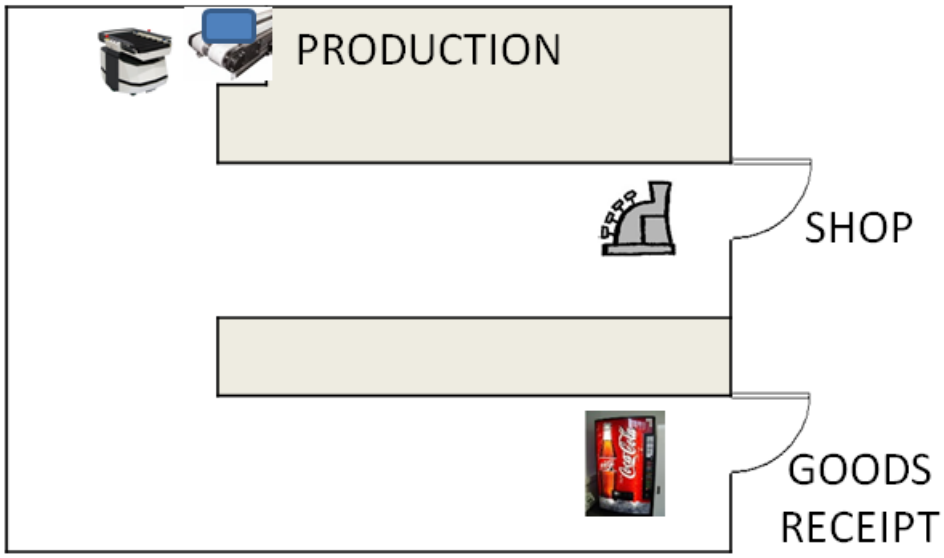

Figure 1.3.: Transport task initial situation

Due to localization or navigation problems the robot misses the aisle to the shop and ends up in the aisle to the goods receipt. At this point it encounters a soda machine, that is a contradiction to the expected cash register. There are several explanations for the observed situation (see Fig. 1.4).

First, the observed soda machine is an illusion, resulting from a sensing failure. Sensing failures either miss objects, or detect objects even if they are not present or mistake one object for another. Such failures can occur due to limited hardware capabilities in combination with sources of interference such as bright lights when using cameras, complex situations, or software faults.

Second, the robot missed the correct aisle and entered the goods receipt aisle. This scenario resembles the true situation in our example and is referred to execution failure. If the outcome of an action differs from the expected one, it is called an execution error. Execution errors may result from many sources. Especially dynamic environments are typically very challenging to robots, if the scene can not be observed as total. According to the ISO norm 26262 (Technical Committee ISO, 2011) we differentiate between faults, errors and failures. A fault resembles an abnormal condition that can cause an element or an item to fail. An error is the discrepancy between a computed, observed or measured

value or condition, and the true, specified, or theoretically correct value or condition. Finally, a failure is the termination of the ability of an element, to perform a function as required.

Third, there is a soda machine near the shop. One reason for the robot's inconsistency could be incorrect initial knowledge. Namely that the soda machine was always located near the shop. Another reason could be an exogenous event. Exogenous events summarize actions, that are not executed by the robot. This could include someone moving the soda machine from the goods receipt area to the shop, or someone installing a second soda machine near the shop. But exogenous events are not necessarily executed by other robots or humans, they can be used to model any kind of events like changing weather (if this information is relevant).

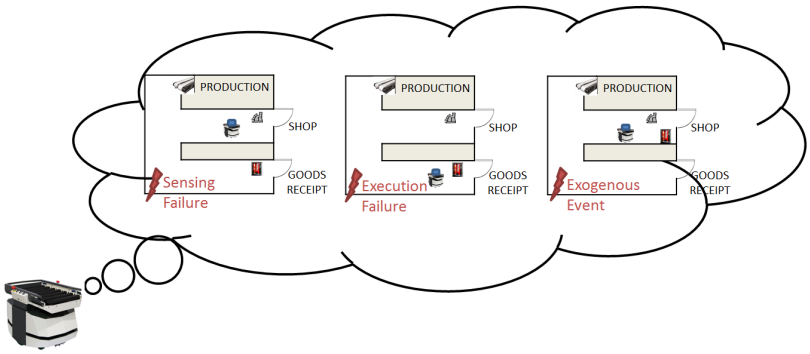

Figure 1.4.: Possible situations explaining the inconsistency.

This means a dependable robot is expected to cope with sensing failures, execution failures, incorrect or incomplete initial knowledge and exogenous events, during its continuous Sensing - Decision-Making - Execution cycle (see Fig. 1.5). Faults in the robot's hardware, software or domain description are beyond the scope of this thesis.

We present a diagnosis and repair framework that is able to detect inconsistencies within the robot's belief and repairs them if possible, where belief is a knowledge base that serves the robot for decision making. The presented framework is based on the IndiGolog high-level programming language which allows for interleaved execution, sensing and planning (de Giacomo et al., 2009). After every execution step a fast consistency check is performed. In case of an inconsistency the most plausible explanation is adapted, minimizing the risk of failures due to faults in the robot's belief.

1.2. Existing Frameworks

Before we continue with the proposed system we take a closer look on frameworks that follow similar strategies.

De Giacomo in (de Giacomo et al., 1998) and similar in (de Leoni et al., 2007) presents an execution monitoring framework for high-level robot programs. It builds up on the logic progamming language Golog and is able to recover from malicious exogenous actions. The framework is based on three, rather strong assumptions, namely (1) that the robot directly perceives all exogenous actions and (2) that sensing is always correct, and (3) there exists a program post-condition together with the property that all programs can be executed offline. After receiving an exogenous event, a predicate *Relevant* checks if the exogenous event prevents the program from termination. This part is done by offline

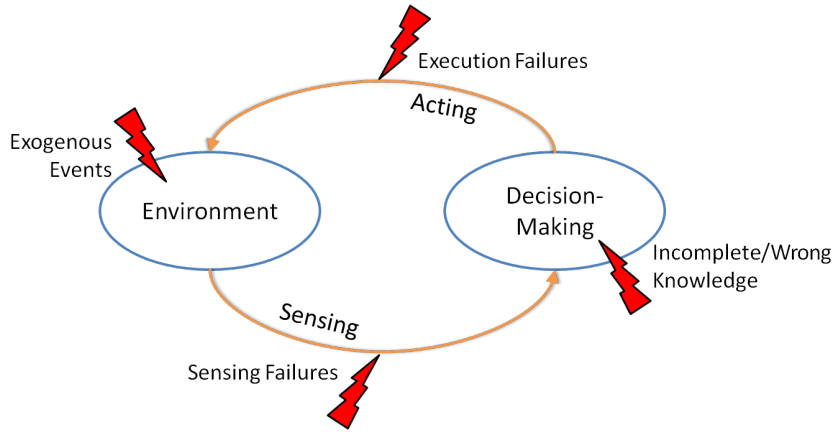

Figure 1.5.: Sensing - Decision-Making - Execution cycle (Gspandl, Podesser, Reip, Steinbauer, and Wolfram, 2012)

evaluation. If the program is still able to terminate, no further steps are necessary. If not, a predicate *Recover* changes the program in a way, that it is terminable and satisfies the program post-condition. The predicate *Recover* again uses offline evaluation. In contrast to the assumption that every reluctant change in the world is due to an exogenous event that can be directly observed, we further discriminate between execution and sensing failures. Furthermore, typically such failures or exogenous events are only partially observable.

Fritz in (Fritz, 2009) presents a planning and execution framework that is able to operate in highly dynamic environments with real world constraints. The emphasis lies on plan optimality during planning as well as during execution. The framework is composed of Plan Generation, Plan Execution, State Estimation, State Evaluation and Recovery components (see Fig. 1.6). Plan Generation is responsible for generating an optimal plan, while Plan Execution is responsible for applying the actions of that plan. State Estimation fuses all sensor values and identifies differences between expectations and actually observed properties. State Evaluation decides whether these deviations are relevant to the current plan. Finally, Recovery executes some kind of replanning, in case the observed deviations are relevant. Especially if planning takes longer than execution cycles, it is very important to know whether planning is necessary or not. Furthermore if replanning is necessary, this can be done efficiently by incorporating past planning results, while preserving optimality. Our framework differs as it focuses on providing the robot a consistent belief. Sensing results are directly integrated only if consistency can be assured.

CRAM, another relevant framework presented by Beetz et al. (2010) is a software toolbox for robotic cognition-enabled everyday manipulation tasks. It supports all life-cycle phases from design, to implementation and deployment. CRAM makes it possible to execute plans written in the CRAM Plan language (CPL) and to reason about belief states and environment observations. In contrast to conventional three tier architectures, CRAM provides features of the upper two levels such as parallel execution, partial ordering of sub-plans, sophisticated failure handling or semantic annotation in one level. Besides CPL, Prolog based KNOWROB is responsible for knowledge processing and reasoning. It either evaluates queries or return bindings of queries. Description logic OWL acts as knowledge representation. Both, CPL and KNOWROB can be fed with a set of extension modules like the "Everyday Manipulation" extension. This means the CRAM kernel itself stays very compact. The complete high-level robot control system is called COGITO (Beetz et al., 2010). The super framework

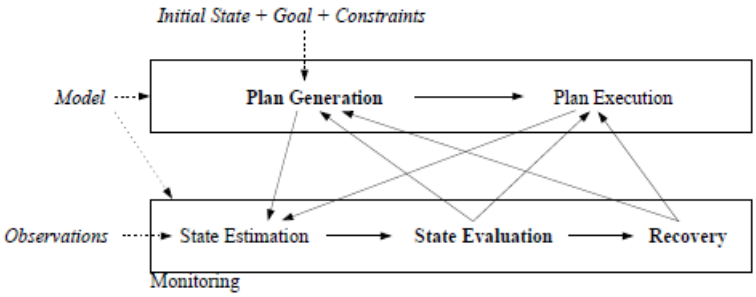

Figure 1.6.: A framework for plan generation and plan execution including monitoring (Fritz, 2009)

of CRAM adds the additional functionality of reasoning over plans. This is necessary for improving plans at runtime. In contrast to our framework, CRAM explicitly models the belief of the robot, but does not reason about inconsistencies within this.

The LAAS architecture is presented in (Alami et al., 1998). Though it contains a functional, execution and decision level, there are major differences to other frameworks. The decision level does not only allow for planning, it is realized by a planner-supervisor structure that allows reaction during planning. The functional level typically comprises all robot actions and perception capabilities. Similar to ROS (Quigley et al., 2009) functionality is grouped into modules that pass data or call services among each other. A module generator is responsible for module synthesis based on a formal module description. A dedicated controller is responsible for module management in order to facilitate transition between module states as idle, initialized, failed, interrupted or executing. Furthermore all relevant module data and events (default settings, parameter updates, processing results, commands) are automatically logged to databases. The execution level acts between the decision level and the functional level. It receives high-level commands and selects, parametrizes and synchronizes functional modules. All incoming requests are checked for logical constraints written in a formal language. Many of these constraints can be generated automatically out of module specifications. While the LAAS architecture pays little attention to repairing inconsistent beliefs, its fault tolerance is an ubiquitous design goal. This is realized by several generators that try to reduce coding errors by automatic generation. Furthermore, it is ensured that every part of the system reacts within a given time span and that it is interruptible. Additional rules, e.g. rules constraining resource usage ensure smooth system behavior.

Of course there are many other high-level robotic frameworks but most of them more or less ignore inconsistencies within a robot's belief or they do not pay attention to dependability. For instance, NASA Jet Propulsion Laboratory in combination with NASA Ames Research Center, Carnegie Mellon and University of Minnesota provide the "CLARAty" architecture for robotic autonomy (Volpe et al., 2001). This system only considers inconsistencies between decision and functional layer.

1.3. Research Objectives

This Section introduces the research objectives that motivated and guided the work of this thesis.

1.3.1. Formal Belief Management

According to the title of this thesis a Belief Management system forms the main interest of this work. Approaching this topic from an AI perspective the formal definition of such a Belief Management system is a key issue. Dependability is in the main focus. This means, that the Belief Management system is expected to provide a consistent belief even in harsh environments.

1.3.2. Belief Management Implementation handling various threats

Building up on a formal definition of such a dependable Belief Management system, the next objective is to have an implementation of this system. The implementation is expected to behave well even in the presence of several threats. This threats consist of incomplete knowledge, execution failure, sensing failure and exogenous events. In case of incomplete knowledge the robot is required to execute tasks, though it may not have full knowledge of all relevant data. For instance, it should be able to deliver items to some destination, without knowing exactly the current position of the item to be delivered. Furthermore a correct overall execution of tasks is expected, even if intermediate steps may fail, like picking an item. The same is expected to be true for sensing systems that do not provide correct measurements all the time, like reporting a non-existing item. Exogenous events, where the environment is changed but not directly observed by the robot, form another threat to be handled by a Belief Management system.

1.3.3. Efficient Belief Management on a real world Robot

Another key objective is the applicability of such an implementation to a real world robot. This means that we need a proper framework around the Belief Management in order to ensure a proper interaction with sensing and execution instances of a real world robot. If robots operate in the same environment as humans do, they are expected to behave in such a way that they do not block their environment. This means we expect the robot to behave efficiently. In terms of a Belief Management system, this issue can be addressed by efficient runtimes of the system.

1.3.4. Systematic Evaluation

Finally, one objective of this thesis is to have a comprehensive evaluation of the proposed system. This evaluation is expected to answer if such a Belief Management system is able to strengthen or even enlarge a robot's capabilities. All the proposed objectives are expected to be systematically evaluated.

1.4. Contributions

This Section gives an overview over the major contributions of this thesis and explains them shortly.

1.4.1. Formal Definition and Implementation of Belief Management

Roughly speaking a Belief Management is a system that provides the robot with a base for decision making. The formal definition of consistency that is crucial for the application of a belief management and the formalization of the belief management itself was published in Gspandl, Pill, Reip, and Steinbauer (2011). Based on this notation we are able to prove several properties of this belief management. This includes the maximum number of hypotheses needed to explain all inconsistencies after a number of k executed actions or the number of all hypotheses that explain inconsistencies arising from a given number of failures. This was published in Gspandl, Pill, Reip, Steinbauer, and Ferrein (2011). The implementation together with the formal definition was published in Gspandl, Pill, Reip, and Steinbauer (2013).

1.4.2. Robot Control Architecture

On the basis of the formal definition, a robot control architecture is achieved. It uses an adapted version of IndiGolog (de Giacomo et al., 2009) that is able to interact with our belief management implementation. A ROS interface covers action execution and sensor information retrieval. This means, this robot control architecture is able to control simulated agents as well as real world robots. Parts of this architecture were published in Wolfram, Gspandl, Reip, and Steinbauer (2011) and Gspandl, Podesser, Reip, Steinbauer, and Wolfram (2012).

1.4.3. Diagnosis Templates

Observations have shown that inconsistencies within the robots belief often arise due to single failures. Furthermore, especially in large domains generating and testing all hypotheses can be very time-consuming. So called Diagnosis Templates can help to reduce the number of necessary tests dramatically. Diagnosis Templates are pre-calculated and explain one inconsistency. At run-time only hypotheses that follow from Diagnosis Templates explaining the current inconsistency are tested. The concept of Diagnosis Templates was published in Reip, Steinbauer, and Ferrein (2012).

1.4.4. Belief Management Optimization

Besides the usage of Diagnosis Templates that have a huge impact on the runtime of the belief management system, several smaller concepts are used to optimize the runtime performance of the presented system. Therefore we use several open-lists to influence the hypotheses evaluation selection in order to achieve faster runtimes. Furthermore, we derive the notation of a hypothesis compression set. Multiple hypothesis can be compressed into one compression set. The result of the compression again forms a hypothesis by its own. The usage of compressed hypotheses can tender many consistency checks unnecessary. In the worst case the runtime of a system applying this feature is not better than the original system.

1.4.5. Domain Knowledge Generation

Domain Knowledge Generation is essential for intelligent systems, and thus essential for a Belief Management system as well. Providing correct and comprehensive data input is crucial for a wide applicability and acceptance of our Belief Management system and additionally for its correct behavior.

Though this topic is a large research topic on its own, first steps in this directions have been undertaken with colleagues. In this way, two approaches were developed and the findings were published in Gspandl, Reip, Steinbauer, and Wotawa (2010) and Gspandl, Monichi, Reip, Steinbauer, Wolfram, and Zehenter (2007) and Gspandl, Hechenblaickner, Reip, Steinbauer, Wolfram, and Zehentner (2012).

1.5. Outline

The remaining thesis is structured as follows: Chapter 2 provides necessary preliminaries. This includes the situation calculus, IndiGolog and history-based diagnosis. The robot delivery domain is presented in Chapter 3 as it forms the running example throughout the thesis. Chapter 4 introduces the formal definition of the belief management. These findings are extended by the application of Guarded Action Theories in Chapter 5. This is followed by Chapter 6 that transfers this system to a real robot. Chapters 7 and 8 tackle performance issues. Chapter 7 presents Diagnosis Templates and Chapter 8 combines hypothesis compression and hypothesis selection. Chapter 9 provides an evaluation of the presented system. Domain Knowledge Generation is tackled in Chapter 10. Finally, Chapter 11 concludes the thesis and depicts directions for further research.

Chapter 2

Preliminaries

Within this chapter we provide theoretical foundations for the remaining part of this thesis. First, we introduce the situation calculus a logic formalism. This is followed by an overview of IndiGolog. IndiGolog is a high-level programming language that allows incremental program execution. Finally, we give an introduction to histoty-based diagnosis. Please note that, as usual, all free variables in formulas are meant to be implicitly universally quantified. Lists of variables are abbreviated as vector: $\vec{x}$ for $x_1, x_2, ..., x_n$.

2.1. Situation Calculus

The situation calculus is a second order language. It was first proposed by McCarthy (1963) and later refined by Reiter (2001). Every change in the world is modeled by named actions. Sequences of actions denoting world histories are called situations. The initial situation is represented by the constant S_0. A special binary function symbol *do* maps situation × action to its successor situation. $s' = do(\alpha, s)$ denotes the resulting situation s' after executing action α in situation s. If not stated otherwise, actions are indicated by α, β, χ. For example, if action $pickup(x)$ states that a robot takes some object x, $do(pickup(x), s)$ relates to the situation after the robot picked up the object in situation s. Situations and actions are first order terms. For example, $do(putdown(x), do(goto(room), do(pickup(x), s)))$ is a situation term for the action sequence $[pickup(x), goto(room), putdown(x)]$ in situation s. Note that due to the recursive behavior of *do*, action sequences have to be read from right to left. Relations and functions that change over time (from situation to situation) are called relational fluents and functional fluents respectively. They take exactly one situation term as last argument. Within the situation suppressed form, this last parameter is omitted. For example the predicate $is\_at(obj, room)$ denoting if object *obj* is in room *room* is dependent of the current history. Please note that we use history and situation synonymously. It might change after the robot picks up an object and moves it to another room.

The executability of actions is denoted by a special predicate *Poss*. $Poss(\alpha, s)$ denotes that action α is executable in situation s (Action Preconditions). For instance, $Poss(pickup(obj), s) \supset \exists room.\ is\_at(obj, room, s) \wedge at(room, s)$ defines that a robot can only pickup an object *obj* if it is in the same room as the object. However the executability of actions may vary with situations. For example, the

robot is required to have enough battery for executing the pickup task. Or if the robot is located in a crowd, it must not be blocked. So enumerating and considering all these possible contexts seems to be impossible at design time for real-world related domains. If preconditions are too specific, they can not be instantiated at run-time as some facts might not be available. The problem of defining them best is called the qualification problem (Lin, 2008). The classical example for this problem is $bird(x) \wedge \neg penguin(x) \wedge \neg ostrich(x) \wedge \neg pekingDuck \wedge \ldots \supset flies(x)$. This means a bird flies if it is not a penguin, an ostrich or ... Given the fact $bird(Tweety)$ and trying to infer $flies(Tweety)$, this cannot be achieved as there is no information about Tweety being a penguin or not. Reiter (2001) distinguishes between important qualifications as $bird(x)$ and minor qualifications as $\neg penguin(x), \neg ostrich(x), \ldots$ and ignores all "minor" qualifications.

Another omnipresent problem when axiomatizing dynamic worlds is the frame problem. Effects are generally modeled by effect axioms. They describe how an action changes a fluent. For instance, $has\_object(obj,s) \supset is\_at(obj,room,do(goto(room),s))$ states that the object *obj* is in room *room* after the robot moves there and whether the robot is in possession of this object. Besides these effect axioms, frame axioms for describing action invariants are needed. Those cover fluents that are not changed by actions. For example the fluent *has_object* is not affected by the action *goto*: $has\_object(obj,s) \supset has\_object(obj,room,do(goto(room),s))$ and $\neg has\_object(obj,s) \supset \neg has\_object(obj,room,do(goto(room),s))$ for the negative case. This results in $2 \times |A| \times |F|$ axioms, where $|A|$ stands for the number of actions and $|F|$ for the number of fluents. On the one hand, the axiomatizer needs to write down all axioms and on the other hand the implementation has to reason efficiently over such a huge number of axioms. A solution to the frame problem seeks to fulfill two goals. First, it should be possible to generate frame axioms out of effect axioms. This reduces error-proneness and saves a lot of painful labor. Second, the solution is expected to lead to a minimum number of axioms.

The first part of the frame problem was addressed by Pednault (Pednault, 1989). Given a fluent *on* describing the state of a lightbulb and an action switch that either switches on the lightbulb on or switches it off, the positive and negative effect axioms are (Reiter, 2001):

$$\neg on(x,s) \supset on(x,do(flip(x),s))$$
$$on(x,s) \supset \neg on(x,do(flip(x),s))$$

This is extended to the logically equivalent forms:

$$\neg on(x,s) \wedge y = x \supset on(x,do(flip(y),s))$$
$$on(x,s) \wedge y = x \supset \neg on(x,do(flip(y),s))$$

Assuming that the formulas indicated above describe all possibilities how action *flip* is able to alter the fluent *on*, we can conclude that, if the lightbulb was on before *flip* and off after *flip*, then $on(x,s) \wedge y = x$ held. This leads to:

$$on(x,s) \wedge \neg on(x,do(flip(y),s)) \supset y = x$$
$$\neg on(x,s) \wedge on(x,do(flip(y),s)) \supset y = x$$

This is again logically equivalent to:

$$on(x,s) \wedge y \neq x \supset on(x,do(flip(y),s))$$
$$\neg on(x,s) \wedge y \neq x \supset \neg on(x,do(flip(y),s))$$

Finally, we received our first frame axioms. It states that the action *flip* has no effect on fluent *on* respectively $\neg on$ whenever y does not equal x. This derivation is specific, but it can be generalized. This is done with a positive ($\varepsilon_F^+(\vec{x},\vec{y},s)$) and negative ($\varepsilon_F^-(\vec{x},\vec{y},s)$) effect axiom for every action $\alpha(\vec{y})$ and fluent $F(\vec{x},s)$. The effect axioms are first-order formulas where all free variables are in $\vec{x},\vec{y},s$.

$$\varepsilon_F^+(\vec{x},\vec{y},s) \supset F(\vec{x},do(\alpha(\vec{y}),s))$$
$$\varepsilon_F^-(\vec{x},\vec{y},s) \supset \neg F(\vec{x},do(\alpha(\vec{y}),s))$$

Again, we use the assumption that both effect axioms express all cases where action α influences fluent F (Causal Completeness Assumption). If $F(\vec{x},s)$ and $\neg F(\vec{x},do(\alpha(\vec{y},s)))$ hold, we can rewrite the above-mentioned formulas to:

$$F(\vec{x},s) \wedge \neg F(\vec{x},do(\alpha(\vec{y}),s)) \supset \varepsilon_F^-(\vec{x},\vec{y},s)$$
$$\neg F(\vec{x},s) \wedge F(\vec{x},do(\alpha(\vec{y}),s)) \supset \varepsilon_F^+(\vec{x},\vec{y},s)$$

This is equivalent to:

$$F(\vec{x},s) \wedge \neg\varepsilon_F^-(\vec{x},\vec{y},s) \supset F(\vec{x},do(\alpha(\vec{y}),s))$$
$$\neg F(\vec{x},s) \wedge \neg\varepsilon_F^+(\vec{x},\vec{y},s) \supset \neg F(\vec{x},do(\alpha(\vec{y}),s))$$

Thus, we are able to automatically compute frame axioms for every action fluent combination that consists of effect axioms. If there is no connection by means of effect axioms, empty effect axioms are required:

$$false \supset F(\vec{x},do(\alpha(\vec{y}),s))$$
$$false \supset \neg F(\vec{x},do(\alpha(\vec{y}),s))$$

This solution presented by Pednault makes it possible to generate frame axioms easily, but with a clear drawback, the high number of frame axioms.

The second part of the frame problem, namely the compact representation of frame axioms (Schubert called them explanation closure axioms), was first addressed by Haas (1987) and later extended by Schubert (1990). They focus on identifying actions that change a fluent. For instance, if a robot is holding an object, but does not in the successor situation, then it either puts the object down or someone took the object.

$$has\_object(x,s) \wedge \neg has\_object(x,do(a,s)) \supset a = putdown(x) \vee a = snatch(x)$$

This can be rewritten to a frame axiom:

$$has\_object(x,s) \wedge a \neq putdown(x) \wedge a \neq snatch(x) \supset has\_object(x,do(a,s))$$

Every action other than *snatch* or *putdown* leaves the fluent *has_object* unchanged under the assumption that actions with different names are different actions (Unique Name Assumption). The generalization of the given example leads to:

$$F(\vec{x},s) \wedge \neg F(\vec{x},do(a,s)) \supset \Phi_F(\vec{x},a,s)$$
$$\neg F(\vec{x},s) \wedge F(\vec{x},do(a,s)) \supset \Psi_F(\vec{x},a,s)$$

This states that $\Phi_F(\vec{x},a,s)$ and $\Psi_F(\vec{x},a,s)$ provide an explanation for every change of F. This can be rewritten to the logically equivalent form:

$$F(\vec{x},s) \wedge \neg\Phi_F(\vec{x},a,s) \supset F(\vec{x},do(a,s))$$
$$\neg F(\vec{x},s) \wedge \neg\Psi_F(\vec{x},a,s) \supset \neg F(\vec{x},do(a,s))$$

Due to an universal quantification over actions, it is possible to reduce the number of frame axioms from $2 \times |A| \times |F|$ to $2 \times |F|$. Thus, we receive a compact representation of frame axioms, but Schubert argued that there is no algorithm to generate them automatically.

Finally, Reiter merged both approaches to receive a compact representation of frame axioms that is easy to provide (Reiter, 1991, 2001). This is shown by an example of an object that breaks (if fragile) after dropping it, or if the object is close to an exploding bomb. Broken objects can be repaired. The positive and negative effect axioms for broken are:

$$fragile(x,s) \supset broken(x,do(drop(r,x),s))$$
$$next\_to(b,x,s) \supset broken(x,do(explode(b),s))$$
$$\neg broken(x,do(repair(x,s))$$

This can be rewritten to the logically equivalent form:

$$[\exists r.a = drop(r,x) \wedge fragile(x,s) \vee \exists b.\{a = explode(b) \wedge next\_to(b,x,s)\}] \supset broken(x,do(a,s))$$
$$\exists r.a = repair(r,x) \supset \neg broken(x,do(a,s))$$

Under the completeness assumption that we have given all effect axioms leading to F respectively $\neg F$, we can state that a change from F to $\neg F$ is due to the actions *drop* or *explode*, and a change from $\neg F$ to F is due to *repair*. We can rewrite the sentence according to Schubert's proposal indicated above:

$$\neg broken(x,s) \wedge broken(x,do(a,s)) \supset$$
$$\exists r.a = drop(r,x) \wedge fragile(x,s) \vee \exists b.\{a = explode(b) \wedge next\_to(b,x,s)\}$$

And for the negative case:

$$broken(x,s) \wedge \neg broken(x,do(a,s)) \supset \exists r.a = repair(r,x)$$

Based on this example we can derive a general case. Therefore we take a step back and investigate the transformation of effect axioms into their normal forms. The positive and negative normal forms are denoted by:

$$\gamma_F^+(\vec{x},a,s) \supset F(\vec{x},do(a,s))$$
$$\gamma_F^-(\vec{x},a,s) \supset \neg F(\vec{x},do(a,s))$$

Starting from effect axioms, we are faced with effect axioms of the form:

$$\phi_F^+ \subset F(\vec{t},do(\alpha,s))$$

α denotes an action term such as $pickup(x)$ or $goto(room)$ and $\vec{t}$ is a term. By expliciting the action and its parameters we receive:

$$a = \alpha \wedge \vec{x} = \vec{t} \wedge \phi_F^+ \supset F(\vec{x},do(a,s))$$

$\vec{x} = \vec{t}$ is an abbreviation for $x_1 = t_1 \wedge x_2 = t_2 \wedge ... \wedge x_n = t_n$ and $\vec{x}$ are new variables that differ from one another and from any variable in the original effect axiom. If $y_1, y_2, ..., y_m$ stand for all free (implicitly universally qualified) variables except the situation variable s, then the following sentence is logically equivalent:

$$\exists y_1, y_2, ..., y_m.[a = \alpha \wedge \vec{x} = \vec{t} \wedge \phi_F^+] \supset F(\vec{x}, do(a,s))$$

To this extent, every effect axiom with free variables $\vec{x}, a, s$ can be written as:

$$\psi_F \supset F(\vec{x}, do(a,s))$$

For each of k positive effect axioms we receive:

$$\begin{aligned} \psi_F^{(1)} &\supset F(\vec{x}, do(a,s)) \\ \psi_F^{(2)} &\supset F(\vec{x}, do(a,s)) \\ &\vdots \\ \psi_F^{(k)} &\supset F(\vec{x}, do(a,s)) \end{aligned}$$

This can be combined to a single, logically equivalent sentence, that defines the normal form of the positive effect axioms for fluent F. The negative case is derived in the same manner.

$$[\psi_F^1 \vee \psi_F^{(2)} \vee ... \vee \psi_F^{(k)}] \supset F(\vec{x}, do(a,s))$$

For simplicity we will use the abbreviated positive and negative forms:

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a,s)) \tag{2.1}$$
$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a,s)) \tag{2.2}$$

Having defined the positive and negative normal form effect axiom for a fluent F, we can apply the causal completeness assumption, stating that both sentences completely describe all cases where F changes from positive to negative or vice versa. This yields to explanation closure axioms:

$$F(\vec{x}, s) \wedge \neg F(\vec{x}, do(a,s)) \supset \gamma_F^-(\vec{x}, a, s) \tag{2.3}$$
$$\neg F(\vec{x}, do(a,s)) \wedge F(\vec{x}, s) \supset \gamma_F^+(\vec{x}, a, s) \tag{2.4}$$

Again it is necessary that actions with different names are different (unique name assumption).

$$\neq \exists \gamma_F^+(\vec{x}, a, s) \wedge \gamma_F^-(\vec{x}, a, s) \tag{2.5}$$

If now T is a first-order theory and entails that in no situation the positive normal form effect axiom for fluent F and the negative normal form effect axiom for fluent F can hold both (2.5), then the normal form effect axioms and the explanation closure axioms logically correspond:

$$F(\vec{x}, do(a,s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg \gamma_F^-(\vec{x}, a, s). \tag{2.6}$$

Sentence 2.6 is called the successor state axiom for fluent F. It describes all changes to a fluent in a compact way and can be generated according to the procedure indicated. Thus, they provide an

elegant solution to the frame problem. This holds if the effect axioms have the form that a fluent F becomes true or false after execution of a given action a, if some condition R held before:

$$R(\vec{x},s) \supset (\neg)F(\vec{x},do(a,s)) \tag{2.7}$$

Furthermore actions must be deterministic. So actions with uncertain effects such as flipping a coin cannot be considered. Taking the above mentioned example we receive the following successor state axiom for the fluent *broken*:

$$broken(x,do(a,s)) \equiv \exists r.a = drop(r,x) \wedge fragile(x,s) \vee$$
$$\exists b.\{a = explode(b) \wedge next\_to(b,x,s)\} \vee broken(x,s) \wedge \neg \exists r.a = repair(r,x)$$

2.1.1. Formal Language and Basic Action Theory

The situation calculus is a second-order language with equality. Within this thesis we follow the definition of Reiter (2001). It consists of the sorts action, situation and a sort object for all domain specific objects. The Basic Action Theory (BAT) provides the basic axiomatization of the language. The language specific alphabet consists of:

- variable symbols
- 2 function symbols of sort situation (constant S_0 and do: action $\times situation \rightarrow situation$)
- a binary predicate $\sqsubset$: $situation \times situation$
- a binary predicate symbol $Poss : action \times situation$
- for each n $\geq$ 0, countable infinite predicate symbols with arity n of the form $(action \cup obj)^n$ (situation independent relations)
- for each n $\geq$ 0, countable infinite function symbols with arity n of the form $(action \cup obj)^n \rightarrow obj$ (situation independent functions)
- for each n $\geq$ 0, finite or countable infinite function symbols with arity n of the form $(action \cup obj)^n \rightarrow action$ (action functions)
- for each n $\geq$ 0, finite or countable infinite predicate symbols with arity n of the form $(action \cup obj^n \times situation$ (relational fluents)
- for each n $\geq$ 0, finite or countable infinite function symbols with arity n of the form $(action \cup obj)^n \times situation \rightarrow obj$ (functional fluents)

Variable symbols allow arbitrary use of domain specific variables. S_0, as mentioned earlier, denotes the initial situation. The function *do* yields to successor situations. Besides fluents that have a situation as last argument, no other function or relation is allowed to use situations as arguments. A binary predicate $\sqsubset$ allows for situation ordering. Action preconditions are denoted by *Poss*. Situation independent relations and functions model facts that do not change over time, e.g. prime(x) or sinus(x). Besides the given specific alphabet part, the alphabet contains the standard alphabet of logical symbols.

The foundational axioms for situations consist of four axioms and are denoted by Σ:

$$do(a_1, s_1) = do(a_2, s_2) \supset a_1 = a_2 \wedge s_1 = s_2 \tag{2.8}$$

$$\forall P.[P(S_0) \wedge \forall a, s.P(s) \supset P(do(a, s))] \supset \forall s.P(s) \tag{2.9}$$

$$\neg s \sqsubset S_0 \tag{2.10}$$

$$s \sqsubset do(s, s') \equiv s \sqsubseteq s' \tag{2.11}$$

The first axioms 2.8 defines unique names for situations. This means situations are only the same if they consist of the same actions. That is different to states (as used in different languages), which describe the values of fluents. Two situations might lead to the same values of all fluents, but they are different as long as their action history is different. An action history is described by recursive do functions starting from the initial situation. The next axiom 2.9 leads to the whole set of situations by induction, starting in the initial situation. The initial situation is ensured to be the first situation by axiom 2.10. Additionally, axiom 2.11 leads to an ordering of situations, where $s \sqsubseteq s'$ that is not contained in the language's alphabet, is an abbreviation for $s \sqsubset s' \vee s = s'$. All four axioms are domain-independent. Combining the given axioms we can see situations form a tree. The initial situation acts as root node. This rood node is extended layer by layer with all possible actions.

Situations can consist of an arbitrary number of actions. Syntactically every action can be used, but very often we are only interested in situations that are executable. This means, starting in the initial situation, the first action can be executed and every following action can be executed after its predecessor action has finished. Based on the predicate *Poss*, we introduce an abbreviation *executable* that states that all actions from the given situation's action history are executable:

$$executable(s) \equiv \forall a, s^*.do(a, s^*) \sqsubseteq s \supset Poss(a, s^*) \tag{2.12}$$

The basic theory of actions (BAT) is a collection of all axioms defining the situation calculus:

$$\mathcal{D} = \Sigma \cup \mathcal{D}_{SS} \cup \mathcal{D}_{ap} \cup \mathcal{D}_{una} \cup \mathcal{D}_{S_0} \tag{2.13}$$

Besides the foundational axioms for situations, it contains the successor state axioms $\mathcal{D}_{SS}$ for functional and relational fluents, the action precondition axioms $\mathcal{D}_{ap}$, the unique names axioms for actions $\mathcal{D}_{una}$ and the initial situation $\mathcal{D}_{S_0}$. Successor state axioms for relations are already defined in 2.6. Successor state axioms for functional fluents are defined in a similar manner and have the following form

$$F(x_1, x_2, ..., x_n, do(a, s)) \equiv \Phi_F(x_1, x_2, ..., x_n, a, s)$$

In this case $\Phi_F(x_1, x_2, ..., x_n, a, s)$ is a formula uniform in s. Uniform in s means that the given formula is determined by situation s and does not depend on any situation s' with $s \neq s'$. Action preconditions are sentences of the form

$$Poss(\alpha(x_1, x_2, ..., x_n), s) \equiv \Pi_\alpha(x_1, x_2, ..., x_n)$$

Here $\alpha(x_1, x_2, ..., x_n), s)$ is an action function symbol with arity n and $\Pi_\alpha(x_1, x_2, ..., x_n)$ a formula that is uniform in s. Unique name axioms for actions ensure that actions with different names relate to different actions. The initial situation describes the root of execution. It consists of first-order sentences that are uniform in S_0, thus having no dependency on any other situation than the initial one. For consistency reasons it is necessary that for every fluent, the condition of a maximum of one successor state axiom holds.

2.1.2. Regression

Regression is a necessary prerequisite for reasoning. It transforms formulas uniform in s into a formula uniform in S_0 $s = do([a_1, a_2, ..., a_n], S_0)$. We abbreviate the situation term $do(a_n, do(a_{n-1}, ..., do(a_2, do(a_1, \sigma))))$ with $do([a_1, a_2, ..., a_{n-1}, a_n], \sigma)$ for ease of readability. This way we can evaluate formulas after a sequence of actions was performed, which is a requirement for planning and automated reasoning. A formula $\mathcal{W}$ that is uniform in situation s and contains a fluent F is replaced by a logical equivalent formula $\mathcal{W}'$ that contains the successor state axiom of fluent F instead of the fluent itself. For a relational fluent $F(\vec{t}, do(\alpha, \sigma))$ with successor state axiom $F(\vec{x}, do(a, s)) \equiv \Phi(\vec{x}, a, s)$, the fluent $F(\vec{t}, do(\alpha, \sigma))$ in $\mathcal{W}$ is reduced to $\Phi(\vec{x}, \sigma, \sigma)$ in $\mathcal{W}'$. Note that $\mathcal{W}'$ is closer to S_0 than $\mathcal{W}$. This step is repeated as long as the formula simply depends on situation S_0. Formulas in S_0 can be evaluated using only $\mathcal{D}_{S_0}$ and $\mathcal{D}_{una}$. Formulas are regressable if each situation term has the syntactic form $do([a_1, a_2, ..., a_n], S_0)$ with actions $a_1, a_2, ..., a_n$, and for each $Poss(\alpha, \sigma)$ predicate α has the form $A(t_1, t_2, ..., t_n)$. There is no quantification over situations and there is no usage of predicate $\sqsubset$ or of the equality operator on situations.

Reiter (2001) defined a regression operator $\mathcal{R}$ that reduces regressable formulas $\mathcal{W}$ uniform in s, into formulas uniform in S_0. The regression operator builds up on BAT.

- $\mathcal{W}$ is an atom.
  - If $\mathcal{W}$ is an situation independent atom then
    $\mathcal{R}[\mathcal{W}] = \mathcal{W}$
  - If $\mathcal{W}$ is a relational fluent of the form $F(\vec{t}, S_0)$ then
    $\mathcal{R}[\mathcal{W}] = \mathcal{W}$
  - If $\mathcal{W}$ is a functional fluent of the form $f(\vec{t}, S_0)$ then
    $\mathcal{R}[\mathcal{W}] = \mathcal{W}$
  - If $\mathcal{W}$ is a *Poss* atom of the form $Poss(A(\vec{t}), \sigma)$ for action $A(\vec{t})$ and situation σ with precondition axiom $Poss(A(\vec{x}, s) \equiv \Pi_A(\vec{x}, s)$, then
    $\mathcal{R}[\mathcal{W}] = \mathcal{R}[\Pi_F(A(\vec{t}), \sigma)]$
  - If $\mathcal{W}$ is a relational fluent of the form $F(\vec{t}, do(\alpha, \sigma))$ with successor state axiom
    $F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s)$ then
    $\mathcal{R}[\mathcal{W}] = \mathcal{R}[\Phi_F(\vec{t}, \alpha, \sigma)]$
  - If $\mathcal{W}$ is a functional fluent of the form $f(\tau, do(\alpha', \sigma'))$ with every mentioned situation of the form $do([a_1, a_2, ..., a_n], S_0)$ and successor state axiom $f(\vec{x}, do(a, s)) = y \equiv \phi_f(\vec{x}, y, a, s)$, then
    $\mathcal{R}[\mathcal{W}] = \mathcal{R}[\exists y.\phi_f(\vec{t}, y, \alpha, \sigma) \wedge \mathcal{W}|_y^{f(\vec{t}, do(\alpha, \sigma}]$ where $\phi|_t^{t'}$ denotes the replacement of all occurrences of t' by t in formula ϕ. In our case every function fluent f is replaced by its result y.
- $\mathcal{W}$ is non-atomic. It is defined inductively over logical connectives. Note that the use of connectives $\neg$, $\wedge$ and $\exists$ is sufficient in order to formulate logically equivalent sentences to all other logical connectives or quantifiers.
  - $\mathcal{R}[\neg \mathcal{W}] = \neg \mathcal{R}[\mathcal{W}]$
  - $\mathcal{R}[\mathcal{W}_1 \wedge \mathcal{W}_2] = \mathcal{R}[\mathcal{W}] \wedge \mathcal{R}[\mathcal{W}]$
  - $\mathcal{R}[\exists v.\mathcal{W}] = \exists v.\mathcal{R}[\mathcal{W}]$

Regression delivers logical equivalent formulas that depend on the initial situation only. Though the regression depends on successor state axioms $\mathcal{D}_{ss}$, the subsequent evaluation is only based on $\mathcal{D}_{una}$ and $\mathcal{D}_{S_0}$. Regressed formulas are first-order and the second-order induction axiom from Σ is not in use. Thus regression can be done in first-order.

2.2. IndiGolog

IndiGolog is an agent programming language and one of the most recent Golog (see Levesque et al. (1997)) variants. Golog is based on an extended version of the situation calculus that allows for complex actions and procedures. It offers high-level structures to program a robot's behavior such as *if then else* and *while* statements or subprograms. The functionality of Golog was first extended by ConGolog in (de Giacomo et al., 2000). ConGolog added support for concurrency and interrupts. Both Golog and ConGolog are executed offline, i.e. the complete program is evaluated first, before any execution is triggered. This limitation is intolerable in dynamic real-world robotic setups. IndiGolog, an incremental and online Golog variant, tackles this drawback. That means, IndiGolog only commits to one action, executes it and selects the next action. This obviously leads to situations where tasks cannot be established, due to the greed for executing an action. In case the user wants a robot to examine more than the next action, a search operator exists. This search operator facilitates the possibility to do offline evaluation similar to Golog within a defined lookahead. This allows the programmer to switch smoothly between deliberation and reactive behavior (Sardiña, 2005). The formal semantics of IndiGolog builds up on ConGolog and is explained in Chapter 2.2.1. The key components of IndiGolog are presented in Chapter 2.2.2.

2.2.1. IndiGolog Formal Semantics

The formal semantics of IndiGolog builds up the work done in ConGolog. Therefore, it uses two special predicates *Trans* and *Final*. $Trans(\delta, s, \delta', s')$ evaluates if by a single program execution step leads from program δ in situation s to program δ' in situation s'. $Final(\delta, s)$ checks if program δ terminates in situation s. The predicate *Trans* is defined by the axioms $\mathcal{T}$:

1. Empty program:
   $Trans(nil, s, \delta', s') \equiv False$
2. Primitive actions:
   $Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$
3. Test/wait actions:
   $Trans(\phi?, s, \delta', s') \equiv \phi[s] \wedge \delta' = nil \wedge s' = s$
4. Sequence:
   $Trans(\delta_1; \delta_2, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma; \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee Final(\delta_1, s) \wedge Trans(\delta_2, s, \delta', s')$
5. Non-deterministic branch:
   $Trans(\delta_1 | \delta_2, s, \delta', s') \equiv Trans(\delta_1, s, \delta', s') \vee Trans(\delta_2, s, \delta', s')$
6. Non-deterministic choice of argument:
   $Trans(\pi v.\delta, s, \delta', s') \equiv \exists s.Trans(\delta_x^v, s, \delta', s')$
7. Non-deterministic iteration:
   $Trans(\delta^*, s, \delta', s') \equiv \exists\gamma.(\delta' = \gamma; \delta^*) \wedge Trans(\delta, s, \gamma, s')$

8. Synchronized conditional:
   $Trans(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif }, s, \delta', s') \equiv \phi[s] \wedge Trans(\delta_1, s, \delta', s') \vee \neg\phi[s] \wedge Trans(\delta_2, s, \delta', s')$
9. Synchronized loop:
   $Trans(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s, \delta', s') \equiv$
   $\exists\gamma.(\delta' = \gamma; \textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}) \wedge \phi[s] \wedge Trans(\delta, s, \gamma, s')$
10. Concurrent execution:
    $Trans(\delta_1 \| \delta_2, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma \| \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee \exists\gamma.\delta' = (\delta_1 \| \gamma) \wedge Trans(\delta_2, s, \gamma, s')$
11. Prioritized concurrency:
    $Trans(\delta_1 \rangle\!\rangle \delta_2, s, \delta', s') \equiv$
    $\exists\gamma.\delta' = (\gamma \rangle\!\rangle \delta_2) \wedge Trans(\delta_1, s, \gamma, s') \vee$
    $\exists\gamma.\delta' = (\delta_1 \rangle\!\rangle \gamma) \wedge Trans(\delta_2, s, \gamma, s') \wedge \neg\exists\zeta, s''.Trans(\delta_1, s, \zeta, s'')$
12. Concurrent iteration:
    $Trans(\delta^{\|}, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma \| \delta^{\|}) \wedge Trans(\delta, s, \gamma, s')$

The predicate $Trans(\delta, s, \delta', s')$ retrieves an action from program δ or preprocesses program δ to retrieve an action in a later step. The remaining program is unified with δ'. $\phi[s]$ denotes the formula ϕ in situation s. The above-mentioned axioms read as:

1. An empty program cannot evolve.
2. If the program consists of a single action only, this actions is executed and the remaining program is empty.
3. Checks if formula ϕ holds in situation s. If yes the remaining program is empty.
4. If δ_1 is not finished, it is evolved and the resulting program is prepended to δ_2. Otherwise δ_2 is evolved.
5. Evolves either δ_1 or δ_2.
6. Replaces all occurrences of variable v in program δ by some instantiation x (δ_x^v) and evolves it.
7. Evolves δ and prepends the remaining program to the δ^*.
8. If formula ϕ holds, δ_1 is evolved, otherwise δ_2.
9. If formula ϕ holds, δ is evolved.
10. Either δ_1 or δ_2 is evolved. In contrast to non-deterministic branches the resulting program of the evolved section is again concurrently executed to the other program section.
11. In contrast to concurrent execution, δ_2 is evolved only if δ.
12. Evolves one step in $\delta^{\|}$ and starts concurrently executing the remaining program together with the whole loop. This can lead to an unbounded number of parallel loop executions.

The predicate *Final* checks if the current program can terminate in situation s. It is defined by the axioms $\mathcal{F}$:

1. Empty program:
   $Final(nil, s) \equiv True$
2. Primitive action:
   $Final(a, s) \equiv False$

3. Test/wait action:
   $Final(\phi =, s \equiv False$
4. Sequence:
   $Final(\delta_1; \delta_2, s) \equiv Final(\delta_1, s) \wedge Final(\delta_2, s)$
5. Non-deterministic branch:
   $Final(\delta_1 | \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$
6. Non-deterministic choice of argument:
   $Final(\pi v.\delta, s) \equiv \exists x.Final(\delta_x^v.s)$
7. Non-deterministic iteration:
   $Final(\delta^*, s) \equiv True$
8. Synchronized conditional:
   $Final(\textbf{if } \phi \textbf{ then } \delta_1 \textbf{ else } \delta_2 \textbf{ endif }, s) \equiv \phi[s] \wedge Final(\delta_1, s) \vee \neg\phi[s] Final(\delta_2, s)$
9. Synchronized loop:
   $Final(\textbf{while } \phi \textbf{ do } \delta \textbf{ endWhile}, s) \equiv \neg\phi[s] \vee Final(\delta, s)$
10. Concurrent execution:
    $Final(\delta_1 \| \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$
11. Prioritized concurrency:
    $Final(\delta_1 \| \delta_2, s) \equiv Final(\delta_1, s) \vee Final(\delta_2, s)$
12. Concurrent iteration:
    $Final(\delta^{\|}, s) \equiv True$

The axioms indicated read as:

1. The empty program is a final configuration
2. A primitive action is not a final configuration
3. A test/wait action is not a final configuration
4. A final configuration is achieved if both δ_1 and δ_2 are final configurations.
5. It is a final configuration if any branch is a final configuration.
6. It is a final configuration if the replacement of all variables v with some value x is a final configuration.
7. Non-deterministic iteration is a final configuration.
8. If ϕ holds in s, δ_1 has to be final. If ϕ does not hold, δ_2 has to be final.
9. If ϕ does not hold in s, δ has to be final.
10. Both δ_1 and δ_2 have to be final configurations.
11. Both δ_1 and δ_2 have to be final configurations.
12. Concurrent iteration is a final configuration.

Besides the axioms for *Trans* and *Final* IndiGolog introduces axioms for sensing. This is a special fluent $SF(a,s)$ in combination with axioms that describe the correlation of sensing consequences in a situation s by action a. For instance

$$SF(senseDoor(d),s) \equiv Open(d,s)$$

Actions without appropriate use of sensing information use $SF(a,s) \equiv True$.

Equipped with axioms for *Trans* and *Final* as well as for sensing we are able to extend the Basic Action Theory $\mathcal{D}$. We introduce an extended Basic Action Theory $\mathcal{D}^* = \mathcal{D} \cup \mathcal{C} \cup \{Sensed\}$. $\{Sensed\}$ represents all axioms necessary for sensing. $\mathcal{C}$ is made up of axioms for reifying programs as terms, as well as sets $\mathcal{T}$ and $\mathcal{F}$ with axioms for *Trans* and *Final*.

The notion of history is special in IndiGolog. Instead of plain actions, it uses pairs (a,μ) of an action a together with its sensing result μ. Combining the actions $a_1,...,a_n$ with sensing results $\mu_1,...,\mu_n$ leads to history $\sigma = (a_1,\mu_1)\cdot...\cdot(a_n,\mu_n)$.

2.2.2. IndiGolog Components

The implementation of IndiGolog is based on LeGolog (Levesque and Pagnucco, 2000) and provides a framework for acting on real robots. That was tested with a variety of robots such as Lego Mindstorm (Lego, 2015) or ER1 Evolution (2015) from Evolution Robotics, which was acquired by iRobot in 2012 (iRobot, 2015). IndiGolog is written in Prolog. The compatibility with SWI Prolog (Wielemaker et al., 2012; SWI-Prolog, 2015) allows simple interfacing to Java or C. The framework consists of the parts top-level main cycle, the language semantics, the temporal projector, the environment manager, the set of device managers, and the domain application, where only the device managers and the domain application are domain specific. The components are shown in Figure 2.1.

The top-level main cycle implements all the presented semantics. Furthermore, it incorporates exogenous events and sensing actions. The main cycle is implemented in prolog with the predicate $indigo(E,H)$ with a program E and a history H. This predicate uses the predicate $Trans/4$ and $Final/2$ (as explained above) in order to realize the execution.

```
indigo(E,H) :- handle_exogenous(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_rolling(H,H2), !, indigo(E,H2).
indigo(E,H) :- catch(final(E,H), exog, indigo(E,H)).
indigo(E,H) :- catch(trans(e,H,E1,H1), exog, indigo(E,H)),
  (var(H1) -> true ;
   H1 = H -> indigo(E1,H);
   H1 = [A|H] -> exec(A,S), handle_sensing(H,A,S,H2), indigo(E1,H2)).
```

First, the main cycle applies all exogenous events to the history through the predicate *handle_exogenous*. If the predicate succeeds, the cut operator (!) inhibits backtracking, as exogenous events cannot be undone. Then the *indigo* is called recursively. If *handle_exogenous* fails, because there are no exogenous events, the second clause checks if the current history should be rolled forward. As evaluation on formulas depends on regression, the runtime of regression depends directly on the length of the history. Thus, if the history is too long, the initial situation S_0 is moved forward by progression (Reiter, 2001). Therefore, successor state axioms of actions that have already occurred are applied to

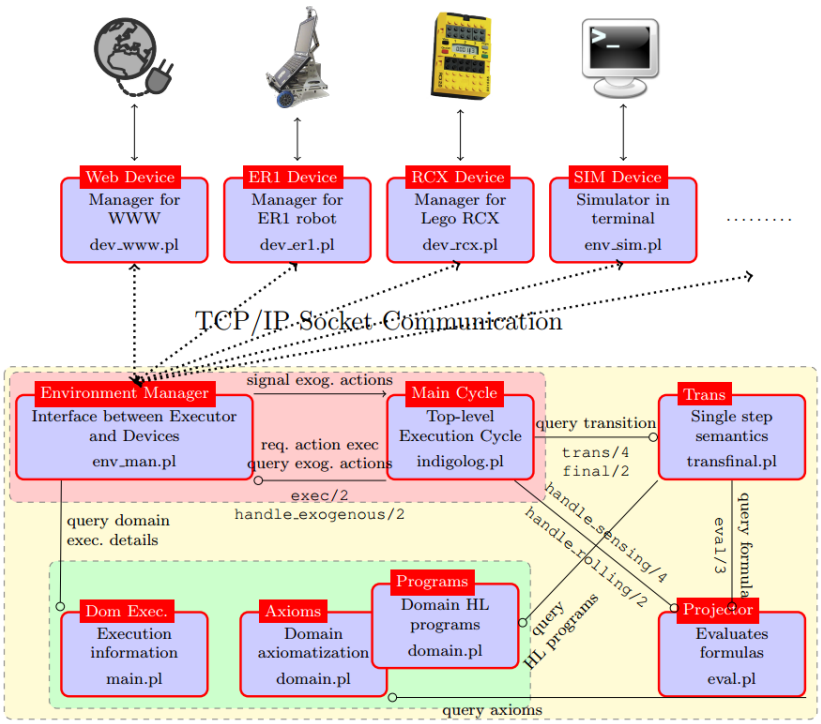

Figure 2.1.: The IndiGolog implementation architecture. Links with a circular ending represent goal posted to the circled module (de Giacomo et al., 2009)

the initial situation. This is carried out by the temporal projector, that is explained afterwards. Again if rolling forward succeeds, this is followed by a cut and a recursive call to *indigo*. Otherwise the third clause of *indigo* becomes active. Third, the current program is tested for termination. The *catch* predicate is responsible for exception handling. It the evaluation of *Final* overlaps with the arrival of an exogenous event, an exception is thrown that interrupts the evaluation of *Final*. The second argument unifies the name of the exception (in our case *exog*) and calls the recovery argument at the third position. The fourth clause does the actual execution by applying the *Trans* predicate. This call to *Trans* is again encapsulated in a *catch* clause. If the evaluation of *Trans* succeeds, the successor history is checked for unification. This means that if $H1$ is still a free variable, there was a problem in *Trans* and further evaluation of the *indigo* clause is canceled by evaluating true. Otherwise if the history after the *Trans* call is the same as before the call, no new action is necessary in real world and therefore *indigo* is called recursively. If the history was extended in *Trans*, the new action A is split from $H1$ and executed in real world by the predicate *exec*. Sensing results for the executed action are incorporated into the current history by the predicate *handle_sensing*. Finally *indigo* is called recursively to continue execution.

The temporal projector manages the robot's belief about the world and evaluates formulas. As the interface is quite abstract, it makes it possible to implement a broad range of formalisms (besides the basic action theory). The interface contains the predicates $eval/3$, *handle_sensing* and *handle_rolling*. As mentioned above, *handle_rolling* is responsible for keeping the history length at a suitable size. This is done by rolling up the history by progression if it is exceeds a specific length. Imagine a light bulb that can be switched by the action *flip* with $\neg on$ in S_0 and the situation $do([flip,flip],S_0)$. If this

history is rolled by one action, $\neg on$ switches to *on* in the new S_0' and the new action history is now $([flip], S_0')$. Note that the state (all fluent values) is the same in both situations.

handle_sensing ensures that the sensing information is processed. The simplest way to achieve this, it to add the sensing information to the history. IndiGolog's standard implementation adds the sensing information to the current history and carries out conditioned successor state axioms to do regression. In order to provide support to incomplete knowledge, the standard temporal projector implementation keeps the value for every fluent in its initial situation database. If the value is not known, more than one entry per fluent can exist (see Sardiña et al. (2004)). For instance if the value of a relational fluent is unknown, two entries with fluent values true and false are saved. Conditional successor state axioms for sensing actions allow to reject one special fluent value entry or reduce all possible values to only one remaining value. They have the form:

$$settles(a, r, f, v, w)$$
$$rejects(a, r, f, v, w)$$

settles takes an action a, with sensing result r, and sets fluent f to value v if w is known to be true. Similarly *rejects* takes an action a, with sensing result r, and invalidates value v for fluent f if w is known to be true. Successor state axioms for actions are modeled in a straight forward manner:

$$causes(a, f, v, w)$$

causes states that fluent f takes value v after executing action a if w is possibly true. Finally, $eval(F,H,V)$ evaluates if a fluent F is known to be V at history H. This predicate is relevant, as it provides the evaluation functionality for the predicates *Trans* and *Final*.

The environment manager in conjunction with the device managers is responsible for communication with real robots, software agents, the web, etc. As illustrated in Figure 2.1, the environment manager abstracts the device manager. Each device manager is responsible for a dedicated hardware participant. It handles the specific device communication and implements a high-level interface, so that the environment manager can invoke it. The device manager can also handle the communication to a graphical interface as shown in Sardiña and Vassos (2005). In order to restrict the use of the framework to a minimum, the communication between the environment manager and the device managers uses TCP/IP, to allow any TCP/IP-supporting technology to be used as device manager. The environment manager on the other side of the communication is written in prolog. It is connected to the main cycle, but acts in an independent thread to reduce communication problems due to computations within the main cycle. Generally, the environment manager transfers execution commands to the device managers and sends back sensing information as well as exogenous events to the main cycle. The domain application models the domain-specific part of an application. First, it provides an axiomatization of the dynamics of the world. This includes all actions with their preconditions and successor state axioms and the initial situation. It contains the high-level program and its subroutines that determine the robot's behavior. It also summarizes all necessary information to execute actions in real world. This includes mapping high-level symbols to the low-level execution codes.

2.3. History-Based Diagnoses

Especially in dynamic environments conventional diagnosis approaches that look at the question "what is wrong" have their limit. Due to limited sensor or actuator capabilities (see Goel et al. (2000); Verma

et al. (2004)) for approaches that focus on sensor and execution faults), or influences from (object) interaction, actions may fail though their components behind it are intact. So, a more interesting and helpful question instead is "what happened" according to McIlraith (1999). Based on this principle Iwan (2002) proposed the concept of history based diagnoses.
In classical diagnosis (Reiter, 1987) a predicate $AB(x)$ is used to denote an abnormal component x. If a component is marked abnormal the result cannot be trusted. This works fine for systems with a clear work flow, for instance when the result of one logical gate is fed into another logical gate. In robotic environments marking components turns out to be a somewhat more difficult. Imagine a robot taking a cup from a table, but it does not succeed. We can easily blame the gripper (including its control algorithm) by applying the predicate AB in order to provide a consistent world model. But what can be inferred from this world model. Is it valid to draw any conclusions? For example the robot might have dropped every other object from the table, while trying to get the cup. Or if the gripper is strong enough it might have moved/turned the robot by colliding with a fixed solid body and pushing itself away. Furthermore, from a cognitive perspective it remains unclear if the malfunction of the gripper has any influence on a solar eclipse (frame problem). History based diagnoses on the other hand aims to explain the consequences of malfunctions. From a theoretical point of view, it is important that by providing a consistent action sequence the situation calculus provides a solution to the frame problem. And in practical terms, concrete explanations allow much faster corrections (if intended).
History-based diagnosis (Iwan, 2002) tries to find a consistent variant of the inconsistent current action sequence. This is done by inserting actions (at any point within the history) and by replacing actions through variants of them. In this manner even long delays between action failures and potential indirect effects can be handled. Imagine a robot with the task to bring an object B from room $R1$ to room $R2$, with the initial situation that both the robot and the object B are in room $R1$. After an action sequence

$$\bar{\eta}^* = [pickup(B), goto(R2), putdown(B)]$$

the robot receives the feedback (an observation) that the object B is not in $R2$, though it was expected to be according to $\bar{\eta}^*$. Some explanations are:

```
(1) The robot lost its way and entered R3 instead of R2.
(2) The robot failed to grasp B during the pickup-action.
(3) The robot lost B on its way to R2.
(4) Somebody took away B after the robot had put it down in R2.
```

Here in case (1) and (2), a variation of the original action caused the failure (goto the wrong room and no-grasp instead of grasp). In case (3) and (4) an external source caused the error by snatch the object from the robot or taken the object after it was delivered. These four explanations can be transferred into diagnoses:

$$\bar{\delta}^{(1)} = [pickup(B), goto(R3), putdown(B)]$$
$$\bar{\delta}^{(2)} = [pickup', goto(R2), putdown']$$
$$\bar{\delta}^{(3)} = [pickup(B), snatch(B), goto(R2), putdown']$$
$$\bar{\delta}^{(4)} = [pickup(B), goto(R2), putdown(B), somebodytook(B)]$$

Note that a bar denotes a sequence of actions. The action $putdown'$ stands for a variant of the action $putdown(B)$ as the robot has nothing to put down in cases (2) and (3), because it was unable to take the object or it was snatched on the way. Within the situation calculus there is no distinction between actions and events, both are covered by the concept of actions. Though there may be more explanations for the missing object B in room $R2$ than the ones mentioned, not all action sequences explaining an observation are considered to be diagnoses. For example

$$\bar{\alpha} = [pickup(B), goto(R2), putdown(B), pickup(B), goto(R1), putdown(B)]$$

is not considered to be a diagnosis, because the robot actively moves object B to room $R1$, so it would be no surprise to the robot that B is not in $R2$. Deliberate actions that are under the control of the robot are no valid variations to a history. Therefore, we expect diagnoses to meet the following requirements:

- form a possible history (based on the domain description)
- explain the observation
- incorporate the current history (all actions/events or variants, e.g. $goto(R3)$ is a variation of $goto(R2)$)
- add additional events called insertions, e.g. $snatch(O)$ or $somebodytook(O)$

2.3.1. Formal Definition

We use the abbreviation $\bar{\alpha}.\alpha = [\alpha_1, \alpha_2, ..., \alpha_n, \alpha]$ for $\bar{\alpha} = [\alpha_1, \alpha_2, ..., \alpha_n]$. History based diagnoses extends the basic action theory by the predicates *Varia* and *Inser*. *Varia* represents an action variation axiom and *Inser* an insertion axiom. Syntactically they are similar to the predicate *Poss*.

$$Varia(a, A(x_1, x_2, ..., x_n), s) \equiv \Theta_A(a, x_1, x_2, ..., x_n, s)$$

states that under condition $\Theta_A(a, x_1, x_2, ..., x_n, s)$ action a is a variation for action $A(x_1, x_2, ..., x_n)$ in situation s. For example

$$Varia(a, putdown(x), s) \equiv a = putdownNothing$$

Analogous

$$Inser(a, s) \equiv \Theta(a, s)$$

defines that under condition $\Theta(a, s)$ action a is a valid insertion in situation s. For instance

$$Inser(a, s) \equiv \exists z.a = snatch(z) \wedge has\_object(z, s)$$

A history is an action sequence $\bar{\eta}$. An observation is a situation-suppressed formula ϕ. Diagnoses explain an observation that is contradicted by the original history, i.e. $\mathcal{D} \models \neg\phi[\bar{\eta}]$. For the example indicated above we have:

$$\bar{\eta}^* = [pickup(B), goto(R2), putdown(B)]$$
$$\phi^* = \neg at(B, R2)$$

$\mathcal{D} \models at(B, R2, do(\bar{\eta}^*, S_0))$, which is equal to $\mathcal{D} \models \neg\phi^*[\bar{\eta}^*]$ though $\mathcal{D} \models \phi^*[\bar{\eta}^*]$ is expected to hold according to $\bar{\eta}^*$. History based diagnoses build up on the concept of extended variations.

Definition 1. *An extended variation of a ground action sequence* $\bar{\alpha} = [\alpha_1, \alpha_2, ..., \alpha_n]$ *is a ground action sequence* $\bar{\delta} = [\delta_1, \delta_2, ..., \delta_m]$ *such that:*

- *a mapping* $\iota : \{1, ..., n\} \to \{1, ..., m\}$ *exists with* $\iota(1) < \iota(2) < \cdots < \iota(n)$
- *for each* $i \in \{\iota(1), \iota(2), ..., \iota(n)\}$ *with* $i = \iota(j)$*:*
  δ_i *is a valid variation of* α_j *in the situation after the action sequence* $[\delta_1, \delta_2, ..., \delta_{n-1}]$*,*
  i.e., $\mathcal{D} \vDash Varia(\delta_i, \alpha_j, do([\delta_1, \delta_2, ..., \delta_{i-1}], S_0))$
- *for each* $i \in \{1, 2, ..., m\} \setminus \{\iota(1), \iota(2), ..., \iota(n)\}$*:*
  δ_i *is a valid insertion in the situation after the action sequence* $[\delta_1, \delta_2, ..., \delta_{i-1}]$*,*
  i.e.,$\mathcal{D} \vDash Inser(\delta_i, do([\delta_1, \delta_2, ..., \delta_{i-1}], S_0))$

This can be grouped by the predicate $ExtVariation(\bar{\delta}, \bar{\alpha})$

$$
\begin{aligned}
ExtVariation([], []) &\doteq True \\
ExtVariation([], \bar{\alpha}.\alpha) &\doteq False \\
ExtVariation(\bar{\delta}.\delta, []) &\doteq Inser(\delta, (\bar{\delta}, S_0)) \wedge ExtVariation(\bar{\delta}, []) \\
ExtVariation(\bar{\delta}.\delta, \bar{\alpha}.\alpha) &\doteq [Varia(\delta, \alpha, do(\bar{\delta}, S_0)) \wedge ExtVariation(\bar{\delta}, \alpha)] \vee \\
&\quad\ [Inser(\delta, do(\bar{\delta}, S_0)) \wedge ExtVariation(\bar{\delta}, \bar{\alpha}.\alpha)]
\end{aligned}
$$

The first two equations define that an empty action sequence is only an extended variation of itself. The next equation determines that, if the original sequence is empty only insertions are allowed to form the extended variation. Finally, for a non-empty original action sequence, extended variations are defined recursively after applying *Varia* or *Inser*.

Definition 2. *An explanatory history-based diagnosis for an observation* ϕ *and a history* $\bar{\eta}$ *is an extended variation* $\bar{\delta}$ *of* $\bar{\eta}$ *such that* $\bar{\delta}$ *is executable and* ϕ *holds in the situation after the action sequence* $\bar{\delta}$*, i.e.,* $\mathcal{D} \vDash \phi[do(\bar{\delta}, S_0)]$.

$$ExplDiagnosis(\bar{\delta}, \phi, \bar{\eta}) \doteq ExtVariation(\bar{\delta}, \bar{\eta}) \wedge executable(\bar{\delta}) \wedge \phi[\bar{\delta}]$$

Chapter 3

The robot delivery domain

Throughout this thesis we use the running example of an indoor delivery robot as presented in the introduction. The robot is expected to pick up objects from a certain location, move the object from room to room to a goal destination and put the objects down. This happens by applying the actions *pickup*, *goto* and *putdown*. A map of our institute serves as layout for most of the experiments (see Figure 3.1). Every room is named by an identifier, such as $w1$. Connections from room to room are realized by an adjacent predicate (e.g. $adjacent(w2,w3)$). Equivalent to rooms, objects are listed by constants (e.g. $letter$).

The challenge of delivering an object is called a task. The more actions a robot has to execute, the more problems may occur. Therefore we increase the difficulty by always grouping three tasks into a so-called mission. This reduces the likelihood of a lucky strike. In addition to situation-independent properties, we use a set of fluents that is situation dependent as well. This set is formed by *has_object*, *is_at* and *at*. $has\_object(o)$ stands for the robot holding object o, $is\_at(o,r)$ for object o being in room r and $at(r)$ for the robot being in room r. For ease of readability we use the situation and situation-suppressed form of fluents synonymously; for example $at(r,s)$ is written $at(r)$. Unless stated otherwise, the variables o, o_1, o_2, ..., o_n denote objects. Similarly, r, r_1, r_2, ..., r_n are used for rooms. The action preconditions are defined as follows:

- $Poss(pickup(o),s) \equiv obj(o)\exists r. \wedge is\_at(o,r,s) \wedge at(r,s) \wedge \neg\exists o_1.obj(o_1) \wedge has\_object(o_1,s)$
- $Poss(putdown(o),s) \equiv obj(o) \wedge has\_object(o,s)$
- $Poss(goto(r),s) \equiv room(r) \wedge \exists r_1.at(r_1,s) \wedge adjacent(r,r_1) \wedge room(r_1)$

The action *pickup* can be executed, if there is an object o in the same room r as the robot and the robot is not holding an object yet. The execution of *putdown* is less restrictive. It is sufficient that the robot holds the object to be set down. The precondition of the action *goto* ensures the target room r is adjacent to the robot's current location r_1.

According to history based diagnoses we have a set of action variations and insertions. For simplicity we often call the combined set of variations and insertions changes. Possible action variations are *pickupNothing*, *putdownNothing* and $pickupWrongObject(o)$, whereas *snatch* and *exogMoveObject* are insertions:

- $Varia(pickupNothing, pickup(o), \top)$

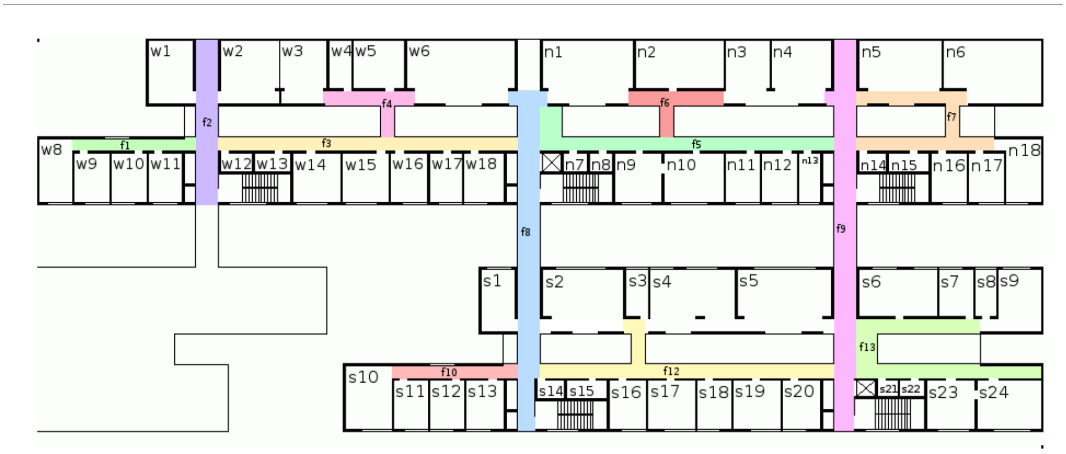

Figure 3.1.: Floor layout of the Institute of Software Technology and its neighboring departments, Graz University of Technology.

- $Varia(putdownNothing, putdown(o), \top)$
- $Varia(pickupWrongObject(o), pickup(o_1), o \neq o_1 \wedge at(r) \wedge is\_at(o,r))$
- $Inser(snatch(o), has\_object(o))$
- $Inser(exogMoveObject(o,r), \neg has\_object(o))$

pickupNothing describes if the robot fails to pickup the intended object. This may occur due to an inaccurate action execution. There is no condition that restricts the occurrence of this variation. Thus, whenever the robot executes action *pickup*, it may fail and end up executing *pickupNothing*. The action variation *putdownNothing* is unrestricted too. Similar to the pickup case it describes a failed putdown action. *pickupWrongObject* is somewhat more complicated. It describes the robot picking up a wrong object. Therefore the wrong object o has to be different than the intended object o_1 and it has to be in the same room r as the robot. The insertion *snatch* indicates that someone steals an object from the robot. It may happen every time the robot is holding an object o. Another insertion is *exogMoveObject*, denoting that an object o is moved to a room r without the knowledge of the robot. This can happen by another robot or some person. The condition that the robot does not hold the object can be applied very often. The more objects and rooms the domain contains, the more instantiations are possible. Thus, this very general insertion is difficult to handle.

The successor state axioms of the fluents described above are as follows:

- $has\_object(o, do(a,s)) \equiv a = pickup(o) \vee a = pickupWrongObject(o) \vee has\_object(o,s) \wedge a \neq putdown(o) \wedge a \neq snatch(o)$
- $is\_at(o,r,do(a,s)) \equiv a = exogMoveObject(o,r) \vee has\_object(o,s) \wedge a = goto(r) \vee is\_at(o,r,s) \wedge \neg\exists r_1 . r_1 \neq r \wedge (a = goto(r_1) \vee a = exogMoveObject(o,r_1))$
- $at(r, do(a,s)) \equiv a = goto(r) \vee at(r,s) \wedge \neg\exists r_1 . r_1 \neq r \wedge a = goto(r_1)$

$has\_object(o)$ holds in situation $do(a,s)$, if action a was the action $pickup(o)$ or $pickupWrongObject(o)$. In this case it does not matter if the robot intended to pickup object o or not, the consequence is the same. If *has_object* held in the previous situation, it holds also in the given situation, if action a is neither $putdown(o)$ nor $snatch(o)$. This means holding the object is only interrupted if the robots actively

puts down the object, or the object is taken from the robot. $is\_at(o,r)$ holds in situation $do(a,s)$, if the object o was brought to room r by the action $exogMoveObject(o,r)$ or if the robot held the object and moved to room r in the action $goto(r)$. Furthermore, $is\_at(o,r)$ holds if it held in the previous situation and the object was not moved to another room r_1 by $goto(r_1)$ or $exogMoveObject(o,r_1)$. While *is_at* is used for objects, *at* describes the location of the robot itself. $at(r)$ holds if the robot moves to the room r by action $goto(r)$ or if the robot was in the room and did not leave it with action $goto(r_1)$ for another room r_1. Please note that successor state axioms contain insertions and action variations too.

Chapter 4

Belief Management

The belief management system is responsible for providing the high level control with a consistent knowledge base that serves for decision-making. Such a knowledge base is typically a logical one. In our case we have the set of relational and functional fluents at a specific situation. The management aspect comes into play when there are inconsistencies within this belief. Then, measurements are needed to detect inconsistencies and repair the belief again. An inconsistency arises if a fluent leads to contradicting values such as $is\_at(x, r_1)$ and $\neg is\_at(x, r_1)$ or additional general rules are violated. Such a general rule might state that an object cannot be in two rooms at the same time. Therefore, $is\_at(x, r_1) \wedge is\_at(x, r_2) \wedge r_1 \neq r_2$, represents an inconsistency, where x is an object and r_1 and r_2 are rooms. Inconsistencies are typically handled by a diagnosis and a repair step. Besides the execution of these two steps, we present a formalism to administer results of it. The remainder of this chapter is structured as follows. First, we give an overview of related research within this research area in Section 4.1. In Section 4.2, we define inconsistencies formally, derive the belief management system and present the integration of this belief management system into the IndiGolog framework. This chapter is based on work published in Gspandl, Pill, Reip, and Steinbauer (2011) and Gspandl, Pill, Reip, Steinbauer, and Ferrein (2011).

4.1. Related Research

Within the related research, we investigate in two directions. On the one hand, we analyze strategies that concentrate on diagnosis or justifications as Horridge et al. (2009) call them. Whereas Lamperti and Zanella (2003) define a diagnosis to be a set of behavioral modes assigned to components that explain the observation, we adhere more to the looser definition of Pencolé and Cordier (2005) where a diagnosis is defined to be the set of paths of the system explaining the observed behavior. On the other hand, approaches that handle the belief as a whole are surveyed.

4.1.1. Action Sequence Based Diagnosis

McIlraith (McIlraith, 1999) was one of the first who turned her diagnosis interest towards "what happened" as described above. The notion of explanatory diagnosis describes action sequences

that result in a behavior that was not expected but observed. A system is described as a quadruple $(\Sigma, HIST, COMPS, OBS)$, where Σ roughly corresponds to the basis action theory $\mathcal{D}$. *HIST* stands for the history of performed ground actions $[a_1, a_2, ..., a_k]$ that were performed in S_0. *COMPS* enumerates the components in use, as a finite set of constants. Finally, OBS_F is formula uniform in s. The task of interest is to find a sequence of actions $E = [\alpha_1, \alpha_2, ..., \alpha_n]$ that is added to the situation $do(HIST, S_0)$ to satisfy an observation.

$$\Sigma \models Poss([HIST.E], S_0) \wedge OBS_F(do([HIST.E], S_0))$$

Poss is used as an abbreviation that corresponds to the definition of *executable* indicated above. Note that actions are only added to the end of the current situation $do(HIST, S_0)$.
According to Occam's Razor, shorter (simpler) explanations are preferred over longer ones. This is clamped by the definition of *simpler* that uses the functions *ACTS* returning the set of actions in use and *LEN* providing the length of the given action sequence. For given action sequences $HIST_A(a_1, a_2, ..., a_n)$ and $HIST_B(b_1, b_2, ..., b_m)$, situation $s_A = do(HIST_A, S_0)$ is simpler than situation $s_B = (HIST_B, S_0)$ if and only if $ACTS(HIST_A) \subseteq ACTS(HIST_B)$ and $LEN(HIST_A) < LEN(HIST_B)$. Based on the definition of simple, chronologically simple explanatory diagnosis are defined. The definition ensures that two different chronologically simple explanatory diagnoses result in different solutions to a given problem. This is achieved as it is not valid to extend a valid diagnosis by unnecessary actions. For a system $(\Sigma, HIST, COMPS, OBS)$ a situation $s = do([HIST.E], S_0)$ and a situation $s' = do([HIST], E')$, E is a chronologically simple explanatory diagnosis E, if and only if E is an explanatory diagnosis and there is no explanatory diagnosis E' with s' simpler than s. Based on the definition of chronologically simple explanatory diagnosis, the diagnosis problem can be cast into a planning problem with the goal to find an action sequence fulfilling $\exists s.OBS_F(s)$ starting from situation $do(HIST, S_0)$. The verification of a diagnosis is simply done by regression and theorem proving in the initial situation.

The idea of formulating the diagnosis problem as planning problem was extended by Sohrabi et al. (2010). The approach extends and generalizes the presented one. While McIlraith (1999) requires observations to hold in the final situation, this limitation is relaxed by defining intervals of situations $OBS[S_0, s]$ denoting the interval from situation S_0 to situation s. Furthermore, commonsense and expert knowledge is incorporated into the belief. In contrast to McIlraith (1999) a system description *DS* consists of $(\Sigma, OBS[S_0, s])$ only, where Σ refers to the basic action theory. Action terms $\mathcal{A}$ within Σ are parted into the sets $\mathcal{A}_{normal}$ and $\mathcal{A}_{faulty}$ to distinguish between normal and faulty system actions. The sequence of recognized actions *HIST* is ignored completely. Incomplete initial knowledge is handled by a formula $H(S_0)$ that combines assumptions on the initial situation. These assumptions are expressed by sentences of the form $[\neg]F(\bar{c}, S_0)$ for a fluent F and a set of constants $\bar{c}$. A diagnosis for a system $(\Sigma, OBS[S_0, s])$ is a tuple $(H(S_0), \bar{\alpha})$ with $\bar{\alpha} = [\alpha_1, \alpha_2, ..., \alpha_k]$ such that $\Sigma \cup H(S_0) \models \exists s.s = do(\bar{\alpha}, S_0) \wedge executable(s) \wedge OBS[S_0, s]$. Additionally, $\Sigma \cup H(S_0)$ is required to be consistent. Based on a reflexive, transitive and preference relation $\preceq$ a diagnosis D is preferred if there is no other diagnosis D' with $D' \prec D$. Two classes for preference relations are presented. The first ones are domain-independent relations such as minimal fault diagnosis. This means the smaller the set $\mathcal{A}_{faulty}$ the better it is. The next criterion is shorter diagnoses (less actions in $\bar{\alpha}$) or simpler diagnoses (less different actions in $\bar{\alpha}$). The other class of preference relations are domain dependent relations. They can either be addressed by probability distributions that are hardly available or by domain knowledge. Here, domain knowledge is encoded in the qualitative preference language $\mathcal{LPP}$ (Bienvenu et al., 2006). $\mathcal{LPP}$ contains linear temporal language connectives to model temporal dependencies between actions and fluents. After encoding preferences in $\mathcal{LPP}$ sentences, these sentences are put into a

sequence to incorporate the needs of a preference relation. The diagnosis task is realized by planning. Therefor the system description is transferred into a PDDL domain description according to Röger et al. (2008). The incomplete initial situation is handled by assigning specific values to unknown fluents. This leads to multiple initial situations with all fluent value combinations. Planning itself is done by state-of-the-art planners. Besides the fact that scaling the initial situations is a serious problem, ignoring the actively executed history brings drawbacks. For instance actions that do not result in observations are ignored. In case such an action is necessary for executing another action, the ignored action has to be executed again, as the robot has no knowledge about its execution.

Of course there are also other techniques to approach the diagnosis problem. Grastien in (Grastien et al., 2007) shows the transformation into a propositional satisfiability problem. As they investigate discrete event systems such as their running example, which is a computer system consisting of 20 identical components, the notion of "actions" is replaced by events. An event is either observable (denoted by the set Σ_o) or unobservable (summarized by the set Σ_u). A system *SD* further consists of A, δ and s_0, where A is a finite set of state variables, s_0 is the initial state and $\delta \subseteq \Sigma_o \cup \Sigma_u \rightarrow 2^{\mathcal{L} \times 2^L}$ assigns each event a set of pairs $\langle \phi, c \rangle$. A state is a function that takes a state variable and outputs the constants 1 and 0 ($s : A \rightarrow 0, 1$). L is the set of all literals, where a literal is a state variable or its negation. $\mathcal{L}$ indicates the language over A using standard logical connectives. The pair $\langle \phi, c \rangle \in \delta(e)$ of an event instance e, denotes that if and only if ϕ holds in s, c becomes true. This idea is formalized by specific successor state axioms. By examining the event instances, the problem is relaxed to a partial order problem. If event instance e_1 does not interfere with event instance e_2 their mutual event sequence is irrelevant. Queries to the SAT solver contain the system description, the observations and a formula encoding the fault level i $\Phi_{\Delta_i} = \Phi_{SD} \wedge \Phi_{OBS} \wedge \Phi_{FL_i}$, where i defines the maximum number of allowed faults. If i equals 0 the system worked correctly. Starting from 0, i is typically increased to find diagnoses with as few faults as possible. As soon as the formula is satisfiable, the algorithm is stopped due to the fact that a valid solution is found. A solution is a set of events with fault level i that is consistent with the observations. The upper bound number of events can be derived from the number of observations (p) and the maximum number of unobservable events between two observations (x). The upper bound is (x+1)p. This number can be lowered by unobservable events that do not interfere as they can be treated together.

4.1.2. Belief Revision

Scherl and Levesque (2003) use an explicit notion of knowledge. This is achieved by discriminating between normal actions and knowledge-producing actions and fluents. While normal actions do not influence knowledge fluents, knowledge-producing actions have no influence on normal fluents. If some fluents are unknown or can take multiple possible values, they are realized by enumerating all those values in multiple situations. For instance if the status of a relational fluent $at(shop)$ is unclear, 2 situations are used. The first situation evaluates to $at(shop)$, the second one to $\neg at(shop)$. A relational fluent F is known to be true, if and only if it is true in all situations. The same holds for the negative case and equivalently for functional fluents. A binary accessibility relation establishes the connection between situations, stating that if the robot is in situation s, it might be in a situation s'. In this manner knowledge can be treated as a fluent. The special fluent $K(s', s)$ denotes that situation s' is accessible from situation s. Such as normal fluents, K takes a situation as last argument. The first argument can be seen as an auxiliary variable. Based on the K fluent they introduce the abbreviation $Knows(P, s)$

indicating that P is known to be true in situation s. E.g.

$$Knows(broken(y),s) \doteq \forall s'.K(s',s) \supset broken(y,s')$$

This approach can also be generalized for the equality predicate and for formulas. For instance

$$\begin{aligned} Knows(number(Bill) = number(Mary),s) \doteq \\ \forall s'.K(s',s) \supset number(Bill,s') = number(Mary),s') \\ \exists x Knows(\exists y[next\_to(x,y) \wedge \neg broken(y)],s) \doteq \\ \exists x.\forall s'.K(s',s) \supset \exists y[next\_to(x,y,s') \wedge \neg broken(y,s')] \end{aligned}$$

Two knowledge-producing actions are introduced. First, there are actions that make the truth value of a formula known. For instance, $sense_P$ makes the truth value of P known. *Kwhether* abbreviates if the value of a fluent P is known.

$$Kwhether(P,s) \doteq Knows(P,s) \vee Knows(\neg P,s)$$

Second, there are actions that make the value of a term known. For instance $read_\tau$ for a term τ makes the value of τ known. Similar to *Kwhether* there is an abbreviation $Kref(\tau,s)$.

$$Kref(\tau,s) \doteq \exists x.Knows(\tau = x,s)$$

where x does not appear in τ. These knowledge-producing actions affect the accessibility relation K from above. Imaging an initial situation with 2 fluents P and Q and three situations s_1, s_2 and s_3 as shown in Figure 4.1. The fluent P is true in all situations, so $Knows(P)$ holds. The fluent Q is different in situation s_2 to the situations s_1 and s_3, but they are assumed to be K-accessible. After executing the action $sense_Q$ we remove the situation s_2 from the K-accessible set, because its value of Q contradicts the sensing result. In this manner $Knows(Q,do(sense_q,S_0))$ holds after executing $sense_Q$ on the remaining K-accessible relation set (s_1,s_3). For normal actions, the K-relation stays the same as long as the action can be executed (*Poss*) in all K-related situations. Based on these ideas of knowledge-producing actions and normal actions, the successor state axiom of K is defined as follows:

$$K(s'',do(a,s)) \equiv \exists s'.s'' = do(a,s') \wedge K(s,s') \wedge Poss(a,s') \wedge SR(a,s) = SR(a,s')$$

The function *SR* subsumes all sensing functions for both action types, though they are insignificant for normal actions. If and only if there is a situation s' that is K-related to s and a predecessor of s'', then s'' is K-related if and only if the action a is executable in s' and the sensing result of a is the same in s and s'. By separating knowledge-producing actions from normal actions, plus the fluent K together with its successor state axiom, knowledge can be seen as a subpart of an adapted Basic Action Theory.

Grosskreutz and Lakemeyer (2001) tackle the problem of knowledge representation and its update in a probabilistic setting. Furthermore they give insight into the pGolog framework, that is an extension to the high-level programming language Golog. The pGolog framework follows a classical three tier architecture where the communication between low-level processes and high-level controllers is carried out with a set of defined registers. If a high-level controller writes a value into a register, an execution is triggered. Otherwise the robot receives some sensing information. During online execution time is modeled discretely by exogenous time update events. This time representation is essential

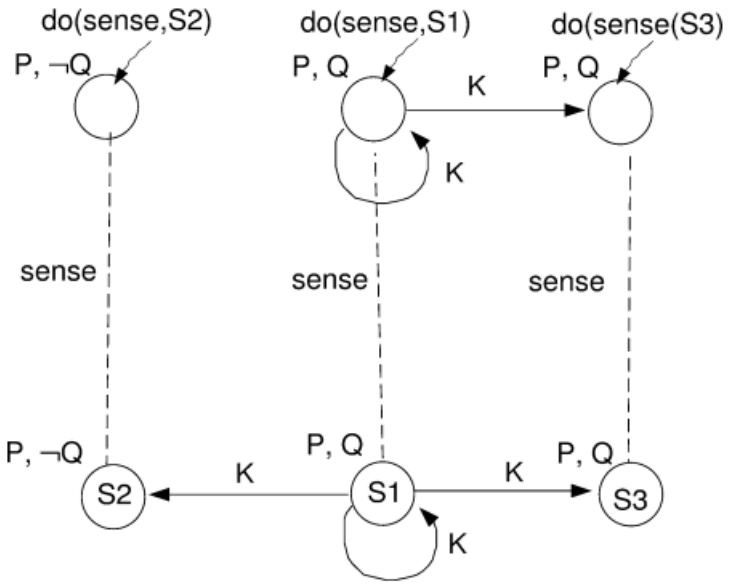

Figure 4.1.: Knowledge-Producing Actions and the K relation (Scherl and Levesque, 2003)

to model the behavior of low-level processes. The special fluent $ll(s,s')$ characterizes low-level processes in situation s if the robot thinks it is in situation s'. Their programs, so called belief-based programs, condition the execution by epistemic tests on the robot's belief. Inspired by Bacchus et al. (1999), the execution is realized by a $transPr(\sigma,s,\delta,s')$ function that returns the probability to reach situation s' with the remaining program δ starting from situation s with program σ. The probability calculation is done by special function p. The tricky part is in formalizing the successor state axioms for the fluents p and ll. Therefore a helper function *adv&filter* is used. In case of a normal action, *adv&filter* returns the probability of the last situation times the transition probability. Otherwise, in case of a sensing action, all inconsistent situations are removed. The successor state axioms for p and ll read as follows:

$$\begin{aligned} p(s^*,do(a,s)) = p \equiv \exists ll^*.adv\&filter(s^*,ll^*,do(a,s)) = p \wedge p > 0 \vee \\ \forall ll^*.adv\&filter(s^*,ll^*,do(as)) = 0 \wedge p = 0 \\ ll(s^*,do(a,s)) = ll^* \equiv adv\&filter(s^*,ll^*,do(a,s)) > 0 \vee \\ \forall ll'.adv\&filter(s^*,ll',do(a,s)) = 0 \wedge ll^* = nil \end{aligned}$$

The first line of p tackles the case of an existing successor s^* of s indicated by a positive probability. If no such successor exists, the probability is zero (second line). Similarly the successor state axiom of ll leads to a result if there is a legal transformation (first line). Otherwise the successor is nil (second line).

The approach of Scherl and Levesque (2003) is limited in the sense that it allows for belief update only. This means the belief of a robot is updated or extended by the actions it executes. If one situation within the K-relation contradicts some sensing information it is abandoned. But there is no possibility to revise the whole belief if it was wrong. If the world has not changed, but the robot discovered new facts that contradict the actual belief, the approach fails by believing everything. This limitation was tackled by Shapiro et al. (2000). They use the K-relation $K(s',s)$ of Scherl and Levesque (2003)

defining situations s' that are possible in situation s but call it B-relation. Furthermore, they introduce the concept of plausibility levels. Every situation receives a plausibility (*pl*) enabling a ranking between different situations. A situation is said to be more plausible if the plausibility value is smaller. Plausibilities are specified on initial situations and do not change during execution, as denoted by the successor state axiom:

$$pl(do(a,s)) = pl(s)$$

In contrast to Scherl and Levesque (2003), the belief is not defined by all situations of the K-relation, but only by those situations with the lowest plausibility. This is expressed by the belief operator *Bel* for a formula ϕ:

$$Bel(\phi,s) \doteq \forall s'.[B(s',s) \wedge (\forall s''.B(s'',s) \supset pl(s') \leq pl(s''))] \supset \phi[s']$$

If the current belief contradicts a sensing information, the situation is dropped and the situation with the lowest plausibility forms the new belief. This is shown by the following example in Figure 4.2.

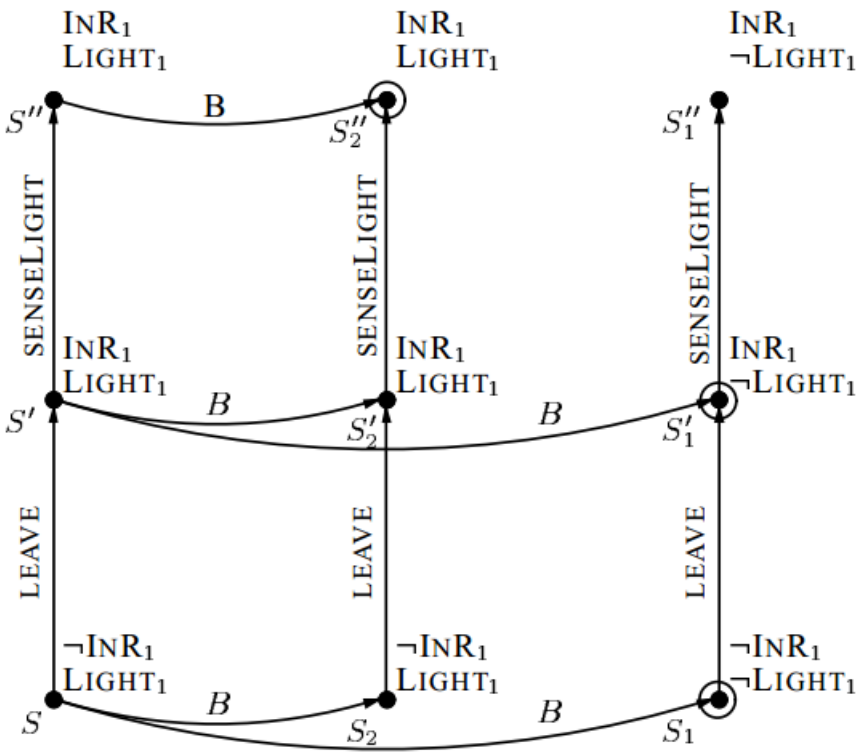

Figure 4.2.: An example for belief update and belief revision (Shapiro et al., 2000).

There are three initial situations S, S_1, S_2 with S_1 and S_2 B-related to S. There are two rooms, namely R_1 and R_2, with a light in each room that can be on or off. With the action *leave*, the robot moves from one room to another. Using the sensing action *senseLight*, it is possible to check if the light in the robot's current room is on. In all initial situations, the robot thinks it is not in room R_1, but has different assumptions about the light in room R_1. The plausibility of situation S_1 is 0 and $pl(S_2) = 1$. Thus, the belief of the robot derives from situation S_1. This is indicated by the circle around the

situation. First the robot leaves room R_2, entering R_1. Though there is an update on the current room of the robot, its belief is still determined by situation S_1. After sensing that the light is on in room R_1, the B-relation between S_1'' and S'' does not hold any more. S_2'' is still in a B-relation to S'', so it becomes the new favorite and revises S_1''. By applying belief update and belief revision in arbitrary order, the approach is able to handle iterated belief change.

Though Shapiro et al. (2000) is able to handle iterated belief change, the number of belief revisions is limited to the number of initial situations. If a situation turns inconsistent with some sensing information, it is dropped from the B-relation and the next best situation is selected. For k initial situations, there are a maximum of k-1 belief revisions. It is expected that the agent has full knowledge about all actions happening in the world and that sensing actions are always correct. Shapiro and Pagnucco (2004) still expect sensing actions to be correct, but they loosen the requirement of a completely observable world by incorporating exogenous actions. Whereas endogenous actions are executed by the robot and thus are observable by the robot itself, exogenous actions are executed by other robots or the environment. The robot can only infer these actions by observations. If the robot detects an inconsistency, the current situation is not necessarily dropped. It is extended by exogenous actions, if they allow a consistent interpretation of the last sensing result. This leads to the observations that situations with a B-relation contain the same endogenous actions, but may differ in their exogenous actions. According to Occam's razor simpler situations are preferred (situations with less exogenous actions). This is achieved by a function *height* that returns the length of an action sequence. As the number of actions for a sequence increase, the height increases as well an the situation becomes less preferable. Building on the plausibility function of Shapiro et al. (2000) a situation s' is preferred over a situation s ($s' \sqsubseteq s$):

$$s' \sqsubseteq s \doteq pl(s') < pl(s) \vee (pl(s') = pl(s) \wedge height(s') \leq height(s))$$

After having defined a preferable relation on situations, we further need a new successor state axiom for the B-relation that incorporates exogenous actions. Exogenous actions are denoted by predicate $Exo(a)$ that holds if a is an exogenous action. An action is either endogenous or exogenous:

$$Endo(a) \doteq \neg Exo(a)$$

A sequence of exogenous actions between a situation s and a situations s' is defined by:

$$ExoSeq(s,s') \doteq s \preceq s' \wedge \forall a, s_1 . s \prec do(a,s_1) \preceq s' \supset Exo(a)$$

The last endogenous action a between situation s and situation s' is defined by:

$$LastEndo(a,s,s') \doteq Endo(a) \wedge ExoSeq(do(a,s),s')$$

Equipped with the presented predicates, it is possible to derive the new B-relation. The function *root* is a helper to derive the initial situation of a situation. The B-relation holds if both situations contain only exogenous actions, or the B-relation holds on the subsituation after the last endogenous action and the sensing result (SF) of the last endogenous action is the same.

$$\begin{aligned} B(s',s) \equiv [&(ExoSeq(root(s),s) \wedge ExoSeq(root(s'),s')) \vee \\ &(\exists s_1', s_1, a . LastEndo(a,s_1',s') \wedge LastEndo(a,s_1,s) \wedge (SF(a,s_1') \equiv SF(a,s_1)) \wedge B(s_1',s_1))] \end{aligned}$$

As plausibility values may be difficult to define, Demolombe and Pozos Parra (2005) presented a rather theoretical approach to belief revision without plausibility levels. First, they propose two

different successor state axioms for real situations and for imaginary ones. And second, if a sensing action, or also called knowledge-producing action, contradicts a fluent in the actual belief, this fluent is simply overwritten. Distinguishing between successor state axioms for real and for imaginary situations accommodates the partial observability of the world. Thus, actions can only be added to the robot's situation if it observes the action. This is denoted by the successor state axiom definition of the K-relation (equivalent to Shapiro et al. (2000) B-relation). The *i* stands for a robot:

$$K(i,s'',do(a,s)) \equiv \exists s'.K(i,s',s) \wedge ((observe(i,a,s) \wedge s'' = do(a,s')) \vee (\neg observe(i,a,s) \wedge s'' = s'))$$

If the robot *i* observes the action *a* in situation s, it forms the successor situation, otherwise the successor situation stays the same. Real situations are denoted by the predicate *real*. The successor state axiom for a fluent F in real situations is extended to:

$$real(s) \supset F(\vec{x},do(a,s)) \equiv \gamma_F^+(\vec{x},a,s) \vee F(\vec{x},s) \wedge \neg\gamma_F^-(\vec{x},a,s)$$

The "subjective" successor state axiom for a robot *i* and fluent *F* is:

$$\begin{aligned} real(s) \supset (K(i,s',s) \supset \\ (F(\vec{x},do(a,s')) \equiv \gamma_F^+(i,\vec{x},a,s') \vee (a = sense_F(i) \wedge F(\vec{x},s) \wedge \\ \neg(\gamma_F^-(i,\vec{x},a,s') \vee (a = sense_F(i) \wedge \neg F(\vec{x},s)))))) \end{aligned}$$

For a K-related situation s', the fluent F holds if (1) $\gamma_F^+(i,\vec{x},a,s')$ holds or if (2) F holds in the real situation and F is sensed by action *a* or if (3) the fluent F held in the predecessor situation and was neither negated by $\gamma_F^-(i,\vec{x},a,s')$ or (4) by its negation in the real situation and a corresponding sensing action *a*. Successor state axioms can be different for different robots too. E.g. a baby might have a different interpretation of actions in the world than its mother. This is achieved by the argument *i* in $\gamma_F^+(i,\vec{x},a,s')$ and $\gamma_F^-(i,\vec{x},a,s')$. As sensing actions directly overwrite the sensed fluent, it is sufficient to have only one possible (imaginary) situation to model the robots belief. In contrast to Shapiro et al. (2000), this has the advantage that no plausibility values have to be defined, but with the clear drawback, that there is no possibility to model different hypotheses with connected fluents. For example a robot might know that it is either in room R_1 or in room R_2. Furthermore, it knows that the light is on in room R_1 and off in room R_2. Initially it believes that it is in room one (light on) and it senses no light. According to Demolombe and Pozos Parra (2005) the revised belief would state that the robot is incorrect in room one and the light is off, whereas in Shapiro et al. (2000) the robot would think that it is correctly in room two and the light is off.

4.2. Formal Belief Management

According to the related research literature mentioned above we treat the robot's belief as the (truth) values of all of its fluents. Our belief is expressed by exactly one situation (initial situation plus action sequence), but we keep a pool of different situations (hypotheses) in order to cope with unforeseen events. Hypotheses are ranked according to a change value that resembles the amount and the severity of adaptions that were applied to every hypothesis. The hypothesis with the lowest change value defines the robot's belief. We do not use an explicit binary notation between different hypotheses, but provide a generation scheme together with a set of rules that unite all valid hypothesis. Belief update is realized as usual by applying successor state axioms to the current belief. As said before, we keep a pool of hypotheses, thus every hypothesis is updated (not only the belief-forming hypothesis). In

contrast to the presented approaches above, we are able to handle arbitrary numbers of iterated belief revisions. This is especially crucial in settings, where the robot is expected to do long shifts. Another feature is the ability to handle incorrect sensing. Incomplete initial knowledge can easily be handled by enumerating all hypotheses.

After giving an insight into the principles of our belief management, we will formalize the components. A pool of consistent hypotheses builds the basis of the system and gives the robot the possibility to recover from incorrect decisions. For instance if the robot chooses the most likely hypothesis, but realizes later on that the decision was incorrect, the second most likely hypothesis is still available and so on. The fittest hypothesis out of this pool forms the belief of the robot, where fitness is defined by a low change value. After every action execution, hypotheses are updated according to the observed actions. In case of an inconsistency, the affected hypothesis is replaced by a set of consistent substitutions. Those substituents are derived pursuant to history-based diagnosis. Thus, we either replace actions by one of its variations, or add extra actions to achieve consistency. Every adaption is penalized by adding a strict positive value to the change value of a hypothesis. The number of hypotheses within the pool is the result of the number of initial hypotheses, minus the number of inconsistent ones, plus the number of newly generated hypotheses. The implementation uses a fixed threshold to limit the total number of hypotheses.

The belief management algorithm is embedded in some execution system works as follows (Gspandl, Pill, Reip, and Steinbauer, 2011). In Line 2 we update all hypotheses h_i of our pool H through all the newly executed actions. This does not include the favorite hypothesis h_f, which is updated by the execution system itself. Then, in Line 4 we check the consistency of our favorite h_f. If it is not consistent, we generate new hypotheses from the inconsistent one in Line 5 and select a new favorite in Line 6.

Algorithm 1: *beliefManagement*

```
1 foreach h_i ∈ H ∖ h_f do
2 |   updateHypothesis(h_i)
3 end
4 if ¬Cons(h_f) then
5 |   generateHypotheses(h_f)
6 |   h_f ← selectFavorite(H)
7 end
```

A central element within the presented system is the notion of consistency. On the one hand an inconsistent situation triggers the diagnosis and repair process within the belief management, on the other hand consistency is used as acceptance criteria for newly generated hypotheses (situations). We have two criteria that define consistency. First, we call a situation inconsistent if a sensing action contradicts its expected value. This means starting in the initial situation the robot has some idea about itself and its environment. This idea is formed by all fluent values and is called belief. This belief is updated according to successor state axioms by every action the robot executes itself or the robot detects (exogenous actions). In case of a sensing action, the robot either has information or has no information about the sensed fluent. In case the robot knows the fluent and the value differs from the sensed value, we have an inconsistency. Second, we have a set of common sense rules that describe basic properties of the domain. For instance if the robot senses that there is some weight on its conveyor, $has\_object(o)$ must be correct for some object. This means, either the robot knows for

object *o*, it is carrying this object, or there is an object *o* and the robot has no knowledge if is carrying this object. We call those general rules invariants. Putting those two rules together we can formalize a predicate consistent.

Definition 3. *A history* σ *is consistent if and only if* $\mathcal{D}^* \models Cons(\sigma)$ *with* $Cons(\cdot)$ *inductively defined as:*

1. $Cons(\varepsilon) \doteq Invaria(S_0)$
2. $Cons(\bar{\delta}.\alpha) \doteq Cons(\bar{\delta}) \wedge Invaria(\bar{\delta}.\alpha) \wedge$
   $[SF(\alpha, \bar{\delta}) \wedge RealSense(\alpha, \bar{\delta}) \vee$
   $\neg SF(\alpha, \bar{\delta}) \wedge \neg RealSense(\alpha, \bar{\delta})]$

The definition of *Cons* is recursive. In the base case of an empty history, there is no action and thus no sensing action either. Therefore, we only have to check for invariants in the initial situation. In the inductive case, we call *Cons* recursively on the previous situation and check the actual situation. This includes the invariants check, plus the comparison between the modeled belief *SF* and the real world values *RealSense* we receive from our sensors. $RealSense(\alpha, \bar{\delta})$ returns true if the sensing action α for a fluent F returns the value $\top$. This is similar to the definition of the modeled belief $SF(\alpha, \bar{\delta})$ (de Giacomo et al., 2009) that leads to the value of a fluent F according to the history $\bar{\delta}$. Ferrein (2008) gives an overview of how to integrate real sensing values into IndiGolog.

Given the definition of a consistent history hypothesis, we take a step forward and provide a rating for hypotheses. As explained above, the belief of our system is based on the most reasonable hypothesis. Thus, we need a measurement function. Similar to the notion of consistency, this measurement function iterates inductively over the given hypothesis. As a sum of all changes, we call *cv* the sum of all variations and insertions. All remaining endogenous actions are not considered, as they do not differ between hypothesis and its origin in terms of history-based diagnosis. This is a simple consequence of history based diagnoses. The function *val* assigns costs to every insertion and variation. In this manner we can distinguish the severity of different changes. Instead of summed values, one can use probabilities too, but they may be difficult to determine, especially if some of them depend on each other. Thus, in order to prevent misleading assessments we use the notion of change values. For example a valid explanation based on a total break down of a robot is usually less probable than a delocalized robot and thus holds a higher change value. The change value is a relative measurement between a hypothesis and a situation (please note that both share the same syntax). If all hypotheses originate from the same situation (typically this is the robot's plain, perceived situation) it is possible to compare an arbitrary number of hypotheses.

Definition 4. *Let* $cv : situation \times situation \to \mathbb{R}^+$, $cv(\bar{\delta}, \bar{\alpha}) = v$ *with* $v \geq 0$ *evaluate the difference between a diagnosis* $\bar{\delta}$ *and a history* $\bar{\alpha}$*. cv is called the* change value *and is inductively defined as:*

1. $cv(\bar{\delta}, \bar{\delta}) = 0$
2. $cv(\bar{\delta}.\delta, \bar{\alpha}) = cv(\bar{\delta}, \bar{\alpha}) + val(Inser(\delta, \bar{\delta}))$
3. $cv(\bar{\delta}.\delta, \bar{\alpha}.\alpha) = cv(\bar{\delta}, \bar{\alpha}) + val(Varia(\delta, \alpha, \bar{\delta}))$
4. $cv(\bar{\delta}.\delta, \bar{\alpha}.\delta) = cv(\bar{\delta}, \bar{\alpha})$

Inser and Varia are as indicated in Chapter 2.3. We assume standard axiomatization of integers together with their standard operations.

In the first line, we assign the change value of zero, if the hypothesis is equal to the situation. In the second line, we handle the case if δ is an insertion. In this manner the value of the insertion is simply added to the value of the remaining sequence. Equivalently, variations are handled in the same way as insertions in the next line. If the current action δ does not differ between the two arguments, the change value is reduced to the remaining action sequence. For example the change value for the action sequence

$$[goto(warehouse), pickup(chocolate), exogMoveObject(candies, quality\_control), \\ goto(shop), putdownNothing]$$

and the original situation of the robot

$$[goto(warehouse), pickup(chocolate), goto(shop), putdown(chocolate)]$$

with

$$val(Inser(exogMoveObject(o,r),s)) = 3 \quad and \\ val(Varia(putdownNothing, putdown(o),s)) = 1$$

equals 4. A hypothesis containing only one instead of the two given changes would have a lower change value. This is a common practice in ranking diagnoses (de Kleer and Williams, 1987).

Equipped with a definition of consistency and a cost function that assigns positive costs to hypotheses, we are able to give a definition of a history-based diagnosis. A diagnosis is valid if and only if it is executable and consistent. In this manner all generated hypothesis could potentially have happened. Therefore, we have to focus on managing all hypotheses by selecting the best one.

Definition 5. *Let $\bar{\delta}$ and $\bar{\alpha}$ be histories. Let $Diag(\bar{\delta}, \bar{\alpha}, v) \doteq ExtVariation(\bar{\delta}, \bar{\alpha}) \wedge Exec(\bar{\delta}) \wedge Cons(\bar{\delta}) \wedge v = cv(\bar{\delta}, \bar{\alpha})$, denoting that $\bar{\delta}$ is an extended variation of σ that is executable and consistent. $\bar{\delta}$ is a proper* history-based diagnosis *(or diagnosis for short) based on $\bar{\alpha}$ if and only if $\mathcal{D}^* \models Diag(\bar{\delta}, \sigma, v)$.*

Again we take the example from the *cv*. Given the original action sequence

$$\sigma = [goto(warehouse), pickup(chocolate), goto(shop), putdown(chocolate)]$$

followed by sensing no chocolate. This sensing obviously contradicts the expected sensing

$$RealSense(sense(chocolate), \sigma)) \not\equiv SF(sense(chocolate), \sigma)$$

Possible diagnoses are:

- $[goto(warehouse), pickupNothing, goto(shop), putdownNothing]$
- $RealSense(sense(chocolate), \sigma)) \equiv SF(sense(chocolate), \sigma)$ (wrong sensing)
- $[goto(warehouse), pickup(chocolate), goto(shop), putdown(chocolate), \\ exogMoveObject(chocolate, quality\_control)]$

Depending on the domain, the number of diagnoses can be very high. We provide an upper bound for this number that is calculated from the length of the action sequence, the number of variations and insertions and a variable k, that defines the maximum number of insertions between two successive actions.

Theorem 1. *Let l be the length of the history σ, n be the number of different possible exogenous events, k be the maximum number of insertions between two consecutive actions of σ, and m be the maximum number of variations of an action. Then, the number H of potential diagnosis candidates is*

$$H = ((m+1) \cdot \sum_{i=0}^{k} n^i)^l.$$

Proof. For every action there are a maximum of m variations plus the action itself. Every action can again be followed by 0 to k insertions. We use a helper variable i to sum over all numbers of insertions to sum up before the next action occurs. The number of insertions is thus exponential in i. So far, we have listed all enumerations for a history of size 1. The whole term is exponential in the history length l. □

The number H of potential diagnosis candidates is exponential in the number of allowed exogenous events between two actions and furthermore exponential in the size of the history. In order to create an applicable system that is able to operate with a given memory size, we handle the possible state explosion by a fixed-size pool of diagnoses candidates. This pool holds n diagnoses that can be derived from a situation. Assuming an enumerable number of insertions and variations this pool *Pool* can be defined as a formula over all possible diagnoses. A *Pool* with fixed size n consists of n tuples of diagnoses and their change values respectively.

Definition 6. *Let σ be a history. Then Pool is defined as*

$$\begin{aligned} &Pool(\sigma) = ((\bar{\delta}_1, v_1), \cdots, (\bar{\delta}_n, v_n)) \doteq \\ &\quad Cons(\sigma) \wedge \bar{\delta}_1 = \sigma \wedge v_1 = 0 \wedge \cdots \wedge \bar{\delta}_n = \sigma \wedge v_n = 0 \vee \\ &\quad \neg Cons(\sigma) \wedge Diag(\bar{\delta}_1, \sigma, v_1) \wedge \cdots \wedge Diag(\bar{\delta}_n, \sigma, v_n) \end{aligned}$$

Due to notational conventions, the pool always consists of exactly n situations. If the situation σ is consistent, the *Pool* consists of n identical tuples in order to fill up the pool. This is the situation itself together with the change value 0. If situation σ is inconsistent, the pools fills with n diagnoses. In the IndiGolog implementation the pool is filled by the most plausible diagnoses, where plausible refers to the change value of a diagnosis.

Out of this pool the belief defining diagnosis is chosen. Therefore we select the diagnosis with the lowest change value and call it preferred diagnosis.

Definition 7. *Let σ be a history. The preferred diagnosis prefDiag is defined as:*

$$\begin{aligned} &prefDiag(\sigma) = \bar{\delta} \doteq \\ &\quad \exists \bar{\delta}_1, v_1, \ldots, \bar{\delta}_n, v_n. Pool(\sigma) = ((\bar{\delta}_1, v_1), \cdots, (\bar{\delta}_n, v_n)) \wedge \\ &\quad [\bar{\delta} = \bar{\delta}_1 \wedge v_1 \leq v_2 \wedge \cdots \wedge v_1 \leq v_n \vee \\ &\quad \bar{\delta} = \bar{\delta}_2 \wedge v_2 \leq v_1 \wedge v_2 \leq v_3 \wedge \cdots \wedge v_2 \leq v_n \vee \\ &\quad \quad \vdots \\ &\quad \bar{\delta} = \delta_n \wedge v_n \leq v_1 \wedge \cdots \wedge v_n \leq v_{n-1}] \end{aligned}$$

We have seen that the number H of possible diagnoses can increase very quickly. On this account, the number of diagnosis due to a given number of faults seems to be more interesting in the majority of cases. For a number c of changes we can compute this amount of diagnosis τ. If the pool size is greater or equal to τ the completeness of the approach can be guaranteed for c faults.

Theorem 2. *Let σ be a history and p be the number of diagnoses of Pool(σ). Let c be the maximum number of all insertions i and variations v to a history σ and let k, l, m and n be as indicated in Theorem 1. Further, let*

$$\tau = \sum_{c'=1}^{c} \sum_{i=0, v=c'-i}^{c'} \binom{l}{v} m^v \binom{i+l-1}{i} n^i.$$

If $c \leq k, l$ then τ is the exact amount of possible hypotheses. τ is an upper bound for $c > k, l$. With $p \geq \tau$ we can guarantee that our approach is complete.

Proof. We have to sum up over all instantiations from one fault to c faults. The current index is assigned to the variable c'. We consider variations v and insertions i separately. Their sum is equal to the current number of faults c'. A maximum of m variations per actions can be arranged m^v times and be distributed over the history length l. Similarly the i insertions result in n^i combinations and can be distributed over the new length $i + l - 1$. The product of both terms leads to the combined set of possible hypotheses. If the pool size is greater or equal to this maximum number of hypotheses, then obviously all hypotheses can be considered to be in the pool. □

Chapter 5

Belief Management in a Guarded Action Theory

The Basic Action Theory provides a proven axiom set in order to model dynamic changing worlds. However, it does not address information retrieval by sensing as well as incomplete causal laws explicitly. This fact is tackled by Guarded Action Theories (Sardiña, 2000). For this reason, Guarded Action Theories form the formal concept of IndiGolog. As Guarded Action Theories can be seen as an extension to the Basic Action Theory, some IndiGolog statements differ from the formal definition in the Basic Action Theory. This chapter focuses on the application of our belief management in IndiGolog based on the underlying Guarded Action Theory. After giving an overview of Guarded Action Theories in Section 5.1, we describe IndiGolog's successor state axioms and our notion of inconsistency within the framework in Section 5.2. We close this chapter with IndiGolog's main loop that is responsible for executing user programs together with our extension to it in Section 5.3. Parts of this chapter were published in Gspandl, Pill, Reip, and Steinbauer (2013).

5.1. Guarded Action Theory

Guarded Action Theory (GAT) extends the presented Basic Action Theory by explicitly handling information gathering through sensing and the capability to express incomplete causal laws (see Sardiña (2000)). Instead of having to execute sensing actions continuously, sensors are treated as always being on, similar to a laser that continuously sends data. Sensing results are integrated into the situation term, as described below. Incomplete causal laws make it possible to lose information. This means, that though the value of a fluent is known in situation s (just one valid assignment), the fluent may be unknown in the successor situation $s' = do(a, s)$ (many valid assignments). It can be necessary due for two reasons. On the one hand, the effects of an action may be unknown. For instance, the result of an action may depend on an unknown fluent, such as setting a desired speed on the motors can lead to the desired velocity or to a stopped robot in case the emergency stop is pressed. If the robot does not know the state of the emergency stop, it can not infer the resulting velocity. On the other hand, other agents may change the world in an unobservable way (for example the state of a door in a multi-agent environment might change eventually after some robot's interaction with the door). As the Guarded Action Theory is an extension to the Basic Action Theory, it contains all axioms of the Basic Action

Theory except successor state axioms. These are replaced by more general guarded successor state axioms. Guarded successor state axioms for a fluent F have the form (Sardiña, 2000):

$$\alpha(\vec{x},a,s) \supset [F(\vec{x}),do(a,s)) \equiv \psi(\vec{x},a,s)]$$

Guarded sensed fluent axioms for a fluent F that replace the SF predicate have the form:

$$\beta(\vec{x},s) \supset [F(\vec{x},s) \equiv \rho(\vec{x},s)]$$

Here, β is a sensor-fluent formula, α and ψ are fluent formulas and ρ is a sensor formula. A sensor fluent formula is a formula that contains a maximum of one situation term that is a variable. Similar to the Basic Action Theory this situation term is only allowed as last argument of a fluent as well as an argument of a sensing function. Sensing functions are unary functions with a situation as single argument, e.g. loaded(s), for a boolean load sensor that signals if a robot is loaded with a container. A sensor formula is a sensor-fluent formula without fluents. Finally, a fluent formula is a sensor-fluent formula without any sensor function. They fall into the category of Reiter's uniform formulas (see Reiter (2001)). α and β are called guards that condition the formulas. Incorporating guarded successor state axioms and guarded sensed fluent axioms leads to the guarded action theory $\mathcal{D}_G$:

- $\mathcal{D}_0$ the initial situation S_0 axioms
- $\mathcal{D}_{Poss}$ the action precondition axioms
- $\mathcal{D}_{GSSA}$ the guarded successor state axioms, where each fluent can have none, one or many axioms
- $\mathcal{D}_{GSFA}$ the guarded sensed fluent axioms, where each fluent can have none, one or many axioms
- $\mathcal{D}_{una}$ the unique name axioms of actions
- $\mathcal{D}_{FUN}$ foundational, domain-independent axioms

The solution to the frame problem as presented above is still applicable. Please see Sardiña (2000) for details. Situation terms within the BAT contain all actions that have occurred since the initial situation S_0. In order to evaluate a formula in a given situation s, we have to ask what is the truth value of the formula after executing a given action sequence. In the context of GAT the question behind evaluating a formula $\phi(s)$ has to be extended to the following: what is the truth value of formula $\phi(s)$ after executing a given action sequence and receiving a given sequence of sensor outcomes. As the situation term is no longer sufficient to model everything that has happened, Sardiña (2000) introduces the notion of history. As situations can easily be extracted from histories. The terms are sometimes used synonymously.

Definition 8. *A history σ is a sequence $(\vec{\mu_0} \cdot (A_1,\vec{\mu_1}) \cdot (A_2,\vec{\mu_2}) \cdots (A_n,\vec{\mu_n})$ with ground action terms A_i with $(0 \le i \le n)$, sensor readings vectors of (real) values $\vec{\mu_i} = (\mu_{i1},\mu_{i2},\ldots,\mu_{im})$ with $0 \le i \le n$ and $0 \le j \le m$. Here, μ_{ij} corresponds to the j-th sensor after the i-th action (Sardiña, 2000).*

This means, within the notion of a history, all received sensor values are added to the situation. The information can be separated again by the application of *end* and *Sensed*.

Definition 9. *Let σ be a history, then the ground situation term $end[\sigma]$ is (Sardiña, 2000):*

$$end[\varepsilon] = S_0$$
$$end[\sigma \cdot (a,\mu)] = do(a,end[\sigma])$$

where ε denotes the empty history.

In case of the empty history *end* equals the initial situation. Otherwise we extract action by action and add it recursively to the situation term. For example, for an object x and the origin and destination rooms o and d and the sensing result $\mu_1,...,\mu_4$, $end[\mu_0 \cdot (goto(o),\mu_1) \cdot (pickup(x),\mu_2) \cdot (goto(d),\mu_3) \cdot (putdown(x),\mu_4)] = do(putdown(x),do(goto(d),do(pickup(x),do(goto(o),S_0))))$.

Definition 10. *Let σ be a history, then the ground sensor formula $Sensed[\sigma]$ is (Sardiña, 2000):*

$$Sensed[\sigma] = \bigwedge_{i=0}^{n} \bigwedge_{j=1}^{m} h_j(end[\sigma_i]) = \mu_{ij},$$

where σ_i denotes the subhistory from the initial situation till action i ($\sigma_i = (\vec{\mu_0} \cdot (A_1,\vec{\mu_1}) \cdot (A_2,\vec{\mu_2}) \cdots (A_i,\vec{\mu_i}))$ and h_j stands for the sensor function of the j-th sensor.

Similarly to *end*, *Sensed* iterates over the history, separates actions from sensing results and combines all sensing results into one formula.

Furthermore we have to extend the definition of executability from situations to histories. This is done by applying the *Poss* predicate to all action of the sequence: a history σ is executable in a GAT $\mathcal{D}$ iff either $\sigma = \vec{\mu_0}$ or $\sigma = \sigma' \cdot (A,\vec{\mu})$ and σ' is an executable history and $\mathcal{D} \cup Sensed[\sigma'] \models Poss(A,end[\sigma'])$.

5.2. Inconsistency in IndiGolog

In a sense Guarded Action Theory serves as appetizer to IndiGolog. Let us recall from Chapter 2.2.1 how IndiGolog models the effect of actions. It uses $causes(a,f,v,w)$, where a fluent f takes value v, after executing action a, if w holds. Here the condition w is made explicitly in form of a guard, as presented above. Generally, if a fluent is not known for sure in IndiGolog, all possible instantiations are stored to express the different possibilities. Multiple instantiations per fluent can be sharpened by sensing actions that either invalidate a single instantiation, or all instantiations besides one. Besides the guards in the successor state axioms, sensing results are also handled explicitly. However, every action is followed by exactly one sensing result with the value 0 or 1, instead of a list of results with one reading per sensor. In this way, sensing actions are required. Therefore, the definition of *Sensed* is slightly different from the definition above (de Giacomo et al., 2009):

Definition 11. *Sensed denotes a formula containing all sensing information of a given situation.*

$$\begin{aligned} Sensed[\varepsilon] &= True \\ Sensed[\sigma \cdot (a,1)] &= Sensed[\sigma] \wedge SF(a,end[\sigma]) \\ Sensed[\sigma \cdot (a,0)] &= Sensed[\sigma] \wedge \neg SF(a,end[\sigma]) \end{aligned}$$

In Chapter 4.2, we derived the definition of an inconsistency. A situation is said to be inconsistent if either an invariant is violated or a sensing value contradicts the expected one. In order to evaluate invariants it is sufficient to evaluate them for every situation. In case of sensing we have to apply IndiGolog's version of guarded sensed fluent axioms. As presented above, we have

$$\begin{aligned} &settles(a,r,f,v,w) \\ &rejects(a,r,f,v,w) \end{aligned}$$

In this case, after executing sensing action a with sensing result r, if condition w holds, the fluent f is either fixed to value v in case of *settles*. Similarly, *rejects* abandons v as possible value of fluent f. For instance, $settles(perceiveObject(X), 1, is\_at(X,R), True, at(R))$ states that if the robot perceives an object X and it is located in room R, then the location predicate *is_at* of object X in room R is set to True.

In order to detect inconsistencies due to sensing we have to inspect the current situation after every sensing action. This is done by investigating all applications of *settles* and *rejects*. By application we mean that a sensing action a with sensing result r has occurred and condition w holds. In case of *rejects* we have to ensure that the fluent value we reject is not the only possible instantiation of that fluent, before the sensing action is executed. As a sensing action does not change the world, we do not expect fluents to change due to sensing actions too.

Definition 12. *check_rejects ensures that the sensing result r of sensing action a does not reject the only valid fluent value v of fluent f.*

$$\begin{aligned} check\_rejects(\sigma) \doteq end[\sigma] = do(a,s') \wedge Sensed[\sigma] = (\mu_0, \mu_1, \ldots, r) \wedge \\ [rejects(a,r,f,v,w) \wedge w(\vec{x},s')] \supset \neg(f(\vec{x},s') = v)] \end{aligned}$$

where, σ is a history that is split into its situation term and the sensing formula. If *rejects* is applicable and w holds, we check that the rejected fluent value combination does not hold. As actions in the situation calculus are required to have a strict order within the situation term, it is no limitation to compare the fluent value before the sensing action is executed with the applied result of the sensing action.

The inspection of *settles* applications works similar to *rejects* applications. Here we have to assure, that there is no fluent value different to the settled value that holds before the execution of the sensing action. Please note, that this is not necessarily the same as that the value itself has to be possible before the sensing action. This follows the fact that a value is said to hold only if there is no other fluent value entry than the examined one.

Definition 13. *check_settles ensures that the sensing value r of sensing action a that leads to fluent value v of fluent f is a valid instantiation of fluent f.*

$$\begin{aligned} check\_settles(\sigma) \doteq end[\sigma] = do(a,s') \wedge Sensed[\sigma] = (\mu_0, \mu_1, \ldots, r) \wedge \\ [settles(a,r,f,v,w) \wedge w(\vec{x},s')] \supset \neg(\exists v'.(f(\vec{x},s') = v')) \end{aligned}$$

So far, we are able to check the outcome of sensing actions by applying *check_settles* and *check_rejects*. We can combine them with *Invaria* in order to have a single predicate *check_last_action*, that is responsible for the newly added action.

Definition 14. *check_last_action ensures that the last action of situation σ does not lead to an inconsistency.*

$$check\_last\_action(\sigma) \doteq check\_rejects(\sigma) \wedge check\_settles(\sigma) \wedge Invaria(end[\sigma])$$

In order to evaluate a complete situation σ, we can apply *check_history*. It uses *check_last_action* action by action:

Definition 15. *check_history ensures that a given history s is consistent.*

$$check\_history(\varepsilon) \doteq Invaria(S_0)$$
$$check\_history(\sigma \cdot (A,\mu)) \doteq check\_history(\sigma) \wedge check\_last\_action(\sigma \cdot (A,\mu))$$

where ε resembles the empty history, A is an action with the corresponding sensing result μ. After transferring the definition of consistency into the guarded IndiGolog world, we are equipped to extend IndiGolog's main loop with our belief management.

5.3. IndiGolog's extended main loop

In Chapter 2.2.2 we described the main loop of the IndiGolog interpreter. As it is implemented in Prolog, the following formulas use the Prolog syntax. First, it integrates exogenous events into the situation. Second, it progresses the initial situation if the history length exceeds a predefined threshold. Third, it checks for termination by applying the *Final* predicate. And finally, it applies the *Trans* predicate in order to execute the next step of the program. The interpreter loops until *Final* holds or there is no longer a valid transition.
A typical transition either checks a condition in case of an *if* or *while* statement, or it checks the precondition of a specific action. All this evaluations expect a consistent situation term. As we have seen before, this situation term may turn inconsistent due to an incorrect initial situation, an execution failure, a sensing failure or unobserved exogenous events.
Therefore, in order to provide the evaluation with a consistent situation term, we add another step to the main cycle that checks the current history for consistency. If the situation is inconsistent, we apply history-based diagnosis to generate new situations out of the current one and add the new situations into our situation pool. Then the system takes the situation with the smallest change value and checks it for consistency. If it is consistent, we can continue within the main cycle, otherwise we keep generating history-based diagnoses, selecting the best one and checking it.
The extended IndiGolog's main loop looks like this:

```
indigo(E,H) :- handle_exogenous(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_consistency(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_rolling(H,H2), !, indigo(E,H2).
indigo(E,H) :- catch(final(E,H), exog, indigo(E,H)).
indigo(E,H) :- catch(trans(e,H,E1,H1), exog, indigo(E,H)),
  (var(H1) -> true ;
   H1 = H -> indigo(E1,H);
   H1 = [A|H] -> exec(A,S), handle_sensing(H,A,S,H2), indigo(E1,H2)).
```

We added the *handle_consistency* call to our belief management. It takes a possibly inconsistent history, and returns a consistent one, if one exists. If there is no consistent history, the same inconsistent history is returned, in order to prevent the robot from canceling its order. Following the implementation of IndiGolog, using an inconsistent history means, that we use the last perceived values though they might contradict previous expectations. This approach is inspired by humans. Quite often we are not able to understand every facet of a scene, resolving every single inconsistency between our expectations and our observations, but it many cases we can still perform the intended actions.

```
handle_consistency(H,H1) :- most_plausible_situation(S),
  must_check_history(H,S,N) ->
    check_consistency_n(H,S,N,H1,S1) ; H1 = H.
```

The predicate *handle_consistency* takes the current action sequence H as input and provides an action sequence H1 as output. The given argument H contains an action sequence only and does not contain the initial situation. All entries of the *Pool* have an unique id (we call it index) that allows to retrieve corresponding initial situation, action sequence and change value. Therefore we first extract this index S of the current situation, which we refer to as *most_plausible_situation*. The most plausible situation is a situation S of all possible situations (*possible_situation*). All entries of all initial situations are stored in a database, and can be accessed by this index. Next by using *must_check_history*, we check whether checking the given history is necessary. If no action was added since the last check, we simply set the consistent history H1 to the given history H. If new actions were added, *must_check_history* unifies the parameter N with the number of actions to be checked. This dramatically reduces the computation costs, as we avoid evaluating the same subhistory over and over. Finally, *check_consistency_n* takes the current history together with the situation index and the number of new actions to be checked and unifies the new history favorite.

Before we present *check_consistency_n*, we will introduce *extend_history*. This clause adds a newly executed action to all histories of our pool. The pool's history indexes are enumerated by the clause *possible_situation*. The corresponding histories are mapped through the clause history taking the history's index. In order to add newly executed actions, we simple iterate over all possible situations, take the stored history H, append the given action A and store the merged history again. The usage of the clause flatten simply ensures that the resulting history Merged is a list without sublists. In this way the action A could be a list of actions as well.

```
extend_history(A) :- forall(possible_situation(S), (
  retract(history(S,H)), flatten([A|H], Merged), assert(history(S,Merged)))).
```

The clause *check_consistency_n* checks the last N actions for consistency and adds all of the new N actions to all histories. This is done recursively. The first line presents the base case with the number N of new actions equal to 0. In that case no action is necessary, hence the output history and its index (arguments four and five) are set to the input arguments one (history) and two (history index). Otherwise, we reduce the number of actions to be checked by one, as shown by variable M and call the clause recursively. After returning from the recursive call, we add the last action A that is split from the remaining history R to all histories from our pool with the clause *extend_history*. Then we call *get_favorite_history* that does the actual inspection returning the favorite situation index S1 and finally retrieve the resulting history H1 from the updated index S1.

```
check_consistency_n(H,S,0,H,S) :- !.
check_consistency_n([A|R],S,N,H1,S1) :- M is N -1,
  check_consistency_n(R,S,M,_,SOld), extend_history(A),
  get_favorite_history(SOld,S1), history(S1,H1).
```

Now we introduce the clause *handle_situation_to_be_removed* that is used by *get_favorite_history*. In case a situation turns out to be inconsistent, we have to remove this situation (S) from the pool and select a new favorite hypothesis (Res).

```
handle_situation_to_be_removed(S,Res) :-
  setof(X, possible_situation(X),LX), length(LX,1), !, Res=S.
handle_situation_to_be_removed(S,Res) :- retract_situation(S),
  most_plausible_situation(Next), get_favorite_history(Next,Res).
```

First, we check if the situation to be removed is the last one. We simply check if the size of the set of all possible situations of the pool equals one. If this is the case, we keep the situation by setting the result history index to the input index. Otherwise the situation is dropped from the hypothesis pool by *retract_situation*, we select the next favorite situation by *most_plausible_situation* and again call *get_favorite_history*. This call is necessary as there is no guaranty that all situations in our pool are still consistent due to newly added actions. Of course it would be possible to check them all, after every added action, but this would lead to an extreme computational overhead. Therefore, we only check consistency on demand in a lazy fashion. Finally, we will discuss the *get_favorite_history* clause, that selects the next belief-defining history:

```
get_favorite_history(S, S) :- history(S,H), check_history(H,S), !.
get_favorite_history(S,Res) :- generate_history_variations(S),
  handle_situation_to_be_removed(S,Res), !.
```

The first clause retrieves the history from the pool and checks it for consistency, by calling *check_history* as introduced in Chapter 5.2. If the check is valid, we can return the input situation. Otherwise, hypotheses are generated according to history-based diagnoses and are added to the pool by *generate history_variations*. Then we drop the invalid situation in case we have a consistent one by *handle situation_to_be_removed*.

Figure 5.1 sums up all given clauses and combines them in a callgraph that defines the presented belief management system.

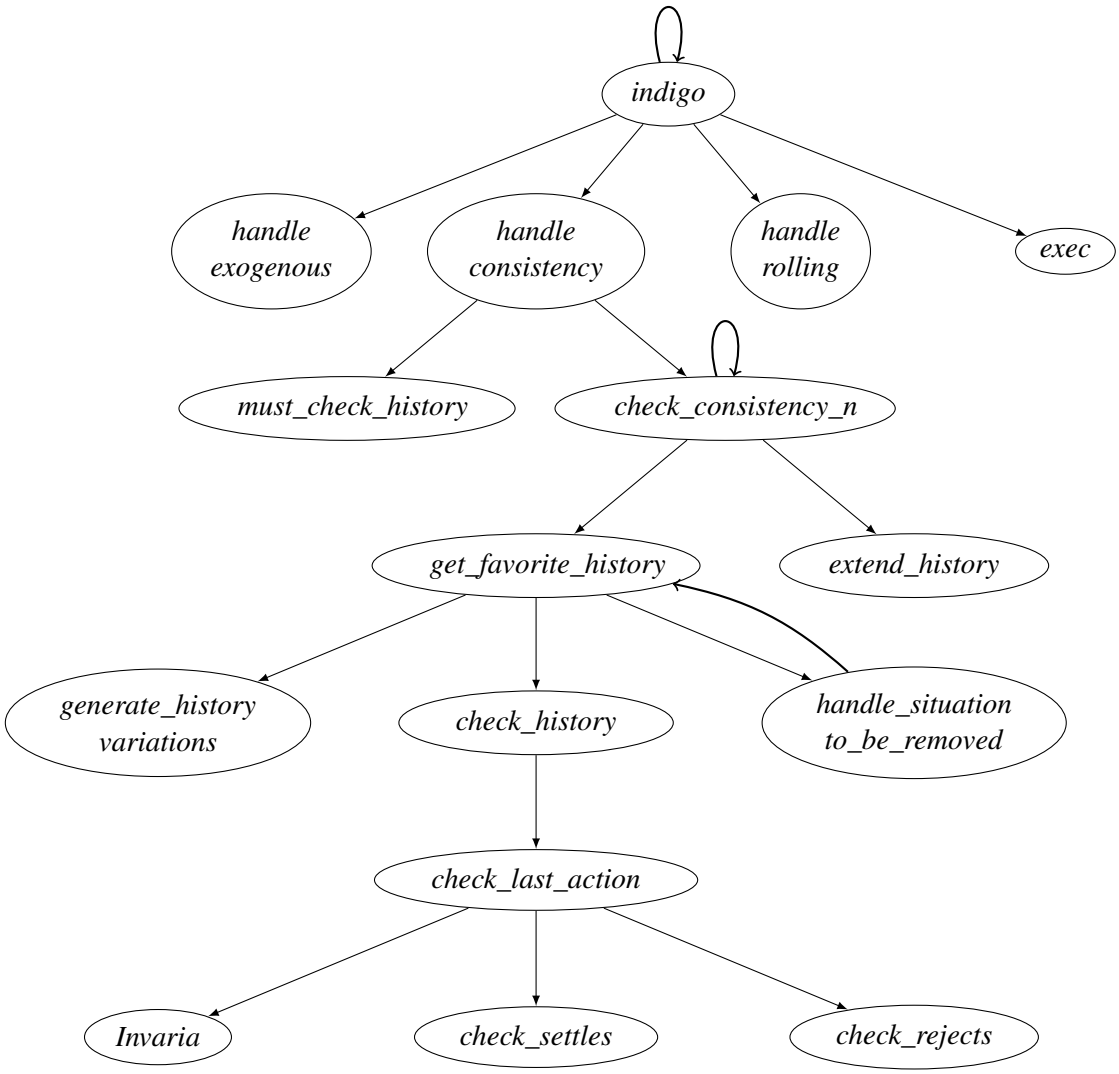

Figure 5.1.: Callgraph belief management in IndiGolog

Chapter 6

Robot Control Framework

After introducing a belief management formally, we will go one step further and apply it to a physical robot. We propose a system architecture, describe the interfaces and take a closer look at exemplary instantiations. Before this architecture is presented, we provide preliminaries that are crucial for this chapter. We start by examining teleo-reactive (T-R) programs (Nilsson, 1994) in Chapter 6.1. T-R programs are a popular way to program a robot and they provide useful properties in combination with belief management. They offer the possibility to react to changes in the belief in a straightforward manner. Chapter 6.2 gives an introduction into grounding. Grounding defines the connection between a logical representation and its embodiment in reality (Russel and Norvig, 2003) that is relevant if we want to execute actions on a real robot and sense the execution results. This is followed by Chapter 6.3 where all parts are put together to form a robot control framework that is able to cope with execution and sensing failures, incomplete initial knowledge and exogenous events. This chapter is based on work published in Gspandl, Podesser, Reip, Steinbauer, and Wolfram (2012), that was nominated for the "Best Cognitive Robotics Paper" award.

6.1. T-R Programs

T-R programs, as described by Nilsson (1994), provide a convenient way to control robots, while reacting to continuous feedback from the environment. The approach tries to combine ideas from the AI community with circuit-based systems such as GAPPS (Kaelbling, 1988; Kaelbling and Rosenschein, 1990). Therefore, we can find similarities to production systems as used by Nilsson (1984) for the famous SRI robot Shakey, or the STRIPS planning as presented by Fikes et al. (1972). T-R programs support parameter binding as well as recursion. Programs can evolve at run time, so it is not essentially necessary to anticipate all possible situations at construction time. Another reason for the popularity of the approach is that creating T-R programs is easy and very intuitive. This not only supports human programmers but allows for automated planning and learning.

T-R sequences are robot programs that lead to a specific goal (teleo) while continuously taking the

environment into account (reactive). They consist of productions:

$$K_1 \rightarrow a_1$$
$$K_2 \rightarrow a_2$$
$$\ldots$$
$$K_i \rightarrow a_i$$
$$\ldots$$
$$K_m \rightarrow a_m$$

Here K_i denote conditions, while a_i are actions. Thus, if condition K_i holds, action a_i is to be executed. The program is always scanned from the top, searching for the first condition to hold. K_1 denotes the goal condition, thus a_1 is typically a nil action. The condition K_m is typically always true.

Rather than thinking in discrete actions, Nilsson views the system from a circuitry perspective as shown in Figure 6.1. Conditions are continuously evaluated in order to achieve direct reactions to the environment.

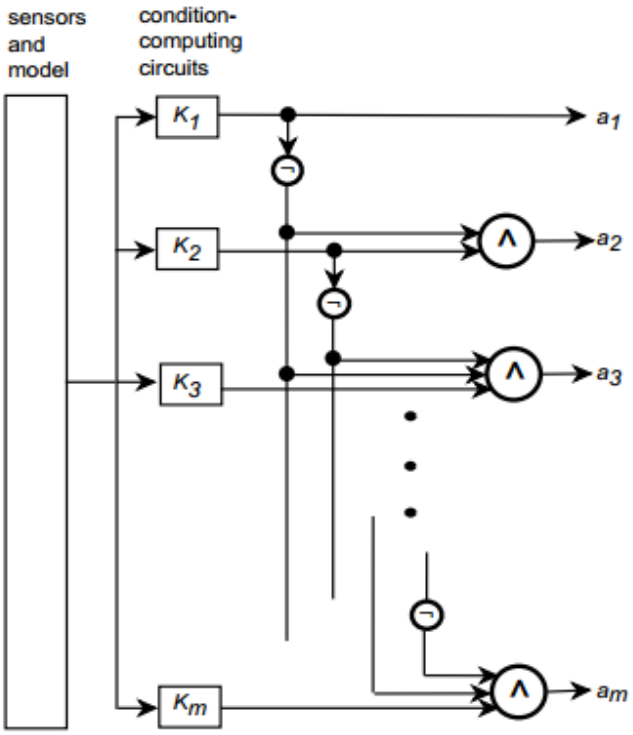

Figure 6.1.: TR Sequence in Circuitry (Nilsson, 1994)

The following example shows a T-R program named *task* that allows a robot to deliver an object *obj* to a target *dest*. If the object is already at the desired target location (the goal condition) no action is necessary. If the condition does not hold, the next condition is evaluated. In our case, we check if we reached the target location while holding the object. If this holds, the object is put down. One step before releasing the object at the target location, is to move to the target with the object. This can be done after picking up the object. If the robot loses the object while moving, the condition

$has\_object(obj) \wedge \neg is\_at(obj, dest)$ evaluates to false, so that the next condition is checked and the robot tries to pick the object again. Thus a robot executing the given T-R program is able to react to the environment while focusing on a given goal. The same holds if the object is brought to the target location while the robot is approaching it. As soon as the object reaches the target location, the robot stops by reaching the nil action. The last condition is the weakest one, it simply unifies the current location of the object to be transported.

$$
\begin{aligned}
&task(obj, dest)\\
&\quad \neg has\_object(obj) \wedge is\_at(obj, dest) \rightarrow nil\\
&\quad\quad has\_object(obj) \wedge is\_at(obj, dest) \rightarrow putdown(obj)\\
&\quad\quad\quad has\_object(obj) \wedge \neg at(obj, dest) \rightarrow goto(dest)\\
&\quad \exists room.at(room) \wedge is\_at(obj, room) \rightarrow pickup(obj)\\
&\quad\quad\quad\quad \exists room.is\_at(obj, room) \rightarrow goto(room)
\end{aligned}
$$

A T-R program works backward from a specific goal, where each action is expected to achieve the next higher condition. For instance action $goto(t)$ that is executed if condition $has\_object(o) \wedge \neg is\_at(o,t)$ holds, is expected to eventually achieve the condition $has\_object(o) \wedge is\_at(o,t)$, if it is executed long enough. Formally, an action a_i is supposed to reach a condition K_j with $j < i$. Conditions are regressions of higher conditions through actions (Nilsson, 1994). A T-R program is said to fulfill the regression property if each condition K_i with $i > 1$ is the regression of a condition K_j with $j < i$. A completeness property states that all m conditions of a T-R program $K_1 \vee \cdots \vee K_i \vee \cdots K_m$ form a tautology. This ensures that there is an action that can be executed. A T-R program is said to be universal, if and only if it is complete and the regression property holds.

As shown in the example above, where a robot is instructed to deliver an object to a destination, T-R programs can contain parameters. Their entries are not only restricted to primitive actions, but can again be T-R programs. This way, recursions are enabled, as shown in the following example (Nilsson, 1994). Here the *goto* routine is extended to a new program *amble* where the robot moves straight if the path is clear, or avoids obstacles by adding new points to the path through the routine *newPoint*. Alternatively to binding parameters in the condition part, the recursion example expects a variable *position* to continuously being evaluated to the current position of the robot.

$$
\begin{aligned}
&amble(dest)\\
&\quad\quad at(dest) \rightarrow nil\\
&\quad clear\_path(dest) \rightarrow goto(dest)\\
&\quad\quad\quad \top \rightarrow amble(newPoint(position, dest))
\end{aligned}
$$

Similarly to rule-based representation, T-R programs can also be represented as graphs. Nodes are formed by conditions, whereas arcs between the nodes are realized by actions. The goal condition forms the root node of the graph. Each action is expected to account for the next higher node. If more than one action is necessary to achieve the node, then all subgraphs are added by the corresponding action arc to the node. Instead of a one-path graph, this leads to a tree that is called teleo-reactive tree (T-R tree). T-R trees are executed by identifying the shallowest true node and executing the action

of the root-directed arc. If there is some notion of cost, the lowest-cost true node could be selected instead of selecting the shallowest true one. The regression, completeness and universal properties are transferred directly from T-R sequences to T-R trees.

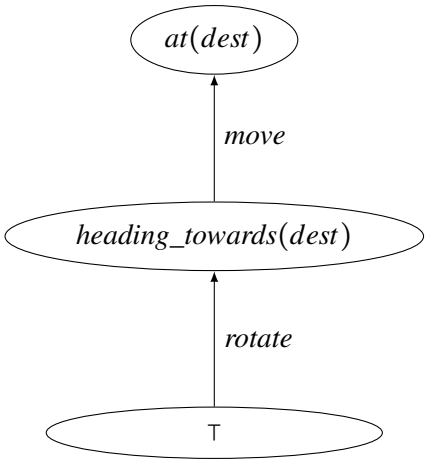

Figure 6.2.: T-R-sequence graphical representation of *goto* (Nilsson, 1994)

Reaching the desired location ($at(dest)$) forms the goal condition of the TR graph in Figure 6.2. If this property holds, the robot reached its goal and no action is necessary. If not, the robot checks if it is already looking at the target. If yes, the robot starts its motors with the action *move*. Otherwise it turns to the target.

To summarize, T-R programs can be represented in different ways. They are very easy to understand, are goal directed and react immediately to the environment. This is why T-R programs were chosen to act as execution control instance within our framework.

6.2. Symbol grounding

The symbol grounding problem (Harnad, 1990) accompanies the AI community since its beginning. It deals with the problem of how to provide symbols with meanings. Symbols are any objects that are part of a symbol system. This means a hole in a paper can be a symbol as well as a word in a computing machine. There are syntactical rules that specify the manipulation of symbols. Their interpretation falls in the area of grounding.

One of the first researchers who approached this problem was Alan Turing in his article Computing Machinery and Intelligence (Turing, 1950), where he proposed the rather philosophical question “Can machines think?”. He answered the question by a behavioral intelligence test (called Turing Test). In this test a questioner interviewed its counterpart for five minutes. After the interview the questioners had to answer the question whether they were communicating with a person or a machine. If the interrogator is wrong in more than 30% of the the time the machine succeeds. Thus, in order to pass the test a machine does not necessarily have to understand the scene and its symbols in use, it only has to use them. In order to act as if it was intelligent forms the weak-AI hypothesis in philosophy. In contrast to weak-AI stands the strong-AI hypothesis where a system is expected to really think instead of simulated thinking (Russel and Norvig, 2003).

The idea of weak-AI was continued by McCarthy et al. (1955) with the proposal for an AI summer workshop. There, they stated that "Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it." At the rise of chat rooms many chatbots successfully deceived humans. The chatbot CYBERLOVER was even smart enough to elicit enough personal information, so that identities could be stolen and law enforcement started its work.

In contrast to this enthusiastic view of intelligent machines, Turing formulated some objections too. A very strong objection is the argument disability where Turing creates a list of desirable characteristics such as being friendly, making mistakes, learning from expertise, telling right from wrong, falling in love, making someone fall in love with it, using words correctly, having the same level of diversity of behavior as humans, doing something really new. Many of these properties can be assigned to intelligent machines. For example learning and telling right from wrong to Deep Blue (Campbell et al., 2002), the chess computer that won against the world chess champion Garry Kasparov, or TD-Gammon that was able to play Backgammon at expert level (Tesauro, 1995) using artificial neural networks that were trained by temporal difference learning. Machines like the teddy bear are also known to make someone fall in love with it. Regarding the correct usage of words, the New York Times describes iPhone's virtual assistant Siri with the words unbelievable, performing an incredible range of tasks or just mind-blowing (Pogue, 2011). Nevertheless, behaviors like falling in love are still uncommon for machines.

Another objection raised by Turing is the mathematical objection. It is based on Gödel's incompleteness theorem (Gödel, 1931). It states that within a formal axiomatic system including arithmetic F, a Gödel sentence $G(F)$ with the following two properties can be created. First, $G(F)$ is a sentence of F, but cannot be proved within F. Second, if F is consistent, then $G(F)$ is true (Russel and Norvig, 2003). Furthermore Turing stated the argument of informality. There he claims that human behavior cannot be described by a set of rules. As machines can be seen as rule-executing systems, they are unable to act like humans (Qualification problem).

In contrast to the notion of weak AI, the Turing test does not go far enough to representatives of the strong AI community. They claim, that it is still a simulation of thinking. Machines passing the test, miss the consciousness of passing it. This is formulated by Geoffrey Jefferson as: "Not until a machine could write a sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain - this is, not only write it but know that it had written it." (see Russel and Norvig (2003); Jefferson (1949)).

6.3. Robot Control Framework

Equipped with knowledge of how to derive qualitative data from quantitative inputs and the ability to reach a goal state using T-R programs, we put things together to describe a framework that allows a robot to act in real life situations. This framework has to ensure that a robot is able to perceive the world, process this data in order to provide a solid basis for decision-making and finally execute this decision. In parallel, we seek to be invariant to external negative influences and incomplete or incorrect initial knowledge. Furthermore, a robot acting according to this framework is expected to overcome execution and sensing failures. In order to overcome these four threats already indicated within the Sensing - Decision-Making - Execution cycle in Figure 1.5, the framework consists of two layers. A low-level system is responsible for sensing and execution and is realized within the Robot Operating System ROS. An introduction to ROS and relevant components that are necessary to

control a mobile robot is given in the next section. The high-level system that is implemented in our extended IndiGolog version covers decision-making and belief management as described within the previous sections. T-R programs play a key role within the whole system, as they allow for reacting immediately to changes in the robot's belief. That means, it is irrelevant to the program execution by T-R programs if the belief was changed. As the program execution always focuses on the current favorite hypothesis, this hypothesis can be replaced without problems. This way we can optimize belief management and program execution separately. Section 6.1 explains that T-R programs are goal oriented and thus always execute the first executable action closest to the goal state. If the world state changes, actions change seamlessly, keeping an uninterrupted emphasis to reach the goal. Between the high-level and the low-level layers we find a sensing-decision-action loop. Sensing results are propagated up, whereas actions are sent down. This complete setup is indicated in Figure 6.3.

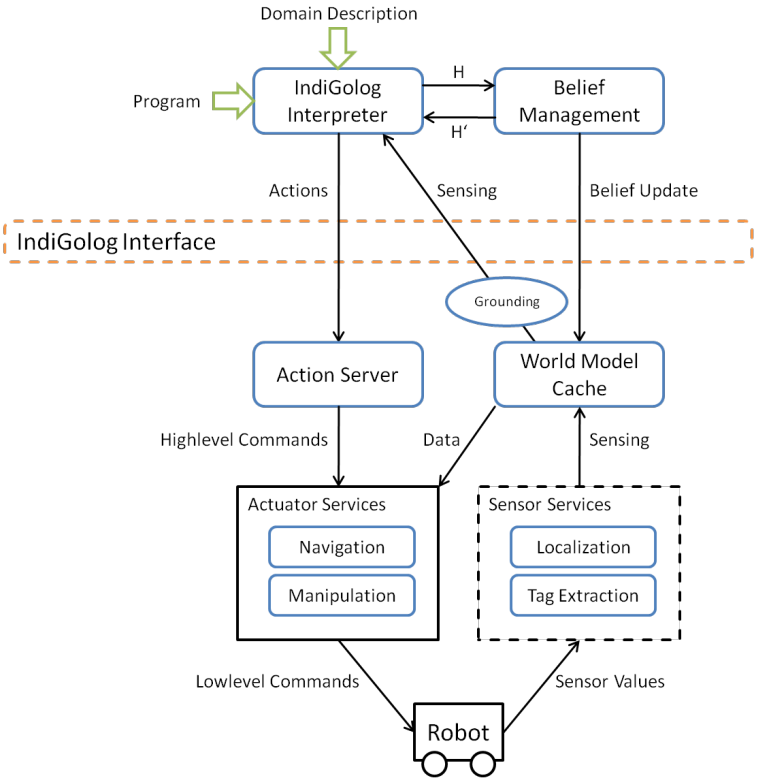

Figure 6.3.: Robot Control Framework System Architecture

6.3.1. Low-Level System

The low-level system is responsible for sensing the environment and executing the decisions derived by the high-level system. It consists of a set of subsystems that are examined one after the other. As this low-level system is mainly realized within the prominent robotics framework ROS, we start with a short introduction into ROS (see Quigley et al. (2009); Marder-Eppstein et al. (2010)).

ROS

ROS consists of a communication system with implementations in C++ and Python. There exist further, though not complete implementations in Java, Lua and many other languages. BSD was chosen as license. This means it is open source, but can be used commercially without restrictions. This way the framework has a broader scope of interest not limited to university but equally interesting to industry. The communication system is augmented by a broad range of tools that allow for communication introspection or provide insight into data through visualizations. Such features are definitely necessary for successful robotic developments, but the most important argument for the great success story is the set of sophisticated algorithms. This includes core algorithms for localization and navigation, as well as many different algorithms for specific robotics applications. Most of them are written as universal as possible, so they can be applied to a broad range of different robotic plattforms very easily by setting the key parameters. But users still need a solid robotic background as they have to set up the system, but they do not need to write all basic, essential algorithms on their own. Roboticists unfamiliar to ROS need only a few days in order to get a robot running, before they can focus on the topics of interest.

ROS was started in 2007 by the Stanford Artificial Intelligence Laboratory within the Stanford-AI-Robot project (STAIR). It received an important boost through the initiative of Willow Garage. Willow Garage was founded by a former Google employee and declares itself a robotic research institute for non-military but personal robotics. Besides open-source software, open source hardware is a crucial element within its company policy. In 2010 the company shipped the PR2 (personal robot two) to eleven research institutes around the globe (see Figure 6.4). In return to receiving a robot, these institutes committed themselves to release own developments to the global robotic community. The capabilities of a PR2 are still impressive, but in 2010 they turned recent developments upside down. Robots where able to fulfill a new range of activities such as fetching beer from a fridge or preparing pancakes. A PR2 consists of a mobile platform with two arms, where each arm has seven degrees of freedom. Several camera and laser systems can be found within the robot's head. In combination with microphones a lot of interesting human-robot interaction research was carried out. Another laser is mounted on the base of the robot and serves for navigation. The impressive computation power of two quad core i7 Xeon servers with eight cores each and 24GB of RAM is located within the base as well. In addition to a 500 GB hard disk, it consists of a 1,5 TB disk that can be easily changed in order to provide researchers with fast data analysis.

We can definitely say that Willow Garage and ROS lifted robotic research to a higher level, but many problems still remain. On the one hand, it is very easy to set up a prototype that can do fancy things in a demo setup, but it is very hard to receive a dependable system in a productive environment. Though communication can be inspected to some degree, it is not trivial to receive an overview of the global system - this includes fundamental questions, such as: "Are all components running correctly?". Furthermore most of the work concentrates on low-level activities, but there is only little work on higher-level activities like the proposed framework. Recently the Open Source Robotics Foundation sponsored by prominent names Bosch, Darpa, Google Atap or NASA, took over Willog Garage's activities and focusing on topics like multi-robot teams, real time systems, non-ideal networks. Those efforts definitely attract less public interest, but are a major step towards widespread usage of robotic applications.

Figure 6.4.: Personal Robot 2 (PR2) fetching beer from a fridge

Robot Driver

Opposite of high-level decision-making and belief management instances and within the low-level system we find the robot driver. It is responsible for hardware communication and abstracts the hardware as much as possible in order to reduce the influence of different platforms to the remaining of the system. This means transforming information from sensors as well as forwarding appropriate commands to execution systems such as motor controllers, load-carrying devices or simple commands like shutting down the system. Similar to operating systems, hardware abstraction is a crucial requirement for successful deployment of a system to different platforms. For example apart from specific parameter sets and a few special implementations, incubed IT, a company developing robotics navigation solutions, runs their navigation system with the same code on various robots. Starting from a robot transporting special goods with a weight of 15 kg, to a robot carrying standard totes for 600 times 400 mm with up to 50 kg, to a bigger robot that carries a weight of 200 kg and finally a big robot that carries euro pallets (see Figure 6.5).

Sensor Services

The information read from the robot driver is forwarded to a set of sensor services that process this information. For the delivery domain that serves as running example, we have four main sensor sources. First, we derive odometry information from the wheels of the robot. Similar to a car, we count the turns and can therefore guess the current position based on our last position. If the kinematics of the robot is similar to a differential drive, odometry information of turns is very noisy. So we could additionally use a gyroscope to stabilize the turn information. Nevertheless, a position estimate based on odometry information is nothing more than a guess, that becomes worse and worse as time goes on due to slippage, rough terrain or a systematic error if not calibrated. Second, there is the laser scanner that provides distance information for a given scan range. Third, there is a camera that produces images and three-dimensional depth information. And fourth, we receive information from the robot's load-handling device. While the information from the load-handling device is usually handled

Figure 6.5.: Robot group shot

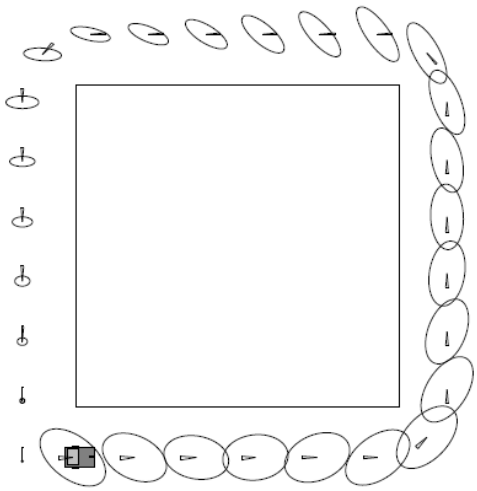

Figure 6.6.: Robot position estimates and corresponding covariance on a rectangular path without correction (Gutmann, 2000)

on its own (a set of light barriers denote if the robot is loaded or not), position calculation (called localization) is a more complex task that requires multiple sensors. Therefore, even for odometry that acts as a first guess of the robots position with a high frequency we do not use raw values, but a sensor combination. This first guess is corrected by laser information with a lower frequency. A setup where multiple sources are used to receive a better result than from a single source is called filter. Localization is typically based on either a Kalman Filter (Kalman, 1960; Maybeck, 1990) or a particle filter (Gordon et al., 1993).

The Kalman Filter is a highly efficient filter that keeps one single instance of its state vector of dimension n plus a covariance matrix of dimension n times n. The state vector clearly represents the value to be filtered, whereas the covariance matrix represents the uncertainty of the state vector. The filter works in two steps. The first step is a prediction, where steering inputs are used to predict the future state of the system. In its basic form the Kalman Filter allows linear prediction functions which can be computed by a matrix multiplication. More complex functions require the extended form of the filter. The variance of a prediction has to be known and is simply added. Thus, the prediction increases the covariance of our state. In the extended version the Jacobian of the prediction function is used. The second step is the correction step, where a measurement is used to correct the current state and to reduce the covariance. Similar to before, this is computed by an efficient matrix multiplication. Again the variance of the measurement has to be known. The lower the sensor's variance in comparison to the covariance of the state, the higher is the influence of this measurement on the state vector. Typically, a prediction is followed by a correction and again a prediction, so the whole filter consists of only a few matrix operations. As computation power is always limited in mobile robotics this is an attractive argument. Nevertheless, practical applications show that Kalman filters are difficult to apply as most of the systems are non-linear and therefore require an extended Kalman Filter that again requires a set of differentiations, which may be quite difficult to derive. Furthermore all movements and sensor readings have to be normally distributed. This leads to two problems. First, it might be difficult to derive the normal distributions of every filter participant and second, non-normally distributed inputs might deviate the filter. If applications of the Kalman Filter in mobile robotics always require high level experience and quite often a lot of corrections as well. Figure 6.6 shows a robot on a rectangular path with the corresponding covariance denoted by ellipses. As there is no correction, the uncertainty increases and increases.

Computationally more complex but a lot easier to apply is the particle filter (see Thrun et al. (2000)). Instead of holding a single instance of the state to be filtered in conjunction with a covariance matrix, the particle filter holds many instantiations of the state vector together with a weighting factor. Instead of describing the uncertainty in every dimension and dimension combination, we have a single factor that states the quality of a single state vector that we call particle. Again we have the two steps of prediction and correction. Within the prediction, we apply the control commands to every particle. As odometry is more precise than control commands, odometry information is used for prediction. In order to increase robustness, we do not only move the particles by the control commands but add some noise to every move. The motivation in behind is that if we have enough particles many might guess the correct movement. If we plot the particles on a two dimensional plane before and after prediction, we see an increase of the hull in the majority of cases. The bigger this hull, the bigger is the uncertainty of the state. As every particle is a guess of the state, the distribution of the samples models the uncertainty. In the correction phase, we apply the sensor input to every particle and check the fit at every particle's position. This fit is used to update the weight of the particle (the better the fit, the higher the weight increase). After a full circle of prediction and correction, the particles for the next round are typically sampled by their weight. This means, particles with a higher weight have a

higher chance of occurring once or more often within the particle set of the next round. This way more and more particles center around the true state, though it is necessary to ensure some diversity across the particles. The simplest strategy for deriving a single state estimate from the set of all particles, is taking the particle with the highest weight. This strategy has the clear drawback that best particle might jump along time. A more elaborated strategy is to cluster particles, calculate the weight of each cluster and calculate a weighted average of the cluster weighted highest. This leads to a much stabler system with affordable additional computation effort. All in all, in contrast to the Kalman filter, the particle filter is relatively simple to implement and can be applied to non-linear and non-gaussian problems with a arguable amount of additional computational costs. Furthermore, if the number of particles is high enough, it can be applied to the global localization problem, where a robot does not know the initial position. Figure 6.7 shows the case where the robot does not know its initial pose and thus spread all particles over the complete map. The longer the robot drives and thus the more observations are available, the less particle clusters remain. Finally, the true pose of the robot is found in the right picture.

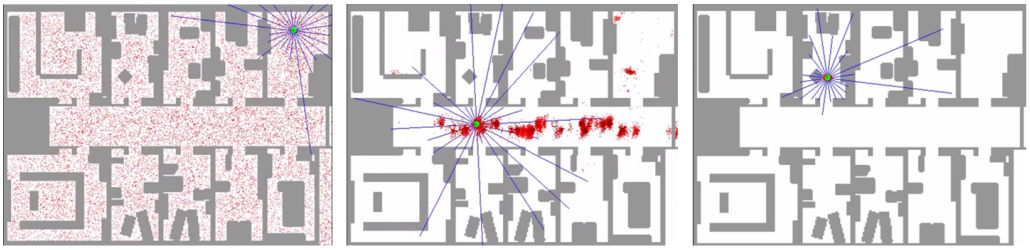

Figure 6.7.: Global localization using a Particle filter: from a complete unknown pose to a known one (Thrun et al., 2001). In the left picture, the robot does not know where it is. All particles are distributed over the whole map. In the middle, most of the particles have centered around possible poses. Finally, in the right picture, there remains one cluster around the true pose of the robot. The corresponding video can be found at `https://www.youtube.com/watch?v=nWvLX6xmoAw`

The next capability that is necessary for our system is object recognition. This can be realized by augmented reality AR tags (Azuma, 1997). We have a camera that captures pictures, tries to identify AR tags and calculates the AR tag position in world coordinates. An AR tag is defined by a special layout (typically black and white) that can be extracted by vision algorithms (see Figure 6.8). If the size of the tag is known, it can be placed in three dimensional space originating from the camera. The position of the camera itself can be retrieved by the position of the robot within its application scenario and the transformation from the robot's base coordinate system to the camera's camera system. Transformations from one coordinate system to another are simply done by matrix multiplications (Gentle, 2007). The quality of the position recognition depends directly on the quality of the localization. A simple implementation would use the last reading as the position of the object. More sophisticated approaches use averages or filters as described above. Both approaches have to ensure that object motions are treated correctly. For example if the robot stands still and it can watch the moving object, an average with a fixed number of samples would delay the object's motion as old and thus wrong measurements are incorporated into the result.

Additionally the environment recognition for navigation comprises the laser sensor and the three

Figure 6.8.: A milk box labeled with AR tags (Wolfram, 2011)

dimensional picture of the RGB-D camera. The aim of this sensor fusion is a map that serves as the robot's decision base for motion commands. The laser delivers a quite reliable view of the environment, but this view is limited to a plane. Objects that lie below or above this plane cannot be detected by the laser. This three-dimensional part is a specialty of RGB-D cameras. Figure 6.9 shows an office chair that is very difficult to detect by a two dimensional laser, but is easily detected by the camera. Similar conclusions are drawn in Quigley et al. (2009). So the combination of those two systems can be used to guide robots through dynamic environments.

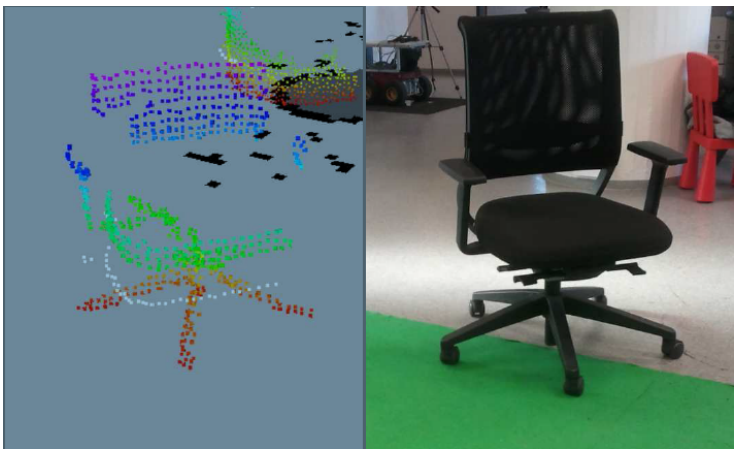

Figure 6.9.: An office chair from the view point of a RGB-D camera (Wolfram, 2011). The projection on the ground is shown by white points.

World Model Cache

All the processed information from the sensor services is fed to the World Model Cache. This World Model Cache is responsible for storing quantitative and qualitative information. This means the component plays a key role within the architecture as incorrect information leads to incorrect decisions and or to incorrect executions respectively. In contrast to a rather general description of the sensor services in the previous section, we will take a closer look at this part of the systems. The design and implementation was part of Máté Wolfram's master project, so an extended documentation can

be found in this master thesis (Wolfram, 2011).

The storing task can be refined in storing object data, their attributes, relations and aliases. Every object within the system has a unique ID. Additionally to this unique ID, an object may have several aliases. The aim of such aliases is to offer every module access data based on its internal name instead of converting the ID each time the world model is accessed. For example the augmented reality tag discovery system may be interested in AR tag ids only, whereas the AR does not make a difference for the reasoning system, but it is interested in the name and type of the object. Tag #64 could be a calculator within the reasoning system and this connection is stored in the World Model by aliases. This corresponds to the symbol grounding problem. A central management was chosen in order to ease interaction between modules and reduce error-proneness. Adding and removing aliases is done by standard ROS communication techniques.

Attached to an unique object ID, we find object data. In the simplest form this can be the pose of the object in any coordination system. Similar to resolving aliases, the World Model is responsible for doing coordinate transformations. Using ROS this is a relatively simple task, as there is a rather powerful transformation library (Foote, 2013). It keeps track of transformation matrices between coordination systems over time and offers methods to transform poses from one coordinate system into another coordinate system at any given time. By setting a fixed coordinate frame and a target time one can also transform from one frame into the same frame taking the movement of other frames into account. This is very convenient for filtering sensor measurements, such as read AR tags while the robot is moving. Additional object information is stored by attribute entries, where each attribute consists of a name and a value. An additional time stamp gives insight when the attribute was changed the last time. This time stamp can be used to ensure that only the latest information is stored (network latencies could mix up sequences between different observation sources) and to give every module the possibility to assess the value of the information. A special token *not_set* is used to store the information that no value is available. This feature is used by the belief management system in order to overwrite faulty observations.

Whereas attribute entries store information of one object, relations between objects are tackled by relation entries. Relation entries follow the subject, predicate, object triple, though the subject is not directly part of an relation entry but is implicitly available as the subject holds its relation entries. Typical predicate values are *in*, *closeTo* or *attachedTo* but of course they are not limited to spatial relations. An additional relation property allows for refining the relation. This could be a distance for *closeTo* or a side for *attachedTo*. Again there is a timestamp ensuring that only the latest assignment is stored. The services for retrieving information from the World Model are more or less straight forward. They provide the possibility to retrieve an object's pose in any coordinate frame, retrieve its attributes and relations and view a list of all aliases. Equivalently to retrieving information from the component, it offers services to add or update information. Every access either contains the unique ID of an object or an alias.

Though the world model is realized within the ROS framework, its internals are decoupled from ROS. It is implemented in Java and offers two kinds of visualization. First there is a tabular graphical user interface that lists all attributes and relations. Additional markers of the object's poses are published and can be visualized using ROS' standard visualization tool called rviz (see http://wiki.ros.org/rviz/UserGuide for details). This allows for fast online observation of execution (see Figure 6.11). The internal design of the World Model follows the model, view, controller principle and is shown in Figure 6.10.

Typically information is not fed directly into the World Model, but takes its route via an information

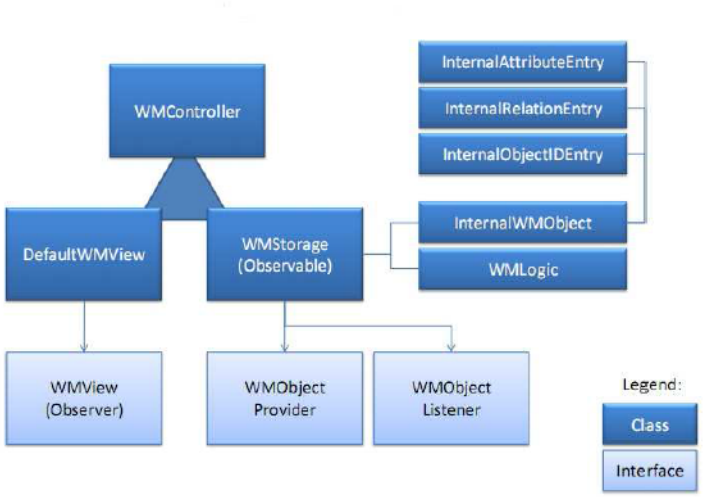

Figure 6.10.: The world model's internal design (Wolfram, 2011)

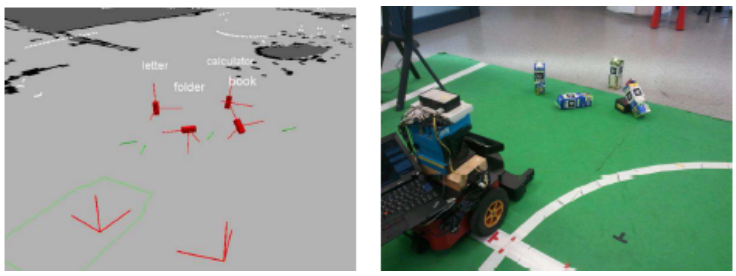

Figure 6.11.: A robot and a group of objects in visualization and reality (Wolfram, 2011)

translator. Such an information translator takes raw information from the sensor services and brings them into a format the World Model can process. Those modules are placed between sensor services and World Model forming a sensor pipeline as shown in Figure 6.12.

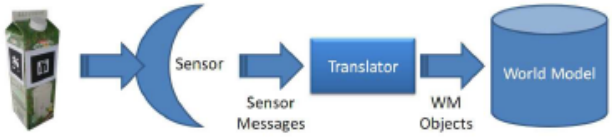

Figure 6.12.: Sensor pipeline from object into its world model representation (Wolfram, 2011)

Actuator Services

After describing the sensor pipeline, we cut short the typical data flow to the high-level system and back to the low-level system. This way we assume that some oracle will send the right decision at the right time and we can thus continue our system description of the low-level layer with the actuator services. Of course this module highly depends on the type of robot in use, but assuming we have a mobile robot within the delivery domain, we can identify several main tasks. Namely, have the robot drive from A to B, grasp objects and release them at certain target. Though different instantiations of this capabilities are necessary for different objects and transport routes, thinking in terms of standard containers and a standard procedure for picking up and putting down these containers, robots have the possibility to take over a broad range of transport tasks.

So, first we need the ability to pick or drop items. As they are highly interconnected we seek for a common solution for both of them. Disregarding the manual transfers where a human places objects onto a robot manually, we are interested in automatic object transfers. There are many different solutions for achieving this goal. The most common ones are grippers as we can find at the pioneer robots of the institute or a kind of a conveyor system that is mounted on the robot such as the robots of incubed IT. In presence of a conveyor it is easy to unload objects from the robot. If the counterpart does not provide active support to unload the object, it is sufficient to move the object off the conveyor by using the conveyor's motor. As soon as the object's center of mass is beyond the robot's footprint, the object is moved by gravity. If the counterpart is automated as well, some kind of signal exchange is used in order to start the transfer. The most elaborate way to transfer objects is to use a robotic hand that grasps the object on its own and moves it from the robot to the storage or vice versa. Depending on the degrees of freedom, this leads to a multi-dimensional planning problem. This is a complex problem but leads to the biggest flexibility in carrying out transfers. An overview of all of the transfer/hardware setups is shown in Figure 6.13. All setups have in common that a certain degree of fine positioning is necessary for the robot to be in the right spot when starting the transfer. This requires accurate information from the World Model and the right controlling algorithm behind it. During the transfer we typically use additional sensors such as light barriers or pressure sensors in order to back the execution.

The second important task besides transferring objects is to move from A to B. This task is required for different occasions. First, before picking an object, the robot has to move to its target location. Second, after picking the object we have to move to the target location. And optionally third, the

Figure 6.13.: Robots with different transfer capabilities

robot has to move to service locations such as charging stations when its battery is low. As charging stations may be located at transfer points this third move is optional. Another optional target could be a labeling machine. Typically, the navigation task is solved by two planners. A global planner is responsible for calculating a complete path from the robots current position to its desired target. The task of the local planner is to compute a motion command that is optimal for the local surroundings and the received path of the global planner. As in every planning problem, the representation of the environment plays a key role. In mobile robotics, costmaps have shown to be very useful. They represent the environment in a grid where each cell of the grid contains a cost value. The higher the cost, the less attractive this cell is to the robot and the more the robot will try to avoid this cell. The costs of the cells are received by the different sensors of the system. There are numerous ways to merge the sensors. A very efficient way to combine different sensors was developed within the ROS ecosystem. Here, different sensors serve different layers in a costmap. A set of different layer implementations exist in order to store and prepare the information as efficiently as possible. On a constant frequency the layers are merged to form a single costmap interface that is used by the planners. Special values within the costmap can be used to denote cells that must not be traversed such as walls or other obstacles, or cells that should not be traversed as there is no information about them, because they could not be reached during the mapping process. A map does not necessarily exist at the beginning of the robot's task, but can instead be created during the robots driving task. Yet, as map creation is done very quickly and the environment of the robot is known in the majority of cases, mapping is carried out before the robot starts driving. Figure 6.14 shows the costmap within a narrow environment. Light regions denote obstacles, whereas dark spots can be traversed by a robot. The descent from dark to light can be traversed, but is less preferable. The green dots refer to the current laser measurement of the robot. The robot itself is located in the lower and darker part of the image and is painted by a green polygon. The green line originating from the robot is the current global path of the robot.

Assuming that a more or less circular robot with radius r can turn on the spot, we can increase all obstacles by this radius r and thus reduce the global path-finding problem to finding a path of connected cells from the cell that contains the current location of the robot to the cell that contains the robot's goal position. Inflating the map by a fixed radius is much easier than checking the correct robot footprint at every position during the search process. So the search itself can be done by a standard search algorithm. Equipped with a path, we need to handle the path execution. In the simplest case this could be a controller function that keeps the robot on the path and stops in case of an obstacle. As this behavior does not resemble the expected behavior of an intelligent mobile robot, we need to come up with a more sophisticated solution. The next step could be an elastic band (Quinlan and Khatib,

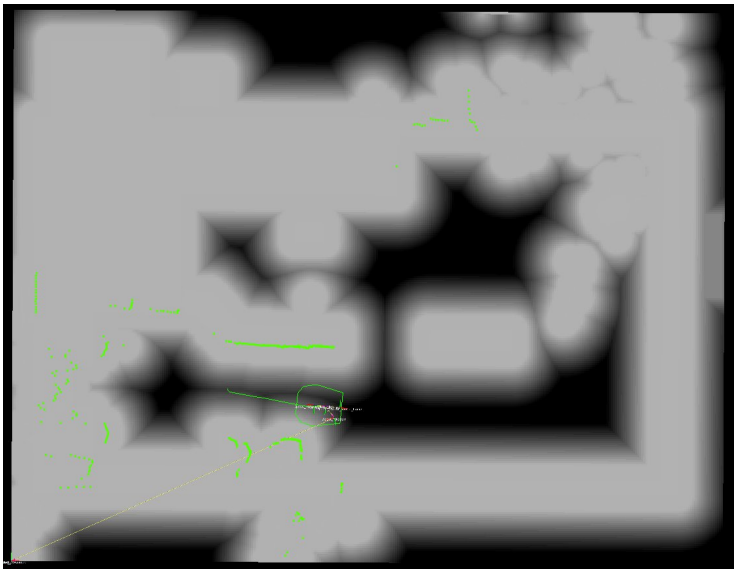

Figure 6.14.: Global Costmap with robot, path and laser points.

1993) implementation where a set of artificial forces is used to adopt the global path in a local window. The path starts with the global path generated by the path planner. Artificial forces that originate from obstacles are used to deform the path. The result is a collision-free, smooth path that can serve as input to a controller function. Another quite popular strategy for deriving velocity commands based on a given map and additional sensor input is the Dynamic Window Approach (Fox et al., 1997) or, with small differences in configuration, called Trajectory Rollout (Gerkey and Konolige, 2008). This approach takes the current velocity of the robot and calculates the reachable upper and lower velocity borders within a small time frame. Within those velocity borders a discrete (equally distributed) set of velocities is taken and applied to the current position and velocity of the robot for some time. The resulting trajectories are assessed according to a cost function and the velocity of the best trajectory is applied to the robot. Typical inputs to the cost function are distances to obstacles, the goal and the global path. Though the configuration of the cost function may not be easy, the method is very popular because this comprehensible method is very easy to implement and provides quick results in terms of an appealing robot navigation behavior.

6.3.2. IndiGolog Interface

Armed with knowledge about the high-level and the low-level system, we can now concentrate on the interface between the two layers. In Figure 6.3 this interface is denoted by a horizontal bar named IndiGolog Interface. The interface is responsible for communication between the layers. Therefore, it is split into three sub-components namely (1) an interface to trigger low-level action executions, (2) an interface to return abstract sensing results and (3) an interface to manipulate the continuous World Model Cache.

The action execution interface holds a mapping of IndiGolog actions to a concrete action server. Action servers are implemented in ROS and are responsible for execution in real world, where an execution can consist of many substeps. For example, the action $pickup(object)$ is executed by an action server by first, moving to an approach position that is stored within the World Model Cache

relative to the object. Second, opening the gripper, third, moving closer to the object till the robot touches the object, which can be identified through light barriers and finally closing the gripper again and lifting the object. Lifting the object is important so that the object is not missed during driving. The Action Server again forwards the sub tasks to the actuator services. As soon as the action is finished, its return value is sent back to the high-level system, which continues with consistency checking and decision-making.

The abstract sensing results interface is responsible for supplying the high-level system with qualitative information such as *is_at*(*chocolate*,*quality_control*). The grounding can be as sophisticated as described in Section 6.2, though practice shows that simple thresholds provide a good solution for a wide range of cases. Still we have to take care of imprecise measurements that lead to jumping values. For instance if the *chocolate* is located at the border between *quality_control* and *warehouse* its logical location may change quite often between the two rooms due to system noise. This problem was tackled by Steinbauer et al. (2005) by applying predicate hysteresis. In contrast to a threshold, the output of a hysteresis not only depends on the input but on the current output value as well. This concept is wide-spread in control theory and leads to delayed output reaction.

The World Model Module has a special instance within its system, the World Model Logic that is responsible for translating quantitative into qualitative information. Based on the sensor information that reaches the World Model via the Sensor Services a set of calculation instances derives many additional facts such as approach positions for a grasp. As the spatial context is crucial to the execution of the robot, most calculations are geometric ones. Next, all available information is fed into a set of conditions, that simplifies the available information into predicates, which can be used by the high-level system. Figure 6.15 gives an overview of the internal structure of the World Model Logic. New evaluations are forwarded to the high-level system. In order to be used by the high-level system, we further extend the IndiGolog main loop by *handle_exogenous_sensing*. The predicate *handle_exogenous_sensing* checks if there exists some sensing information provided by the low-level system but not actively triggered. If such a sensing information exists, the information is added to the history by sensing actions, similar to actively triggered sensing. The complete extended IndiGolog main loop looks as follows:

```
indigo(E,H) :- handle_exogenous(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_exogenous_sensing(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_consistency(H,H2), !, indigo(E,H2).
indigo(E,H) :- handle_rolling(H,H2), !, indigo(E,H2).
indigo(E,H) :- catch(final(E,H), exog, indigo(E,H)).
indigo(E,H) :- catch(trans(e,H,E1,H1), exog, indigo(E,H)),
  (var(H1) -> true ;
   H1 = H -> indigo(E1,H);
   H1 = [A|H] -> exec(A,S), handle_sensing(H,A,S,H2), indigo(E1,H2)).
```

Finally, the continuous World Model Cache manipulation interface is responsible for returning high-level conclusions back to the low-level system. For example, if an object was believed to be in the *warehouse*, but belief management corrected the location and stated that the object is within *quality_control*, the low-level system has to be informed about this change. Qualitative information can be seen as an abstraction of quantitative information, therefore the correct quantitative value can only be generated in the minority of cases. For this reason we try to generate a value that is as good as possible. Of course such functions could be arbitrary complex, but we seek for simple solutions that

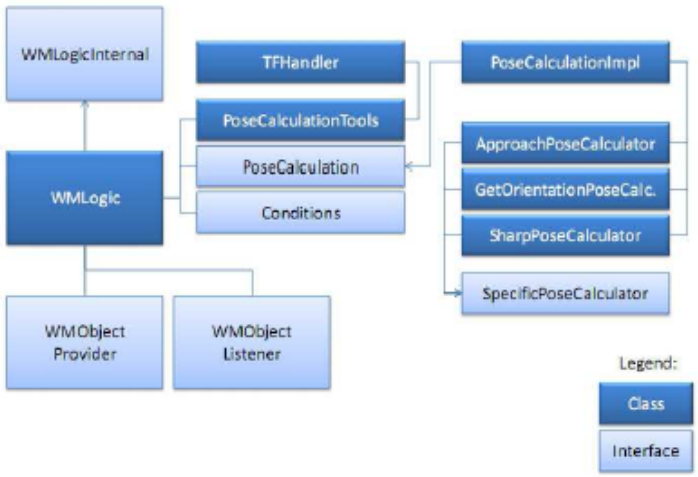

Figure 6.15.: Transforming quantitative into qualitative information by World Model Logic (Wolfram, 2011)

can be applied to the majority of cases. If an object is found to be in a room, we place the object at the center of the room. Besides adding new facts or negating existing ones, the high-level system can also conclude that there is no valid information about an object attribute. In this case the attribute is simply set to the special token *not_set*.

Chapter 7

Diagnosis Templates

Belief Management as presented in Chapter 4 uses a pool of possible hypotheses to find the correct belief. This pool is created by applying every exogenous action and every action variation. Obviously, the resulting number of hypotheses to be checked for consistency is very high (see Theorem 1). Furthermore consistency checks require a high number of computations as the initial state has to be checked for consistency, in addition, for every action we have to check if it is executable and if all invariants hold. The longer the history, the more computation is necessary for a single action, as property checking is done by regression. This means, every hypothesis, for which we can safely omit the consistency check is a big gain in terms of runtime. This chapter is based on work published in Reip, Steinbauer, and Ferrein (2012).

We have seen that most of the time just a single fault caused a system to be inconsistent, but also hypotheses with a high number of faults were generated. This observation is consistent with experiences the authors made in Mies et al. (2008). Equally, most of the generated (inconsistent) hypotheses have nothing to do with the observed inconsistency. For example we check if a missed grab of a $container1$ can be repaired by moving a totally uncorrelated $container2$ from one room to another. The larger the basic action theory and the more variations and exogenous actions exist, the more hypotheses are generated. Fritz (2009) states "We often face a lot of unpredictable exogenous events, most of which are unrelated to the problem the agent is facing and thus do not matter."

We think we can do a better job than the presented brute-force search. Every inconsistency we detect differs in at least the value of one predicate. For example, $is\_at(container1, quality\_control)$ is sensed to be true but was expected to be false. So we are interested in actions that have an influence on the *is_at* predicate bound to the objects $container1$ and *quality_control*. The effect of actions is defined within the basic action theory. This way we can precalculate all exogenous actions and action variations that can eventually lead to a change of the *is_at* predicate. The result of such a precalculation are action sequences that contain exactly one exogenous action or one action variation, which we call diagnosis templates. During runtime we simply fetch diagnosis templates that correspond to the current inconsistency, generate all hypotheses based on the templates and check them for consistency.

Before having a closer look on diagnosis templates, we examine the different fields of related research in Section 7.1. Then, in Section 7.2, we will define the diagnosis templates formally and present an algorithm to generate them. We will end the chapter with diagnosis template application in Section 7.3.

7.1. Related Research

First we will examine the field of plan diagnosis. Similar to our system, within plan diagnosis research a robot or an agent receives sensor input over time and has to compare the input to expected sensor inputs. In plan diagnosis, the input is a plan, typically in STRIPS notation (see Fikes et al. (1972)) or a similar representation. From this plan, the robot can calculate the expected outcomes after each action. A plan is typically seen as a system of components. So it is possible for every plan state to indicate whether this plan state is a normal state or an abnormal state as conventional model-based diagnosis (see Reiter (1987)). de Jonge et al. (2009) differentiate between two directions. First, primary plan diagnosis deals exactly with the problem of identifying health states for all plan states, where the set of different health states can be larger than just normal and abnormal. Second, secondary plan diagnosis aims at detecting the underlying reasons for misbehaving actions. They differentiate between agent, equipment and environment to be the cause of the problem. If an equipment is said to behave abnormally and the use of this equipment occurs in the remaining plan, a problem may occur.

Similar to classical model-based diagnosis one tries to find minimal diagnosis. This means one tries to find the set with a minimal number of abnormally qualified plan steps. Roos and Witteveen (2009) go one step beyond and define the notion of mini-maxi diagnosis. Whereas mini refers to the minimal diagnosis, maxi states that maximal informative plan diagnoses are demanded. Maximal informative refers to diagnoses with maximal predictive content. As the outcome of an abnormally qualified plan step is undefined, maximal predictive content is advantageous. The combination of the two criteria is done as follows: from all maxi diagnoses they chose the minimal one.

Very often plan diagnosis does not capture a single robot or agent but a team of them. In this case each agent is expected to monitor and diagnose its own actions. As the results of actions may not be directly observable or execution results of actions performed by other agents are not available yet, every agent has to keep a set of different possible action sequences (called trajectories) (see Micalizio and Torasso (2008)). Therefore every definition of an action explicitly contains a set of status variables that is used to define the preconditions and effects of an action.

Although plan diagnosis attempts health qualification for every plan step in contrast to a consistent action sequence that resembles all observations in diagnosis templates, there are a few similarities to diagnosis templates and their application. In plan diagnosis one needs to infer over action effects in order to find a minimal diagnosis, whereas in diagnosis templates we need this investigation to repair the current belief state. Furthermore due to partial observability in plan diagnosis and a set of consistent hypotheses in diagnosis templates both need to manage a pool of candidates.

Within the notion of fluent calculus, that is related to situation calculus, Fichtner et al. (2003) presented a formalization to derive explanations for unexpected situations. Their approach employs active sensing to minimize the impact of such events. Sensing actions and uncertainty is in the focus of Weld et al. (1998) as well. They extended GraphPlan to handle sensing actions and uncertainty, but they do not maintain the consistency of a system's belief. A similar approach was taken by Dearden and Boutilier (1994). They combined planning and uncertainty. Murphy and Hershberger (1999) provide another example for handling sensing failures. Hardware failures in general, are tackled by Muscettola et al. (1998). Based on model-based reasoning, their architecture detected failures in the hardware of the space probe Deep Space One and recovered from them. This system finds the best current operation based on model-based programming. The idea is to define only safe operation requirements rather than hand coding all recovery strategies. Especially in dynamic real-time environments the combination of planning and uncertainty is crucial. For example Ferrein et al. (2004) propose a sys-

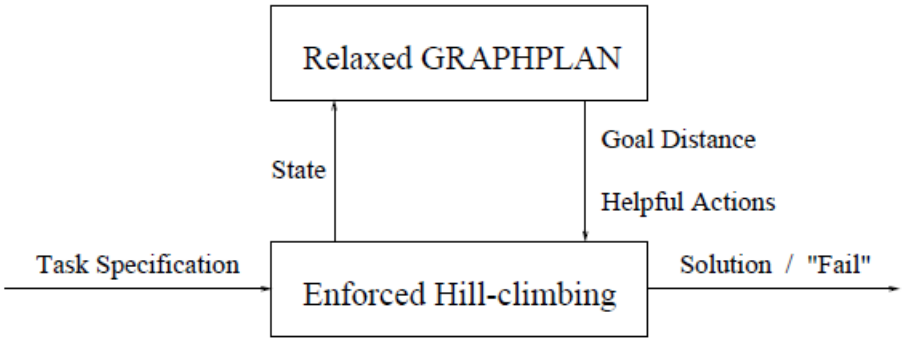

Figure 7.1.: Fast forward's base system architecture (Hoffmann and Nebel, 2001)

tem which keeps a few alternatives and chooses the best one according to a performance measure. Another robot-related diagnosis approach was presented by Liu and Coghill (2004). They introduced a qualitative model of a planar robot's kinematics which can be used for diagnosis purposes.

Within model-based diagnosis, diagnosis templates share a relation to kernel diagnoses. A kernel diagnosis is a partial diagnosis with the property that the only partial diagnosis which it covers is itself (De Kleer et al., 1992). Thus, kernel diagnoses can be seen as compact descriptions that cover inconsistent observations. Similar diagnosis templates describe the root causes to unexpected observations. This means every diagnosis template consists of one action that was not intended by the executing agent or robot. In Williams and Ragno (2007) kernel diagnoses are combined with best-first search. Enumerating all diagnoses is very expensive in terms of computation costs. For this reason a minimal number of solutions is computed with the goal to identify them as quickly as possible.

Diagnosis Templates exploit domain specific properties in a general form, so that the robot's belief can be corrected much faster than without taking the domain into account. The idea is not limited to belief management, but is already used heavily in planning. In both heuristics select relevant actions for the particular problem instead of taking all actions into account. The first successful application was done within the fast forward planning system (Hoffmann and Nebel, 2001), that won the AIPS-2000 planning competition. It should be noted that besides the authors exceptional knowledge of planning algorithm, the design of the algorithm was driven by observing the planner's behavior doing benchmark tasks instead of theoretical considerations. The architecture (see Figure 7.1) consists of two main parts. Starting from the current state, an optimized relaxed version of graphplan serves as heuristic method. It provides a goal distance and a set of helpful actions. The main search algorithm uses enforced hill-climbing. The goal distance is used as heuristic value in a natural way. The helpful actions are used for pruning to keep the observed search space as small as possible. The interplay between hill-climbing and graphplan is continued until the search succeeds or fails.

The idea of exploiting domain specific properties for heuristic based planning was extended in the fast downward algorithm in (Helmert, 2006). The name of the algorithm still contains the "fast" from (Hoffmann and Nebel, 2001). The "downward" derives from a hierarchical strategy where top-level goals decompose the problem until every subproblem is a basic graph search task. The algorithm carries out three steps, (1) translation, (2) knowledge compilation and (3) search as shown in Figure 7.2. The translation part converts the planning problem that is given in PDDL2.2 (see Fox and Long (2003); Edelkamp and Hoffmann (2004)) into an internal structure, namely a multi-valued plan-

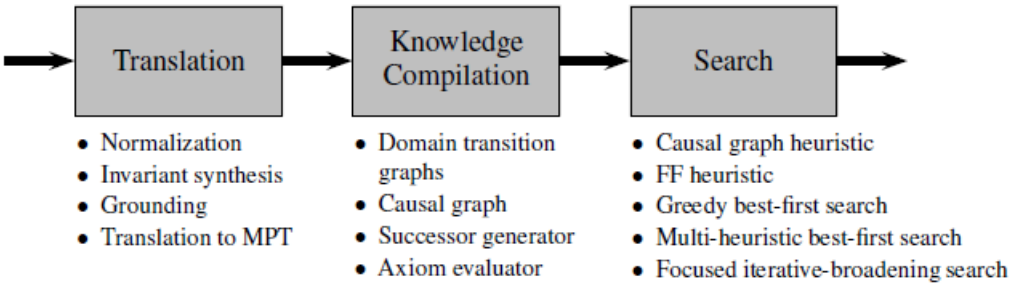

Figure 7.2.: Fast downward's execution steps (Helmert, 2006)

ning task as presented in Edelkamp and Helmert (2000). This means instead of having multiple binary state variables such as *container*1_*in_a*, *container*1_*in_b*, ..., *container*1_*in_z* they use one variable *container*1 $\in \{a,b,\ldots,z\}$. Based on this translation, the multi-valued planning task is defined as a 5-tupel $\Pi = \langle V, s_0, s_*, A, O \rangle$, where V is a finite set of state variables, each with its finite domain D_v, s_0 is the initial state, s_* the goal, A is a set of multi planning task axioms over V and finally O the set of operators over V with preconditions and effects. The knowledge compilation forms a kind of preprocessing step. It converts the planning task into data structures that are useful during search. First, we have domain transition graphs for each state variable. A domain transition graph contains the dependencies between the values of one state variable. Second, similar to domain transition graphs, causal graphs contain the dependencies between different variables. Third, successor generators are data structures that compute the set of applicable operators in a given world state. Fourth, the values of derived variables are computed by axiom evaluators. The search itself can be done by three different search algorithms, which are scheduled in parallel. First, Greedy best-first search orientates on the algorithm with same name from (Russel and Norvig, 2003) with a few adaptions to incorporate the data structures from the knowledge compilation step. Second, multi-heuristic best-first search is a variation of the first algorithm where multiple heuristics are combined to guide the search. Third, focused iterative-broadening search is an experimental algorithm that does not use heuristics, but instead narrows the set of operators by applying causal graphs. To this extend, the fast downward algorithm is important to diagnosis templates as the knowledge compilation step inspired a similar step for diagnosis templates.

7.2. Diagnosis Template Definition

Diagnosis Templates can be seen as patterns that guide the hypotheses generation process and prune hypotheses that have no effect on the current belief inconsistency. Figure 7.3 gives an overview of the different hypothesis categories. The dimensions of each category do not relate to any calculation and differ from domain to domain, but they reflect the personal assessment based on experiences. The large category in blue corresponds to all hypotheses. This category, contains every possible combination of action variations and exogenous actions, irrespective of whether the hypothesis is consistent or not. A subset of all hypotheses is the set of consistent hypotheses. This set is generally much smaller, but nevertheless can be very large, especially if the domain contains a lot of exogenous action with very specific effects, like within an in-house delivery domain the movement of airplanes has no influence on any delivery action, but can be inserted after every action execution. This set

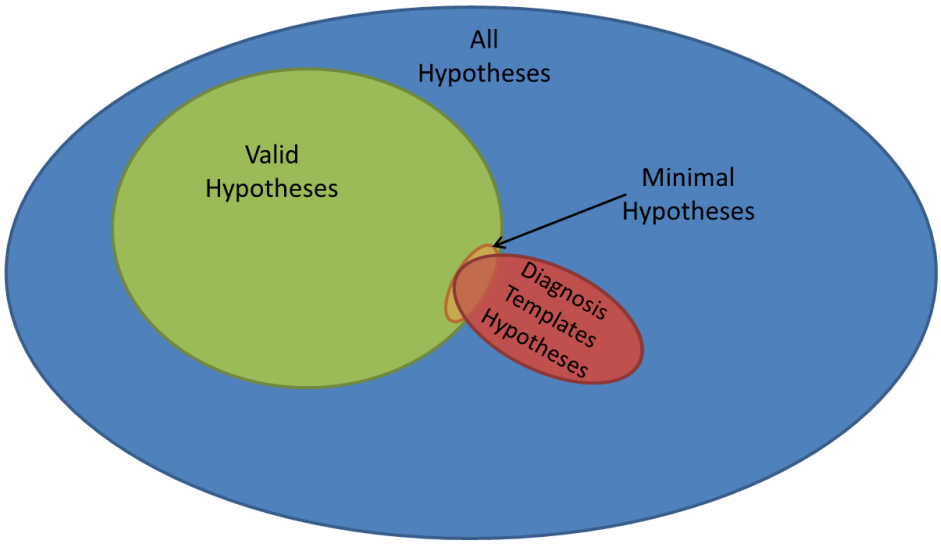

Figure 7.3.: Overview of hypotheses categories. The desired set of minimal consistent hypotheses is just a small subset of all hypotheses.

contains valid hypotheses, but most of the entries may over-explain the given inconsistency. Our goal set are consistent hypotheses that are minimal in the sense that for every hypothesis there is no consistent hypothesis that is an action subset. Diagnosis Templates do not approach the task from the consistency side, but from a problem-oriented side. Thus, they allow for generating hypothesis that all center around the given inconsistency. Still many of them may be inconsistent, so our solution is only a subset of all hypotheses generated by diagnosis templates.

Please note the small discrepancy between the set of minimal consistent hypotheses and the intersection of hypotheses derived by diagnosis templates and consistent hypotheses. This originates from the fact that hypotheses generated from diagnosis templates contain exactly one change, that is either exactly one action variation or exactly one action insertion. If there are inconsistencies that are due to two or more changes, such cases are not handled by diagnoses templates. For this reason, an approach driven by diagnosis templates is incomplete. However, this limitation is not so hard. First, the diagnosis templates approach can easily be extended to handle multiple enumerable faults. And second, of course this depends on the domain, but it is not very often that two or more connected faults arise at the same time, which makes it possible to differentiate them.

So, after sketching the motivation of diagnosis templates and determining their place within the space of all hypotheses, we seek for a formal definition. Therefore, we will provide a few definitions that in conjunction will provide the definition of diagnosis templates. Diagnosis templates form a set of explanations, how to derive some fluent F in terms of action sequences. The dependencies between actions and fluents are extracted from the basic action theory that contains a domain description. These dependencies can be exploited to form general explanations for a fluent. Consecutively, we define the dependencies formally (see Reip et al. (2012)).

Definition 16. *Let $F(\vec{x},s)$, be a fluent and $a(\vec{z})$ be an action. Then, we define $DepAct_F(\vec{x},a(\vec{z}),s)$ indicates whether or not fluent F directly depends on action $a(\vec{z})$ in situation s.*

$$DepAct_F(\vec{x},a(\vec{z}),s) \doteq Poss(a(\vec{z}),s) \wedge \neg F(\vec{x},s) \wedge F(\vec{x},do(a(\vec{z}),s)).$$

So action $a(\vec{z})$ leads to fluent $F(\vec{x})$. This is true, iff fluent $F(\vec{x})$ holds after but not before executing action $a(\vec{z})$.

For example, if the action $pickup(obj)$ is executable in some situation s and the robot is not in possession of the object as indicated by $\neg has\_object(obj,s)$, then $DepAct_{has\_object}(obj,pickup(obj),s)$ holds in situation s, because the robot possesses the object after picking it up ($has\_object(obj,s')$ with $s' = (pickup(obj,s))$). For reasons of readability we omit the negative fluent cases in these definitions. They are defined analogously. Next, we apply the definition from above to a set of fluents.

Definition 17. *Let $F_1(\vec{x_1},s),\ldots,F_n(\vec{x_n},s)$, be fluents and $a(\vec{z})$ be an action. Then the fluents $F_1,\ldots,F_n$ directly depend on an action $a(\vec{z})$ in situation s iff at least one of the fluents $F_1(\vec{x_1}),\ldots,F_n(\vec{x_n})$ depend on action $a(\vec{z})$:*

$$DepAct^{\oplus}{}_{F_1,\ldots,F_n}(\vec{\chi},a(\vec{z}),s) \doteq \bigvee_{i=1}^{n} DepAct_{F_i}(\vec{x_i},a(\vec{z}),s)$$

The vector $\vec{\chi}$ serves as abbreviation for $\vec{x_1},\ldots,\vec{x_n}$.

For example, the predicate $at(r,s)$ directly depends on the action $goto(r)$. This means, every set containing the predicate $at(r,s)$ directly depends on the action $goto(r)$. After defining if a fluent depends on an action, we change the direction and investigate under which circumstances an action or the execution of an action depends on a fluent.

Definition 18. *Let $F(\vec{x},s)$, be a fluent and $a(\vec{z})$ be an action. Then, $DepPoss_F$ depends if the execution of action $a(\vec{z})$ depends on fluent $F(\vec{x})$.*

$$DepPoss_F(\vec{x},a(\vec{z}),s) \doteq Poss(a(\vec{z}),s) \wedge F(\vec{x},s) \wedge \neg Poss(a(\vec{z}),do(switch\_val_F,s))$$

A set of special actions $switch\_val_F$ for every fluent F is assumed to be part of the basic action theory. These actions alter the value of the fluent F. The dependency holds if action $a(\vec{z})$ can be executed in the presence of fluent $F(\vec{x})$ but not in the absence of the fluent, which is enabled by the special action $switch\_val_F$.

This dependency is true for action $putdown(obj)$ that depends on the fluent $has\_object(obj)$. The next dependency relation to be examined is the dependency of a fluent $E(\vec{y})$ on fluent $F(\vec{x})$ via action $a(\vec{z})$.

Definition 19. *Let $E(\vec{y},s)$ and $F(\vec{x},s)$, be fluents and $a(\vec{z})$ be an action. Then, $DepEffect_{E,F}$ denotes iff fluent $E(\vec{y})$ is an effect of action $a(\vec{z})$ and depends on fluent $F(\vec{x})$:*

$$DepEffect_{E,F}(\vec{y},\vec{x},a(\vec{z}),s) \doteq F(\vec{x},s) \wedge E(\vec{y},do(a(\vec{z}),s)) \wedge \neg E(\vec{y},do([switch\_val_F,a(\vec{z})],s)).$$

For example, the predicate $is\_at(obj,r,s')$ for an object obj, a room r and a situation s' is changed by the action $goto(r)$ if the predicate $has\_object(obj,s)$ holds in the preceding situation ($s' = do(goto(r), s)$). Thus $DepEffect_{is\_at,has\_object}(obj,r,obj,goto(r),s)$ holds. This effect dependency can again be generalized to a set of fluents. This means the predicate becomes true if one fluent out of a set of fluents depends on the given fluent $F(\vec{x})$ via action $a(\vec{z})$.

Definition 20. *Let $E_1(\vec{y}_1,s),\ldots,E_n(\vec{y}_n,s),F(\vec{x},s)$ be fluents and $a(\vec{z})$ be an action. Then, $DepFl_{E_1,\ldots,E_n,F}(\vec{y}_1,\ldots,\vec{y}_n,\vec{x},a(\vec{z}),s)$ states that the fluents $E_1,\ldots,E_n$ are an effect of action $a(\vec{z})$ and depend on $F(\vec{x})$.*

$$DepFl_{E_1,\ldots,E_n,F}(\vec{y}_1,\ldots,\vec{y}_n,\vec{x},a(\vec{z}),s) \doteq DepPoss_F(\vec{x},a(\vec{z}),s) \vee DepEffect_{E_1,F}(\vec{y}_1,\vec{x},a(\vec{z}),s) \vee \ldots \vee DepEffect_{E_n,F}(\vec{y}_n,\vec{x},a(\vec{z}),s)$$

Equal to the previous *is_at* example, we can add any predicate $any(\vec{y})$ and $DepFl_{is\_at,any,has\_object}(obj,r,\vec{y},obj,goto(r),s)$ holds.

Equipped with a set of dependency definitions between fluents and actions, we are ready to define a dependency tree for a fluent F. A dependency tree contains all action sequences, starting with exactly one action variation or one action insertion that explains this fluent.

Definition 21. *An edge-labeled and node-labeled non-recursive, fully expanded tree T_F is a fluent dependency tree for fluent F iff the following properties hold:*

1. *The root is labeled F.*
2. *If n is a node of T_F with label $F_1,\ldots,F_n$, then every edge from node n to successor node n_{succ} is labeled with an action term a with $\exists s,\vec{x}_1,\ldots,\vec{x}_n,\vec{z}.DepAct^{\oplus}{}_{F_1,\ldots,F_n}(\vec{x}_1,\ldots,\vec{x}_n,a(\vec{z}),s)$.*
3. *The successor node n_{succ} with edge a of an arbitrary node n labeled $E_1,\ldots,E_m$ in the tree is labeled the following way:*
   $$n_{succ} = \begin{cases} \$ & \exists s,a_1,\vec{z}.Varia(a,a_1(\vec{z}),s) \vee Inser(a,s)) \\ F_1,\ldots,F_n & \Phi \end{cases}$$
   where $F_1,\ldots,F_n$ contain all fluents with Φ abbreviating the formula
   $$\begin{aligned} &\exists \vec{x}_1,\ldots,\vec{x}_m,\vec{y}_1,\ldots,\vec{y}_n,\vec{z},s.DepFl_{E_1,\ldots,E_m,F_1}(\vec{x}_1,\ldots,\vec{x}_m,\vec{y}_1,\vec{z},a,s) \\ &\wedge \ldots \wedge DepFl_{E_1,\ldots,E_m,F_n}(\vec{x}_1,\ldots,\vec{x}_m,\vec{y}_n,\vec{z},a,s) \end{aligned} \tag{7.1}$$
4. *If n is labeled with \$, it is a leaf.*
5. *Let n be a node in T_F. Then, define $H(n)$ as the set of edge labels from n to the root.*

We assume a finite number of actions. The fluent dependency tree $T_{\neg F}$ is defined analogously.

Every edge is labeled by an action. Edges to the root node, are labeled with actions that directly lead to the root fluent. The preconditions of these actions are achieved by the fluents that label the next node level. The algorithm for generating a fluent dependency tree for fluent F directly follows the definition and is shown in Algorithm 2 (Reip et al., 2012). The main input is the target fluent. The initially empty sets of edges E and vertices V form the resulting tree. Additionally, we feed the algorithm with a few additional sets that tremendously facilitate the readability of the algorithm. The set *Actions* and the set *Fluents* can be directly derived from the basic action theory. Please note that we assume finite domains here. The set *Changes* combines all variations and insertions (see Chapter 4) into a single set. The set $DepAct^{\oplus}(E_1,\ldots,E_n)$ follows from the corresponding predicate and contains all actions that lead to $E_1 \vee \ldots \vee E_n$. Equally, the set $DepFl(E_1,\ldots,E_n,a)$ enumerates all fluents, so that action a leads to $E_1 \vee \ldots \vee E_n$. This again is derived from the identically-named predicate.

Algorithm 2: gen_dependency_tree(root, V, E)

input : parent (root) node $(E_1, \ldots, E_n)$
the initially set V of vertices
the initially empty set E of labeled edges
$V \times V \times Label$
a set *Actions* of all actions
a set *Fluents* containing all fluent formulas
a set *Changes* containing all variations and insertions
a set $DepAct^{\oplus}(E_1, \ldots, E_n)$ containing all actions on which fluents $E_1, \ldots, E_n$ depend
a set $DepFl(E_1, \ldots, E_n, a)$ containing all fluents on which fluents $E_1, \ldots, E_n$ depend through action a

```
1  foreach a ∈ Actions do
2      if a ∈ DepAct⊕(E_1,...,E_n) then
3          if a ∈ Changes then
4              V ← V ∪ {new_node_$};
5              E ← E ∪ {root, new_node_$, a};
6              continue;
7          end
8          child ← ∅ ;
9          foreach f ∈ Fluents do
10             if f ∈ DepFl(E_1,...,E_n, a) then
11                 child ← child ∪ {f};
12             end
13         end
14         if child ≠ ∅ ∧ ¬Cycle(child) then
15             V ← V ∪ {node_child};
16             E ← E ∪ {root, node_child, a};
17             gen_dependency_tree(node_child, V, E);
18         end
19     end
20 end
21 if |root| = 1 then                                   // no child node
22     delete root;
23 end
```

The algorithm starts with the target fluent as first node caption, e.g. *gen_dependency_tree*(*is_at*$(obj, r), [], [])$ for generating the fluent dependency tree of fluent *is_at*. This node is expanded recursively in a depth-first manner until a valid leaf is found, a cycle is detected by means of node captions or there is no further valid expansion. A cycle can be interpreted as follows: in order to reach a specific (set of) fluent(s) the same set of fluents has to occur already before. Thus, in case of a cycle, the node is deleted. Due to the recursive nature of the algorithm, it is possible that the parent node including its whole branch is deleted as well. The algorithm starts by probing every action if it leads to at least one of the fluents of interest (Lines 1-2). If the action is already a variation or insertion (Line 3), it is added to the tree by adding a new leaf node with label \$ to the set of nodes and a vertex from the current node to the leaf node labeled with the current action to the set of actions. Then the next action is checked. Otherwise we create an initially empty label for the new child node in Line 8. Next, we check every fluent if it dominates the action's precondition (Line 10). If yes, the fluent is added to the child label (Line 11). After checking all fluents, the child's label is finished. In order to be added to the tree it has to pass two tests. First, the label must not be empty and second, the label has to differ from all its parents' labels (Line 14). If the criteria are met, the label is added to the tree and the algorithm is called recursively in Line 17. After returning from the recursive call, it is checked

if the node could be successfully expanded. If not, the node itself is also deleted (Line 21).

In order to clarify the algorithm, we will show the fluent dependency tree generation based on the example of the fluent $is\_at(obj,r)$. So we start with a root label $is\_at(obj,r)$. Then we check all actions if they fit. Within the action check, the action $goto(r)$ is the first action that influences the root predicate ($DepAct^{\oplus}_{is\_at}(obj,r,goto(r),s)$ holds). As $goto(r)$ is neither a variation nor an insertion, we try to create a child label and can identify $has\_object(obj)$ as the only influencing fluent - $is\_at(obj,r)$ which is only influenced by action $goto(r)$ if the robot is carrying the object obj. Thus, we add a new node $has\_object(obj)$ that connects to the root node via an edge labeled $goto(r)$. This intermediate step is shown in Figure 7.4.

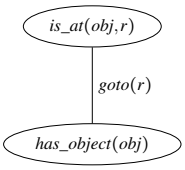

Figure 7.4.: Dependency tree after the first node expansion.

After successfully adding the node $has\_object(obj)$ to the tree, we continue the depth-first search and try to expand the new node. The first influencing action we can find on this node is $pickupWrongObject(obj)$. This action is already a variation of the action $pickup(obj)$ and thus a change action. Therefore, we add a new leaf with the caption \$. The edge between the leaf and the $has\_object(obj)$ node is labeled with the action name, namely $pickupWrongObject(obj)$. This state of the algorithm is shown in Figure 7.5.

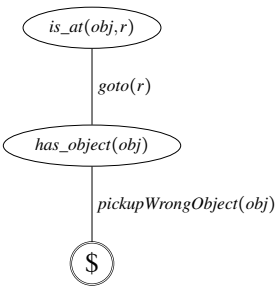

Figure 7.5.: Dependency tree after first leaf is found.

The action $pickupWrongObject(obj)$ was successfully added with a new leaf, so we can continue checking further actions. The next action that influences our current node $has\_object(obj)$ is the action $pickup(obj)$. This action is no variation or insertion, so we look for enabling fluents. And we find them as the conditions below hold.

- $DepFl_{has\_object,is\_at}(obj,obj,r_1,pickup(obj),s)$,
- $DepFl_{has\_object,at}(obj,r_2,pickup(obj),s)$ and
- $DepFl_{has\_object,\neg has\_object}(obj,obj,pickup(obj),s)$.

A label consisting of three predicates is clearly not empty and furthermore this label does not occur within the current branch yet. We can again add a new node with label $is\_at(obj,r_1)$,$at(r_2)$, $\neg has\_object(obj)$ connected to the previous node by edge $pickup(obj)$. The result of this step is shown in Figure 7.6.

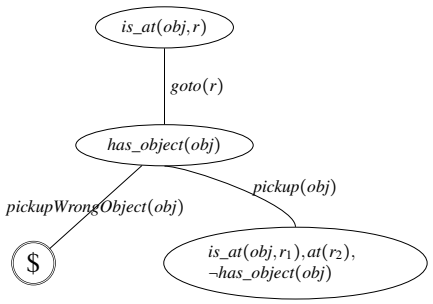

Figure 7.6.: Fluent Dependency Tree after expanding to the second level.

In the next step, we try to expand the new node $is\_at(obj,at(r_1),\neg has\_object(obj)$. Again we check all actions for its influence on the given predicate set. First in an arbitrary order, we identify the action $exogMoveObject(obj)$. As the action is an insertion, we can add a second leaf with caption \$ attached through an edge $exogMoveObject(obj)$. The next identified action is $goto(r_1)$ that leads to the fluent $has\_object(obj)$. Though this child caption does not pass the cycle test, as the same caption already occurs at the parent of the current node. Therefore, this child is not appended. Another fitting action is $putdown(obj)$ that leads to the child label $has\_object(obj)$ through $DepFl_{is\_at,at,\neg has\_object,has\_object}(obj,r_1,r_2,obj,obj,putdown(obj),s)$. This potential child node meets the same fate as the child node before. It does not pass the cycle test and therefore is not added. Figure 7.7 shows this intermediate step with our new leaf. The two rejected child nodes are indicated by dashed ellipses.

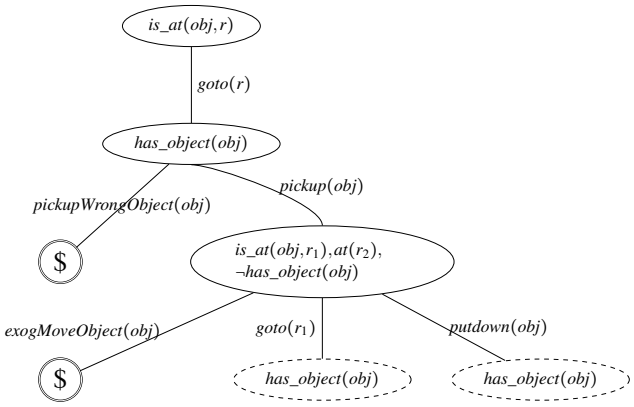

Figure 7.7.: Fluent Dependency Tree after adding the second leaf. Rejected nodes are drawn with dashes.

Now, there is no other valid expansion for node $is\_at(obj,r_1)$, $at(r_2)$, $\neg has\_object(obj)$ so the recursive call returns to node $has\_object(obj)$. As there is no valid expansion for this node as well, we return to the root node $is\_at(obj,r)$. This node is not fully expanded yet. There is one more influencing action namely $exogMoveObject(obj,r)$. This action is an insertion so we can add a third leaf. Now as the root node cannot be further expanded, the algorithm finishes. The complete fluent dependency tree $T_{is\_at(obj,r)}$ is shown in Figure 7.8.

Generating fluent dependency trees that form explanations for a given fluent is an important subtask on the way to generating diagnosis templates. For a node n of $T_{F(\vec{x})}$, $H(n)$ is defined to be the sequence

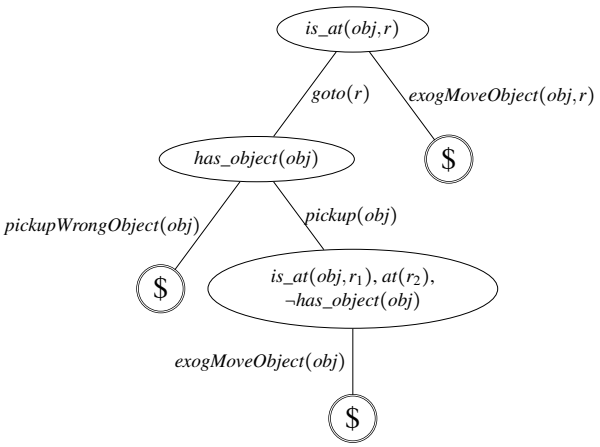

Figure 7.8.: Fluent dependency tree for fluent $is\_at(obj,r)$.

of edge labels on the path in $T_{F(\vec{x})}$ from node n to the root. If n is a leaf, $H(n)$ contains exactly one variation or insertion (cf. Definition 21 in (Reip et al., 2012)). The following definition shows that diagnosis templates can be directly inferred from fluent dependency trees.

Definition 22. *Let $T_{F(\vec{x})}$ be a dependency tree for fluent $F(\vec{x})$. For every leaf node $n_1,\ldots,n_m$ of $T_{F(\vec{x})}$, $dt_i^{F(\vec{x})} = H(n_i)$ with $i \in \{1,\ldots,m\}$ is a Diagnosis Template. $DT^{F(\vec{x})}$ denotes the set of all Diagnosis Templates for a fluent F.*

The fluent dependency tree $T_{is\_at(obj,r)}$ from the previous example leads to three Diagnosis Templates:

- $[pickupWrongObject(obj), goto(r)]$,
- $[exogMoveObject(obj), pickup(obj), goto(r)]$ and
- $[exogMoveObject(obj)]$.

For *Diagnosis Template* $[a_1,\ldots,a_n]$, by definition, action a_1 is always a variation or an insertion. Actions $[a_2,\ldots,a_n]$ act as a kind of condition or enabler for action a_1 to possibly fulfill $F(\vec{x})$. For instance, if $is\_at(obj,r)$ does not hold, it is achieved through $pickupWrongObject(obj)$ only if $pickupWrongObject(obj)$ is followed by the action $goto(r)$. In the following theorem we show that every action sequence that consists of exactly one variation or insertion and leads to a fluent $F(\vec{x})$ can be reduced to a *Diagnosis Template* $dt_i^{F(\vec{x})} \in DT^{F(\vec{x})}$. The theorem guarantees that all possible explanations for $F(\vec{x})$ occur in the *Diagnosis Templates*.

Such a sequence is called a minimal explanation, if it is derived in the following way: an executable and consistent action sequence $[a_1,\ldots,a_n]$ deriving fluent $F(\vec{x})$ with $\exists s.\neg F(\vec{x},s) \wedge F(\vec{x}, do([a_1,\ldots,a_n],s))$ is first reduced to $[a_k,\ldots,a_m]$ with $k \geq 1 \wedge m \leq n$. a_k is further required to be an insertion or a variation and a_m has to lead to the fluent $(\exists s.\neg F(\vec{x},do([a_1,\ldots,a_{m-1}],s)) \wedge F(\vec{x}, do([a_1,\ldots,a_m],s)))$. Second, we remove all actions a_i with i from m to k that are not necessary to achieve fluent $F(\vec{x})$, denoted by $\exists s.F(\vec{x},do([a_k,\ldots,a_{i-1},a_{i+1},\ldots,a_m],s))$. The resulting sequence must still be executable and consistent. m is chosen in a way that the final length of the sequence is minimal. Thus, such sequences start with an insertion or variation and consist only of actions that are necessary to fulfill fluent $F(\vec{x})$. These sequences form exactly the diagnosis templates. So the important question is, do we miss any hypotheses if we restrict our search to diagnosis templates.

Theorem 3. *The set of Diagnosis Templates $DT^{F(\vec{x})}$ contains all minimal action sub-sequences with exactly one insertion or variation leading to situations s^*, where $F(\vec{x},s^*)$ holds. Thus, every action sequence $a_1,\ldots,a_n$ with $\exists s.\neg F(\vec{x},s) \wedge F(\vec{x},do([a_1,\ldots,a_n],s)$ and exactly one variation or insertion is a super-sequence of one $dt_i^{F(\vec{x})} \in DT^{F(\vec{x})}$.*

Proof. The proof is by contradiction. We try to find an executable and consistent action sequence $[a_1,\ldots,a_n]$ with exactly one variation or insertion that leads to fluent $F(\vec{x})$ without any sub-sequence being a *Diagnosis Template*. We show that the sequence $[a_1,\ldots,a_n]$ already assumed to be a minimal explanation either contains an action that is not part of the fluent dependency tree $T_{F(\vec{x})}$ or that $[a_1,\ldots,a_n]$ does not cover a full path within $T_{F(\vec{x})}$.

The root of the fluent dependency tree is labeled $F(\vec{x})$ (rule 1 of Definition 21). The last action a_n directly leads to $F(\vec{x})$, so it fulfills $\exists s.DepAct^{\oplus}{}_F(\vec{x},a_n,s)$. Thus, there is an edge from the tree root labeled a_n (rule 2 of Definition 21). The node following this edge is labeled with fluents that either enable a_n's precondition or a_n's effect $F(\vec{x})$ (rule 3 of Definition 21). Analogously, we know from the action sequence that a_{n-1} is necessary so that a_n can be executed (precondition), or it effects $F(\vec{x})$. Hence, we know that action a_{n-1} is a valid edge label of the tree, too. Equivalently we know for every action a_i with $i = n-2,\ldots,1$ that there is an action a_j with $j > i$ whose precondition or effect is enabled by a_i. Thus, every action $a_i \in \{a_1,\ldots,a_{n-2}\}$ is also part of the diagnosis tree. As a_1 is an insertion or variation, we derive from rule 3 and 4 of Definition 21 that it leads to a leaf n. So $H(n)$ forms a full valid path through the fluent dependency tree. Hence, this contradicts the assumption, as there is no consistent executable action sequence leading to fluent $F(\vec{x})$ that is not a super-sequence of an *Diagnosis Template*. □

Proving, that we can reduce every minimal and valid hypothesis to a diagnosis template, gives us the possibility to reduce the search space for a valid hypothesis tremendously. If we recall the hypotheses categories from Figure 7.3 and add the restriction of single faults, we can see that the overlap of diagnosis template hypotheses and the valid hypotheses form the set of minimal valid hypotheses. So instead of checking all hypotheses, we can narrow our search to the subspace of diagnosis template hypotheses. Please note that the difference in size depends on the domain and in an specially designed domain there may be no difference. Although quite often we see that the bigger the domain, the fewer the interconnections, like the English language. Generally, natural language processing matrices are very sparse (see Jurafsky and Martin (2000)).

7.3. Diagnosis Template Application

Equipped with the generation of diagnosis templates we will investigate how these diagnosis templates can be used to diagnose inconsistent situations. A situation is denoted by the initial situation plus an action sequence. An inconsistent situation can be repaired by replacing an action by its variation or inserting exogenous actions (Gspandl, Pill, Reip, Steinbauer, and Ferrein, 2011).

Following a single fault policy, we allow exactly one change. Examining the definition of consistency (Definition 3) we see that an inconsistency either results from a mismatch between the expected and real sensing values or from a violated invariant. In the former case, we have two possibilities in order to diagnose the inconsistency. First, if there exists a sensing variation, we can change the sensing result. That is a trivial task with no implication to the remaining action sequence. Second, we can alter the action history, so that the sensed fluent provides the same result as in the sensing result.

This means we have to change a fluent value. In the latter case of an invalid invariant, we can reformulate the invariant formula into conjunctive normal form (CNF), which is a conjunction of clauses. In order to fulfill the conjunction every clause has to be true. As in CNF every clause is required to be a disjunction of literals it is rather simple to identify all the literals that may let the formula fail. To sum it up, beside the simple case where we have to change a sensing result, we can correct a situation to be consistent by fulfilling a fluent (or a set of fluents).

Recalling the normal form of successor state axiom

$$F(\vec{x}, do(\alpha, s)) \equiv \varphi^{+}(\alpha, \vec{x}, s) \vee F(\vec{x}, s) \wedge \neg\varphi^{-}(\alpha, \vec{x}, s)$$

we can fulfill a fluent either by an action that leads to the fluent or by maintaining the fluent value by omitting the action (or the effects of the action) that negates the fluent. Two designated algorithms are presented that lead to the described behavior. Whereas algorithm *fulfill* changes an action sequence in order to lead to a fluent, algorithm *maintain* keeps the value of a fluent. So the resulting hypotheses set is the sensing variation, plus the result of algorithm *fulfill* and algorithm *maintain*. This behavior for finite domains is shown in Algorithm 3, which is called each time an inconsistency is detected. The current situation forms the input. It is split into the action sequence ($\vec{a}$) and the initial situation S_0. First, the set H of hypotheses is initialized empty. Afterwards, the set of fluents that create the inconsistency is identified in Line 2. If the inconsistency is due to a mismatch between expected and real sensing value, we add the real fluent value. In case of an invariant we reformulate the invariant as conjunctive normal form and add all literals of disjunctions of literals that evaluate to false. Consider an initial situation where the robot is located in the *quality_control* ($at(quality\_control)$), holding the object *container1* ($has\_object(container1)$). The current action sequence is $[putdown(container1), goto(shop)]$. In the shop, the robot senses the *container1* ($sense(container1)$), which was expected to be in the *quality_control*. A possible sensing variation could be *sense_nothing* that is added if applicable in Line 4. Then for every conflicting fluent we generate the hypotheses by applying the algorithms *fulfill* (Line 7) and *maintain* (Line 8). The details of those algorithms follow immediately.

Algorithm 3: $generate(\vec{a}, S_0)$

```
1  H ← ∅
2  [F_1, ..., F_n] ← identifyFluents
3  if IsSensingAction(a_m) then
4  |  H ← H ∪ applySensingVariation(a_m, [a_1, ..., a_{m-1}], S_0)
5  end
6  foreach f(x⃗) ∈ [F_1, ..., F_n] do
7  |  H ← H ∪ fulfill_{F(x⃗)}(DT^{F(x⃗)}, a⃗, ∅, S_0)
8  |  H ← H ∪ maintain_{F(x⃗)}(a⃗, ∅, S_0)
9  end
10 return H
```

Algorithm *fulfill* (Alg. 4) tries to apply diagnosis templates as shown in Fig. 7.8. In this case we had three diagnosis templates, namely

- $[pickupWrongObject(obj), goto(r)]$,
- $[exogMoveObject(obj), pickup(obj), goto(r)]$ and

Algorithm 4: $fulfill_{F(\vec{x})}(DT^{F(\vec{x})}, \vec{a}, \vec{v}, S_0)$

```
1  H ← ∅
2  dtSet ← ∅
3  s ← do([a_1,...,a_m], S_0)
4  s_{-1} ← do([a_1,...,a_{m-1}], S_0)
5  foreach dt_i ∈ DT^{F(x)} do
6      l ← |dt_i|
7      if l > 1 then
8          if unify(dt_i.α_l, a_m) then
9              dtSet ← dtSet ∪ {dt_i ∖ dt_i.α_l}
10         else
11             dtSet ← dtSet ∪ dt_i
12         end
13     else if |dt_i| = 1 then
14         if ∃x.Varia(dt_i.α_1, a_m(x), s_{-1}) then
15             H ← H ∪ applyVaria(s_{-1}, dt_i.α_1, v)
16         else if Inser(dt_i.α_1, do(a_m, s_{-1})) then
17             H ← H ∪ applyInser(s_{-1}, dt_i.α_1, v)
18         end
19     end
20 end
21 if |a| > 1 then
22     H ← H ∪ fulfill_{F(x)}(dtSet, [a_1,...,a_{m-1}], [a_m, v], S_0)
23 end
24 return H
```

- $[exogMoveObject(obj)]$.

Every diagnosis template consists of one action variation or insertion plus an optional list of actions that follows the change. This non-change list can be seen as condition, so that the variation or insertion can be applied successfully. Thus, first we check if this non-change list occurs in the current action sequence and if yes, we try to apply the change action (variation or insertion). The algorithm takes the initial situation together with the current action sequence, the set of diagnosis templates explaining the current inconsistency and an initially empty set of visited actions, that is used for completing diagnosis candidates within the algorithm. We will explain the algorithm based on the inconsistency of the previous example, where the initial situation is defined by $[at(quality\_control), has\_object(container1)]$ and the action sequence is $[putdown(container1), goto(shop)]$. This was followed by the sensing the *container1* in the shop. So we try to make $is\_at(container1, shop)$ hold.

The algorithm iterates recursively over the current action sequence and within each call it iterates over all diagnosis templates. We try to unify the last action (a_l) of diagnosis templates that are longer than one action with the current action in Line 7f. So the diagnosis template $[pickupWrongObject(obj), goto(r)]$ can be applied as $goto(r)$ and $goto(shop)$ can be unified. Therefore, we can reduce the non-change condition list by this one *goto* action and append the shortened diagnosis template to the diagnosis template list for the next iteration (Line 9). $[exogMoveObject(obj), pickup(obj), goto(r)]$

can be applied equally as the previous template. The third template, namely $[exogMoveObject(obj)]$, is different, as it contains only the change action, so we have to check if this action can be applied. This results in the first diagnosis $[putdown(container1), goto(shop), exogMoveObject(container1, shop)]$ derived by Line 17 that applies the given exogenous action. After iterating over all diagnosis templates, it is checked if the current action sequence is longer than one. If yes, we have further actions to be checked and call the algorithm recursively (Line 22). Within the next call we have two remaining diagnosis templates ($[pickupWrongObject(obj)]$ and $[exogMoveObject(obj), pickup(obj)]$) and the current action sequence is $[putdown(container1)]$. The first diagnosis template cannot be applied as $pickupWrongObject(obj)$ is no valid variation for the action $putdown(container1)$ and the second template cannot be applied because $pickup(obj)$ cannot be unified with $putdown(container1)$. The remaining action sequence is not greater than one, so no further recursive call is made. After returning from the recursive call, all hypotheses from the recursive call are added to the current hypotheses set. Finally, this hypothesis set is returned as result.

After having a possibility to change an action sequence in order to lead to a fluent, we investigate the second case where a fluent is expected to maintain its value. We do this by preventing the fluent from being altered by an action. For this purpose we check every action if it changes the relevant fluent. After identifying the action we either try to change it by a variant of it, or we try to prevent this fluent changing effect by changing the situation before the execution of the identified action, so that the effect can not take place. This is shown in algorithm 5. Similarly to Algorithm 4, it takes the initial situation S_0, the current action sequence $a_1, \ldots, a_m$ and the sequence of visited actions $v_1, \ldots, v_p$ (initially empty) as input. First, the hypothesis set is initialized, followed by abbreviations for the current and the predecessor situation s and s_1. Then we check if the current action changes the fluent value $F(\vec{x}, s)$ to false (Line 5). If there is an executable action variation that prevents this change it is added to the hypotheses set (Line 6). Here the set of visited actions is necessary to complete the action sequence of the new hypothesis. Furthermore we try to avoid the effect of the fluent negating action. Therefore, relevant fluents to enable the effect are identified in Line 10. If such a fluent is found, it is prevented by maintaining the negative fluent in Line 11 and furthermore by fulfilling the negative fluent in Line 12. The results of both cases are added to the hypotheses set. If the main fluent $F(\vec{x}, s)$ is not negated by the current action and the remaining action sequence is not empty, the algorithm is called recursively (Line 17). Finally the generated hypotheses set is returned in Line 20. Please note that returned hypotheses are not necessarily consistent. Such a consistent check is expected to happen after all hypotheses are generated. This is simply due to the fact that the same hypothesis may be generated from both *fulfill* and *maintain* algorithms. So if we store the resulting hypotheses in a set before checking them for consistency, we can omit an expensive double check.

A final remark has to be made about the unrelated single-fault assumption that is made throughout this chapter. This means, every time an inconsistency occurs, this inconsistency can be repaired if it is due to a single fault. It is no problem to repair two consecutive faults if there is sufficient sensing. For instance, the first sensing action unveils the first fault that can be repaired by our belief management. Then, immediately afterwards, the next action is again a sensing action that unveils another fault. Here again, belief management can repair the resulting inconsistency. The unrelated part of an unrelated single-fault addresses exactly this part, namely that every single fault has to be repairable by one change. It is no problem to handle multiple faults one by one, which can be done if the faults are unrelated.

Algorithm 5: $maintainFluent_{F(\vec{x})}(\vec{a}, \vec{v}, S_0)$

$H \leftarrow \varnothing$
$\vec{a}_l \leftarrow [a_1, \ldots, a_{m-1}]$
$s \leftarrow do([a_1, \ldots, a_m], S_0)$
$s_{-1} \leftarrow do([a_1, \ldots, a_{m-1}], S_0)$
if $F(\vec{x}, s_{-1}) \wedge \neg F(\vec{x}, s)$ **then**
  if $\exists \alpha, \vec{y}.Varia(\alpha, a_m(\vec{y}), s_{-1}) \wedge Exec(\alpha, s_{-1})$ **then**
    $H \leftarrow H \cup [a_1, \ldots, a_{m-1}, \alpha, v_1, \ldots, v_p]$
  end
  foreach $f \in Fluents$ **do**
    if $\exists \vec{y}, \vec{z}.DepEffect_{F,f}(\vec{x}, \vec{y}, a_m(\vec{z}), do(\vec{a}, S_0))$ **then**
      $H \leftarrow H \cup maintain_{\neg f(\vec{y})}(\vec{a}_l, [a_m, \vec{v}], S_0)$
      $H \leftarrow H \cup fulfill_{\neg f(\vec{y})}(DT^{\neg f(\vec{y})}, \vec{a}_l, [a_m, \vec{v}], S_0)$
    end
  end
else
  if $|\vec{a}| > 1$ **then**
    $H \leftarrow H \cup maintain_{F(\vec{x})}(\vec{a}_l, [a_m, \vec{v}], S_0)$
  end
end
return H

Chapter 8

Efficient Belief Management

After defining diagnosis templates in Chapter 7 that ensure great progress in terms of efficiency, we look more closely at efficiency and present a few add-ons that have proven to be useful. We present two heuristics that deal with the problem of when to investigate which hypothesis. This is especially useful if the domain consists of fluents that allow multiple different instantiations. The combination of these heuristics is quite important to the performance of the whole system.

8.1. Hypothesis Compression

Within Chapter 7, we tried to reduce the amount of generated hypotheses that have to be checked for consistency. Furthermore, we know that we generate hypotheses for inconsistent situations. After this generation we have to choose a new favorite as the old one has become inconsistent (see Algorithm 1). The new favorite is again kept until it becomes inconsistent. This means only if exactly one hypothesis results from every repair process, or if we always choose the incorrect hypothesis, we will be able to use all hypotheses from the pool to form the belief. As this coincidence might happen, but is very unlikely to happen the question arises if we need to generate all valid hypotheses or whether we can restrict the resulting hypotheses set without loss of generality or performance. So the idea is to compress a set of hypotheses into one hypothesis. Parts of this chapter were published in Gspandl, Podesser, Reip, Steinbauer, and Wolfram (2012).

Assuming that the possibility of compressing a set of hypotheses into a single hypothesis exists, we take a look at the decompression process first. Obviously, we would like to decompress the hypothesis set just in case it is really necessary. As needed the definition directly depends on the favorite selection policy, we have to take this into account. In Section 4.2, costs were presented as selection criteria. So, if we can compress the hypothesis set in a way that the compression result itself forms a hypothesis we do not need to alter the remaining belief management system. Furthermore, if all hypotheses share the same costs, we can easily place the decompression process around the selection of the new favorite.

So coming back to the compression itself, we were already able to identify two requirements. First, all hypotheses of a compression set have to share the same costs. Furthermore, if the compressed set itself is a normal hypothesis, the decompression algorithm needs to operate without any additional

knowledge. Imagine the following action sequences that are all followed by the sensing result indicating *containerX* is in the *shop*. All action sequences are supposed to start in an equal initial state and all of them form a consistent situation.

- $[goto(storage), pickup(container1), goto(shop), exogMoveObject(containerX, shop)]$
- $[goto(storage), pickup(container1), exogMoveObject(containerX, shop), goto(shop)]$
- $[goto(storage), exogMoveObject(containerX, shop), pickup(container1), goto(shop)]$
- $[exogMoveObject(containerX, shop), goto(storage), pickup(container1), goto(shop)]$

As the costs of hypotheses resulting from summing up the cost value of each change, all four hypotheses share the same costs, as they share the same insertion $exogMoveObject(containerX, shop)$. Furthermore, we see that all action sequences consist of the same actions. They only differ in the position of the insertion, where the change moves further to the front in every example. This means we can derive the second action sequence from the first sequence by moving the change of one position towards the beginning of the sequence. The same holds for the third and the second, as well as the fourth and the third sequence. From a human perspective it makes no difference if the *containerX* was brought to the *shop* a few minutes earlier or later, because we can decouple the action sequence of the robot and the exogenous action. According to Occam's Razor, we prefer the very last position for a change, as the effects of the change can influence less other actions. An action can only influence another action if it occurs before. For instance, in the sequence $pickup(container1), goto(shop)$ the action $pickup(container1)$ might influence the action $goto(shop)$. If the sequence is $goto(shop), pickup(container1)$, then the action $pickup(container1)$ has no influence on the action $goto(shop)$. Thus, for action sequences with the same endogenous actions and the same change that occurs in different positions, we found an exemplary solution that allows us to represent a set of hypotheses as a single correct hypothesis and to decompress them, when needed. After outlining the idea we will now formally define this compression.

First, we will start with the definition of a function that returns the position of an action within a given action sequence. If the action occurs more often than once the function will return the position of the last occurrence. In case the action sequence does not contain the action, the function will return ∞.

Definition 23. *Let α, denote an action, while $a_1, \ldots, a_n$ is an arbitrary action sequence. For some situation term s, action α occurs at position p in the action sequence $a_1, \ldots, a_n$, iff*

$$\begin{aligned} pos(\alpha, do([a_1, \ldots, a_n], s)) = p \doteq \\ p = n \wedge \alpha = a_n \vee \\ p = n-1 \wedge \alpha = a_{n-1} \wedge \alpha \neq a_n \vee \\ p = n-2 \wedge \alpha = a_{n-2} \wedge \alpha \neq a_n \wedge \alpha \neq a_{n-1} \vee \\ \vdots \\ p = \infty \wedge \alpha \neq a_n \wedge \alpha \neq a_{n-1} \wedge \cdots \wedge \alpha \neq a_1 \end{aligned}$$

Equipped with a function that gives us the position of an action within an action sequence, we can define a position relation within an action sequence, denoting that action α_1 occurs before α_2 in action sequence $a_1, \ldots, a_n$. In case the action sequence does not contain α_2 but contains α_1 the predicate will evaluate to true. In case α_1 is not present the predicate will always evaluate to false.

Definition 24. *Let α_1 and α_2 be two actions and $a_1,\ldots,a_n$ an action sequence. α_1 is defined to be before α_2, iff*

$$\begin{aligned} before(\alpha_1,\alpha_2,do([a_1,\ldots,a_n],s) \doteq \\ \exists p_1.pos(\alpha_1,do([a_1,\ldots,a_n],s) = p_1 \wedge \\ \exists p_2.pos(\alpha_2,do([a_1,\ldots,a_n],s) = p_2 \wedge p_1 < p_2 \end{aligned}$$

The next predicate checks if two action sequences consist of the same endogenous actions. If both action sequences consist of the same actions, the test is trivial and we can stop. Otherwise, we step through all actions recursively. If action sequence $a_1,\ldots,a_n$ ends with an action variation or an insertion, the last action is cut and the predicate is evaluated with the remaining sequence. The same is true for action sequence $b_1\ldots b_m$. If neither action sequence $a_1\ldots a_n$ nor action sequence $b_1\ldots b_m$ ends with a change, the last action in both sequences has to be equal and the remaining sequences are evaluated.

Definition 25. *Let $a_1,\ldots,a_n$ and $b_1\ldots b_m$ be action action sequences. They are said to be equal regarding endogenous actions, iff*

$$\begin{aligned} EndogEquality(do([a_1,\ldots,a_n],s),do([a_1,\ldots,a_n],s)) &\doteq \top \\ EndogEquality(do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) &\doteq \\ (Inser(a_n,do([a_1,\ldots,a_{n-1}],s)) \vee & \quad Varia(a_n,b_m,do([a_1,\ldots,a_{n-1}],s))) \wedge \\ EndogEquality & \quad (do([a_1,\ldots,a_{n-1}],s),do([b_1,\ldots,b_{m-1}],s)) \\ EndogEquality(do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) &\doteq \\ (Inser(b_m,do([a_1,\ldots,b_{m-1}],s)) \vee & \quad Varia(b_m,a_n,do([b_1,\ldots,b_{m-1}],s))) \wedge \\ EndogEquality & \quad (do([a_1,\ldots,a_{n-1}],s),do([b_1,\ldots,b_{m-1}],s)) \\ EndogEquality(do([a_1,\ldots,a_n,\alpha],s),do([b_1,\ldots,b_m,\alpha],s)) &\doteq \\ EndogEquality & \quad (do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) \end{aligned}$$

Provided with a set of definitions, we put them together in order to define if two action sequences consist of the same endogenous actions that are in the same order and the same changes that are in the same order as well. So, first the same order of all changes is checked action by action in both sequences by applying the predicate *before* to every change. This is followed by the predicate *EndogEquality* that checks the equality of all endogenous actions.

Definition 26. *Two action sequences $a_1,\ldots,a_n$ and $b_1,\ldots b_m$ are said to be change equal for the action variations or insertions $\alpha_1,\ldots,\alpha_k$, iff*

$$\begin{aligned} ChangeEquality(\alpha_1,\ldots,\alpha_k,do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) \doteq \\ \bigwedge_{i=1,\ldots,k-1} before(\alpha_i,\alpha_{i+1},do([a_1,\ldots,a_n],s) \wedge \\ before(\alpha_i,\alpha_{i+1},do([b_1,\ldots,b_m],s) \wedge \\ EndogEquality(do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) \end{aligned}$$

After having defined an equality between two action sequences with regard to change actions, we define a certain order between action sequences. As the endogenous actions are the same in both sequences, we concentrate on the change actions. Two action sequences are said to be sorted if every change action in the first action sequence occurs in the second action sequence at the same position or later. So, after applying the predicate *ChangeEquality* we compare the positions of all change actions. As noted in the introduction of this section, we prefer changes to occur as late as possible within an action sequence, as it can influence less other actions.

Definition 27. *Let $a_1,\ldots,a_n$ and $b_1,\ldots,b_m$ action sequences and $\alpha_1,\ldots\alpha_k$ a sequence of action variations or insertions, then the action sequences are said to be Sorted, iff*

$$\begin{aligned}
Sorted([],do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) &\doteq \top\\
Sorted([\alpha_1,\ldots,\alpha_k],do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)) &\doteq\\
ChangeEquality([\alpha_1,\ldots,\alpha_k],do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s))\wedge&\\
\exists p_a.pos(\alpha_1,do([a_1,\ldots,a_n],s)) = p_a\wedge&\\
\exists p_b.pos(\alpha_1,do([b_1,\ldots,b_m],s)) = p_b\wedge&\\
(p_a \geq p_b \wedge Sorted(\alpha_2,\ldots,\alpha_k,do([a_1,\ldots,a_n],s),do([b_1,\ldots,b_m],s)))&
\end{aligned}$$

So, finally we are able to define the compression function. The idea behind this is to take a set of action sequences that fulfill the *ChangeEquality* predicate and to represent them by a single action sequence. We do this by sorting all action sequences and select the first one. This means the resulting action sequence fulfills the *Sorted* predicate with every other action sequence from the compression set. For space reasons we further abbreviate action sequences by vectors, such as the abbreviated action sequence $a_1,\ldots a_n$ is written as $\vec{a}$.

Definition 28. *Let $\vec{a_1}$, $\vec{a_2}$, …, $\vec{a_m}$ be action sequences and $\vec{\alpha}$ be a sequence of change actions, then the compression function compress takes all the sequences and results in action sequence $\vec{z}$ as follows:*

$$\begin{aligned}
compress([\vec{\alpha}],do([\vec{a_1}],s),do([\vec{a_2}],s),\ldots do([\vec{a_m}],s)) = do([\vec{z}],s) &\doteq\\
(\vec{z}=\vec{a_1}\wedge Sorted([\vec{\alpha}],do([\vec{a_1}],s),do([\vec{a_2}],s))\wedge\ldots\wedge Sorted([\vec{\alpha}],do([\vec{a_1}],s),do([\vec{a_m}],s)))\vee&\\
\vdots&\\
(\vec{z}=\vec{a_m}\wedge Sorted([\vec{\alpha}],do([\vec{a_m}],s),do([\vec{a_1}],s))\wedge\ldots\wedge Sorted([\vec{\alpha}],do([\vec{a_m}],s),do([\vec{a_{m-1}}],s)))&
\end{aligned}$$

So we are able to compress a set of action sequences into a single action sequence. Let us move the perspective to the hypothesis generation side and assume that we want to insert an insertion into an action sequence in order to generate the respective hypotheses. Then it is easy to see, that all hypotheses will form a set of hypotheses that can be compressed. Furthermore we can easily compute the result of the compression set, without generating all the hypotheses before. This helps a lot as there is less load on the pool of our belief management system. Furthermore, it might happen that this action sequence is never selected to be the favorite one. As all hypotheses of the compression set share the same cost value, no hypotheses of the compression set would have been selected as well. So we can save a potentially high number of evaluations that is crucial to the performance of the belief management system. As the number of hypotheses within the belief management pool may be very large, quite a high number of hypotheses is never selected to be the favorite hypotheses that forms the belief of the robot.

The better the selection strategy, the less (incorrect) hypotheses are selected as favorites. Nevertheless we expect hypotheses to be selected, because otherwise it would not make sense to store them in our hypotheses pool. In case a compressed action sequence is selected to be the favorite hypothesis, we have to answer two questions. First, which hypothesis should form the robot's belief and second, what is to be done if this favorite hypothesis turns inconsistent again. The first question could be answered by any hypothesis out of the compression set, though as they are equal regarding changes and as the result of the compression function follows Occam's razor, there is no motivation to select any other hypothesis than the result of the compression function. The initial situation for the second question is as follows: if the favorite turns inconsistent we have a compression set of hypotheses and the result of the compression function is inconsistent. So, although the result of the compression function is inconsistent, there may be another hypothesis within the compression set that is consistent. Again, as all hypotheses within this set share the same cost value, any hypothesis from this set is a potential candidate to be the new favorite. For that purpose we need to have a strategy that is able to recover the compression set from the result of the compression function.

Taking the initial example of this section, this means we want to recover the set

- $[goto(storage), pickup(container1), goto(shop), exogMoveObject(containerX, shop)]$
- $[goto(storage), pickup(container1), exogMoveObject(containerX, shop), goto(shop)]$
- $[goto(storage), exogMoveObject(containerX, shop), pickup(container1), goto(shop)]$
- $[exogMoveObject(containerX, shop), goto(storage), pickup(container1), goto(shop)]$

from the result of the compression function, namely the action sequence

- $[goto(storage), pickup(container1), goto(shop), exogMoveObject(containerX, shop)]$

Generally this process is referred as decompression. In our case this is a rather trivial process, as we only have to shift the positions of insertions and action variations. Accordingly, to the definition of the compression function, we have to preserve the order of change actions (insertions and action variations) and endogenous actions separately. Based on the definitions of compressing and decompressing a set of change equal actions two interesting questions arise. First, how many action sequences can be compressed into one action sequence and second, how many change equal action sequences can be decompressed from one action sequence respectively.

Theorem 4. *Let n be the length of the action sequence σ, k be the sum of insertions and action variations occurring in σ. Then, the number of decompressed action sequences u from σ is bounded by*

$$u \leq \binom{n}{k}$$

Proof. Assuming that every action is an insertion, we can assign all k actions a position within the n slots. This leads to $\frac{n!}{(n-k)!}$ different action sequences. As there exists a fixed order within the insertions, we have to reduce the number by $k!$ leading to $\frac{n!}{(n-k)! \times k!} = \binom{n}{k}$. An action variation cannot be placed as freely as an insertion, because they are bound to an endogenous action in the original action sequence. So we have less possibilities to choose a position for action variations, thus $\binom{n}{k}$ is the upper bound for all decompressed action sequences resulting from a compressed action sequence of length n with k changes. □

8.2. Hypotheses Management

We have seen that taking all possible hypotheses into account to form an explanation for an existing inconsistency poses a few challenges. First we were able to reduce the number of hypotheses that have to be checked for inconsistency by applying diagnosis templates in Section 7.3. Next, we moved the focus to the pool itself that contains all valid hypotheses. In Theorem 2 we defined the upper bound for the pool size. This rather huge number was already tackled by compressing multiple hypotheses into one representative. The bigger the pool size of our system the more inconsistencies can be explained, but the more calculation time is necessary to manage the pool. Please keep in mind, that if the pool is too small, the correct hypothesis might be thrown out of the pool. So, of course within a fail-safe system where the robot is allowed to spend as much time as necessary to execute every task, the pool size will be chosen as high as possible. However, almost all robots cannot afford taking such liberty and are expected to behave as efficiently as possible similar to other machines. So, we are interested in providing explanations for as many inconsistencies as possible in a pool that is as small as possible.

In Section 4 we defined a function *val* that assigned a positive cost value to every change. Summing up these change values gives an indicator of the plausibility of a hypothesis: the smaller this value, the more plausible the hypothesis. This approach works fine, if all costs are calculated at the same time, but has clear drawbacks if not. Costs are calculated during generation. Recalling Algorithm 1, we stay with one hypothesis as long as it is consistent. If it is inconsistent, we apply history-based diagnosis and check the next favorite hypothesis (lowest costs) for consistency.

Imagine the following situation (see Figure 8.1): we start with a hypothesis H_0 that turns inconsistent, where the number denotes the costs. By applying the presented approach, we are able to fill the hypothesis pool with the hypotheses H_1, H_6 and H_7. As H_1 shares the lowest costs, it forms the new favorite. After executing a few more actions, H_1 turns inconsistent again and H_2, which is generated from H_1 is the new favorite and so forth. The salient point is that while hypothesis after hypothesis turns inconsistent, old uninteresting hypotheses stay within the pool. We call a hypothesis uninteresting, if it was generated many cycles before and never formed the favorite, such as H_6 and H_7 in our example. So after detecting an inconsistency in H_5 that leads to H_8 we have to check H_6 and H_7 first as they share lower cost values than the newly generated hypothesis H_8. They might explain the current situation, though this is unlikely as they were generated four inconsistencies earlier.

Therefore, we seek for a better selection strategy. To this extent we approach our hypothesis pool as a search problem. This means that the pool can be seen as a sorted open list that is processed till a valid hypothesis is found. Inconsistent hypothesis can again generate new hypotheses that are inserted into the open list. On the other hand, we can state that an inconsistency can emerge for two reasons. First, it might arise from a new issue, or second it can result from a wrong previous decision. Please note that multiple consistent hypotheses can coexist, so that the real action sequence can only be detected after additional (sensing) actions.

In case of a new issue, the situation is as follows. The belief of the robot that is encoded by an initial situation and an action sequence is consistent until something happens. Then the belief turns inconsistent. So, assuming that the extended Basic Action Theory is able to describe the situation, we can apply history-based diagnosis to the last consistent situation in order to incorporate the new issue. All resulting hypotheses from history-based diagnosis on the last consistent favorite hypothesis are included in the open list. This procedure resembles exactly the basic management algorithm we presented in Algorithm 1, but instead of a cost-sorted open list, we prefer an open list that is sorted by

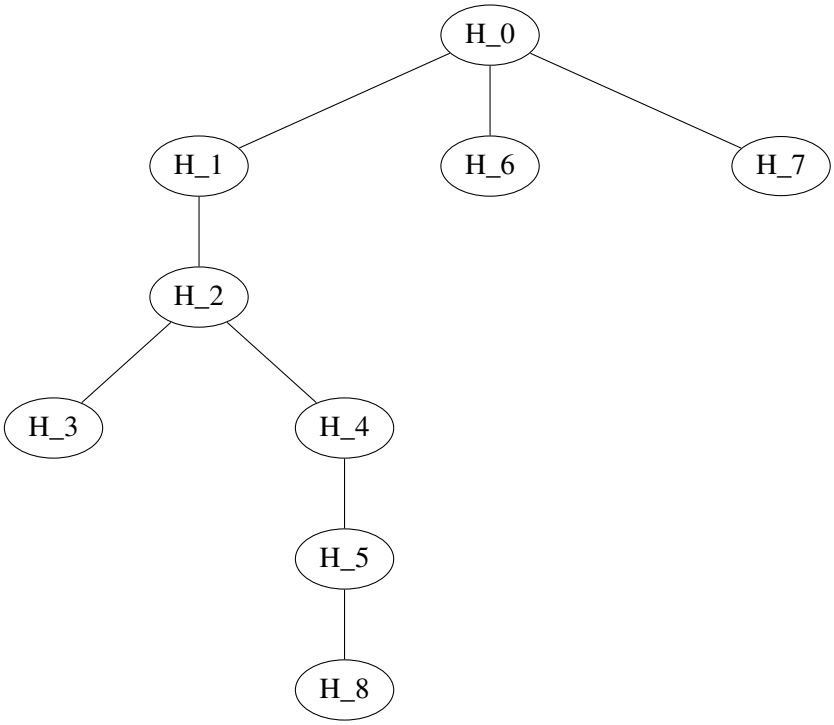

Figure 8.1.: Exemplary hypothesis generation over time, where time evolves on the vertical axis from top to bottom. This means H_1, H_6 and H_7 are generated at the same time and all of them are generated before H_2. The hypothesis cost is part of the name. For instance, H_5 owns cost 5. A child node is added to the parent node if the parent turns inconsistent and the child node hypothesis was generated out of the parent node. For example, if H_4 turns inconsistent, H_5 is generated. The default selection strategy always selects the hypothesis with the lowest cost. If H_5 turns inconsistent, H_8 is generated and H_6 becomes the next favorite although it was already generated 4 inconsistencies earlier!

age first.

On the other hand we saw that a hypothesis can turn inconsistent because we made the wrong choice in a previous selection process. This means many hypotheses were able to describe the world consistently one or more actions ago, but new actions revealed further discrimination potential. So no new hypothesis has to be generated, we simply have to correct one of the last selections. Assuming an identical distribution of sensing actions over all actions and an identical coverage of directly perceptible fluents by sensing actions, we can infer that wrong selections are more likely to occur later than earlier. For instance, if we take the last selection, it is easy to see that it is followed by less or equal actions that allow for discrimination than any selection before.

So now, we can sum up our requirements. If we made an incorrect decision in selecting a new favorite, no new hypotheses have to be generated. It is enough to revisit the last decisions and probe the hypothesis with the next lowest cost: this is a rather fast process. On the other hand, if something unexpected happened, we have to generate new hypotheses and if such hypotheses can be generated from the last favorite, we would like to select one of these as new favorite. This can easily be achieved by two open lists. The first open list contains all hypotheses from the last generation and is sorted by costs. We call it fast open list. The second open list forms the previously described hypothesis pool,

but is sorted by age and is called pool. In case of an inconsistency we process the fast open list and check if it contains a consistent hypothesis. If yes, the hypothesis is selected as the new favorite and the execution can continue. If not, we process the pool and check if it contains a consistent hypothesis. Inconsistent hypotheses are removed from the pool and history-based diagnosis is applied. In this way we can reduce the number of unnecessary generation steps.

This behavior is realized by the algorithms *selectFromFastOpenList* and *efficientBeliefManagement*. Algorithm *selectFromFastOpenList* (see Algorithm 6) is a helper algorithm that is applied by Algorithm *efficientBeliefManagement* (see Algorithm 7). It takes the fast open list and the hypothesis pool as input. The algorithm iterates over the fast open list and checks if the given hypothesis is consistent. If it is inconsistent, it is removed from the fast open list and from the pool (see Lines 6-8). If it is consistent, the hypothesis is returned in Line 11. If the fast open list is initially empty (see Line 2) or if none is found (see Line 13), it returns *nil*.

Algorithm 6: *selectFromFastOpenList*

input : A cost sorted fast open list of hypotheses O
An age sorted pool of hypotheses H
output: A hypothesis h_f or *nil*

```
1  if |O| = 0 then
2  |   return nil
3  end
4  h_f ← selectFavorite(O)
5  while |O| ≥ 1 ∧ ¬Cons(h_f) do
6  |   O ← O ∖ h_f
7  |   H ← H ∖ h_f
8  |   h_f ← selectFavorite(O)
9  end
10 if Cons(h_f) then
11 |   return h_f
12 end
13 return nil
```

Algorithm 7 is an extension of the basic belief management Algorithm 1. The update step in Line 2 is equal to the basic algorithm. If the current hypothesis h_f is inconsistent, it checks if the fast open list contains a consistent hypothesis. If this is true, we are done. If there is no consistent hypothesis on the fast open list a set of new hypothesis is generated in Line 7. This set defines the new fast open list and is added to the pool additionally. If this new set contains a consistent hypothesis, such a hypothesis is selected in Line 10. If no hypothesis can be found, the new hypothesis is selected from the pool, similar to the basic algorithm (see Line 12).

Algorithm 7: *efficientBeliefManagement*

```
1  foreach h_i ∈ H ∖ h_f do
2  |   updateHypothesis(h_i)
3  end
4  if ¬Cons(h_f) then
5  |   h_f ← selectFromFastOpenList(O, H)
6  |   if h_f = nil ∨ ¬Cons(h_f) then
7  |   |   N ← generateHypotheses(h_f)
8  |   |   O ← N
9  |   |   H ← H ∪ N
10 |   |   h_f ← selectFromFastOpenList(O, H)
11 |   |   if h_f = nil ∨ ¬Cons(h_f) then
12 |   |   |   h_f ← selectFavorite(H)
13 |   |   end
14 |   end
15 end
```

Chapter 9

Evaluation

After presenting a belief management that provides a robot with a consistent belief under harsh conditions, we will investigate if this belief management is useful for a robot. It has to cope with execution and sensing faults, incomplete initial belief and exogenous events. We are interested whether the robot can execute the given tasks or not together with the corresponding runtimes. Instead of different domains, we stay with the robot delivery domain and change the difficulties the robot has to master. Furthermore, we interpret the results, so that conclusions are possible for different domains. The contents of this chapter have been published in the papers Gspandl, Pill, Reip, and Steinbauer (2011), Gspandl, Pill, Reip, Steinbauer, and Ferrein (2011), Wolfram, Gspandl, Reip, and Steinbauer (2011), Gspandl, Podesser, Reip, Steinbauer, and Wolfram (2012), Gspandl, Pill, Reip, and Steinbauer (2013) and Reip, Steinbauer, and Ferrein (2012).

We start by asking ourselves if a belief management system makes sense for a robot. For this purpose, we compare the performance of an agent provided with a belief management with an agent without in a simple simulated environment. Next, after having shown that a robot can benefit from belief management, we increase the complexity of the scenario and add additional threats. Then we leave the simulated world and apply the presented system to a real world robot. Here, we show that the robot is able to cope with all four different types of threats and additionally we let the robot execute a long term test. Finally, we examine the effect of the presented efficiency measures on runtimes.

9.1. Belief Management System - Useful or Not?

In order to evaluate the benefit of a robot equipped with a belief management system, we compared the performance of an agent with and an agent without belief management system in a delivery domain. The contestant equipped with a belief management system used a simple implementation without any optimizations such as diagnosis templates, compression or a special hypothesis management as described in Section 8.2. The simulated environment resembles the institute's map and consists of 24 rooms that are connected by 5 hallways. The map is shown in Figure 9.1. For every task the robot had to move to the task's initial location, pick up the requested object, move to the task's destination and put down the object there. The robot was able to accomplish this by choosing between $goto(room)$, $pickup(obj)$ and $putdown(obj)$ actions. The *goto* action moves the robot between adjacent rooms. The

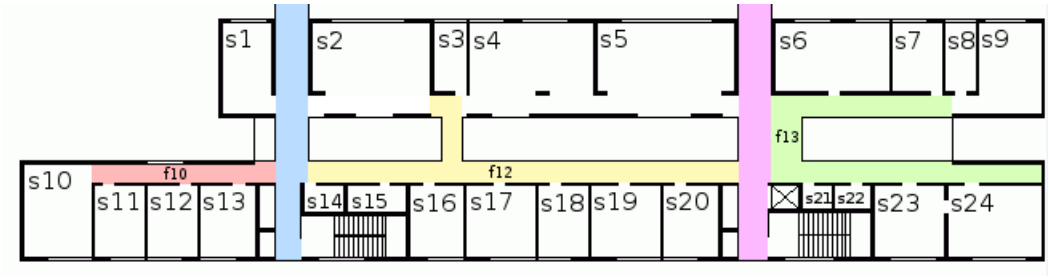

Figure 9.1.: Floor layout of the Institute of Software Technology, Graz University of Technology

action *pickup*(*obj*) lets the robot hold the given object if the robot is not already holding an object, and the action *putdown*(*obj*) sets down the given object from the robot at the requested location. Besides the robot's actuators, it had two sensors. The first sensor was a vision sensor that detects objects that are in the same room as the robot. The second sensor was a pressure sensor, that indicated if the robot is loaded or not. The sensing was carried out continuously. This means according to an adjustable parameter n the robot received sensing information every n actions. This continuous sensing feature resembled the typical way a robot behaves. We added this kind of sensing to the IndiGolog framework as a new feature.

We call a group of three transportation tasks a mission. The robot received the tasks of the mission in advance and executed them until no further tasks are open. In order to reduce the influence of the high-level program on the robot to a minimum, the robots used simple T-R programs as shown in Section 6.1. This means the task was finished whenever the object was located at the required destination and the robot was not holding it. Otherwise if the robot was in possession of the object and at the destination, it put the object down, or if the robot held the object, but was not at the destination yet, it moved towards the destination and so on.

During execution, the robot was faced with sensing and execution faults. We grouped them into three fault scenarios. Scenario F1 consisted of execution faults. With a probability of 40%, the robot failed to pick up an object. With a probability of 20% it picked the wrong object. In scenario F2 we added another execution fault, namely a putdown fault with a probability of 30%. Scenario F3 introduced a sensing fault with a probability of 5%, where the vision of the robot failed. Independent of sensing and execution faults, we defined different sensing rates. Here we differentiated between three sensing scenarios. S1 represented sensing after every execution action. Sensing after every second and after every third execution action respectively was denoted by S2 and S3.

The success of the robot was measured by successful missions, where a mission was called successful if all three objects were located at the correct destination. The mission failed if objects were misplaced, the execution exceeded a two minute timeout, or if the interpreter aborted, because there was no further executable action. A typical execution trace consisted of about 40 actions. We compared a base robot with no belief management system against a robot equipped with a belief management system. Both had to execute 50 different missions. The missions were generated randomly. In order to reduce the influence of outliers, each mission had to be executed ten times with different seeds that controlled the random number generation which in turn controlled the occurrence of faults.

Table 9.1 shows the results of both contestants in a simulated environment with sensing after every executed action. In the easiest fault setting F1, both implementations were able to finish all missions successfully. These numbers are not surprising since every execution fault is sensed immediately.

	F1	F2	F3
Base Agent	100%	51%	18%
Belief Management Agent	100%	100%	96%

Table 9.1.: Simulation results for base agent versus belief management agent with sensing rate S1 (sensing after every action execution)

T-R programs do not include a state, so all action selections depend on the environment only and thus are always correct. So actually one would expect the same results for fault setting F2, but the base implementation did not meet the expectations. The reason for this can be found in the pressure sensor, which works in detail as follows: if a robot holds an object, it receives the corresponding sensor information; if the robot does not hold anything, it receives no sensing information. This knowledge is represented by an invariant and can be used to detect putdown faults. This resembles a typical example how belief management is able to derive additional value out of sensing information respectively some action sequence. The salient issue is, as soon as the complete observability of the environment is no longer given, the robot equipped with a belief management system outperforms a robot without a belief management system. The introduction of sensing faults in F3 increased the gap between the two contestants even more.

We can see that belief management does not provide any advantages in settings close to complete observability. As soon as the observability decreases belief management can lead to substantial improvements. This is achieved by two features: the first deals with inconsistency detection and the second with the handling of inconsistencies. On the one hand, the belief management can help to detect inconsistencies. For instance, invariants can reveal inconsistencies that again can be corrected and thus the robot can make better decisions due to a correct belief. On the other hand we have to think how we can cope with inconsistencies. The detection of inconsistencies is easy as sensing contradicts the expectations, but what can we do as soon as inconsistencies are revealed. Basic strategies could either stick to sensing all the times, or prefer the most plausible situation, but it is obvious that choosing the correct hypothesis is difficult at all times. Here, belief management gives us the chance to make incorrect decisions on best effort basis and correct them later on.

Next, we further examined the influence of sensing on the belief. We saw that the less the environment is observable, the more a robot profits from a belief management. We further decreased observability by increasing the sensing step from S1 to S2 and S3. The corresponding results are shown in Table 9.2 and Table 9.3. We start our observation with the base agent. In F3 we can observe a base success rate of roughly 20% more or less regardless of the sensing step. This 20% can be achieved due to a fortunate course of events even in the more difficult settings. In contrast to the constant success rate in the occurrence of all faults, we can see a continuous decrease of the success rate from 100% to 38% to 31% in F1 and 51%, 24% and 20% in F2 respectively. While the base agent faced a steady decrease, the belief management agent handled more or less all execution faults without problems (F1 and F2) and finished nearly all missions successfully. Compared to execution faults, sensing faults are a somewhat more difficult to handle. This follows the fact that the less correct sensing information is available, the less information we have in order to repair inconsistent situations. We can observe this fact by comparing S3/F3 and S2/F2. The belief management agent faced a decrease in success rates also in the presence of sensing. Although there is still a substantial gap to the base agent (54 percentage points in the most difficult setting).

After execution and sensing faults, the next natural step is to handle exogenous events. In case of

	F1	F2	F3
Base Agent	38%	24%	15%
Belief Management Agent	100%	100%	80%

Table 9.2.: Simulation results of base agent versus belief management agent with sensing rate S2 (sensing after every two actions)

	F1	F2	F3
Base Agent	31%	20%	17%
Belief Management Agent	99%	99%	71%

Table 9.3.: Simulation results base agent versus belief management agent with sensing rate S3 (sensing after every three actions)

an action or sensing fault, we can place an action or sensing variation at the position of the action or sensing action itself. Exogenous events can be much more difficult, because there is no limitation in their position within an action sequence. This means whenever their preconditions are valid, exogenous events can be inserted into the respective action sequence. This can lead to a huge number of hypotheses that might exceed the size of the pool. For this reason we started with a relatively simple exogenous event *snatch*. While the robot was carrying the object, the object was snatched and was left in the same room. We applied this exogenous event with a probability of 20% in a new fault scenario F4. In addition to this new exogenous event F4 contained all faults used in F3.

Again we compared the performance of a robot with belief management with the performance of a robot without belief management. Except the additional exogenous event, the setup remained the same. Due to a higher complexity of the environment, we increased the timeout to 120 minutes. The results are given in Table 9.4. Clearly, the base agent was overstrained with all the given threats. A success rate of only 6 % with sensing after every action execution is very low. This value is more than twelve times higher in systems with belief management. We can see a decrease in S2, although it still solved more than 50% of all missions in the very adversarial scenario S3/F4.

	S1	S2	S3
Base Agent	6%	4%	3%
Belief Management Agent	77%	65%	52%

Table 9.4.: Simulation results of base agent versus belief management agent in fault scenario F4.

The timeout in F1-F3 was two minutes, so the runtime was no big issue. In F4, we had to increase the timeout by an order. So, we used 120 minutes to evaluate the performance of the belief management system. For this reason we take a closer look at the run times of our belief management system. The base agent serves again as base value. As this agent strictly followed its tasks regardless of inconsistencies in its belief the runtimes were constant with the drawback of low success rates. One could call the robot an "all or nothing" robot. The average runtime of this all or nothing robot was 29.4 seconds.

Comparing the runtimes of Table 9.5, we see that no setup sensor/fault scenario of the belief management agent was able to beat the runtime of the base agent. Of course this is no surprise, though it is surprising that the overhead of the belief management system is very small throughout the scenarios F1 to F3. This is interesting as the benefit of the belief management is very high in these

	F1	F2	F3	F4
S1	38s ± 7s	44s ± 8s	75s ± 25s	845s ± 444s
S2	48s ± 10s	56s ± 11s	83s ± 22s	644s ± 190s
S3	57s ± 27s	67s ± 16s	87s ± 21s	771s ± 237s

Table 9.5.: Runtime in seconds of the belief management agent in fault scenarios F1 to F4

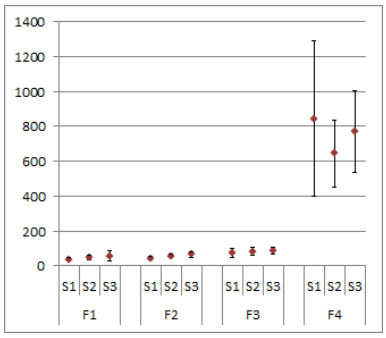

Figure 9.2.: Runtime in seconds of the belief management agent in fault scenarios F1 to F4

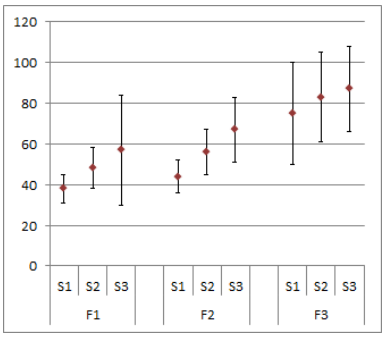

Figure 9.3.: Runtime in seconds of the belief management agent in fault scenarios F1 to F3

scenarios. Albeit the overhead for F1 to F3 is justifiable, it is less justifiable in the scenarios F4 where the runtimes increase dramatically. For a better understanding the data is also shwon in Figure 9.2. As the runtimes dramatically increase in F4, the differences between F1, F2 and F3 are difficult to see. Therefore, Figure 9.3 plots the values F1 to F3, where the upper and lower border represents the average value plus respectively minus the standard deviation.

So we try to compare the performance gain of the belief management system to its runtime loss. Therefor, we introduce performance increase as division of belief management success rate divided by base agent success rate. This ratio is shown in Table 9.6. In scenario S1/F1 there is no potential for improvements as the success rate of the base agent was already 100%. The value increased with higher complexity due to less sensor information in S2 and S3 as well as more faults in F2 and F3. The increase was followed by a short drop in S3/F3, where even the belief management agent had to face a lower success rate of 71 percent. Finally, in F4 the gain was high as the base agent's success rate converged to zero.

	F1	F2	F3	F4
S1	1.0	1.96	5.33	12.83
S2	2.63	4.17	5.33	16.25
S3	3.19	4.95	4.18	17.33

Table 9.6.: Performance gain of the belief management system compared to the base agent. For example, 2 means that the belief management system could successfully finish twice as much missions as the base agent.

Controversial to the gain in performance we have also seen a loss in runtime when comparing our belief management robot to a base robot. This loss is presented in Table 9.7. The table's entries are formed by the division of the belief management runtime by base agent runtime: this means the lower

the number, the more efficient the belief management. While S1/F1 is very efficient, we see a steady increase up to S3/F3. As expected, the values of F4 are very high as the runtimes are very long.

	F1	F2	F3	F4
S1	1.29	1.50	2.55	28.74
S2	1.63	1.90	2.82	21.90
S3	1.94	2.28	2.96	26.22

Table 9.7.: Runtime performance of the belief management system compared to the base agent. For instance, 2 means that the runtime of the belief management system is twice the runtime of the base agent.

Now, equipped with a measurement for performance increase and a measurement for runtime loss we put them together. The underlying question is quite simple, namely "How much runtime am I willing to sacrifice for an improved performance?" A pragmatic approach to this question is a n to n relation, e.g. for twice the performance I am accepting the double amount of runtime (on average!). Although one may prefer a customized approach, we continue with the presented one and use exactly this n to n relation. The corresponding data is shown in Table 9.8. Values above one denote an overall benefit of the system. The bigger the value the higher the reward.

As we have seen before, scenario S1/F1 differs from the remaining ones. Here the setup was relatively simple and the robot could perfectly observe the whole environment. As the base agent finished all missions successfully, it makes no sense to opt for a belief management system that even takes longer in terms of runtime. This drawback is denoted by a system benefit value smaller than one. However, as soon as the setup becomes slightly more complex the belief management has an advantage again. In S2/F2,S3/F2 and S1/F3 the system benefit value is even above two. This means, while the base agent had already difficulties, the belief management agent was able to deliver good results and runtimes. In F4 the runtimes of the belief management agent were too high to receive a system gain value above one.

	F1	F2	F3	F4
S1	0.77	1.31	2.09	0.45
S2	1.61	2.19	1.89	0.74
S3	1.65	2.17	1.41	0.66

Table 9.8.: System benefit: performance benefit times runtime loss of the belief management system compared to the base agent

In a nutshell, we can say that the belief management makes sense as soon as a robot has to execute tasks that are not completely trivial. While action faults and sensing faults can be handled relatively efficiently, exogenous events have a huge negative impact on the agent's runtime. This results from the fact that a significantly higher number of checks is required for exogenous events than for action or sensing variations. While an exogenous event could be placed at any position within an action sequence, variations can only be placed at their endogenous correspondent's position. Theoretically, the O-notation (Tenenbaum, 1995) of application checks is the same, as every action variation might be a variation to every endogenous action. However, please note that this is a rather unrealistic case. For action faults, sensing faults and exogenous events we can state that if runtime is not a key factor, basic belief management should be applied. If either runtime or success rate is essential, we need better strategies.

9.2. Belief Management System on a physical robot?

Besides the details of our belief management in Chapter 4 and 5, we showed in the previous section that a belief management system is helpful for a simulated robot. Within this section we extend this question and evaluate if a belief management system is useful to a real-world robot as well. We evaluate this by posing two subquestions. First, is the robot able to finish a mission while being exposed to all four different kinds of threats as incorrect knowledge, execution failures, exogenous events and sensing failures. Second, we examine the robot's performance on long-term runs.

We executed both experiments with a Pioneer 3-DX robot (MobileRobots, 1999). This robot had a relatively small footprint and was able to turn on the spot thanks to its differential drive. The robot was very agile and thus predestinated for indoor navigation. A gripper allowed for grabbing objects by closing its paddles and lifting the objects. For localization and obstacle avoidance the robot was equipped with a Sick laser range finder. As this laser is limited to obstacles on a plane, we extended the robots perception capabilities by a Microsoft Kinect (Microsoft, 2010) sensor in order to detect obstacles in three-dimensional space. An additional camera allowed for AR tag recognition in order to identify known objects. The setup of the robots strictly followed the description of the robot control framework as depicted in Chapter 6. This setup is shown in Figure 6.3, and a picture of the robot in use in Figure 6.13 on the left side. We used the same T-R-program to find the correct decision for the robot in every situation as in the simulation runs.

9.2.1. Real world robot opposed to four kinds of threats

In the first experiment we tested our robot's capability to handle all four kinds of threats correctly as presented in Section 6.3: (1) incomplete or incorrect knowledge, (2) execution failures, (3) exogenous events and (4) sensing failures. The institute's laboratory served as execution environment for the experiment. It mainly consists of four relevant rooms, namely a seminar room, an aisle containing a kitchen unit (we refer to it as kitchen), an office room and finally a large open space office containing the field of the middle size team, which we refer to as field. Field, office and seminar room are connected to the kitchen as shown in Figure 9.4. Two milk boxes were placed within the environment - a box of unskimmed milk abbreviated by U and and a box of low fat milk abbreviated by L. The robot had the task of delivering these two objects into specified rooms in a given order. While the robot executed these tasks, we altered manually the environment in order to resemble the different threats.

Initially, the robot and the unskimmmed milk were located in the kitchen and the low fat milk was located in the office room. First, the robot had to bring the low fat milk to a specified position in the field. The second task, which had to be executed afterwards, was to bring the unskimmed milk to the office room. Initially, the robot was equipped with two distinct hypotheses regarding the position of the low fat milk. A higher rated but incorrect hypothesis stated that the low fat milk was in the seminar room and another lower ranked hypothesis states the correct position in the office room. This setup resembled incomplete initial knowledge. As soon as the robot received its first task it rushed into the seminar room where it expected to find the low fat milk. Although, when it arrived in the seminar room, there was no sign of the low fat milk. The robot abandoned the hypothesis and selected the next best one, which is in our case the correct one. As the position of the object to be delivered changed, the robot changed its plan and moved to the office room, where it successfully picked up the low fat

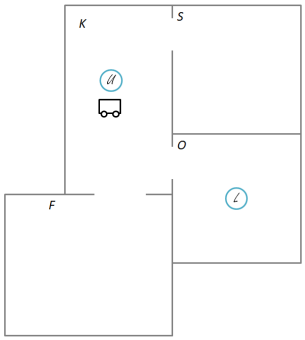

Figure 9.4.: Institute's laboratory that serves as environment for the real world experiment. In the initial situation, the robot and the unskimmed milk are located in the kitchen (K) and the low fat milk is in the office (O).

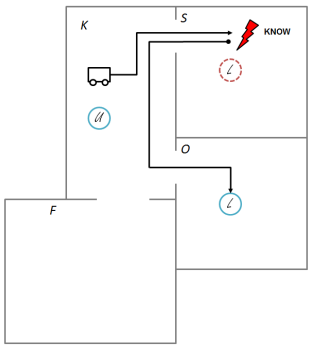

Figure 9.5.: Threat 1: Incomplete knowledge regarding the position of the low fat milk (L). The robot drove from the kitchen (K) to the seminar room (S), where it expected the low fat milk to be, but it is in the office (O) in reality.

milk. The steps so far are shown in Figure 9.5, where incorrect object locations are denoted by dashed circles.

After successfully picking up the low fat milk the robot proceeded to its target destination on the field. While it was trying to release the object, we tricked it again by holding the gripper so it would not open. The robot was unable to release the object, which resembles an execution failure. This failure is labeled EXEC in Figure 9.6. In contrast to the expected positive execution result, the light barrier still signaled an object within the robot's gripper. The sensing formula of the light barrier states that if the sensor is blocked, then the robot holds an object $(lightBar(sf) \Leftrightarrow \exists o : hasObject(o, sf))$. As there is no object o with the property *has_object* the robot faced another inconsistency. This means that the current hypothesis was dropped, history-based diagnosis was applied and the next favorite was selected. In our case, the new favorite was a descendent of the dropped hypothesis. The new hypothesis contained an action variation, namely $putdown(lowfatmilk)$ was replaced by *putdownNothing*. So the robot continued its plan by retrying to execute another $putdown(lowfatmilk)$ action. This time we did not not disturb the robot, so it was able to execute this action successfully and finish the first task.

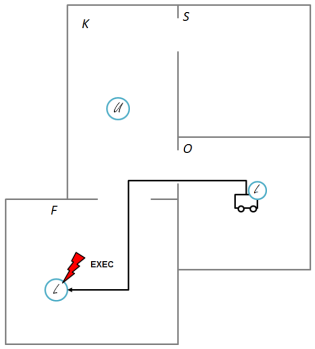

Figure 9.6.: Threat 2: Execution failure - the gripper of the robot is blocked, so it cannot release the low fat milk (F).

The second task was to bring the unskimmed milk to the office room. This task starts with a difficulty, because the initial position of the unskimmed milk was unknown. The location was narrowed down to the kitchen while executing the first task, where the robot traversed the kitchen and sensed the object. In expectation of the unskimmed milk to be found in the kitchen, the robot started moving to this location. In the meantime we manually moved the unskimmed milk from the kitchen into the seminar room (labeled EXOG in Figure 9.7). Hence the robot could not discover any unskimmed milk in the kitchen after its arrival from the field. The sensor result differs from the expectation, which we call an inconsistency by definition. Again the favorite is dropped, history based diagnosis started and a new favorite is selected. Here, history-based diagnosis generates a few equally weighted hypotheses. By applying the exogenous action *exogMoveObject*, which moves an object from one room into another we receive a hypothesis for every room. Therefore, the robot followed the first but incorrect hypothesis into the first room, but it was not successful. Again, it generated a new hypothesis set, however the new hypotheses are ranked worse than the set before, because they contain two exogenous events in contrast to the previous hypotheses that contain one. The hypothesis selected next was generated at the same time as the previous one. It consists of the same actions, but with a different grounding of the *exogMoveObject* exogenous event. The current situation believes the unskimmed milk to be in another room. After some trials the robot chose the correct room and successfully picked up the object.

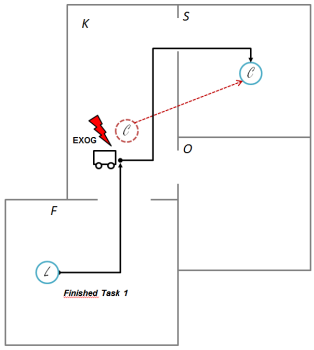

Figure 9.7.: Threat 3: Exogenous event - the robot expected the unskimmed milk (U) to be in the kitchen (K), but it was brought to the seminar room (S). So the robot generated an exogenous event *exogMoveObject* for every room and started driving from room to room, eliminating one wrong hypothesis after the other until it sticked to the correct one.

Three threats are prevented successfully, just one threat is missing. After picking the unskimmed milk successfully, the robot needed to deliver it. The delivery target of this task was the office room. The robot moved to the office room, where it put down the object. Directly after the putdown action, we triggered the light barrier in order to simulate a sensing failure. At the same time the AR unit was able to detect the unskimmed milk standing on the floor. The belief management system came up with two explanations for this inconsistency. First, an execution failure of the putdown action, followed by a sensing failure of the AR unit. Second, a sensing failure of the gripper light barrier. As the second case is more plausible the belief management system selected this concept as the new favorite hypothesis. This means that the belief was repaired and the robot was able to finish its mission by moving to the final destination the kitchen. The situation is depicted in Figure 9.8.

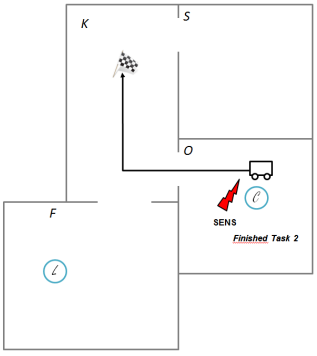

Figure 9.8.: Threat 4: Sensing failure - the robot senses a triggered light barrier due to the unskimmed milk (U) standing on the floor

To sum it up, we can state that the robot solved its first challenge. Confronted with four different kinds of threats, it was able to solve all inconsistencies within its belief and finish its mission. Incorrect initial knowledge, action failure and sensing failure were solved relatively easily, but the exogenous event was very difficult. Here, the robot had to check room by room until it found the object. It has to be noted that this instantiation of an exogenous event is much harder than the *snatch* action in the

earlier examples, because it can be applied much more often (every combination of an object and a room in nearly every situation).

9.2.2. Real world robot long term experiment

While the first experiment was quite artificial, as all four kinds of threats were introduced manually, we attempted for a more natural setup in the second experiment. In this experiment we verified that the robot is able to handle threats to its belief management system that arise from natural execution over a long period of time. So the question behind this hypothesis is "Is a robot that is equipped with a belief management system a reliable robot?". This experiment was executed on the soccer field of our institute's laboratory. We defined five locations, where four objects were placed that the robot had to move around. This means, the robot was asked to pick up object by object and bring it to the empty location. These tasks were continually executed for a typical shift period of eight hours. As one shift is already a good indicator for the reliability of robot, we repeated this experiment three times. As the robot had no ability to charge itself, it started beeping when the battery was low. While beeping, we stopped the robot by a keystroke. Then we did a hot swap of the robot battery and continued the execution with another key stroke. At the same time the battery of the controller notebook was changed. Figure 9.9 gives an impression of this battery change. The execution of the experiment was strenuous, as any human error that led to a power loss, required the experiment to be restarted again on the next day. Details can be found in (Wolfram, 2011).

Figure 9.9.: Battery hot swap during long term robot experiment (Wolfram, 2011)

On average, the robot executed 627 actions per run and finished 147 tasks. As watching a robot moving milk boxes from one location to another is not very thrilling, we had to find another way to indicate whether a task is successful. As the tasks are repetitive and consecutive tasks depend on each other, we opted for a recursive validation. We define that tasks of the last round are finished successfully if the final configuration is correct. This means all spots are occupied by one box, except the next one to be handled. Every task before the last round is defined as successful if the next task with the same box is successful. This follows from the fact that the preceding task fulfills the precondition of the proceeding task. Of course this definition is very restrictive and pessimistic. Although this is no problem, as we are interested in building a stable system that can handle tight deadlines in terms of success rates without problems. This definition allows the executing staff to focus only on the final configuration. In order to reduce the influence of human errors, all numbers of this experiment were retrieved from log-files. The experiment was successfully carried out three times for eight hours, with an average of 5.5 corrections per run.

9.3. Evaluation Diagnosis Templates

So far, we can state that a belief management system can boost the performance of a virtual robot in an adversary environment. Furthermore, we were able to show that it is not only possible to use the proposed system in simulation but on real robots as well. On the one hand our robot successfully coped with numerous of human interventions and on the other hand it was able to execute transfer tasks of milk boxes for several days. Although those results sound very promising, we need to mention the runtime. For example, the exogenous event in Section 9.2.1 forced the robot into a pause of approximately twenty seconds for reflection. So we are interested now, if the presented Diagnosis Templates can rise up to the expectations and boost the runtimes. Therefore, we go back to simulation and carry out comprehensive tests.

Although the execution logic cannot be neglected, it is not our main interest. Hence we continue with the same kind of tasks and let the robot deliver objects from A to B. In order to accomplish such a task, the robot had to go to the source location, pick up the object, move to the destination where it then put down the object. Again three tasks in a row build a mission. Our objective was to make the setup very difficult. Hence we used the same threats as the most difficult example in Section 9.1, where a robot picked up an incorrect object with a probability of 20% or failed to pick it up totally with a probability of 20%, was unable to put down an object with a probability of 30%, or the object was snatched from the robot with a probability of 15%. The robot received sensing information every three actions, which again equaled the most difficult setting mentioned above. On top of these difficulties, we added another exogenous event namely $exogMoveObject(obj, room)$ that moved an object *obj* from its current location to a room named *room* with a probability of 2% without being sensed by the robot. This event is very difficult as the only condition constraining this exogenous action is that the object is not carried by the robot. Furthermore, this event directly depends on the number of objects and rooms. Within the real robot experiments we have seen that such an exogenous event causes long runtimes even if the number of rooms is small. On average, the probability of 2% per object results in one undetected movement every step in case of 50 objects involved. As our robot received sensing information every third action, three objects were moved on average.

So we selected four different scenarios with an increasing number of objects and rooms. In the first scenario we have three objects in twenty rooms. This is followed by ten objects in 29 rooms, 29 objects in 50 rooms and finally 60 objects in 71 rooms. We name the scenarios according to the number of rooms (20 rooms, 29 rooms, 50 rooms and 71 rooms). These scenarios were set up to compare two contestants. The contestant BM implemented a belief management as presented in Section 5.1. Its competitor T used diagnosis templates as presented in Section 7. The diagnosis system was able to handle one fault per repair. In order to provide fair conditions contestant BM's implementation was also altered to handle single faults only. We start with the main comparisons such as the number of successful missions and runtimes and underpin them afterwards with numbers such as number of generated diagnoses. We ran every scenario 100 times per contestant. As the complexity of the scenarios is significantly more difficult than before, the timeout was raised to two hours.

Taking the first results from Table 9.9, we see a clear advantage of diagnosis templates. Over all four scenarios contestant T accomplished around 90 of 100 settings. On the other hand, contestant BM solved only 50 percent of the settings in the easiest scenario. Please note that the easiest scenario here is far more challenging than the most difficult example in Section 9.1. 50 rooms and 29 objects were already difficult enough to let the simple belief management robot fail in nearly every setting due to timeouts. A few lucky strikes could be finished relatively quickly.

	20 rooms	29 rooms	50 rooms	71 rooms
BM	0.50 ± 0.50	0.29 ± 0.42	0.03 ± 0.06	0.05 ± 0.10
T	0.90 ± 0.18	0.92 ± 0.14	0.88 ± 0.22	0.90 ± 0.17

Table 9.9.: Successful missions plus standard deviations - basic belief management (BM) versus diagnosis templates (T) (Reip et al., 2012)

Next, we reinforce the presented results by examining the sub-results of missions, namely successful tasks. In order to successfully finish a mission, the robot had to finish three tasks successfully. Hypothetically, it is possible that the basic belief management agent was able to successfully finish two tasks in every run, but failed to finish the third one. Although this is clearly not satisfying, the performance difference to diagnosis template robot would not be as high as indicated in Table 9.9. The results of the task comparison are shown in Table 9.10.

	20 rooms	29 rooms	50 rooms	71 rooms
BM	2.18 ± 0.83	1.88 ± 0.75	0.52 ± 0.68	0.46 ± 0.65
T	2.89 ± 0.20	2.90 ± 0.18	2.85 ± 0.26	2.89 ± 0.20

Table 9.10.: Successful tasks plus standard deviations - basic belief management (BM) versus diagnosis templates (T) (Reip et al., 2012)

Similar to the mission success results, these results show a constant performance of the diagnosis template agent over all examples. On the other hand, the performance of the basic belief management agent declines as soon as the difficulty of the setting increases. Whereas in setting 20 rooms the basic belief management agent was able to fail in less than one task on average, these results could not be held in setting 29 rooms. The settings 50 and 71 were even less successful. In this case contestant BM could not even finish a single task on average. This can either be due to the timeout of 7200s or that no executable action is left.

After investigating the performance of the diagnosis template robot versus the basic belief management robot, we are interested in the runtimes. Does the diagnosis template robot require longer execution times in order to solve the examples, that its predecessor base belief management robot could not solve? First, we only compared the runtimes of successful missions. This means timeouts are not included. The results are shown in Table 9.11. Generating the diagnosis templates that are used during the execution runs is carried out quickly, it takes less than a second. Hence we generate all diagnosis templates at the beginning of every run and therefore the total runtime of a diagnosis template robot run includes the runtime for diagnosis generation.

	20 rooms	29 rooms	50 rooms	71 rooms
BM	$417.5s \pm 303.9s$	$2770.3s \pm 2314.7s$	$27.7s \pm 4.3s$	$314.0s \pm 472.0s$
T	$28.9s \pm 16.2s$	$39.6s \pm 32.1s$	$164.8s \pm 204.8s$	$274.4s \pm 176.7s$

Table 9.11.: Runtimes plus standard deviations in seconds of successful missions - basic belief management versus diagnosis templates with standard deviations (Reip et al., 2012).

The runtimes of the diagnosis template agent roughly follow a linear increase in the number of objects. However, this is completely different with the basic belief management agent. Here we see a steep increase in runtime from 20 to 29 rooms, followed by an incredible runtime boost with

50 rooms and a reasonable runtime with 71 rooms. The question is, what is the dominator of the basic belief management agent's runtimes. This leads to a twofold answer. First, the runtimes are clearly dominated by the complexity increase of the examples. This explains the increase between 20 and 29 rooms. As soon as 50 rooms are reached, we have seen that the basic agent was completely overloaded and was only able to solve easiest setups. And in case the setup was easy enough the basic agent solved it pretty fast. Therefore the deviation is also small. In the setting of 71 rooms, the basic agent again solved only simple setups very quickly, plus a few where it took much longer to solve them. So we see again an increase in the average runtime, but in the deviation as well.

In order to get a better picture of the runtimes of the basic agent we compare all runtimes of successful and unsuccessful runs in Table 9.12. Now, basic agent runtimes also follow a steady increase. With 20 rooms, the difference between all runtimes and successful runtimes is only 71 seconds on average. This sounds reasonable because the basic agent is able to solve a good number of the 20 room setups. We have seen before that the performance declines with 29 rooms. This relates with an increase of runtimes of nearly 1000 seconds per run. Finally, the runtimes of 50 and 71 rooms are the combination of timeouts plus a few easy setups that can be solved quickly.

	20 rooms	29 rooms	50 rooms	71 rooms
BM	$488.0s \pm 386.6s$	$3523.0s \pm 2588.2s$	$6647.2s \pm 1046.7s$	$6826.9s \pm 731.8s$
T	$28.7s \pm 15.8s$	$38.5s \pm 30.0s$	$187.8s \pm 249.8s$	$348.9s \pm 300.4s$

Table 9.12.: Runtimes plus standard deviations in seconds of successful and unsuccessful missions with a timeout of two hours (7200 seconds) - basic belief management (BM) versus diagnosis templates (T) with standard deviations (Reip et al., 2012)

The timeouts are investigated separately in Table 9.13. Up to 60 objects in 71 rooms combined with a load of injected failures and exogenous events can give an agent a hard time to finish its mission, especially when the objects to be delivered are moved across the execution environment. In such cases the agent has to browse room by room in order to locate the true position of an object. Additionally while moving from room to room, new problems may arise. This can explain long execution times. Nevertheless the diagnosis template agent was far from running into timeouts regularly. Only in the biggest setup a timeout happened once in hundred runs. This was completely different to the basic belief management agent. Confronted with huge amount of inappropriate hypotheses that had to be checked for consistency without success or evaluating hypotheses that came up with more or less the same explanation cost this agent a lot of processing time. Even a rather large timeout of two hours troubled this agent, as nearly every fifth execution in the setup of 50 rooms led to a timeout. This number increased to over 90% as soon we had 50 rooms or more.

	20 rooms	29 rooms	50 rooms	71 rooms
BM	$0\% \pm 0\%$	$19\% \pm 31\%$	$91\% \pm 17\%$	$93\% \pm 13\%$
T	$0\% \pm 0\%$	$0\% \pm 0\%$	$0\% \pm 0\%$	$1\% \pm 1\%$

Table 9.13.: Timeouts (7200 seconds) plus standard deviations - basic belief management (BM) versus diagnosis templates (T) with standard deviations (Reip et al., 2012)

After investigating the most important numbers as runtimes and success rates, we can clearly state that the diagnosis template agent outperforms the basic belief management agent and is in a league of its own. Regarding the number of successful missions, the diagnosis template agent is completely

out of reach. It is able to solve scenarios with satisfying success rates where the basic agent can score only lucky strikes. This tremendous increase in performance is accompanied by improved runtimes too. There is such a difference that is difficult to compare the two instances. In the last scenario with 71 rooms the runtime of the base belief management agent is nearly 20 times higher than the runtime of the diagnosis template agent. And this number is already limited by the timeout. We have seen that a belief management is helpful in contrast to a system without belief management, but as soon the setups get harder a sophisticated strategy is necessary. Diagnosis templates can be identified as such a sophisticated strategy. As it is very difficult to compare a diagnosis template agent with a basic belief management agent, we continue by comparing different setups of diagnosis template agents in order to receive a better understanding of the system.

We keep the setup of 20, 29, 50 and 71 rooms but use a different number of objects. We call the room - objection combination mentioned above scenario A and the additional combination scenario B. Table 9.14 gives an overview of the combinations in use. It is the same combination that was used in (Gspandl, Podesser, Reip, Steinbauer, and Wolfram, 2012) for the scalability experiments.

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	3	10	29	60
Scenario B	5	18	32	90

Table 9.14.: Object room combinations for the diagnosis template agent (Gspandl, Podesser, Reip, Steinbauer, and Wolfram, 2012).

The runtimes of scenario A were already shown in Table 9.12. In Table 9.15 we include the runtimes of scenario B and show additionally the maximum values. Especially the maximum values are interesting as they are relatively high in scenarios with 50 rooms or more. So we have to ask if these max values are outliers or if they occur regularly. Especially in Scenario A 50 rooms, where the runtime is longer than in the corresponding Scenario B, which has more objects. The high standard deviation already let us assume that the influence of peak(s) is essential. In order to evaluate these assumptions, the runtimes of setup A 50 rooms can be found in Figure 9.10. Here we can clearly see that the majority (88%) of runtimes is below 100 seconds and just a few of them are above. As the gap between the different runtimes per setup is very high, the logarithmic scale allows for better reading. We can see that peaks have a high influnce on average runtimes.

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	$28.9 \pm 16.2 (190)$	39.6 ± 32.1 (588)	$164.8 \pm 204.8 (7047)$	$274.4 \pm 176.7 (5735)$
Scenario B	$28.2 \pm 16.1 (263)$	31.1 ± 14.7 (537)	$101.7 \pm 76.8 (2598)$	$392.9 \pm 209.42 (4668)$

Table 9.15.: Runtimes in seconds of diagnosis template agent with standard deviation and max value

Apart from a few outliers the average runtime increase of the diagnosis template agent over all scenarios in Table 9.15 is linear. This can be supported by linear regression. The coefficient for objects is 3.86 and the coefficient for rooms is 0.85. In combination the two coefficients are able to explain 95 percent of the data. But of course the runtime is the sum of many subparts such as decision making, execution, consistency checks, hypothesis generation and hypothesis selection. Execution itself can almost be neglected as we act in a simulated environment. Decision-making in combination with execution depends on the action sequence length. The longer the sequence, the more decisions have to be made. Every decision is built on regression calls that take some time. From the first

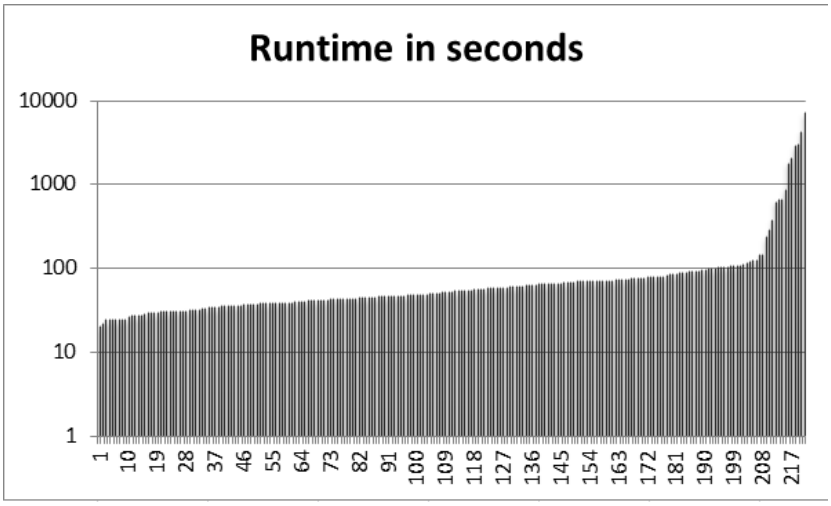

Figure 9.10.: Diagnosis Template runtimes 50 rooms, 29 objects. Please note the algorithmic scale.

experiment with an agent without belief management, we know that it takes about 29 seconds in order to finish a mission. The more complex a scenario becomes, the more time is required for the belief management part. In this context the most time consuming part is to generate new hypotheses and to check them for consistency. We extracted the sub runtimes for hypotheses generation and their consistency checks. The average runtimes are listed in Table 9.16. Maximum values can be found in Table 9.17.

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	$0.21s \pm 0.11s$	$0.36s \pm 0.18s$	$1.99s \pm 0.9s$	$9.29s \pm 4.5s$
Scenario B	$0.18s \pm 0.08s$	$0.44s \pm 0.16s$	$3.76s \pm 3.00s$	$15.12s \pm 6.07s$

Table 9.16.: Average diagnosis generation and consistency check runtimes in seconds of diagnosis template agent with standard deviation

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	$0.75s$	$5.24s$	$14.6s$	$140.48s$
Scenario B	$0.59s$	$2.04s$	$234.65s$	$91.64s$

Table 9.17.: Maximum diagnosis generation and consistency check runtimes in seconds of diagnosis template agent

For diagnosis generation and consistency checking it is easier to imagine a dependency on the number of rooms and objects than for the complete runtime. In the simplest case, the exogenous event *exogMoveObject* leads to more instances the more rooms exist. The higher the number of objects, the higher is the chance to encounter objects in unexpected rooms. Again we applied linear regression in order to investigate the relation between generation runtimes and the number of objects and rooms. In this case we can explain even 97% with the coefficients 0.22 for objects and -0.06 for rooms. The increase in explicable data might be explained as follows. The complete runtime depends heavily on the run of events. For example if an object was moved to the last room within the universe, the robot has to search room for room till it finds the object to be delivered. This means, the total number of

executed actions has an influence on the runtime. On the other hand, the diagnosis generation has to generate all possible hypotheses according to the encountered inconsistencies. This number does not depend on a specific run of events.

Next we investigate the number of executed actions. The values have very high deviations. Not even if we remove a portion of 10 percent of outliers, this fact still remains. Table 9.18 and 9.19 show the number of total actions and actively executed actions (excluding exogenous events and sensing actions) per run respectively. The number of actively executed actions in Table 9.19 is relatively stable across all scenarios. On the other hand, the number of total executed actions increases. This can be explained by a rising number of sensing actions. At each sensing step every object in the same room as the robot was sensed by a single sensing action. Thus, the more objects were seen, the more actions occurred. Although, the number of executed actions does not reveal any new information.

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	$149 \pm 48 (379)$	138 ± 29 (263)	$152 \pm 25 (217)$	$175 \pm 25 (250)$
Scenario B	$145 \pm 33 (302)$	139 ± 22 (198)	$154 \pm 23 (213)$	$184 \pm 29 (252)$

Table 9.18.: Average total number of executed actions including sensing and exogenous events after removing 10 percent of outliers. The numbers are given with standard deviation and max value.

	20 rooms	29 rooms	50 rooms	71 rooms
Scenario A	$49 \pm 16 (124)$	41 ± 9 (78)	$43 \pm 7 (62)$	$48 \pm 7 (67)$
Scenario B	$45 \pm 10 (94)$	40 ± 6 (56)	$43 \pm 6 (58)$	$47 \pm 7 (63)$

Table 9.19.: Average number of actively executed actions (endogenous actions) after removing 10 percent of outliers. The numbers are given with standard deviation and max value.

As a summary, we have seen that belief management can boost the performance of a robot. Furthermore, we could show that diagnosis templates are a crucial element regarding the runtime of the belief management. Besides improving runtimes, the performance was enhanced as well. This follows the basic idea of diagnosis templates, namely continuing with relevant hypotheses instead of evaluating worthless traces. A number of key value evaluations allows a deeper insight in the behavior of these diagnosis templates.

Throughout the evaluation we stick to the delivery domain. From a decision-making perspective this may be a narrow view. However, the diagnosis perspective is completely different. We started with a relatively simple scenario and made it more and more difficult. In the first step we added additional failures and exogenous events. This could be tackled by introducing diagnosis templates. When using diagnosis templates we saw that creating templates is relatively easy and the difficulty lies in its application. A template can be seen as a change together with its condition. This condition is the action sequence that follows the change action. We call it condition as the action sequence has to occur within the situation to be repaired. This occurrence check is done by unification and thus is very efficient as well, similar to inserting change actions into a given action sequence. The most expensive task is the consistency check for a given hypothesis. Therefore we applied several strategies in order to reduce the amount of consistency checks.

Based on this we know that templates with a specific condition are easy to handle. In contrast if the conditioning action sequence is empty, the change is difficult to handle as it can be applied

very often. This difficulty can even be increased if the change action allows multiple different object instantiations. Exactly this circumstance is realized by the exogenous event *exogMoveObject*. It comes without conditional action sequence and takes a room and additionally an object as argument. So the final scenario to be diagnosed is completely different from the initial one.

Chapter 10

Domain Knowledge generation

Building up domain knowledge is very time consuming and error-prone, which is a hurdle for using knowledge based systems. Therefore, we are interested in automatic generation processes that allow more people and domains to use knowledge based systems. Here, every approach that is more efficient than writing domain knowledge by hand is welcome. There is a broad range of approaches in generating domains and additional ones in knowledge sharing such as (Waibel et al., 2011). Another approach within the same funding project was taken by Mühlbacher and Steinbauer (2014b). They proposed to reuse existing common-sense knowledge bases. This reuse of knowledge is realized by a mapping between the IndiGolog-based high-level control and the common sense ontology Cyc (Guha and Lenat, 1990). Specifically Mühlbacher and Steinbauer (2014b) use Cyc to specify invariants. In addition to a proof-of-concept implementation they discuss the circumstances when such a mapping is valid and successful. Though the runtime performance is low, the system is able so improve the detection and diagnosis of inconsistencies in an agent's belief. The contents of this chapter have been published in the papers Gspandl, Monichi, Reip, Steinbauer, Wolfram, and Zehenter (2007), Gspandl, Hechenblaickner, Reip, Steinbauer, Wolfram, and Zehentner (2012) and Gspandl, Reip, Steinbauer, and Wotawa (2010).

In the reminder of this chapter we present two strategies in order to generate domain knowledge. The first approach in Section 10.1 generates agent plans out of soccer tactics sketches. The second approach in Section 10.2 takes tactics descriptions in natural language and translates them into an ontology. In a further step this ontology used to do high-level control of a simulated soccer agent.

10.1. From Sketch to Plan

In (Gspandl, Reip, Steinbauer, and Wotawa, 2010) we presented a methodology to translate soccer tactics sketches or drawn strokes into agent plans that define the behavior of RoboCup Simulation League agents. The inputs are tactics graphs as indicated in Figure 10.1. Sketches like this one are easy to understand for humans. Here the two strikers, as represented by outlined circles, try to create a scoring situation against three defenders drawn by filled circles. The sketch consists of multiple facts and relations such as striker1 is marked and it cannot pass by the defense line, but can dribble along it. The difficult task is to discriminate between relevant and irrelevant facts. For example, nobody cares

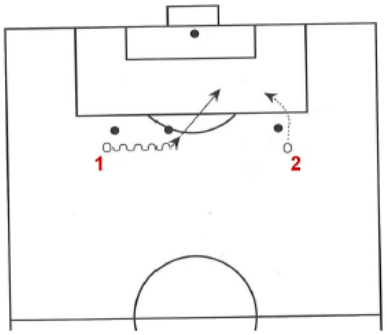

Figure 10.1.: Exemplary soccer tactics sketch that is automatically transformed into a soccer agent plan.

about the exact distance between striker1 and its closest defender. In order to meet domain-specific tasks, a pipeline was installed to meet the requirements of the resulting transformation.

The first step of the transformation pipeline is the extraction of spatial-temporal representation. This representation is domain independent and consists of primitives such as circles, boxes or strokes. The next step takes this domain independent representation and adds domain specific interpretations. In soccer, this relates to discriminating between own and opponent players, assigning different strokes to different actions and so forth. The example in Figure 10.1 leads to the following quantitative facts:

- agent(a1).
- own(a1).
- pos(a1, 28.07, -13.04).
- field(f1, 52.5, -34, 52.5, 34, -52.5, 34, -52.5, -34). // the field is defined by its corners
- action(d1, dribble, 28.07, -13.04, 28.71, -3.59). // an action has a type plus start/end
- ...

The next step, called sketch interpretation transforms the domain-specific recognition results into a domain-specific data structure (plan) that can be used by the agents. This starts by converting the quantitative information into qualitative information. Therefor, we apply several qualitative rules. This results in the following qualitative statements:

- defensive(opponent1) ∧
  defensive(opponent2) ∧
  defensive(opponent3) ∧
  offensive(own1) ∧
  offensive(own2) ∧
  pos(own1, outside penalty left) ∧
  ...
- hasBall(own1) ∧
  dribbleFreeSector(own1, right) ∧
  marking(opponent1,own1) ∧
  ...

In the next step we generate one Plan Tree per sketch. A Plan Tree is the abstracted behavior of several agents describing the given sketch. The resulting Plan Tree for the running example is shown in Figure 10.2. A Plan Tree is generated in four sub-steps. The first sub-step derives possible action templates. This includes the action preconditions, its invariant and its effects. For the action pass this would be the possession of the ball as precondition. The movement of the ball from its origin to the pass partner defines the invariant and the possession of the ball by the pass partner is the postcondition. The second sub-step defines the Plan Tree precondition by the set of qualitative statements. In the third step every action in the Plan Tree is specified. The precondition of every action is the postcondition of the previous action. If it is the first action of an agent, the precondition is equal to the Plan Tree precondition. The invariant of the action is equal to the invariant of the action template. The postcondition is derived by the progression of the action effects on the action's precondition. After defining the precondition, invariant and postcondition, the action is inserted into the Plan Tree according to the executing player and the temporal sequence. Finally in a fourth sub-step, the plan invariants are derived by the combination of all action invariants.

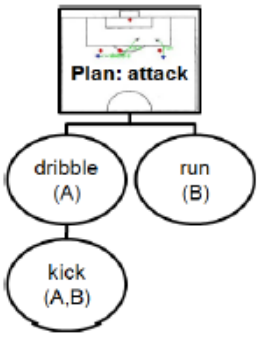

Figure 10.2.: The plan resulting from the transformation of the tactics graph in Figure 10.1 (Gspandl, Reip, Steinbauer, and Wotawa, 2010).

Typically, several plans are used together in order to define the behavior of an agent. This transformation process is represented in Figure 10.5. For the evaluation we applied the presented system to twelve sketches and received a corresponding plan set. On 15 games in RoboCup Soccer Simulation League we evaluated the coverage and the quality of this plan set. The plan coverage defines how many of all offensive situations can be described by the generated plan set. The quality of the plan set evaluates if the identified situation resembles the the correct sketch. The resulting coverage of 30% is shown in Figure 10.3. There we can see that most of the coverage is due to five or six sketches. Figure 10.4 shows the plan quality with an average value of 44%. Although, this number is not very high, it is able to automatically label a considerable number of situations.

After investigating the coverage and the quality of the plan set, we tested if the generated plan set is able to improve the performance of a RoboCup Soccer Simulation League team. Thereto, we measured the the performance of a team consisting of simple agents and the same team enhanced by the generated plan set against a common opponent. Each team had to play 25 games. While the basic team was able to score only 2.65 goals per game on average, the enhanced team scored 2.8 goals on average. This indicates, that enhancing the behavior of a simple agent team by an automatically generated plan set increases the scoring quality of a team.

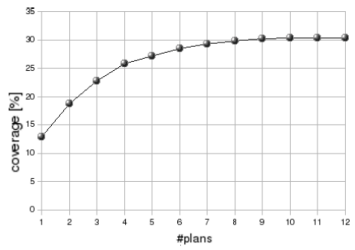

Figure 10.3.: The accumulated coverage of twelve plans in 15 games over all offensive situations (Gspandl, Reip, Steinbauer, and Wotawa, 2010).

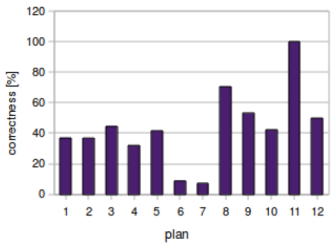

Figure 10.4.: The quality of each of the twelve plans in percentage of correctly classified situations (Gspandl, Reip, Steinbauer, and Wotawa, 2010).

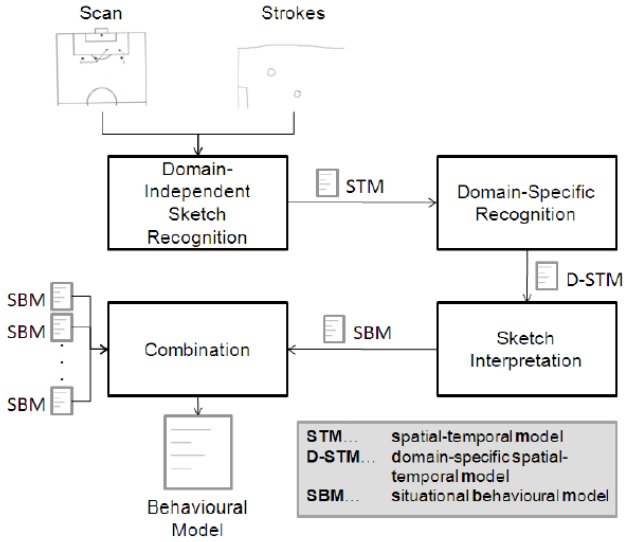

Figure 10.5.: The process of automatically deriving a behavioral model from several tactics graphs or sketches (Gspandl, Reip, Steinbauer, and Wotawa, 2010).

10.2. Ontology population from text

In (Gspandl, Hechenblaickner, Reip, Steinbauer, Wolfram, and Zehentner, 2012) the overall goal was the same, namely to create behavioral plans for a RoboCup Soccer Simulation League team. But in contrast to tactics sketches we used natural language processing. The input to this generation process is soccer literature as represented in Figure 10.6. In order to facilitate human understanding a sketch is added to the description similar to the one we provided before as shown in Figure 10.7. Populating knowledge bases of any kind by natural language processing (Jurafsky and Martin, 2000) is very popular, since a huge amount of natural language is available (see Cimiano (2006); Chen and Mooney (2008)). The presented transformation consists of several steps that are shown in Figure 10.8.

First, we start with a preprocessing that applies several natural language processing techniques; within this preprocessing step, we split up text into atomic action descriptions by detecting stop words such as "and" and "like". Next, we apply the Stanford part of speech tagger (Toutanova and Manning,

```
* 5 passes the ball to 9 (who has come towards him) and moves
  diagonally to get the return pass;
* 9 passes the ball back to 5;
* 10 sprints deep to the right, criss-crossing with 9 who does the
  same thing from the opposite side;
* 5 can choose to pass the ball to either 9 or 10.
```

Figure 10.6.: Natural Language description of opponent field penetration (Fascetti and Scaia, 1998).

Figure 10.7.: Accompanying sketch of opponent field penetration (Fascetti and Scaia, 1998).

2000) that provides the lexical category to each word. Based on this analysis we can remove unnecessary lexical categories and create tupels of subject, predicate and object (SPO). After identifying subjects we can replace pronouns such as "who" by the real actor. For instance, the text (’5 passes the ball to 9’) is replaced by the SPO {5; pass; 9}, or the pronouns from (’who has come towards him’) are substituted by ’9’ and ’5’. Next, we apply temporal corrections to order the tuples by identifying tenses and temporal keywords such as "while". In the last preprocessing step we resolve synonyms by looking up every subject, predicate and object in the WordNet (Miller, 1995) dictionary and replace it by a base form.

After preprocessing we have a set of temporal sorted subject, predicate, object tupels that are mapped to unique base forms. The next step is the population of the ontology itself. To this extend, we generate test strings from the ontology concepts and compare them to the input tuples. For instance, the ontology contains a taxonomy tree ("-" represents a subclass) Object - GameEntity - Player - Wing - Left Wing. Here, every taxonomy entry can consist of multiple test strings. In our case we simply use the names of the entries and compare them to the incoming tuples. Comparison is rated by an adapted version of the Levenshtein distance (see Levenshtein (1966)) on word level. Then we iterate over subject, object, predicate tuples and create an action compound for the new tactic in the ontology. Every action is mapped to a new subclass. Then the temporal context is applied by either creating parallel actions or action sequences. The precondition is not given by the text. It is set in a parallel step by processing the accompanying sketch (e.g. geometric relations). Finally a postprocessing step ensures that the resulting action compound can be executed. For example we have to ensure that the action crisscross is part of both players’ plans although just one player is used as subject in the description (10 criss-cross 9).

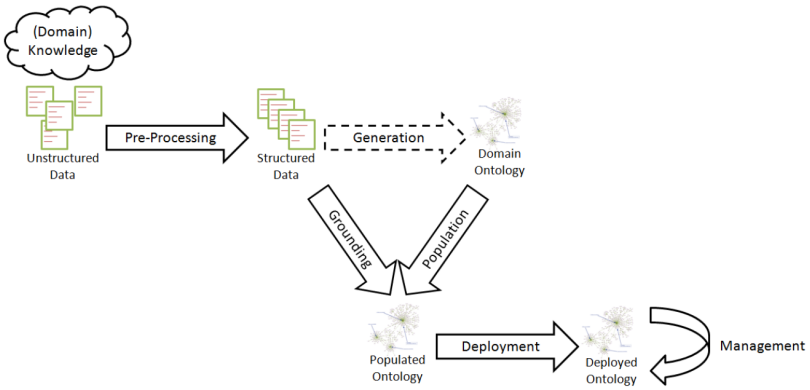

Figure 10.8.: Transformation pipeline from natural text to an ontology ready for execution (Gspandl, Hechenblaickner, Reip, Steinbauer, Wolfram, and Zehentner, 2012)

The presented methodology was evaluated by transferring sixteen tactics descriptions with a total of 129 actions. The results were checked by an expert with the successful result that 71 percent of the actions was perfectly recognized. Another 20 percent was close to being identified, which means that after applying a few additional rules, they could be correctly identified. The temporal sequence was identified correctly in all cases. After checking the transformation we additionally checked if Simulation League Agents are able to execute the transformed plans. We corrected incorrectly transformed plans by hand and set the players automatically to their original positions. Finally, the execution was started without execution noise. This means one test per scenario is sufficient in order to check the success of the execution. After the execution of the run we checked the geometric relations of all players and the ball in order to validate if the run is successful. The results show that 86 percent of the executions were carried out successfully (Gspandl, Hechenblaickner, Reip, Steinbauer, Wolfram, and Zehentner, 2012).

Chapter 11

Conclusion

This chapter starts with a summary of the previous chapters and outlines key findings of this thesis. This is followed by Chapter 11.2, where suggestions for further research are offered.

11.1. Summary

The aim of this thesis is to provide a robot with a consistent description of the current world state in order to ensure that a robot can make the correct or optimal decision. This world state description is normally realized by some kind of World Model. World Models in the classical sense are able to store and provide properties as well as providing reasoning. However, the fact that the knowledge base may consist of contradicting facts represents a common problem in robotics. In order to overcome such contradicting facts or inconsistencies we extended the World Model with a component called belief management. Such a belief management is expected to provide the remaining system with a consistent description of the world along the task-execution by the robot. Providing a consistent world description can be done by consistency checking, diagnosis and repair. If the belief of a robot turns inconsistent, we have to identify the underlying problem by applying a diagnosis and repair the belief. The thesis shows that such a belief management system is a crucial component for truly autonomous robots.

A belief management system gives robots the possibility to overcome several types of threats. These threats are caused by action and sensing failures, incomplete or wrong initial knowledge, or exogenous events and have the potential to turn a robot's belief inconsistent. Whereas different domains offer strategies to handle inconsistencies locally, handling inconsistencies is rather difficult for robots that act in the same environment as humans. Imagine an electronic circuit where the result of a logical gate contradicts the expectation. By applying model based diagnosis we can identify a subset of all involved gates and mark them as abnormal. This works well, because all components involved can be identified easily. However, this is not true for real world scenarios where people might enter the scene, do something and leave it again without giving robots the chance to notice the people. Humans are interesting for a robot if they do something that changes the world. It is very difficult to describe the possible influence of a human to the world. If a person would be marked as working abnormal, we would have no idea about action outcomes of this person. This means, that if we know the person

tried to achieve something, nearly everything could have happened. The same is true for the robot itself, as it can actively change the environment by taking different (unwanted) actions, the actions' outcome are unclear if the robot is marked abnormal.

An explicit notion of action is used by the situation calculus. This first order language uses a description of an initial situation plus a sequence of actions to describe the environment. We found out that this kind of notion works well with robots, as actions are basic concepts of robots and the situation calculus. This action-driven formulation presents new challenges to the diagnosis process of the belief management. Rather than describing components as abnormal, we are interested in the true progression of the world. This again can be described by an action sequence, which exactly leads to history-based diagnosis. For example if a robot arrives at the destination of a transport task without the object to be transported and it figures out that the object might have been lost during the move, it does not only know that the object is not at the destination, but it also knows that the object must be in one of the rooms along the delivery path. Therefore, information on the exact action sequence can reveal additional and important knowledge.

Building up on history-based diagnosis (Iwan, 2002) we start with an initial version of a belief management component. We created a formal description of the belief management. In a further step an IndiGolog implemention was realized. This first version was already able to outperform simulated robots without belief management system in terms of scenarios that could be finished successfully. In a next step we designed a framework for a real robot system that incorporates this belief management system. In this test we found that this system is able to manage all threats that are described above. Though the success rate of this system is satisfying the runtime performance is clearly not. Even in small scenarios, the robot takes a substantial amount of time to identify inconsistent situations, diagnose these situations and repair the belief. As incomprehensible robot behavior clearly lowers the acceptance rate, an intensive investigation of the runtime was undertaken. This investigation showed that a robot needs the greatest part of the time to calculate multiple fault combinations that explain the current situations, though in most of the times an inconsistency is due to a single fault. As a consequence we focused on unconnected single faults. This means we try to explain every inconsistency by a single fault. Numerous single faults may occur during a robot's lifetime.

Furthermore we improved the uninformed search for those single faults by a directed search and introduced the concept of diagnosis templates. Diagnosis templates explain a specific fluent value together with a condition that is necessary to achieve this fluent value. Diagnosis templates can be precomputed efficiently based on a domain description and are simply applied at runtime. History-based diagnosis with diagnosis templates works as follows: if an inconsistency is detected, the inconsistent fluents are identified first. In case of an inconsistency that arises from a sensing action, this is the set of fluents where the expected value differs from the (indirectly) measured one. Second, we select all templates that explain the inconsistent fluent set. Third, we apply the diagnosis templates and check the resulting explanations for consistency. Using this approach we significantly reduce the number of consistency checks for explanations that have nothing to do with the real cause (e.g. failing to put down an object does not explain why the robot is carrying nothing though it is expected to do so). This led to a breakthrough in terms of runtime. In order to ensure that this directed search does not miss any explanations, we proved this circumstance.

In the end we have an efficient system that is able to cope with a variety of threats that are typical in real world environments. The system is embedded in a robot control framework that was successfully tested in numerous simulated experiments as well as on a real robot.

11.2. Further research

Apart from the findings of this thesis, there are several suggestions of further research that should be investigated. A straight-forward direction is to extend the work on diagnosis templates and expand these diagnosis templates to multiple faults. This implies that, during template generation one does not stop the search as soon a change action is found. Equally to non-change actions the search is continued based on the preconditions of change actions. In large domains a maximum number of faults might still be necessary. Similar to template generation, template application also needs to continue after the application of the first change action.

The more hypotheses are generated, the more hypotheses have to be stored within the hypothesis pool. This is no big issue regarding memory, as hard-discs grow larger and larger and cloud storage is an omnipresent term. The limitation results from the management work that is necessary to keep the pool up-to-date and the search for new favorites. Chapter 8 provides first steps into an efficient pool management. This could be realized by recognizing hypotheses that can be removed from the pool without checking them for consistency. Every time an inconsistency arises, we generate new hypotheses - the correct one, plus a possibly empty set of alternatives that explain the same inconsistency. So, as soon we recognize the correct hypothesis we could eliminate the alternative ones from the pool without a time-consuming consistency check.

Similar to the question of which hypotheses to keep, is the question of which inconsistency to investigate. For example imagine the robot's task is to deliver coffee cups from the kitchen to the dining room and it discovers a milk box close to the kitchen sink instead within the fridge. If delivering coffee cups is the robot's only task the position of the milk box is completely irrelevant to the robot and inconsistencies regarding the milk box can be neglected. On the other hand, if the coffee cup delivery task is followed by a milk box delivery task the position of the milk is relevant. Apart from the general halting problem, Indigolog offers a few constructs that make forward checks very difficult like non-deterministic choice of arguments (see Section 2.2.1). The authors of (Podesser et al., 2012) take the upcoming primitive actions as input and infer if the current inconsistency is relevant for further execution. A related strategy is active diagnosis. After explaining an inconsistency by multiple diagnosis, the aim of active diagnosis is to select further sensing actions that allow for decreasing the size of the explaining diagnosis set (Mühlbacher and Steinbauer, 2014a; Kuhn et al., 2008). A similar strategy was proposed by Alur et al. (1995). They discuss discrimination strategies in the context of non-deterministic and probabilistic state machines.

No matter which system, runtime is an issue in the majority of cases. Though the different algorithms for belief management have led to a tremendous gain in runtime performance, the remaining potential is still high. First order logic is very convenient to program, even so the runtime of its algorithms is generally bad. In case of enumerable domains, the user should be given the chance to express its ideas in first order logic, but the execution and computation should take place in some kind of multi-valued representation that can provide fast implementations similar to achievements within the field of planning (Helmert, 2006). The same is true for the regression property of the situation calculus (Reiter, 2001). It is very convenient for theoretic evaluations, but during execution the same things are evaluated over and over. Improving those two factors can speed up the system significantly and hence contribute to the acceptance of this and similar systems.

One very important topic regarding the practicability of the present approach is knowledge generation. Although, first findings were presented in Chapter 10, this process needs to be much more convenient. If a specialist is necessary in order to create a domain description, it is very likely that the

system will only be used on rare occasions. The goal is to have a simple strategy, that allows users to apply the system without any worries of how to get started.

Bibliography

ALAMI, R., CHATILA, R., FLEURY, S., GHALLAB, M., AND INGRAND, F. 1998. An architecture for autonomy. *International Journal of Robotics Research 17*, 315–337. (Cited on page 6.)

ALUR, R., COURCOUBETIS, C., AND YANNAKAKIS, M. 1995. Distinguishing tests for nondeterministic and probabilistic machines. In *Annual ACM symposium on Theory of computing (STOC)*. 363–372. (Cited on page 127.)

AZUMA, R. T. 1997. A Survey of Augmented Reality. *Presence 6*, 355–385. (Cited on page 65.)

BACCHUS, F., HALPERN, J. Y., AND LEVESQUE, H. J. 1999. Reasoning about noisy sensors and effectors in the situation calculus. *Artificial Intelligence 111,* 1âĂŞ2, 171–208. (Cited on page 37.)

BEETZ, M., JAIN, D., MÖSENLECHNER, L., AND TENORTH, M. 2010. Towards performing everyday manipulation activities. *Robotics and Autonomous Systems*. (Cited on page 5.)

BEETZ, M., MÖSENLECHNER, L., AND TENORTH, M. 2010. CRAM – A Cognitive Robot Abstract Machine for Everyday Manipulation in Human Environments. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Taipei, Taiwan, 1012–1017. (Cited on page 5.)

BIENVENU, M., FRITZ, C., AND MCILRAITH, S. A. 2006. Planning with qualitative temporal preferences. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, Lake District, UK, 134–144. (Cited on page 34.)

CAMPBELL, M., HOANE, JR., A. J., AND HSU, F.-H. 2002. Deep blue. *Artif. Intell. 134,* 1-2 (Jan.), 57–83. (Cited on page 59.)

CHEN, D. AND MOONEY, R. 2008. Learning to sportscast: A test of grounded language acquisition. In *In 25th International Conference on Machine Learning*. Helsinki, Finnland. (Cited on page 122.)

CIMIANO, P. 2006. *Ontology Learning and Population from Text: Algorithms, Evaluation and Applications*. Springer, New York. (Cited on page 122.)

DE GIACOMO, G., LESPÉRANCE, Y., AND LEVESQUE, H. J. 2000. Congolog, a concurrent programming language based on the situation calculus. *Artif. Intell. 121,* 1-2 (Aug.), 109–169. (Cited on page 19.)

DE GIACOMO, G., LESPÉRANCE, Y., LEVESQUE, H. J., AND SARDIÑA, S. 2009. IndiGolog: A High-Level programming language for embedded reasoning agents. In *Multi-Agent Programming*, A. E. F. Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, Eds. Springer US, New York, 31–72. (Cited on pages xiii, 4, 8, 23, 42, and 49.)

DE GIACOMO, G., REITER, R., AND SOUTCHANSKI, M. 1998. Execution monitoring of high-level robot programs. In *Principles of Knowledge Representation and Reasoning*. (Cited on page 4.)

DE JONGE, F., ROOS, N., AND WITTEVEEN, C. 2009. Primary and secondary diagnosis of multi-agent plan execution. *Autonomous Agents and Multi-Agent Systems 18,* 2, 267–294. (Cited on page 76.)

DE KLEER, J., MACKWORTH, A. K., AND REITER, R. 1992. Characterizing diagnoses and systems. *Artificial Intelligence 56,* 2, 197–222. (Cited on page 77.)

DE KLEER, J. AND WILLIAMS, B. C. 1987. Diagnosing multiple faults. *Artificial Intelligence 32,* 1, 97–130. (Cited on page 43.)

DE LEONI, M., MECELLA, M., AND DE GIACOMO, G. 2007. Highly Dynamic Adaptation in Process Management Systems Through Execution Monitoring. In *Business Process Management*, G. Alonso, P. Dadam, and M. Rosemann, Eds. Lecture Notes in Computer Science, vol. 4714. Springer Berlin / Heidelberg, Berlin, Heidelberg, Chapter 14, 182–197. (Cited on page 4.)

DEARDEN, R. AND BOUTILIER, C. 1994. Integrating planning and execution in stochastic domains. In *Conference on Uncertainty in Artificial Intelligence*. 162–169. (Cited on page 76.)

DEMOLOMBE, R. AND POZOS PARRA, P. 2005. Belief change in the situation calculus: a new proposal without plausibility levels. In *In Proceedings Workshop of the 17th European Summer School in Logic Language and Information about Belief Revision and Dynamic Logic*. Edinburgh, Scotland. (Cited on pages 39 and 40.)

EDELKAMP, S. AND HELMERT, M. 2000. Exhibiting knowledge in planning problems to minimize state encoding length. In *Recent Advances in AI Planning*. Springer, 135–147. (Cited on page 78.)

EDELKAMP, S. AND HOFFMANN, J. 2004. Pddl2.2: The language for the classical part of the 4th international planning competition. *4th International Planning Competition (IPCâĂŹ04), at ICAP-SâĂŹ04*. (Cited on page 77.)

EVOLUTION. 2015. Working with er-1 (evolution robot). `https://www.cs.drexel.edu/~as397/tutorial.htm`. Accessed: 2015-03-23. (Cited on page 22.)

FASCETTI, E. AND SCAIA, R. 1998. *Soccer Attacking Schemes and Training Execrcises*. Reedswain Publishing. (Cited on pages xv and 123.)

FERREIN, A. 2008. Robot Controllers for Highly Dynamic Environments with Real-time Constraints. Ph.D. thesis, Knowledge-based Systems Group, RWTH Aachen University, Aachen Germany. (Cited on page 42.)

FERREIN, A., FRITZ, C., AND LAKEMEYER, G. 2004. On-line decision-theoretic golog for unpredictable domains. In *Proc. KI-2004*. LNCS. Springer, 322–336. (Cited on page 76.)

FICHTNER, M., GROSSMANN, A., AND THIELSCHER, M. 2003. Intelligent execution monitoring in dynamic environments. *Fundamenta Informaticae 57,* 2–4, 371–392. (Cited on page 76.)

FIKES, R. E., HART, P. E., AND NILSSON, N. J. 1972. Learning and executing generalized robot plans. *Artificial Intelligence 3,* 0, 251 – 288. (Cited on pages 55 and 76.)

FOOTE, T. 2013. tf: The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. 1–6. (Cited on page 67.)

FOX, D., BURGARD, W., AND THRUN, S. 1997. The dynamic window approach to collision avoidance. *Robotics Automation Magazine, IEEE 4,* 1 (Mar), 23–33. (Cited on page 71.)

FOX, M. AND LONG, D. 2003. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research 20*, 61–124. (Cited on page 77.)

FRITZ, C. 2009. Monitoring the Generation and Execution of Optimal Plans. Ph.D. thesis, University of Toronto. (Cited on pages xiii, 5, 6, and 75.)

GENTLE, J. E. 2007. *Matrix Algebra*. Texts in Statistics. Springer. (Cited on page 65.)

GERKEY, B. P. AND KONOLIGE, K. 2008. Planning and control in unstructured terrain. In *In Workshop on Path Planning on Costmaps, Proceedings of the IEEE International Conference on Robotics and Automation (ICRA*. (Cited on page 71.)

GÖDEL, K. 1931. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme. *Monatshefte für Mathematik und Physik 38,* 1, 173–198. (Cited on page 59.)

GOEL, P., DEDEOGLU, G., ROUMELIOTIS, S. I., AND SUKHATME, G. S. 2000. Fault detection and identification in a mobile robot using multiple model estimation and neural network. In *IEEE International Conference on Robotics and Automation*. (Cited on page 24.)

GORDON, N., SALMOND, D., AND SMITH, A. 1993. Novel approach to nonlinear/non-gaussian bayesian state estimation. *IEEE Proceedings F, Radar and Signal Processing 140,* 2, 107–113. (Cited on page 64.)

GRASTIEN, A., ANBULAGAN, A., RINTANEN, J., AND KELAREVA, E. 2007. Diagnosis of discrete-event systems using satisfiability algorithms. In *Proceedings of the 22nd national conference on Artificial intelligence - Volume 1 (AAAI07)*. AAAI Press, Palo Alto, California, 305–310. (Cited on page 35.)

GROSSKREUTZ, H. AND LAKEMEYER, G. 2001. Belief update in the pGOLOG framework. In *KI 2001: Advances in Artificial Intelligence*, F. Baader, G. Brewka, and T. Eiter, Eds. Lecture Notes in Computer Science, vol. 2174. Springer Berlin Heidelberg, 213–228. (Cited on page 36.)

GSPANDL, S., HECHENBLAICKNER, A., REIP, M., STEINBAUER, G., WOLFRAM, M., AND ZEHENTNER, C. 2012. The ontology lifecycle in robocup: Population from text and execution. In *RoboCup 2011: Robot Soccer World Cup XV*, T. Röfer, N. Mayer, J. Savage, and U. Saranli, Eds. Lecture Notes in Computer Science, vol. 7416. Springer Berlin Heidelberg, 389–401. (Cited on pages xv, 9, 119, 122, and 124.)

GSPANDL, S., MONICHI, D., REIP, M., STEINBAUER, G., WOLFRAM, M., AND ZEHENTER, C. 2007. "KickOffTUG — Team Description Paper 2007.". In *International RoboCup Symposium.* Atlanta, USA. (Cited on pages 9 and 119.)

GSPANDL, S., PILL, I., REIP, M., AND STEINBAUER, G. 2011. "Belief Management for Autonomous Robots using History-Based Diagnosis". In *International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)*. Syracuse, NY, USA. (Cited on pages 8, 33, 41, and 101.)

GSPANDL, S., PILL, I., REIP, M., AND STEINBAUER, G. 2013. "Maintaining consistency in a robot's knowledge-base via diagnostic reasoning". *AI communications 26,* 1, 29–38. (Cited on pages 8, 47, and 101.)

GSPANDL, S., PILL, I., REIP, M., STEINBAUER, G., AND FERREIN, A. 2011. Belief Management for High-Level Robot Programs. In *Proc. IJCAI-11*. 900–905. (Cited on pages 8, 33, 86, and 101.)

GSPANDL, S., PODESSER, S., REIP, M., STEINBAUER, G., AND WOLFRAM, M. 2012. A dependable perception-decision-execution cycle for autonomous robots. In *ICRA*. IEEE, 2992–2998. (Cited on pages xiii, xvii, 5, 8, 55, 91, 101, and 115.)

GSPANDL, S., REIP, M., STEINBAUER, G., AND WOTAWA, F. 2010. From sketch to plan. In *In 24th International Workshop on Qualitative Reasoning (QR-2010)*. Portland, Oregon, USA. (Cited on pages xv, 9, 119, 121, and 122.)

GUHA, R. V. AND LENAT, D. B. 1990. Cyc: A midterm report. *AI Magazine 11,* 3, 32–59. (Cited on page 119.)

GUTMANN, J.-S. 2000. Robuste Navigation autonomer mobiler Systeme. Ph.D. thesis, Research Group Foundations of Artificial Intelligence, University of Freiburg, Germany. (Cited on pages xiii and 63.)

HAAS, A. R. 1987. The case for domain-specific frame axioms. In *The frame problem in artificial intelligence. Proceedings of the 1987 workshop*, F. Brown, Ed. Morgan Kaufmann Publishers, San Francisco, 343–348. (Cited on page 13.)

HARNAD, S. 1990. The symbol grounding problem. *Phys. D 42,* 1-3 (June), 335–346. (Cited on page 58.)

HELMERT, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research 26,* 1 (July), 191–246. (Cited on pages xiv, 77, 78, and 127.)

HOFFMANN, J. AND NEBEL, B. 2001. The ff planning system: Fast plan generation through heuristic search. *J. Artif. Int. Res. 14,* 1 (May), 253–302. (Cited on pages xiv and 77.)

HORRIDGE, M., PARSIA, B., AND SATTLER, U. 2009. Explaining inconsistencies in OWL ontologies. In *Scalable Uncertainty Management*, L. Godo and A. Pugliese, Eds. Lecture Notes in Computer Science, vol. 5785. Springer Berlin / Heidelberg, Berlin, Heidelberg, Chapter 11, 124–137. (Cited on page 33.)

IAGNEMMA, K. AND BUEHLER, M. 2006. Editorial Special Issue on the DARPA Grand Challenge. *J. Field Robot. 23,* 9, 655–656. (Cited on page 1.)

IFR STATISTICAL DEPARTMENT. 2012. World robotics industrial robots 2012. (Cited on pages xiii and 2.)

IROBOT. 2015. irobot. `http://www.irobot.com/`. Accessed: 2015-03-23. (Cited on page 22.)

IWAN, G. 2002. History-based diagnosis templates in the framework of the situation calculus. *AI Communications 15,* 1, 31–45. (Cited on pages 25 and 126.)

JEFFERSON, G. 1949. The mind of mechanical man. *British Medical Journal 1,* 25 (June), 1105–1110. (Cited on page 59.)

JURAFSKY, D. AND MARTIN, J. H. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st ed. Prentice Hall PTR, Upper Saddle River, NJ, USA. (Cited on pages 86 and 122.)

KAELBLING, L. P. 1988. Goals as Parallel Program Specifications. In *In Proceedings AAAI-88*. American Association for Artificial Intelligence., 541–550. (Cited on page 55.)

KAELBLING, L. P. AND ROSENSCHEIN, S. J. 1990. Action and planning in embedded agents. *Robotics and Autonomous Systems 6,* 1âĂŞ2, 35 – 48. <ce:title>Designing Autonomous Agents</ce:title>. (Cited on page 55.)

KALMAN, R. E. 1960. A new approach to linear filtering and prediction problems. *Transactions of the ASME–Journal of Basic Engineering 82,* Series D, 35–45. (Cited on page 64.)

KUHN, L., PRICE, B., DE KLEER, J., DO, M., AND ZHOU, R. 2008. Pervasive diagnosis: the integration of active diagnosis into production plans. In *Int. Workshop on Principles of Diagnosis*. (Cited on page 127.)

LAMPERTI, G. AND ZANELLA, M. 2003. *Diagnosis of Active Systems*. Kluwer Academic Publishers, Dordrecht, The Netherlands. (Cited on page 33.)

LEGO. 2015. Lego mindstorms. `http://mindstorms.lego.com/`. Accessed: 2015-03-23. (Cited on page 22.)

LEVENSHTEIN, V. I. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10,* 8 (Feb.), 707–710. (Cited on page 123.)

LEVESQUE, H. J. AND PAGNUCCO, M. 2000. Legolog: Inexpensive experiments in cognitive robotics. In *In Proceedings of the International Cognitive Robotics Workshop(COGROBO*. 104–109. (Cited on page 22.)

LEVESQUE, H. J., REITER, R., LESPÉRANCE, Y., LIN, F., AND SCHERL, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming 31,* 1-3, 59 – 83. (Cited on page 19.)

LIN, F. 2008. *Situation Calculus*. Elsevier, Oxford, Chapter 16, 649–669. (Cited on page 12.)

LIU, H. AND COGHILL, G. M. 2004. Qualitative modeling of kinematic robots. In *Proc. QR-04*. (Cited on page 77.)

LUSSIER, B., CHATILA, R., GUIOCHET, J., INGRAND, F., LAMPE, A., OLIVIER KILLIJIAN, M., AND POWELL, D. 2005. Fault tolerance in autonomous systems: How and how much? In *IN PROCEEDINGS OF THE 4TH IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments*. Nagoya, Japan, 16–18. (Cited on page 3.)

MARDER-EPPSTEIN, E., BERGER, E., FOOTE, T., GERKEY, B. P., AND KONOLIGE, K. 2010. The Office Marathon: Robust Navigation in an Indoor Office Environment. In *International Conference on Robotics and Automation (ICRA)*. Anchorage, AK, USA. (Cited on page 60.)

MAYBECK, P. 1990. The kalman filter: An introduction to concepts. In *Autonomous Robot Vehicles*, I. Cox and G. Wilfong, Eds. Springer New York, 194–204. (Cited on page 64.)

MCCARTHY, J. 1963. Situations, Actions and Causal Laws. Tech. rep., Stanford University. (Cited on page 11.)

MCCARTHY, J., MINSKY, M. L., ROCHESTER, N., AND SHANNON, C. 1955. A PROPOSAL FOR THE DARTMOUTH SUMMER RESEARCH PROJECT ON ARTIFICIAL INTELLIGENCE. (Cited on page 59.)

MCILRAITH, S. A. 1999. Explanatory diagnosis: Conjecturing actions to explain observations. In *Logical Foundations for Cognitive Agents: Papers in Honour of Ray Reiter*. Artificial Intelligence. Springer, Berlin, Heidelberg, 155–172. (Cited on pages 25, 33, and 34.)

MICALIZIO, R. AND TORASSO, P. 2008. Monitoring the execution of a multi-agent plan: Dealing with partial observability. In *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 408–412. (Cited on page 76.)

MICROSOFT. 2010. Kinect for Windows. http://www.microsoft.com/en-us/kinectforwindows/. [Online; accessed 25-January-2015]. (Cited on page 107.)

MIES, C., FERREIN, A., AND LAKEMEYER, G. 2008. Repairing decision-theoretic policies using goal-oriented planning. In *KI 2008: Advances in Artificial Intelligence, 31st Annual German Conference on AI, KI 2008, Kaiserslautern, Germany, September 23-26, 2008. Proceedings*. Lecture Notes in Computer Science, vol. 5243. Springer, 267–275. (Cited on page 75.)

MILLER, G. A. 1995. Wordnet: A lexical database for english. *Commun. ACM 38,* 11 (Nov.), 39–41. (Cited on page 123.)

MOBILEROBOTS, A. 1999. Pioneer P3-DX. http://www.mobilerobots.com/ResearchRobots/PioneerP3DX.aspx. [Online; accessed 25-January-2015]. (Cited on page 107.)

MÜHLBACHER, C. AND STEINBAUER, G. 2014a. Active diagnosis for agents with belief management. (Cited on page 127.)

MÜHLBACHER, C. AND STEINBAUER, G. 2014b. Using common sense invariants in belief management for autonomous agents. In *Modern Advances in Applied Intelligence*, M. Ali, J.-S. Pan, S.-M. Chen, and M.-F. Horng, Eds. Lecture Notes in Computer Science, vol. 8481. Springer International Publishing, 49–59. (Cited on page 119.)

MULLER, J. 2013. Silicon valley vs. detroit: The battle for the car of the future. (Cited on page 1.)

MURPHY, R. R. AND HERSHBERGER, D. 1999. Handling Sensing Failures in Autonomous Mobile Robots. *The International Journal of Robotics Research 18,* 4, 382–400. (Cited on page 76.)

MUSCETTOLA, N., NAYAK, P. P., PELL, B., AND WILLIAMS, B. C. 1998. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence 103,* 1-2, 5 – 47. (Cited on page 76.)

NILSSON, N. J. 1984. Shakey the robot. Tech. Rep. 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025. Apr. (Cited on page 55.)

NILSSON, N. J. 1994. Teleo-reactive Programs for Agent Control. *J. Artif. Int. Res. 1,* 1 (Jan.), 139–158. (Cited on pages xiii, 55, 56, 57, and 58.)

PEDNAULT, E. P. 1989. ADL: Exploring the middle ground between strips and the situation calculus. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*. Morgan Kaufmann Publishers, Inc, 324–332. (Cited on page 12.)

PENCOLÉ, Y. AND CORDIER, M.-O. 2005. A formal framework for the decentralised diagnosis of large scale discrete event systems and its application to telecommunication networks. *Artificial Intelligence Journal 164,* 1-2, 121–170. (Cited on page 33.)

PODESSER, S., STEINBAUER, G., AND WOTAWA, F. 2012. Selective belief management for high-level robot programs. In *Proceedings of the International Workshop on Principles of Diagnosis (DX-12)*. (Cited on page 127.)

POGUE, D. 2011. New iphone conceals sheer magic. (Cited on page 59.)

QUIGLEY, M., CONLEY, K., GERKEY, B. P., FAUST, J., FOOTE, T., LEIBS, J., WHEELER, R., AND NG, A. Y. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*. (Cited on pages 6, 60, and 66.)

QUINLAN, S. AND KHATIB, O. 1993. Elastic bands: connecting path planning and control. In *Robotics and Automation, 1993. Proceedings., 1993 IEEE International Conference on*. 802–807 vol.2. (Cited on page 70.)

REIP, M., STEINBAUER, G., AND FERREIN, A. 2012. Improving belief management for high-level robot programs by using diagnosis templates. In *Proceedings of the International Workshop on Principles of Diagnosis (DX-12)*. (Cited on pages xvii, 8, 75, 79, 81, 85, 101, 113, and 114.)

REITER, R. 1987. A theory of diagnosis from first principles. *Artificial Intelligence 32,* 1 (Apr.), 57–95. (Cited on pages 25 and 76.)

REITER, R. 1991. Artificial intelligence and mathematical theory of computation. Academic Press Professional, Inc., San Diego, CA, USA, Chapter The frame problem in situation the calculus: a simple solution (sometimes) and a completeness result for goal regression, 359–380. (Cited on page 14.)

REITER, R. 2001. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, Cambridge, Massachusetts. (Cited on pages 11, 12, 14, 16, 18, 22, 48, and 127.)

RÖGER, G., HELMERT, M., AND NEBEL, B. 2008. On the relative expressiveness of ADL and golog: The last piece in the puzzle. In *In Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, G. Brewka and J. Lang, Eds. AAAI PRESS, Palo Alto, California, 544–550. (Cited on page 35.)

ROOS, N. AND WITTEVEEN, C. 2009. Models and methods for plan diagnosis. *Autonomous Agents and Multi-Agent Systems 19,* 1, 30–52. (Cited on page 76.)

RUSSEL, S. AND NORVIG, P. 2003. *Artificial Intelligence: A Modern Approach*, Second Edition ed. Prentice Hall International. (Cited on pages 55, 58, 59, and 78.)

SARDIÑA, S. 2000. Indigolog: Execution of guarded action theories. M.S. thesis, Toronto, Ont., Canada, Canada. (Cited on pages 47, 48, and 49.)

SARDIÑA, S. 2005. Deliberation in agent programming languages. Ph.D. thesis, University of Toronto, Toronto, Ont., Canada, Canada. AAINR07609. (Cited on page 19.)

SARDIÑA, S., DE GIACOMO, G., LESPÉRANCE, Y., AND LEVESQUE, H. J. 2004. On the semantics of deliberation in indigolog - from theory to implementation. *Annals of Mathematics and Artificial Intelligence 41,* 2-4 (Aug.), 259–299. (Cited on page 24.)

SARDIÑA, S. AND VASSOS, S. 2005. The wumpus world in IndiGolog: A preliminary report. In *Proceedings the Nonmonotonic Reasoning, Action and Change Workshop at IJCAI (NRAC-05)*. Professional Book Center, Edinburgh, Scotland, 90–95. (Cited on page 24.)

SCHERL, R. B. AND LEVESQUE, H. J. 2003. Knowledge, action, and the frame problem. *Artificial Intelligence 144,* 1âĂŞ2 (Mar.), 1 – 39. (Cited on pages xiii, 35, 37, and 38.)

SCHUBERT, L. K. 1990. Monotonic solution of the frame problem in the situation calculus: An efficient method for worlds with fully specified actions. In *Knowledge Representation and Defeasible Reasoning*, H. E. Kyburg, R. P. Loui, and G. N. Carlson, Eds. Vol. Volume 5. Kluwer Academic Publishers, Dordrecht / Boston / London, 23–67. (Cited on page 13.)

SHAPIRO, S. AND PAGNUCCO, M. 2004. Iterated belief change and exogenous actions in the situation calculus. In 16th European Conference on Artificial Intelligence (ECAI-04), R. L. de Mántaras and L. Saitta, Eds. *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, 878–882. (Cited on page 39.)

SHAPIRO, S., PAGNUCCO, M., LESPÉRANCE, Y., AND LEVESQUE, H. J. 2000. Iterated belief change in the situation calculus. In Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR-2000), A. G. Cohn, F. Giunchiglia, and B. Selman, Eds. *Artificial Intelligence*, 527–538. (Cited on pages xiii, 37, 38, 39, and 40.)

SOHRABI, S., BAIER, J. A., AND MCILRAITH, S. A. 2010. Diagnosis as planning revisited. In *Proceedings of the 12th International Conference on the Principles of Knowledge Representation and Reasoning (KR-10)*. AAAI Press, Toronto, Canada, 26–36. An abridged version of this paper appears in International Workshop on Principles of Diagnosis (DX-10). (Cited on page 34.)

STEINBAUER, G., WEBER, J., AND WOTAWA, F. 2005. "From the real-world to its qualitative representation - Practical lessons learned.". In *18th International Workshop on Qualitative Reasoning*. Graz, 186–191. (Cited on page 72.)

SWI-PROLOG. 2015. SWI-Prolog Robust,Mature,Free. Prolog for the real world. `http://www.swi-prolog.org/`. Accessed: 2015-03-23. (Cited on page 22.)

TECHNICAL COMMITTEE ISO. 2011. ISO 26262 Compliance Achieving Functional Safety for Automotive E/E Systems. (Cited on page 3.)

TENENBAUM, G. 1995. *Introduction to analytic and probabilistic number theory*. Vol. 46. Cambridge university press. (Cited on page 106.)

TESAURO, G. 1995. Temporal difference learning and td-gammon. *Commun. ACM 38,* 3 (Mar.), 58–68. (Cited on page 59.)

THRUN, S. 2010. What we're driving at. (Cited on page 1.)

THRUN, S., BENNEWITZ, M., BURGARD, W., CREMERS, A. B., DELLAERT, F., FOX, D., HÄHNEL, D., ROSENBERG, C., ROY, N., SCHULTE, J., AND SCHULZ, D. 1999. Minerva: A second-generation museum tour-guide robot. In *IEEE International Conference on Robotics and Automation*. (Cited on page 1.)

THRUN, S., BURGARD, W., AND FOX, D. 2001. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. Intelligent robotics and autonomous agents. The MIT Press. (Cited on pages xiii and 65.)

THRUN, S., FOX, D., BURGARD, W., AND DELLAERT, F. 2000. Robust Monte Carlo Localization for Mobile Robots. *Artificial Intelligence 128,* 1-2, 99–141. (Cited on page 64.)

TOUTANOVA, K. AND MANNING, C. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *In Proceedings of the EMNLP/VLC-2000*. Hong Kong, 63–70. (Cited on page 122.)

TREBI-OLLENNU, A. 2006. Special Issue on Robots on the Red Planet. *IEEE Robotics & Automation Magazine 13,* 2. (Cited on page 2.)

TURING, A. M. 1950. Computing machinery and intelligence. One of the most influential papers in the history of the cognitive sciences: http://cogsci.umn.edu/millennium/final.html. (Cited on page 58.)

VERMA, V., GORDON, G., SIMMONS, R., AND THRUN, S. 2004. Real-time fault diagnosis. *IEEE Robotics & Automation Magazine 11,* 2, 56 – 66. (Cited on page 24.)

VOLPE, R., NESNAS, I., ESTLIN, T., MUTZ, D., PETRAS, R., AND DAS, H. 2001. The CLARAty architecture for robotic autonomy. (Cited on page 6.)

WAIBEL, M., BEETZ, M., CIVERA, J., D'ANDREA, R., ELFRING, J., GALVEZ-LOPEZ, D., HAUSSERMANN, K., JANSSEN, R., MONTIEL, J., PERZYLO, A., SCHIESSLE, B., TENORTH, M., ZWEIGLE, O., AND VAN DE MOLENGRAFT, R. 2011. Roboearth. *Robotics Automation Magazine, IEEE 18,* 2, 69–82. (Cited on page 119.)

WALL, M. 2012. Touchdown! huge nasa rover lands on mars. (Cited on page 2.)

WELD, D. S., ANDERSON, C. R., AND SMITH, D. E. 1998. Extending graphplan to handle uncertainty and sensing actions. In *fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*. 897–904. (Cited on page 76.)

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96. (Cited on page 22.)

WILLIAMS, B. C. AND RAGNO, R. J. 2007. Conflict-directed a* and its role in model-based embedded systems. *Discrete Appl. Math. 155,* 12 (June), 1562–1595. (Cited on page 77.)

WOLFRAM, M. 2011. An integral mobile robot platform for research and experiments in the field of intelligent autonomous systems. (Cited on pages xiv, xv, 66, 67, 68, 69, 73, and 111.)

WOLFRAM, M., GSPANDL, S., REIP, M., AND STEINBAUER, G. 2011. "Robust Robotics Using History-Based-Diagnosis in IndiGolog". In *Austrian Robotics Workshop*. Hall/Tirol, Austria. (Cited on pages 8 and 101.)

WURMAN, P. R., D'ANDREA, R., AND MOUNTZ, M. 2007. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. In *Proceedings of the 19th national conference on Innovative applications of artificial intelligence - Volume 2*. IAAI'07. AAAI Press, 1752–1759. (Cited on page 1.)

WYLAND, B. 2008. Warehouse automation: How to implement tomorrow's order fulfillment system today. (Cited on page 1.)