# Proceedings of the Fourth International Cognitive Robotics Workshop

Valencia, Spain
August 23-24, 2004

Editors:

Patrick Doherty, Linköping University, Sweden (Chair)
Gerhard Lakemeyer, Aachen University of Technology, Germany (Co-Chair)
Angel P. del Pobil, Universidad Jaume-I, Spain (Co-Chair)

*

# Preface

This is the fourth in a series of workshops dedicated to the topic of cognitive robotics. The workshop series began in 1998 as part of the AAAI Fall symposium series, continued in Berlin in 2000 where it was co-located with ECAI 2000, and crossed the ocean again in 2002 where it took place in Edmonton, Canada and was co-located with AAAI 2002. This year, we are back in Europe again and the workshop is co-located with ECAI 2004.

Cognitive robotics is concerned with endowing robots and software agents with higher level cognitive functions that enable them to reason, act and perceive in changing, incompletely known, and unpredictable environments. Such robots must, for example, be able to reason about goals, actions, when to perceive and what to look for, the cognitive states of other agents, time, collaborative task execution, etc. In short, cognitive robotics is concerned with integrating reasoning, perception and action with a uniform theoretical and implementation framework.

Traditionally, research in cognitive robotics has emphasized logical formalism as the main representational and reasoning tool and previous workshops have focused specifically on these approaches. We are now seeing more attempts at integrating results from the area with physical robotic systems. Due to this trend, we feel that it is beneficial to the workshop participants and the community at large to widen the scope of the workshop somewhat, not only in the call for papers but also in inviting participants from the robotics and cognitive science communities to serve on the program committee and participate in this years workshop. We hope this is a trend that will continue in future workshops.

36 papers were submitted and 15 were accepted for presentation. We would like to take this occasion to thank both the authors who have submitted papers and also the program committee for their help with ideas and with reviewing.

Patrick Doherty, Linköping University, Sweden (Chair)
Gerhard Lakemeyer, Aachen University of Technology, Germany (Co-Chair)
Angel P. del Pobil, Universidad Jaume-I, Spain (Co-Chair)

*

## Program Committee

Chitta Baral, Arizona State University, USA
Michael Beetz, Munich University of Technology, Germany
Henrik Christensen, Royal Inst. of Technology, Sweden
Patrick Doherty, Linkping University, Sweden
Malik Ghallab, LAAS-CNRS, Toulouse, France
Joachim Hertzberg, Fraunhofer Institute, AIS, Germany
Gerhard Lakemeyer, RWTH Aachen, Germany
Sheila McIlraith, University of Toronto, Canada
John-Jules Meyer, Utrecht University, Netherlands
Nicola Muscettola, NASA Ames Research Center, USA
Bernard Nebel, Albert-Ludwigs-Universitaet Freiburg, Germany
Maurice Pagnucco, University of New South Wales, Australia
Fiora Pirri, Universita di Roma "La Sapienza", Italy
Angel P. del Pobil, Jaume I University, Spain
Alessandro Saffiotti, rebro University, Sweden
Erik Sandewall, Linkping University, Sweden
Alan C. Schultz, Naval Research Laboratory, Wash. DC, USA
Murray Shanahan, Imperial College, UK
Michael Thielscher, Dresden University of Technology, Germany
Brian Williams, CSAIL, MIT, USA

*

## Table of Contents

*

# Paper Session I

August 23, 9:00 - 10:30

- **Specifying Failure and Progress Conditions in a Behavior-Based Robot Programming System**, F. Kabanza, K. B. Lamine
- **Robel: Synthesizing and Controlling Complex Robust Robot Behaviors**, B. Morisset, G. Infante, M. Ghallab, F. Ingrand
- **Patterns in Reactive Programs**, P. Haslum

*

# Specifying Failure and Progress Conditions in a Behavior-Based Robot Programming System

**(Extended Abstract)**

Froduald Kabanza and Khaled Ben Lamine
University of Sherbrooke
Sherbrooke, Quebec, J1K 2R1 Canada
{Kabanza, Khaled.Ben.Lamine}@usherbrooke.ca

**Abstract.** Behavior-based robot programs are designed by splitting the overall robot decision making process among concurrent processes, each of them being in charge of a well defined simple goal-oriented behavior. However, the combination of simple behaviors into more complex ones often incur global interactions that cannot be debugged by just considering individual behaviors. The normal way of dealing with global interactions that may cause failure in the robot execution is to program additional heuristic processes that monitor behaviors to check that they progress normally towards their intended goals. In this paper, we explain how Linear Temporal Logic (LTL) can be used as a declarative language for specifying an interesting class of such monitoring processes. This simplifies the task of writing robot control programs, increases the design modularity by clearly separating the control components from the monitoring ones, and augments the reliability of control programs thanks to a precise and clear LTL semantics.

## 1 INTRODUCTION

Behavior-based robot programming systems follow the principle that the overall robot decision making process should be split among several concurrent processes, each of them being in charge of a well defined simple goal-oriented behavior; it is then the combination of theses processes that achieves more complex task-oriented behaviors [1, 7, 10, 15]. An interesting feature that explains in part the increasing popularity of this approach is undoubtedly the modularity of robot programs that are designed along this principle; one designs a program that controls one behavior, without having to get into details of interacting behaviors that control the other robot aspects. For instance, one can program a *goto* behavior that combines with *obstacle-avoidance* behaviors to make a robot go to a desired destination, yet without getting involved into the code of obstacle-avoidance processes.

The split of behaviors among many concurrent processes introduces, however, all the known hurdles of concurrent processes. Individual behaviors that are tested or proven to work correctly by assuming local conditions may no longer work perfectly when combined with other behaviors. In fact, most complex tasks require heuristic processes that monitor the robot behaviors to check whether they progress normally towards their intended goals, and trigger failure-avoidance or failure-recovery behaviors whenever necessary [3, 11, 13, 16]. However, programming such robot monitoring behaviors still remains a difficult task, because of the complexity of process interactions, which is often exacerbated by the intrinsic imprecision in robot sensors, actuators and positioning devices or the uncertainty in the robot environment.

In order to facilitate the programming of robot monitoring processes, we are trying to develop a general framework for specifying declarative robot monitoring conditions using Linear Temporal Logic (LTL) [12]. We use LTL to specify conditions that a normal, satisfactory robot execution should satisfy; then we monitor those statements in the background of the robot execution to notify of their violation. Processes waiting on such notifications can then be activated to avoid anticipated failures or to recover from them. LTL is expressive enough to handle an interesting subset of monitoring conditions, such as waiting until a particular sequence of state conditions has been observed.

With this approach, the implementation of a monitoring process will not require elaborate knowledge of the internal program structure of the processes being monitored. This approach also seems quite flexible, allowing the specification of both light-weight synchronous monitoring processes as well as more complex asynchronous ones. It is quite simply implemented, without any explicit storage of the robot history; instead, the relevant state features of the history are automatically conveyed by the update of monitoring conditions which are added to the robot internal state. These features make our approach fit naturally with the behavior-based architecture that we use, thus requiring little learning effort from robot software programmers to use our approach for coding logic-based monitoring processes.

The remainder of the paper is organized as follows. In the next section we discuss SAPHIRA [10, 15], the robot programming system used to implement our approach. Then, we discuss the use of LTL to express robot monitoring conditions; we discuss the basic algorithms that are used to handle those formulas in robot monitoring activities, as well as the integration of these algorithms into the SAPHIRA architecture. We conclude with a discussion on most related work and on future directions of this research.

## 2 SAPHIRA ARCHITECTURE

SAPHIRA is a programming system for mobile robots, developed by Konolige and his team at SRI [10, 15]. It includes libraries for controlling a mobile robot at different levels of complexity, going from low-level (e.g., moving a given distance, turning the wheels a given angle, or acquiring raw sensor data) to more complex navigation behaviors (e.g., obstacle avoidance, map registration, path planning) and tasks (e.g., sequencing behaviors using arbitrary C++ programs). Figure 1 shows an abstract view of the SAPHIRA

architecture. SAPHIRA processes can be synchronous or asynchronous. They all have access to the robot state, which consists of the state reflector (position encoder readings, wheel orientation, sonar readings), and the environment representations (map information, sonar interpretation data). State information is updated automatically from robot sensors and from SAPHIRA synchronous or asynchronous processes.

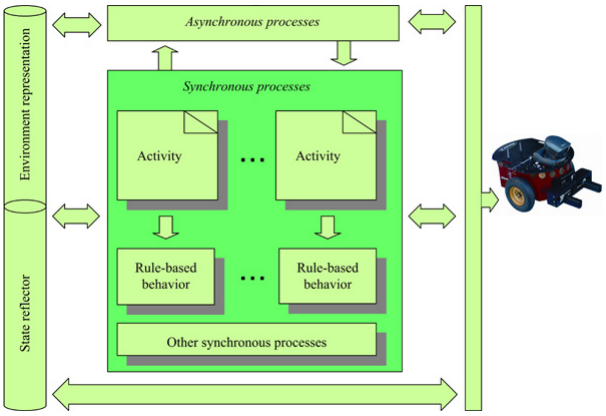



**Figure 1.** SAPHIRA architecture

Synchronous processes are normally used to implement lower-level behaviors that require immediate action on the part of the robot, such as obstacle avoidance or object tracking; they are managed by a SAPHIRA specific scheduler. Asynchronous processes on the other hand are normally used to implement higher-level behaviors (e.g., a task planner deciding on the order objects are picked and delivered by a robot in an office delivery application); they are managed by the host computer operating system (e.g., Linux or Windows).

The synchronous scheduler iterates through each synchronous process every 100 milliseconds. At each cycle, it runs each process, one at time, from the execution point where it suspended it last time to the next suspension point; a priority mechanism is used to resolve conflicts among processes that simultaneously affect the same robot control parameters.

The rationale behind the 100 milliseconds cycle becomes obvious by considering the obstacle-avoidance behavior; if we code an avoidance strategy consisting in turning left each time the sensor readings indicate an obstacle on the right, we would like the time between the detection of the obstacle (one block of instruction executed during one cycle) and the turning of the robot wheel and change of its speed (another block of instruction executed in the next cycle) to last at most one 100 milliseconds, so as the robot's reaction is fast enough before it hits the obstacle. Hence, when writing synchronous processes, one must ensure that the code between two suspension points is fast enough to be executed within the 100 milliseconds cycle.

Synchronous processes are programmed using either rule-based C++ libraries, a declarative C-like language called Colbert, or finite state machines (FSM) which are actually internal representations of processes written using the rule-based C++ libraries and are rarely convenient to code in directly. Asynchronous processes are normally written in higher-level languages (e.g., C++ or Java). In SAPHIRA terminology, rule-based processes are called behaviors, whereas those written using Colbert are called activities. In this paper, we adopt a formal software engineering terminology [13]; a behavior is a set of possible execution sequences that a given SAPHIRA process may go through when interacting with other processes.

## 3 MONITORING PROCESSES

A robot task is programmed by combining several concurrent SAPHIRA processes. For instance, moving a robot to a goal location can be done by combining one behavior that avoids obstacles and one behavior that attracts the robot towards the goal location; the obstacle avoidance behavior is often further split into avoiding close obstacles and staying away from remote obstacles. Adding corridor following, object-seeking, object-grasping and object-releasing processes makes the robot become an object delivery system. Although such a splitting of behaviors is a purely reactive, heuristic one, in many cases it is sufficient to move the robot fast towards its target, while avoiding obstacles. However, in unusual obstacle configurations, the robot can get trapped, oscillating between the obstacle avoidance and goal-tracking behaviors.

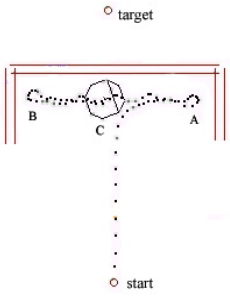



**Figure 2.** Example of navigation failure

Figure 2 illustrates this situation with a U-shape obstacle configuration. The robot is shown as a circle, with a crossing line indicating its heading direction. The dotted curve indicates the robot trajectory so far. Using rule-based C++ goal-reaching and obstacle avoidance processes provided with the SAPHIRA distribution, we reproduced an experiment made by Xu [17]. The robot starts from the indicated point on the figure with the goal of reaching the indicated target position. At the start, the U-shape obstacle is too far away to have an impact on the obstacle avoidance behavior; hence the robot moves in a straight line towards the target only under the effect of the goal-reaching behavior. As it gets closer to the obstacle (bottom of the U-shape), the obstacle avoidance veers the robot to the right to avoid the obstacle (it could also have veered left). The robot continues moving along the bottom of the obstacle, towards the right. On point A, because of the obstacle in front, the robot veers left (being attracted by the goal-reaching behavior), then because of the obstacle still on the left, abruptly veers right, making its heading direction opposite to the target direction. There are no more obstacles in front, and the robot becomes attracted again towards the target, bringing it back to point C. Since it approaches the bottom slightly inclined on its right, it will tend to veer left this time, making a move that is a mirror to the previous one, this time with critical point B playing the role of critical point A. The robot keeps on oscillating this way, between points A, C and B endlessly.

This is kind of situations is not limited to behavior-based robots. It is actually a problem for any navigation approach based on local decisions such potential field methods [11]. If

the obstacle configurations are known, we can use path-planning to escape from such situations. For unpredictable obstacles (e.g., in office delivery environments people can move freely and objects may be displaced without notice), the robot behaviors have to be coupled with monitoring processes to detect such U-shaped obstacle configurations in order to activate recovery strategies.

## 3.1 Program-based monitoring processes

We can detect and escape from U-shape obstacles by using the virtual target approach by Xu [17]. The idea is to monitor the occurrence of the above A and B points in the robot behaviors and then to temporally set a virtual escape target for the robot (see Figure 3).

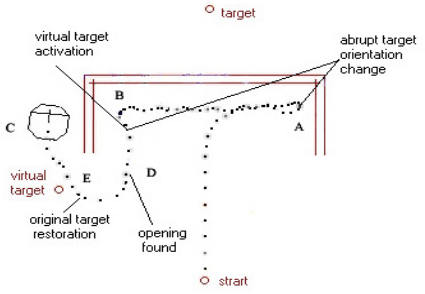



**Figure 3.** Virtual target approach

To experiment with this idea, we wrote a Colbert process (i.e., more precisely a Colbert activity, named *DetectUshape*) which sets a global Boolean variable (*UshapeDetected*) when such points A and B have occurred over a period of time; another Colbert process (*SetVirtualTarget*) waits on *UshapeDetected* becoming true to change temporally the robot target to a virtual target opposite to the current one and to activate another Colbert process (*DetectOpening*), which moves the robot along one side of the U-shape (much like a corridor following behavior), trying to detect an opening; once an opening is detected (point D), this is notified to *SetVirtualTarget***,** which restores the original target. Colbert includes useful primitive for programming the above processes, C-like *while* and *if* flow control instructions, instructions for suspending and resuming processes, instructions for acquiring process states, global variable definitions for inter-process data sharing, and a very useful *waitFor* instruction allowing a process to be suspended until a Boolean condition becomes true.

The above solution still remains simple in many aspects (it's not difficult to trap the robot with a more complex obstacle configuration), nevertheless it illustrates that even very lower-level robot control can deal with logical patterns of behaviors which could be abstracted over using declarative statements.

## 3.2 LTL-based monitoring processes

The process *DetectUshape* monitors a failure condition for robot navigation that can be declaratively expressed as "*eventually, the robot's heading direction keeps on changing abruptly from left to right, or from right to left, immediately followed with the heading opposite to the target*". We want to simplify the program by replacing the process *DetectUshape* by a simple *wait* instruction on such a declaratively stated condition.

This statement specifies something that should not happen in a normal execution; it's a failure condition. We can also express this from a perspective of something that should be maintained true in a normal behavior: "*the robot's heading direction is never continually changing abruptly from left to right, or from right to left, immediately followed with the heading opposite to the target*"; this is a progress condition. Failure conditions and progress conditions are dual, but it's nice to have both of them in a tool for monitoring behaviors, since some execution properties are better captured as failures while others are better expressed as progress conditions.

By integrating declarative failure and progress conditions as Colbert and C++ primitives, we can use them to code succinct monitoring processes, to develop prototypes using them in a development phase and later replace them by Colbert or C/C++ code, or to test the correctness or performance of a robot. For instance, we can simulate a robot control program on randomly generated object delivery requests using a statement like "always when a delivery request is received, it is fulfilled within 20 seconds". We can also express things like "when grasping and delivering object, the robot should wait until an object is visible for five consecutive SAPHIRA cycles, before approaching it;" or "the robot must wait in the corridor until door to room B is opened."

### *3.2.1 Specifying LTL conditions*

The execution of a robot process produces a sequence of robot states. We can thus express failure and progress conditions as declarative statements over sequences of robot states. In Colbert and C/C++ processes, we can already write conditions relevant to just one state using Boolean expressions (in Colbert they have the same syntax as in C/C++). Normal Colbert or C++ Boolean expressions form the basic case for LTL conditions, called *state conditions*. For instance, if x is an integer declared in Colbert, then *(x== 1||x==2)* is an LTL condition. All Boolean variables are also LTL conditions.

LTL conditions are expressed from the perspective of the current robot state. A state condition is true at a current point of execution if the corresponding Boolean expression evaluates to true. *Backward conditions* express properties with respect to the execution history of the current state, whereas *forward conditions* express properties with respect to what will happen from the current state.

A backward condition is a condition involving the logical operators **L** (last), **S** (since), **P** (previous) and **G** (all the time) to refer to the history states. The syntax and intuitive semantics are quite simple. If **c** is a state condition or a backward condition, then **(L c)** is a backward condition, which is true in the current state if **c** is true in the preceding state. If **c** and **d** are state conditions or backward conditions, then **(c S d)** is a backward condition, which is true in the current state if **c** has been true in each previous state since when **d** was true. If **c** is a state condition or a backward condition, then **(P c)** is a backward condition, which is true in the current state if **c** is true in some previous state. If **c** is a state condition or a backward condition, then **(G c)** is a backward condition, which is true in the current state if **c** is true in all previous states. Finally, backward conditions can be combined using the usual Boolean operators **II** (or), **!** (negation) and **&&** (and). The operator **L** is only used in synchronous processes; by preceding state, it then means the state available at the preceding SAPHIRA cycle. For

asynchronous processes, SAPHIRA states are sampled at arbitrary periods.

Conditions can be nested arbitrary, making the resulting semantics a recursive one, with a double basic case on state conditions and on the start state of the execution. This gives us a quite powerful language for expressing behavior properties. For example, we can express the abrupt direction change in the U-shape obstacle failure by using the condition

*(P (targetRealLeft && (L targetRealRight))) ||*
*(P (targetRealRight && (L targetRealLefft))),*

where *targetRealLeft* and *targetRealRight* are Boolean conditions over the robot current position and the target, expressing respectively that, the target is on the rear left of the robot, or on the real right; thus the disjunct *(P (targetRealLeft && (L targetRealRight))* expresses that there is a previous state in which the target was on the rear left and on the rear right in the state just before.

A forward condition is a condition involving the logical operators **N** (Next), **U** (until), **E** (eventually) and **A** (always) to refer to the history states. The syntax and intuitive semantics are also simple. If **c** is a state condition or a forward condition, then **(N c)** is a forward condition, which is true in the current state if **c** is true in the next state. If **c** and **d** are state conditions or forward conditions, then **(c U d)** is a forward condition, which is true in the current state if **c** true in all forward states preceding the first state, if any, where **d** is true. If **c** is a state condition or a forward condition, then **(F c)** is a forward condition, which is true in the current state if **c** is true in some future state. If **c** is a state condition or a forward condition, then **(A c)** is a forward condition, which is true in the current state if **c** is true in all future states. Forward conditions can also be combined using the usual Boolean operators. The operator **N** is only used in synchronous processes; by next state, it then means the state available at the next SAPHIRA cycle.

Forward conditions are like mirror conditions to backward conditions with respect to the current state. In fact, in our case, any property expressed as a backward condition can also be expressed as a forward condition and vice-versa.[1] The previous U-shape example can be expressed forwardly as

*(F (targetRealRight && (N targetRealLeft))) ||*
*(F (targetRealLeft && (N targetRealRight))).*

Even though backward conditions and forward conditions have the same expressive power, it is useful to have them both because some behavior properties are better specified by telling what should happen or not happen for any future execution sequence seen from the start state (i.e., the current state in interpreting the condition), whereas other properties are better expressed by stating what bad sequence of states must have occurred in the past, before concluding in a failure or normal progress in the current state.

[1] This equivalence holds because for **(F c),** c is not required to eventually become true (what is required is that if this ever happens, then we must be able to detect it); similarly, in **(c U d)**, **d** is not required to hold eventually. Without these conditions, forward conditions are slightly more expressive than backward conditions.

### *3.2.2 Progressing conditions*

The above example and the intuitive semantics of this language suggest that we would have to store explicitly the sequence of SAPHIRA states in order to evaluate the truth of LTL conditions. In fact, we need not. Given an LTL formula and a current state, it is possible to tell whether the formula is true in the current state, whether it is false, or whether none of these two situations can be decided yet, by simply computing an update of the condition to be evaluated, state by state. The function that computes such an update condition for backward or forward formula is called a *condition progress function*; it progresses a condition along a sequence of execution on the fly until being able to establish its validity or falsity.

The technique for progressing forward formulas is well-known and has been used in many problems, including robot perception planning [5] and robot monitoring [3]. The update rules are actually quite easily derived from the forward recursive syntax rules and recursive semantics of LTL.

For backward conditions a different technique is needed. We can track the truth of a backward condition over a history, on the fly, forwardly, starting from the initial state of an execution, yet without actually keeping an explicit record of the execution trace. Instead, the necessary information for evaluating the backward condition at a given point of execution will be conveyed by a set of past sub-conditions that are updated at every step of execution.

The idea is first to realize that the truth value of a backward condition is completely determined by the truth value of its sub-conditions. Initially, we determine the sub-conditions that are true in the initial state. For example, **(L c)** is initially false since there is no previous state; a Boolean condition **c** is initially true if it holds in the initial state; **(s S d)** is initially true if **d** is true in the initial state; similarly for **P** and **G** conditions. It takes a constant time to compute the initial set of sub-conditions (exactly one run over the condition).

Next, we pass the set of sub-conditions true in the current state to the next state of execution; we say that we have progressed the set of sub-conditions. Given this, we can determine whether a backward formula is true, false or none of these yet, by evaluating Boolean conditions as usual in the new state and backward sub-conditions using their membership in the progressed set of sub-conditions. This also takes a constant time.

### *3.2.3 Integration into SAPHIRA*

Asynchronous processes are programmed in C/C++ (or other high-level languages that allow dynamically linked libraries). On the other hand SAPHIRA Colbert synchronous processes can invoke arbitrary C/C++ functions and have access to C/C++ structures via dynamically linked libraries. Thus, once one has implemented the LTL progression algorithms for forward conditions and backward conditions, it is not hard to make an interface between them and Colbert or with C/C++. Currently we have only the forward progression functions implemented, and only for Colbert processes.

We programmed in C++ a structure for an LTL condition, indicating the type (backward or forward, although only forward ones are supported at the time being), its mode (failure condition or progress condition) the original condition, the progressed condition (by the LTL progression function, this slot becomes true in state where it is satisfied, false in states where it falsified, an LTL condition otherwise) and other few bookkeeping attributes.

The declaration of an LTL condition initializes the appropriate structure and returns a pointer to it. This pointer can then be used in a *waitLTLCondition.* If **c** is a failure condition, then the instruction *waitLTLCondition(c)* in a Colbert process blocks the process until **c** is progressed to false (which means it is made false in the current state). If **c** is a progress condition, then the instruction *waitLTLCondition(c)* in a Colbert process blocks the process until **c** is progressed to true (which means it is made true in the current state). After the process has passed the wait condition, it can execute user-specified code for handling the condition. Figure 4 illustrates the integration into the SAPHIRA architecture.

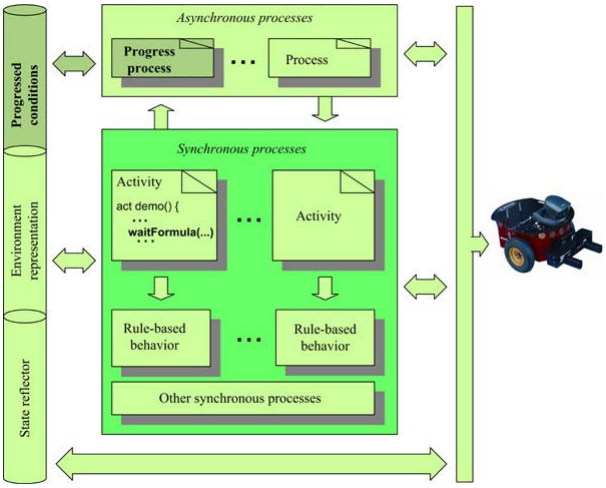



**Figure 4.** SAPHIRA architecture with LTL Progress

## 4 RELATED WORK

Earlier steps of this work were reported in [3,4]. At that time, our system only supported asynchronous monitoring processes, required to record a history of the robot execution (this increases the monitoring complexity), and only handled forward LTL conditions. Even though handling backward conditions does not modify the theoretical expressive power, this allows easier specifications in many cases, particularly when coding robot behaviors based upon past observed states.

Many task-level control languages have been proposed allowing flexible specifications of robot tasks. Among them, the Task Definition Language (TDL) of Simmons and Apfelbaum is a C++ extension [16] with various useful synchronization primitives that facilitate the specification of robot monitoring processes. The language does not support however a declarative specification of behavioral properties such as forward or backward conditions; these have to be encoded directly as programs. Other frameworks that involve formal methods in robot monitoring include the Reactive Plan Language in [2], the probabilistic perception action planner in 54], the extraction of symbolic facts from a history of behaviors' activations [9], and robot monitoring of Golog robot control programs [5]. Both RPL and Golog appear to be more flexible for task-level problems. The work in [11] is much closely related to the framework we propose in that it provides a mechanism for extracting fluents from behavior activation levels; a connection could then be made with our approach by using such fluents as the basis for propositions in monitored progress conditions.

## 5 CONCLUSION

This work is being continued along many avenues, including the following. With synchronous processes, LTL conditions are progressed by Colbert activities. This increases the size of the SAPHIRA stack, consuming precious time of the 100 milliseconds cycle. Since not all conditions need update at every SAPHIRA cycle, we can progress non critical conditions asynchronously, still allowing access of SAPHIRA synchronous processes to the result of this progression.

It also seems feasible to add timing conditions by introducing a global clock variable and having it involved in Boolean conditions that compose progress conditions. For instance, if the clock is initialized to 0 at the start of the robot executions, with units in seconds, we can have a formula like *F=(A(nearTarget || clock < 600))*, where *nearTarget* is a Boolean expression expressing a desired nearness between the robot position and a target position. If this is declared as a failure condition in a process, the instruction *waitLTLCondition(F)* would block the process until 600 seconds have elapsed before the robot is near the target.

## 6 REFERENCESS

[1] R. C. Arkin. *Behavior-Based Robotics*. MIT press, 1998.

[2] M. Beetz. 'Structured reactive controllers: controlling robots that perform everyday activity.' *Agents*, 228-235, 1999.

[3] K. Ben Lamine and F. Kabanza. Reasoning about robot actions: a model-checking approach. In *Advances in Plan-Based Control of Robotic Agents*. LNAI 2466, pages 123-139, 2002.

[4] K. Ben Lamine and F. Kabanza, 'History checking of temporal fuzzy logic formulas for monitoring behavior-based mobile robots'. *Proc. of the 12th IEEE International Conference on Tools with Artificial Intelligence*, 312–319, 2000.

[5] M. Broxvall, L. Karlsson and A. Saffiotti. 'Steps toward detecting and recovering from Perceptual Failures.' *Proc. of the 8th Int. Conf. on Intelligent Autonomous Systems, 2004*.

[6] G. De Giacomo, R. Reiter and M. Soutchanski. 'Execution monitoring of high-Level robot programs.' *Proc. of Principles of Knowledge Representation and Reasoning,* 453-465, 1998.

[7] E. Gat. 'On three-layer architecture.' *Artificial Intelligence and Mobile Robots*, **2**, 1622–1627, 1994.

[8] M. Grabisch. 'Temporal scenario modelling and recognition based on possibilistic logic', *Artificial Intelligence Journal*, **148**(1-2) , 261–289, August 2003.

[9] J. Hertzberg, F. Schönherr, M. Cistelecan and T. Christaller. 'Extracting situation facts from activation value histories in behavior-based robots', *KI-2001:: Advances in Artificial Intelligence, LNAI 2174,* 305–319, 2001.

[10] K. Konolige. Colbert: A language for reactive control in SAPHIRA. In *KI: Advances in Artificial Intelligence*, LNAI, pages 31–52, 1997.

[11] J.C. Latombe. '*Robot Motion Planning*.' Kluwer Academic Press, 1991.

[12] K. Madhava and P. Krishna. 'Perception and remembrance of the environment during real-time navigation of a mobile robot.' *Robotics and Autonomous Systems*, 37(1) :25–51, 2001.

[13] Z. Manna and A. Pnueli. 'The Temporal Logic of Reactive and Concurrent Systems.' Springer-Verlag, 1991.

[14] F.G. Pin and S.R. Bender. 'Adding memory processing behaviors to the fuzzy Behaviorist-based navigation of mobile robots.' In *ISRAM'96 Sixth International Symposium on Robotics and Manufacturing*, 27-30 1996.

[15] E.H. Ruspini, K. Konolige, K. L. Myers and A. Saffiotti. 'The Saphira architecture: A design for autonomy,' *Journal of Experimental and Theoretical Artificial Intelligence*, **9**(1):215–235, 1997.

[16] R. Simmons and D. Apfelbaum. 'A task description language for robot control.' Proc. of Conference on Intelligent Robotics Systems, 1998.

[17] W. L. Xu. A virtual target approach for resolving the limit cycle problem in navigation of a fuzzy behaviour-based mobile robot. *Robotics and Autonomous Systems*, **30**(4) :315–324, 2000.

# Robel : Synthesizing and Controlling Complex Robust Robot Behaviors

**Morisset Benoit**[1] and **Infantes Guillaume** and **Ghallab Malik** and **Ingrand Felix**[2]

**Abstract.** We present the Robel supervision system which is able to learn from experience robust ways to perform high level tasks (such as "navigate to"). Each possible way to perform the task is modeled as an Hierarchical Tasks Network (HTN), called *modality* whose primitives are sensory-motor functions. An HTN planning process synthesizes all the consistent modalities to achieve a task. The relationship between supervision states and the appropriate modality is learned through experience as a Markov Decision Process (MDP) which provides a general policy for the task. This MDP is independent of the environment and characterizes the robot abilities for the task.

## Introduction

Robust robot navigation is a complex task which involves many different capabilities such as localization, terrain modeling, motion generation adapted to obstacles, and so on. Many sensory-motor (*sm*) functions have been developed and are available to perform navigation into structured (e.g. buildings) and unstructured (e.g. outdoor) environments. Since no single method or sensor has a universal coverage, each *sm* function has its specific weak and strong points. The approach presented here improves the global robustness of complex tasks execution in taking advantage of these *sm* functions complementarity.

To achieve these goals, we propose a two-stepped approach named Robel for *RO*bot *BE*havior *L*earning. First, *sm* functions are aggregated in a collection of Hierarchical Tasks Networks (HTN) [17], that are complex plans called *modalities*. Each modality is a possible way to achieve the desired task. One contribution of this work is to use and to synthesize modalities relying on the HTN formalism. In a second step, the relationship between supervision states and the appropriate modality for pursuing the task is learned through experience as a Markov Decision Process (MDP) which provides a policy for achieving the task. The second contribution of this work is an original approach for learning from the robot experiences an MDP-based supervision graph which enables to choose dynamically a modality appropriate to the current context for pursuing the task. We obtain a system able to efficiently use redundancies of low-level *sm* functions to robustly perform high-level tasks.

In the first Section we describe the *sm* functions and briefly detail their forces and weaknesses. In Section 2 we introduce what are the modalities, the one written by hand or better, how we can synthesize them automatically using a planner. Section 3 describes the *controller* and the learning mechanism which has been deployed to choose at run time the appropriate modality for pursuing a task in the current environment. Finally we present the results obtained both on the modalities planning/synthesizing problem and on learning their best use. We conclude with a discussion, and a prospective on this subject.

[1] SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025-3493, USA. email: morisset@ai.sri.com
[2] LAAS-CNRS, 7 avenue du Colonel Roche 31077 Toulouse Cedex 4 FRANCE email: firstname.lastname@laas.fr

## 1 Sensory-Motor Functions

Thanks to years of research in robotics, a large number of *sm* functions are now available, in particular for navigation tasks. To give an idea of this diversity, we briefly present them, according to the main functionality they provide.

Reliable and precise localization is often hard to obtain, numerous methods have been developed to provide this functionality. *Odometry* is easy to use but, due to drift and slippage, is seldom precise enough to perform long-range navigation. *Segment-based localization* is generally reliable in indoor environment [16], but laser occlusion gives unreliable data. Moreover, in long corridors the laser get little data along the corridor axis, thus the drift increases. *Stereo Vision odometry* [11] is more precise than classic odometry, but requires heavy computations which limits drastically the refresh rate of the current position estimate. Moreover, it is very sensible to any parameters which may impede the stereo correlation. *Global Positioning System* can be used for absolute localization, although one need a differential system to have an accurate measure. *Localization on landmarks* such as wall posters in long corridors can provide an accurate localization [2]. However, landmarks are usually only available and visible in few areas of the environment.

According to the type of mission and environment, different type of path planners can be used: *nav* [22] or *m2d* [21]. The first one is better for exploration of unknown regions, but at a high computational cost. The second one is fairly generic and robust, although it may require further processing to get an executable dynamic trajectory to take into account environment changes that occur during navigation.

For locomotion on rough terrains, we use a motion planner/motion generator named *P3D* [5] which is designed essentially for exploration of static environments. Motion control in dynamic environments has been implemented using *ND* [14]. It offers reactive motion capability that remains efficient in very cluttered space, but may fall in local minima. The *elastic band* [19] gives a very robust method for long range navigation, but can be blocked by a mobile obstacle that traps the band against a static obstacle. Last, the dynamic deformation is computationally intensive and may limit the reactivity in cluttered, dynamic environments and may also limit the band length.

## 2 Synthesis of Modalities

A high level task given by a mission planning step requires an integrated use of several *sm* functions among those presented earlier. Each consistent combination of these *sm* functions is a particular plan called a *modality*. A modality is one way of performing the task. A modality has specific characteristics that make it more appropriate for some contexts or environments, and less for others. The choice of the right modality for pursuing a task is far from being obvious. The goal of the *controller* (see Section 3) is to perform such a selection all along the task execution.

We chose to represent modalities as Hierarchical Task Networks. We believe that the HTN formalism is adapted to modalities because of its expressiveness and its flexible control structure [8]. HTNs offer a middle ground between programming and automated planning, allowing the designer to express the control knowledge which is available here. So we can use the same formalism to write a modality by hand or to generate it automatically.

### 2.1 Model of data flow

Creating a consistent modality may be seen as a planning problem using the correct models. To build a coherent modality, we want the planner to produce a plan which properly connect the available modules implementing the *sm* functions. So we need to model the *sm* functions previously described from a data flow point of view. We give a module a semantic by considering it as a black box with some input and output data. By correctly typing the data, and choosing the right *sm* functions, we are able to build a coherent chain of data processing, for a navigation modality. This chain goes from external data to the goal, through sensors, map building and motion generation of a real movement (see Fig. 1).

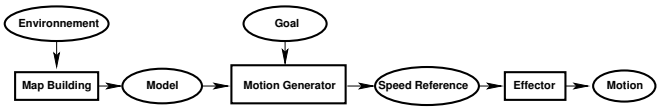



**Figure 1.** Example of a data flow

This scheme might be generalized easily to other kinds of modality, like exploration, manipulation or pure learning, depending on the available functional modules.

To perform this planning task we use SHOP [18], a hierarchical task planner. It uses a description of the domain, operators, and decomposition methods to go from a high level tasks to terminal actions. To get SHOP to produce the correct connections between modules, we set some planning operators:

```
(!connect_as_prod ?func ?data)
(!connect_as_cons ?func ?data)
```

The first one says that a function produces a type of data, the other that a function consumes a type of data. This operators assert the corresponding predicates, to explicit the current state:

```
(connected_producer ?func ?data)
(connected_consumer ?func ?data)
```

We also have to specify correctly the functions. This is achieved as follows:

```
(function lane.fuse)
  (needs lane.fuse (3D_image video_image))
  (produces lane.fuse 3D_model)
```

In the initial state of the problem, the only available data is input data such as *goal*, and we want to reach a state where the *motion* data is available. We implement a backward-chaining mechanism that starts from the final state and choose an operator (a *sm* function) that will provide this data. The new current state needs the input data of our operator. If they are not available, we choose a new function to produce them and so on. The backtrack points of the search are the choice at this time of the *sm* function. If we need a position (for localization), we may choose as different functions as *localization on landmarks* or *visual odometry* to perform it. We may even choose both of them using some data fusion mechanism available on our robots. This regression/decomposition process stops when all required data are available, typically provided by a sensor.

Thus we have the methods:

```
(:method (find_producer ?d)
         ((connected_producer ?f ?d))
         ()
         ((produces ?f ?d))
         ((!connect_as_prod ?f ?d)
          (find_consumable ?f)))
(:method (find_consumable ?f)
         ((needs ?f ?data)
         ((connect_as_cons ?f ?data)
          (find_producer ?data))))
```

The first one is to find the producers for a type of data. If this data is already produced, we stop the decomposition (this is the meaning of the empty parenthesis), else we find a producer and continue. The second one is straightforward.

However, building a modality is not just linking a network of producers/consumers. If this process can synthesize simple modalities, most complex ones require other attributes and parameters to be taken into account.

### 2.2 Other Attributes

We list here the other attributes the modality planning system has to take into account, like the **resources**. A physical device may be required, at the same time, by two different modules in the same modality. Similarly, CPU and Memory usage are interesting resources to model too. Such resources usage could be expressed as constraints in the HTN formalism. The **time** is also important: most modules are synchronous and have their own frequency, which on our robots vary from 50 Hz to less than 1 Hz. A realistic model must take this into account to ensure a correct execution. We may also notice the some processing are done **on request**, when instructed by another module or the executive, or can be **started and stopped** and produce some data at a given period. A **synchronous/asynchronous** attribute is necessary: some execution requests may be synchronized with respect to others (waiting for some data to be available) or may be running at their own pace, using whatever data is then available.

As of today, the current implementation of the modalities synthesizing part of Robel uses the data flow model as well as a simple synchronous/asynchronous and cyclic/on request model. Still, we are able to produce interesting modalities (See Section 4.1). Nevertheless, using a complete model taking into account all the attributes, we expect to be able to automatically produce modalities as rich and as robust than the handwritten ones.

The solution to this problem gives an a priori valid modality (with respect to the model), which can then be "tested" on line, thus allowing the *controller* to learn in which situations it is appropriate to use it. To conclude on this part, we may say that we are now able to automatically generate the executable code of a modality from a few information on how the *sm* work. The next challenge is to learn to use them efficiently.

# 3 The Controller

## 3.1 Qualitative Model of the Environment

We present in this section an example of a controller adapted to an indoor navigation task. To perform this specific task, the design of the control space and the control process itself require the use of a topological graph. Cells are polygons that partition the metric map. Each cell is characterized by its name and a *color* that corresponds to navigation features such as *Corridor, Corridor with landmarks, Large Door, Narrow Door, Confined Area, Open Area and so on*. Edges of the topological graph are labeled by estimates of the transition length from one cell to the next and by heuristic estimates of how easy such a transition is.

## 3.2 The Control Space

The controller has to choose a modality that is most appropriate to the current execution state for pursuing the task. In order to do this, a set of *control variables* has to represent control information for the *sm* functions. The choice of these control variables is an important design issue.

For example, in the navigation task in an indoor environment, the control variables are:

- the **cluttering** of the environment which is defined to be a weighted sum of the distances to nearest obstacles perceived by the laser, with a dominant weight along the robot motion axis;
- the **angular variation** of the profile of the laser range data which characterizes the robot area. Close to a wall, the cluttering value is high but the angular variation remains low. But in an open area the cluttering is low while the angular variation may be high;
- the **inaccuracy of the position estimate**, as computed from the co-variance matrix maintained by each localization *sm* function;
- the **confidence** in the position estimate (because the inaccuracy is not sufficient to qualify the localization, each localization *sm* function supplies a confidence estimate about the last processed position);
- the **navigation color** of current area is used when the robot position estimate falls within some labeled cell of the topological graph, the corresponding labels are taken into account;
- the **current modality** is essential to assess the control state and possible transitions between modalities.

A control state is characterized by the discretized values of these control variables. We finally end-up with a discrete control space which allows us to define a *control automaton*.

## 3.3 The Control Automaton

The control automaton is nondeterministic: unpredictable external events may modify the environment, e.g. someone passing by may change the value of the cluttering variable, or the localization inaccuracy variable. Therefore the execution of the same modality in a given state may lead to different adjacent states. This nondeterministic control automaton is defined as the tuple $\Sigma = \{S, A, P, C\}$:

$S$ is a finite set of control states,

$A$ is a finite set of modalities,

$P : S \times A \times S \rightarrow [0, 1]$ is a probability distribution on the state-transition, $P_a(s'|s)$ is the probability that the execution of modality $a$ in state $s$ leads to state $s'$,

$C : A \times S \times S \rightarrow \Re^+$ is a positive cost function, $c(a, s, s')$ corresponds to the average cost of performing the state transition from $s$ to $s'$ with the modality $a$.

$A$ and $S$ are given by design from the definition of the set of modalities and of the control variables. $P$ and $C$ are obtained from observed statistics during a learning phase.

The Control automaton $\Sigma$ is a Markov Decision Process. As an MDP, $\Sigma$ could be used reactively on the basis of a universal policy $\pi$ which selects for a given state $s$ the best modality $\pi(s)$ to be executed. However, a universal policy will not take into account the current navigation goal. A more precise approach takes into account explicitly the navigation goal, transposed into $\Sigma$ as a set $S_g$ of goal states in the control space. This set $S_g$ is given by a look-ahead mechanism based on a search for a path in $\Sigma$ that reflects a topological route to the navigation goal.

### 3.3.1 Goal States in the Control Space

Given a navigation task, a search in the topological graph provides an optimal route $r$ to the goal, taking into account estimated cost of edges between topological cells. This route will help finding in the control automaton desirable control states for planning a policy. The route $r$ is characterized by the pair $(\sigma_r, l_r)$, where $\sigma_r = \langle c_1 c_2 \dots c_k \rangle$ is the sequence of colors of traversed cells, and $l_r$ is the length of $r$.

Now, a path between two states in $\Sigma$ defines also a sequence of colors $\sigma_{path}$, those of traversed states; it has a total cost, that is the sum $\sum_{path} C(a, s, s')$ over all traversed arcs. A path in $\Sigma$ from the current control state $s_0$ to a state $s$ corresponds to the planned route when the path *matches* the features of the route $(\sigma_r, l_r)$ in the following way:

- $\sum_{path} c(a, s, s') \geq K l_r$, $K$ being a constant ratio between the cost of a state-transition in the control automaton to corresponding route length,
- $\sigma_{path}$ corresponds to the same sequence of colors as $\sigma_r$ with possible repetition factors, i.e., there are factors $i_1 > 0, \dots, i_k > 0$ such that $\sigma_{path} = \langle c_1^{i_1}, c_2^{i_2}, \dots, c_k^{i_k} \rangle$ when $\sigma_r = \langle c_1, c_2, \dots, c_k \rangle$.

This last condition requires that we will be traversing in $\Sigma$ control states having the same color as the planned route. A repetition factor corresponds to the number of control states, at least one, required for traversing a topological cell. The first condition enables to prune paths in $\Sigma$ that meet the condition on the sequence of colors but cannot correspond to the planned route. However, paths in $\Sigma$ that contain a loop (i.e. involving a repeated control sequence) necessarily meet the first condition.

Let $\mathsf{route}(s_0, s)$ be true whenever the optimal path in $\Sigma$ from $s_0$ to $s$ meets the two previous conditions, and let $S_g = \{s \in S \mid \mathsf{route}(s_0, s)\}$. A Moore-Dijkstra algorithm starting from $s_0$ gives optimal paths to all states in $\Sigma$ in $O(n^2)$. For every such a path, the predicate $\mathsf{route}(s_0, s)$ is checked in a straightforward way, which gives $S_g$. It is important to notice that this set $S_g$ of control states is a *heuristic projection* of the planned route to the goal. There is no guaranty that following blindly (i.e., in an open-loop control) a

path in $\Sigma$ that meets route$(s_0, s)$ will lead to the goal, and there is no guarantee that every successful navigation to the goal corresponds to a sequence of control states that meets route$(s_0, s)$. This is only an efficient and reliable way of focusing the MDP cost function with respect to the navigation goal and to the planned route.

### *3.3.2 Finding a Control Policy*

At this point we have to find the best modality to apply to the current state $s_0$ in order to reach a state in $S_g$, given the probability distribution function $P$ and the cost function $C$. A simple adaptation of the *Value Iteration* algorithm solves this problem. Here we only need to know $\pi(s_0)$. Hence the algorithm can be focused on a subset of states, basically those explored by the Moore-Dijkstra algorithm.

The closed-loop controller uses this policy as follows:

- the computed modality $\pi(s_0)$ is executed;
- the robot observes the state $s$, it updates its route $r$ and its set $S_g$ of goal states, it finds the new modality to apply to $s$.

This is repeated until the control reports a success or a failure. Recovery from a failure state consists in trying from the parent state an untried modality. If none is available, a global failure of the task is reported.

### *3.3.3 Estimating the Parameters of the Control automaton*

A sequence of randomly generated navigation goals is given to the robot. During its motion, new control states are met and new transitions are recorded or updated. Each time a transition from $s$ to $s'$ with modality $a$ is performed, the traversed distance and speed are recorded, and the average speed $v$ of this transition is updated. The cost of the transition $C(a, s, s')$ can be defined as a weighted average of the traversal time for this transition taking into account the eventual control steps required during the execution of the modality $a$ in $s$ together with the outcome of that control. The statistics on $a(s)$ are recorded to update the probability distribution function. Several strategies can be defined to learn $P$ and $C$ in $\Sigma$. The first one is used initially to expand $\Sigma$: a modality is chosen randomly for a given task; this modality is pursued until either it succeeds or a fatal failure is notified. In this case, a new modality is chosen randomly. This strategy is used initially to expand $\Sigma$. $\Sigma$ is used according to the normal control except in a state on which not enough data has been recorded; a modality is randomly applied to this state in order to augment known statistics, e.g, the random choice of an untried modality in that state.

## 4 Experimental results

The justification of the whole system relies on the following principle : the use of the complementarity of several navigation modalities increases the global robustness of the task execution. To validate this principle, 5 handwritten modalities have been integrated inboard one of our robot. In order to characterize the usefulness domain of each modality we measured in a series of navigation tasks, the success rate and other parameters such as the average speed, the distance covered, the number of retries. Various cases of navigation have been considered such as for instance, long corridors or large areas, cluttered or not, occluding the 2D characteristic edges of the area or not. These extensive experiments described in details in [15] required several kilometers of navigation. The result is that for each case of navigation met by the robot there is at least one successful modality. On the other hand, no modality is able to cover all cases. This result clearly supports our approach of a supervision controller switching from one modality to another one according to the context.

A second step of experiments is focused on the automatic modalities synthesis by a planning process. These ongoing results are presented in the next section. Finally, the learning capabilities of the controller are illustrated in section 4.2.

### 4.1 Modalities Synthesis

Let us give some examples of synthesized modalities. On Fig. 2, we can see the HTN built by SHOP. The corresponding modality is shown on Fig. 3.

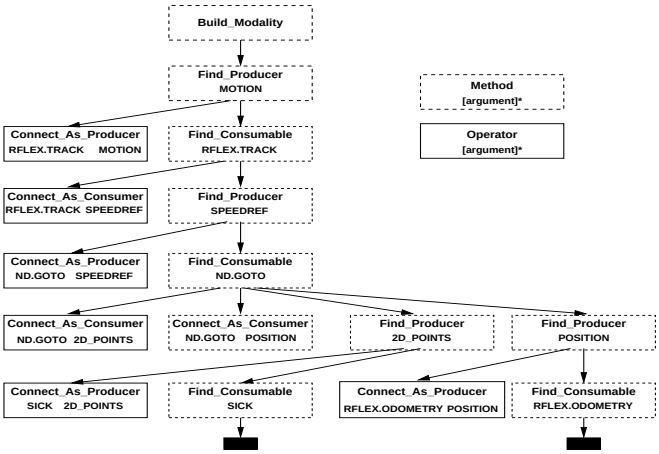



**Figure 2.** HTN built by SHOP for a simple modality

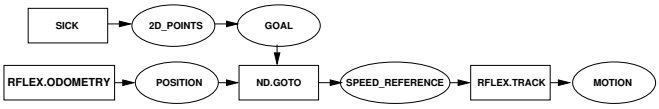



**Figure 3.** A simple modality

The *odometry sm* function does not need any data at input, nor does the sick (laser range finder). The *ND* motion generator uses points from the sick to find out where are obstacles and gives the speed reference to avoid obstacles and go to the goal.

This modality is better suited for exploration of "slightly" dynamic environments, at low speed. This modality is the most simple of the 42 generated for the outdoor mobile robot, using 16 *sm* functions

Another (more complex) modality is shown on Fig. 4. We can see that the robot uses its cameras to take images, and stereo-correlation to have a 3D image. From here, it uses this image to compute a motion (*Stereo Vision Odometry*) which combined with classic odometry will give the localization. The modality also builds a 3D metric model, the corresponding 3D qualitative model and projects it to obtain a 2D qualitative model. This model is used to give long range path made of way points. This way points are consumed one by one by a motion generator that gives speed references using the 3D metric

model (and of course the current position). Then this speed reference is used by the low-level effector to generate effectively the robot motion.

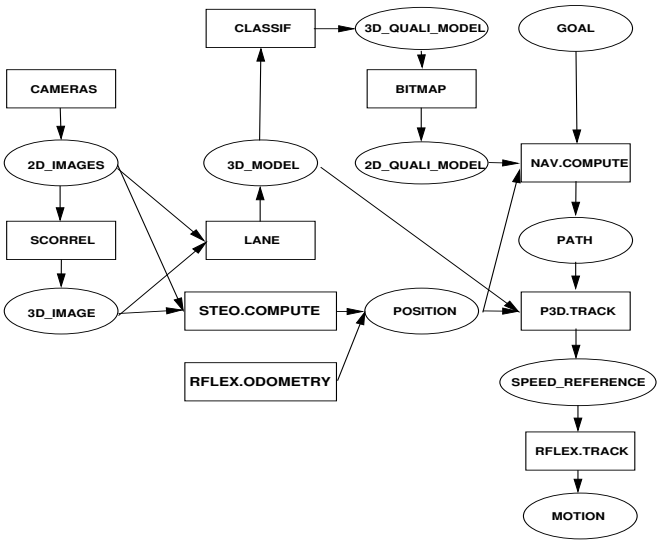



**Figure 4.** A more complex modality

The problem we are facing now is to define criterion and ways for selecting a *good* set of modalities. A too large set will make the learning of the controller unrealistically long and costly, but we obviously need a sufficiently large set to cover the main ways of combining the *sm* functions. For the moment, this selection is performed interactively by the robot designer. This still provides a significant benefit in robustness, programming and debugging time w.r.t. handwriting the modalities.

## 4.2 Controller

. We propose here to illustrate the learning capabilities of the controller through an indoor navigation task. To perform this experiment, we start with an empty automaton and 2 complementary modalities: the first one ($M_1$) is composed by the *elastic band*, *m2d* and the *segment based* localization function, while the second one ($M_2$) works reactively without any path planner. *ND* performs the obstacle avoidance and the robot is localized with the same function as $M_1$. The velocity and the path planner make $M_1$ more efficient in large and open environments. On the other hand, the limited avoidance capabilities of the *elastic band* makes $M_2$ more adapted in highly cluttered environments.

In this three-stepped experiment, the strategy of learning favors the completion of each transition( 3.3.3). If a modality has not been tried for the current state (untried modality), this modality is executed without any computation of $\pi$.

**Phase 1**. The learning starts with a series of 83 navigations in a large open environment. During these navigations, 86 states and 159 transitions are created (Fig. 5). After the $53^{th}$ navigation (noted $n53$) the number of new transitions met by the system tends to be stable. Between $n53$ and $n83$, the computation of $\pi$ returns $M_1$ for any state encountered except for 2 states (in $n60$ and $n70$) whose transitions with $M_2$ were still untried (Fig. 6). The constant selection of $M_1$ by $\pi$ all along the 30 last navigations shows that the controller relevantly learned the superiority of $M_1$ on $M_2$ for the open environments.

**Phase 2**. Some narrow obstacles are added. This new situation generates 14 new states between $n84$ et $n87$ and 10 new transitions are tried until $n93$. During these 9 first navigations, 6 failures are recorded for $M_1$, each time from a state with a high level for the clutter variable. After $n101$ and until $n114$, each time a state with a high level for the clutter variable is encountered, the execution of $M_1$ is stopped by the controller and the obstacle avoidance is systematically performed with $M_2$. No more failures are recorded with $M_1$. As soon as the clutter level recovers a low level, the computation of $\pi$ switches back to a selection of $M_1$. $M_1$ is then kept as long as the value of the clutter state variable stays low. If in the previous phase, $M_1$ was more appropriate than $M_2$, in this second phase, the system is able to learn within 30 navigations, the better efficiency of $M_2$ in cluttered environments.

**Phase 3**. The goal of the third step is to check if the learning of the second phase (avoidance) didn't corrupt the learning of the first phase (open navigation). The obstacles are then removed to recover the same environment as the phase 1 and 58 more navigations are performed by the system. This new step shows that the learning of the phase 1 was not complete: 10 new states are created and 23 untried transitions are completed. Despite these untried transitions, between $n114$ and $n152$, 30 navigations are performed with 100 % of selection for $M_1$ by $\pi$. After $n151$, $M_1$ is constantly selected all along the 21 last navigations. This last step shows that despite an incomplete learning, the efficiency of $M_1$ in the phase 1 has not been forgotten after the learning of the phase 2.

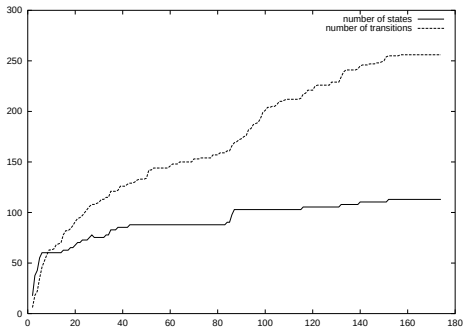



**Figure 5.** Evolution of the size of the graph

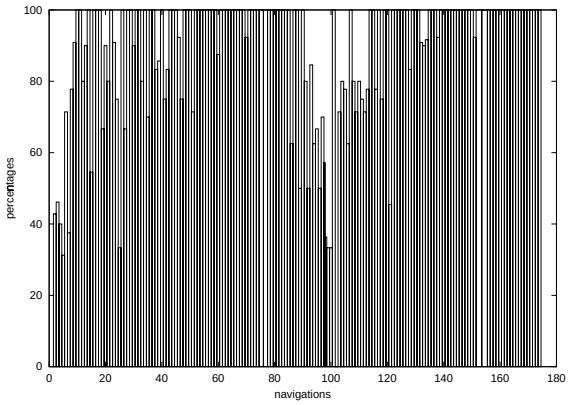



**Figure 6.** Percentage of choice of $M_1$

## Discussion and Conclusion

This paper addressed the issue of producing complex modalities from sensory motors functions, and how to exploit the complementarity of these modalities to perform a task.

We have shown that it is indeed feasible to synthesize modalities from generic specifications, we still need to improve this planning component to take into account attributes such as resource, time and synchronism.

This is certainly not the first contribution that relies on a planning formalism and on plan-based control in order to program an autonomous robot. For example, the *"Structured Reactive Controllers"* [3] are close to our concerns and have been demonstrated effectively on the Rhino mobile robot. The efforts for extending and adapting the Golog language [12] to programming autonomous robots offer another interesting example from a quite different perspective, that of the Situation Calculus formalism [20]. The *"societal agent theory"* of [13] offers also another interesting approach for specifying and combining sequentially, concurrently or in a cooperative mode several agent-based behaviors; the CDL language used for specifying the agent interactions is similar to our Propice programming environment. Let us mention also the *"Dual dynamics"* approach of [10] that permit the flexible interaction and supervision of several behaviors. These are typical examples of a rich state of the art on possible architectures for designing autonomous robots. (see [1] for a more comprehensive survey).

Here also the use of MDPs for supervision and control of robot navigation tasks is not new. Several authors expressed directly Markov states as cells of a navigation grid and addressed navigation through MDP algorithms, e.g. value iteration [23, 6, 7]. Learning systems have been developed in this framework. For example, XFRMLEARN extends these approaches further with a knowledge-based learning mechanism that adds subplans from experience to improve navigation performances [4]. Other approaches considered learning at very specific levels, e.g., to improve path planning capabilities [9]. Our approach stands at a more abstract and generic level. It addresses another purpose: acquiring autonomously the relationship from the set of supervision states to that of redundant modalities. We have proposed a convenient supervision space. We have also introduced a new and effective search mechanism that projects a topological route into the supervision graph. The learning of this graph relies on simple and effective techniques, whose results provide two particular features:

**Portability:** Variables of the control state reflect control information for the *sm* functions. No information dedicated to the environment is present in the control state. In this sense we say that the control state is *abstract*. Thanks to this characteristic, a controller learned in an environment can directly be used in another environment.

**Adaptativity:** In this system, learning and execution are not decoupled : learning of $\Sigma$ parameters is active all along the robot navigations. If a new situation is encountered, corresponding new states are created in $\Sigma$ and the new untried transitions are evaluated and taken into account by the next computations of $\pi$. This unsupervised learning confers a high level of adaptativity to the controller.

In addition to future work directions mentioned above, an important test of Robel will be the extension of the set of tasks to manipulation tasks such as *"open a door"*. This significant development will require the integration of new manipulation functions, the synthesizing of new modalities for these tasks and the extension of the controller state. Another development which seems rather promising is to learn the control space of the controller instead of relying on one given by hand.

## REFERENCES


[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, 'An Architecture for Autonomy', *IJRR*, **17**(4), 315–337, (April 1998).

[2] V. Ayala, J.B. Hayet, F. Lerasle, and M. Devy, 'Visual localization of a mobile robot in indoor environments using planar landmarks', in *IEEE IROS'2000, Takamatsu, Japan*, pp. 275–280, (November 2000).

[3] M. Beetz, 'Structured reactive controllers - a computational model of everyday activity.', in *3rd Int. Conf. on Autonomous Agents*, (1999).

[4] M. Beetz and T. Belker, 'Environment and task adaptation for robotics agents', in *ECAI*, (2000).

[5] D. Bonnafous, S. Lacroix, and T. Simon, 'Motion generation for a rover on rough terrains', in *International Conference on Intelligent Robotics and Systems*, Maui, HI (USA), (October 2001). IEEE.

[6] Cassandra, Kaelbling, and Kurien, 'Acting under uncertainty: Discrete bayesian models for mobile robot navigation', in *Proceedings of IEEE/RSJ IROS*, (1996).

[7] T. Dean and M. Wellman, 'Planning and control', in *Morgan Kaufmann*, (1991).

[8] K. Erol, J. Hendler, and D.S. Nau, 'HTN planning: Complexity and expressivity.', in *AAAI*, (1994).

[9] K. Z. Haigh and M. Veloso, 'Learning situation-dependent costs: Improving planning from probabilistic robot execution', in *In 2nd Int. Conference on Autonomous Agents*, (1998).

[10] J. Hertzberg, H. Jaeger, P. Morignot, and U. R. Zimmer, 'A framework for plan execution in behavior-based robots', in *ISIC-98 Gaithersburg MD*, pp. 8–13, (1998).

[11] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila, 'Autonomous rover navigation on unknown terrains, functions and integration', *International Journal of Robotics Research*, (2003).

[12] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(1-3), 59–83, (1997).

[13] D. C. MacKenzie, R. C. Arkin, and J. M. Cameron, 'Multiagent mission specification and execution', in *Autonomous Robots, 4(1):29 V52*, (1997).

[14] J. Minguez, L. Montano, T.Simeon, and R. Alami, 'Global nearness diagram navigation (GND)', in *ICRA2001, Korea*.

[15] B. Morisset, *Vers un robot au comportement robuste. Apprendre combiner des modalits sensori-motrices complmentaires.*, Ph.D. dissertation, Université Paul Sabatier, Toulouse, novembre 2002.

[16] P. Moutarlier and R. G. Chatila, 'Stochastic Multisensory Data Fusion for Mobile Robot Location and Environment Modelling', in *Proc. International Symposium on Robotics Research, Tokyo*, (1989).

[17] D. Nau, Y. Caoand, A. Lotem, and H. Munoz-Avila., 'Shop: Simple hierarchical ordered planner', in *IJCAI*, (1999).

[18] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell, 'Total-order planning with partially ordered subtasks', in *IJCAI*, Seatle, (2001).

[19] S. Quinlan and O. Khatib, 'Towards real-time execution of motion tasks', in *Experimental Robotics 2*, eds., R. Chatila and G. Hirzinger, Springer Verlag, (1992).

[20] R. Reiter, 'Natural actions, concurrency and continuous time in the situation calculus.', in *KR*, pp. 2–13, (1996).

[21] T. Simeon and B. Dacre Wright, 'A practical motion planner for all-terrain mobile robots', in *IEEE/RSJ IROS*, (1993).

[22] S.Lacroix, I.K.Jung, J.Gancet, and J.Gonzalez, 'Towards long range autonomous navigation', in *7th ESA Workshop on Advanced Space Technologies for Robotics and Automation*, Noordwijk (The Netherlands), (November 2002).

[23] S. Thrun, A. Buecken, W. Burgard, D. Fox, T. Froehlinghaus, D. Henning, T. Hofmann, M. Krell, and T. Schmidt, 'Map learning and high-speed navigation in rhino', in *AI-based Mobile Robots: Case Studies of Successful Robot Systems*, eds., D. Kortenkamp, R.P. Bonasso, and R. Murphy. MIT Press, (1998).

# Patterns in Reactive Programs

**P@trik Haslum**[1]

**Abstract.** In this paper, I explore the idea that there are "patterns", analogous to software design patterns, in the kind of task procedures that frequently form the reactive component of architectures for intelligent autonomous systems. The investigation is carried out mainly within the context of the WITAS UAV project.

## 1 Introduction

A current trend in AI research is the focus on *autonomous systems*: robotic, or "softbotic", systems capable of acting independently and intelligently, in uncertain, varied and dynamic environments. The wish to make agents act intelligently and the requirements imposed by the dynamic nature of the environment combine to create competing needs for *deliberation* and *reaction*, and much attention has been given to the design of system architectures that somehow mediate between these needs. The solutions that have, to date, appeared most successful are variants of so called "layered architectures", where deliberative, reactive and low-level process components run and exchange information asynchronously, *e.g.* [10, 1, 13].

Part of every layered architecture variant is a reactive system, consisting of a base of *procedural knowledge*, rules that prescribe reactions to standard situations or flexible "scripts" for enacting standard tasks, and a mechanism that translates the encoded knowledge into action (*i.e.* commands to low-level system processes), contingent on the state of the environment as perceived via the systems sensors. The core of any reactive system is the procedural knowledge base, and there have been many proposals for languages to express such knowledge. Less attention has been devoted to the problem of how to fill the procedural knowledge base with *content*. The task of writing the actual rules or programs that the reactive layer will execute is often an ardous one, and there are few principles or guidelines to help the programmer who must in the end perform it[2] (notable exceptions are discussed in section 6).

By contrast, principles of programming and design have since long been studied and developed in other areas of computer science, often without reference to the particulars of any implementation language. It may be worthwhile to look likewise at the problem of writing reactive programs. This paper presents a tentative step in this direction: By examining examples of reactive programs, from the literature and from experiences in the WITAS autonomous UAV project, I try to find something akin to "design patterns" [7] for reactive layer procedures. Whether the things found actually qualify as design patterns (or, indeed, have any value or meaning at all) is debatable. This little effort should perhaps be viewed more as a sign that *something* is missing, rather than a definitive answer to what that something should be.

The next three sections provide brief background on languages for programming reactive procedures (section 2), the WITAS project and system architecture currently employed within it (section 3) and design patterns (section 4). Section 5 describes and illustrate some reactive design patterns, while section 6 contains some concluding discussion.

## 2 Rule and Reactive Procedure Languages

A large class of reactive programming languages consist of *condition – action* (or *event – condition – action*) rules. Such rules prescribe an action to be taken whenever the corresponding condition on the systems view of the environment holds. As the complexity of tasks and environment grows, two problems with rule-based programs arise: In a given situation, the condition part of several rules, advising conflicting actions, may hold, or no rules condition may be fulfilled, leaving the system without a response.

A common solution to the first problem is to introduce an *arbitration* mechanism that decides which rule takes precedence in each situation. An example of this approach is the context-dependent blending method developed by Saffiotti *et al.* [15]. In this system, the antecedent of a rule is a combination of (fuzzy) predicates, while the consequent specifies for each possible action how desirable that action is, from the point of view of the rule. Collections of rules are grouped into behaviors, associated with a (fuzzy) context conditions. The desirability assigned to each action by a rule is modified by the degree to which the conditions of the rule and the behavior it belongs to are satisfied, and the action that receives the most support, overall, is the one taken. A different solution, examplified by Lin [11], is to perform a static analysis on the rule base and verify that a conflict can not occur.

Reactive procedures were introduced into AI circles mainly with the PRS system [8]. The RAPS [5] system is similar, and serves as an illustrative example of the approach. A task procedure in RAPS has a "signature", consisting of name and arguments, a context condition, a success condition and a procedure body which is a partially ordered set of steps that may be calls to subtasks, primitive actions, or "wait for condition" statements. RAPS allows having several different procedures for the same task and this is what lends flexibility to their execution: When a call for a particular procedure signature is made, the context conditions of all matching procedures are evaluated against the systems current knowledge state, and among those whose conditions are satisfied, one is chosen for execution. Eventually, the procedure either finishes successfully (its success condition becomes satisfied) or fails. If the chosen procedure failed, the system

[1] Linköpings Universitet (pahas@ida.liu.se)

[2] Some systems, *e.g.* CIRCA [14] or the situated automata of Kaelbling and Rosenschein [9], synthesize the reactive procedures from a specification of the task to be achieved and a model of the environment. This, for better or worse, moves the problem from programming to specification and modelling, but at the same time leads to some restrictions on the expressivity of the task specifications, environment model, and the procedures that the system is able to generate, to make the synthesis problem decidable.

may re-evaluate context conditions to choose another procedure (or even to try the same procedure again) or the call as a whole may fail.

## 3 The WITAS Project and Architecture

The WITAS project aims to develop architechtures and technologies for intelligent autonomous systems in general, and for an autonomous Unmanned Airial Vehicle (UAV) for traffic surveillance in particular[3]. This choice of platform and application area leads to many challenges, but also to a problem that, overall, is in the realm of the possible.

The current WITAS system architecture is the result of many development iterations. Two important characteristics that have emerged are that it is distributed, and that the reactive system plays a central and "driving" role. A distributed architecture is advantageous for several reasons:

- Different system components have different needs: The UAV controller operates under hard real-time constraints, parts of the image processing system may need to run on specialized hardware to achieve acceptable performance, *etc.* In addition, the limited power and payload capacity of the UAV may force some components to reside in a ground-side part of the system, while still interoperating smothly with those components that reside on-board the UAV.
- Distribution lends a certain fault-tolerance, since separate system components can be restarted (or even rebooted) in case of failure, without the need to bring the whole system down.
- An interesting area for future research is the integration of several UAVs, and possibly also other actors such as ground stations (manned or unmanned), into a system capable of acting coherently and efficiently. A distributed architecture leaves many more "hooks" for development in this direction.

To minimize the extra complexity introduced by distribution, a choice has been made to use a CORBA infrastructure[4]. The use of CORBA carries some additional advantages, such as for instance simplifying the transition from a simulated to a real environment.

The "reactive-centric" nature of the architecture is in part an effect of the fact that is distributed, and of the use of CORBA: At the level of interfaces, there is simply not that much difference between *e.g.* the UAV flight control system and a high-level deliberative function such as prediction or a GIS database, and thus the different components are naturally viewed as a collection of "services" for the reactive systems use.

### 3.1 The Modular Task Architecture

The reactive system, like the rest of the WITAS architecture, has been through a number of iterations. The current version, tentatively named the Modular Task Architecture (MTA), is, also like the rest of the WITAS system, distributed, and for pretty much the same reasons.

The common denominator shared by all MTA task procedures is a CORBA interface and a few behavioral restrictions. The task interface is rather basic, containing only operations such as passing arguments, starting and canceling a task. Asynchronous messages are sent from a task to its caller via event channels[5], and a few message types, *e.g.* those signalling task completion or failure, are standardized. This simplicity, however, should not be mistaken for a limitation. Beyond the requirements that MTA places on a task, each task procedure is free to react to events in any form and interact with any component of the WITAS system that is accessible through an interface.

Because the MTA is, in essence, only a standard, task procedures can be implemented in any language (for which there exists CORBA support). Indeed, they have to. As it has turned out that large parts of most task implementations tend to be routine excercises in CORBA programming, we have developed a simple macro language and translator to make writing tasks easier and less prone to cut-and-paste errors[6].

## 4 Design Patterns

There are numerous definitions of what constitutes a design pattern (or just "pattern"). The term was originally used by architect Cristopher Alexander, who has written several books to explain what is meant by it. Gamma, Helm, Johnson & Vlissides, whose book on object-oriented design patterns has probably done most to popularize its use in software engineering, write: "A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. [...] Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use." [7]. A shorter, less object-oriented, description is "A pattern is a named nugget of insight that conveys the essence of a proven solution to a recurring problem within a certain context amidst competing concerns" [2]. Patterns have been recognized at an "application" (or "architectural") level, at the design level, and at the "language" level, *i.e.* in program constructs (where they are often called "idioms") [2].

## 5 Patterns in Reactive Programs

This section describes five different patterns that I have seen in reactive task procedures. The first two examples presented here are more "architectural" in flavour, describing what is essentially structures in the procedural knowledge found in an application domain, while the remaining examples are more design oriented. The first two are also found in the structure of a single task procedure, while the other concern the interplay between two or more tasks.

### 5.1 Scripts

In the introduction, task procedures were described as "flexible scripts". This is a frequently occuring form, the task procedure consisting of a set of steps to be carried out, in sequence or just in partial order, interspersed with waiting for events or conditions.

[3] For a more detailed description of the project, see Doherty *et al.* [4] or `http://www.ida.liu.se/ext/witas/`

[4] The Common Object Request Broker Architecture (CORBA) is an object-oriented middelware standard laid down by the OMG (`http://www.omg.org/gettingstarted/corbafaq.htm`). The interfaces of CORBA objects are specified in the Interface Definition Language (IDL), which maps to whatever language is used to implement the object.

[5] Channels are specified in the CORBA Event Service standard (`http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm`), and allow decoupled passing of arbitrary data from one or more senders to one or more receivers.

[6] The language, called TSL, is based on XML. The translator consists basically of an XSLT processor and a library of templates for each implementation language used.

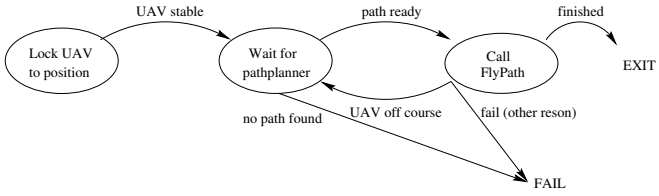



**Figure 1.** The `NavToPoint` Task Procedure

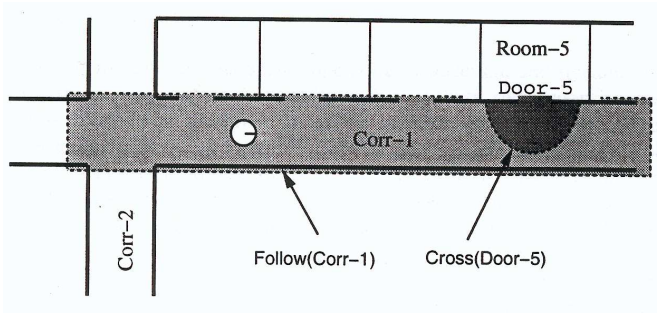



**Figure 2.** The `follow` and `cross` behaviors combined into a script. Reprinted from Saffiotti *et al.* (1995).

Figure 1 shows a schematic of a task procedure for safely navigating the WITAS UAV to a goal position. The normal execution of this task forms sequence of three steps, corresponding to the three states: First, the UAV is stabilized (locked onto its current position), then a request for a safe trajectory from this position to the goal is made to the pathplanning service, and finally a subtask, `FlyPath`, is invoked to execute the trajectory returned by the pathplanner. The possible exception to this normal case is if any of the steps fails, *e.g.* if the pathplanner can not find a flyable trajectory, or if the `FlyPath` task fails. In most cases, this leads to the task as a whole failing, but in one certain case, when `FlyPath` fails because the UAV has drifted too far off course, it only causes the task to "back up" and try again (the reason why it does not back up all the way to the first state is that the `FlyPath` task locks the UAV into a stable hovering mode when failing in this way).

Script-like task procedures can appear also in rule-based systems, though they may be less obvious. The following example of a script implemented by fuzzy behaviors (due to Saffiotti *et al.* [15]), involves a mobile robot navigating in an office environment: The robots goal is to enter a certain room, which it may achieve by a behavior `cross(doorway)`. The context of applicability of this behavior, however, is limited to the close vicinity of the doorway. Another behavior, `follow(corridor)` is applicable in any part of the corridor, and will when effected lead the robot down the corridor, eventually to reach the doorway. A procedure to achieve the goal from the wider context is created by conjoining to the context of `follow` the negation of the context condition for `cross`, and applying the context-dependent blending method to the two behaviors. The result acts like a script, applying first the `follow` behavior until a situation within the context of `cross` is reached, then the `cross` behavior. This is illustrated in figure 2.

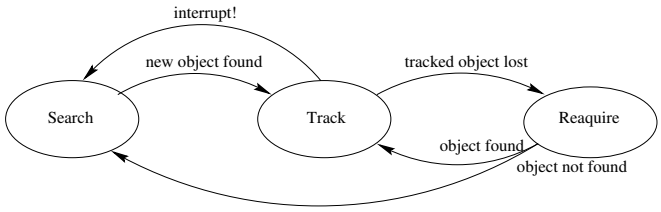



**Figure 3.** The `FindTrack` Task Procedure

## 5.2 Mode Switchers

While many task procedures take the form of scripts (though in some cases more elaborate, *e.g.* with alternate branches for different conditions), some are very definitely not of this kind. Another common cathegory are "mode switchers": tasks that continuously change between a set of different working modes, depending on circumstances, and that often do not have any wired-in terminating condition but carry on indefinitely, until interrupted from without.

Figure 3 shows an example of such a task procedure for the WITAS UAV, which uses the image processing system to alternatly search for objects (defined either by motion or by color) in the camera image, and tracking an object for as long as possible. It has three modes: Searching for an object, tracking the object once found, and "reaquire", which is entered immediately upon losing track and in which the task searches for an object similar to the one just lost near the objects last known image position, for a (short) limited time. While in the tracking mode, an interrupt command causes the task to drop the currently tracked object and resume search (a different external command causes the task to terminate, but this is not illustrated in the figure).

## 5.3 Fail and Retry

Reactive procedures are typically designed to accomplish a particular task, in a limited range of operational circumstances. This limitation is key to keeping the complexity of individual procedures managable. Were we to try to deal with every imaginable contingency that may arise in carrying out a desired task, procedures would quickly become unwieldingly complex. Also, the appropriate response to an abnormal situation may vary depending on the overall plan that the task is part of. For this reason, most reactive systems have a notion of task procedures *failing*. In *e.g.* RAPS or the WITAS system, failures are signalled explicitly, while in Saffiotti *et al's* system a behavior may be defined as "failing" when the degree of satisfaction of its context condition becomes too low.

A pattern related to failing is "catch and retry". It involves two task procedures, one of which ("the caller") has invoked the other ("the callee") and is waiting for it complete. There are many reasons that may cause the callee to fail, but for some of them, the caller can take measures to repair the failure, appropriate to the purpose that the caller called the callee for. Thus, if the callee fails, the caller determines the reason for the failure, and if it is of a kind that the caller knows how to deal with, it takes some action to remedy the problem and restarts, or calls again, the callee. If the failure is of any other kind, the caller itself fails.

An example of this pattern has already been shown, in the `NavToPoint` task procedure: If the `FlyPath` task fails due to the

UAV drifting off course, the procedure solves the problem (by asking the pathplanner for a new path, from the current position and invoking `FlyPath` again). Note that, again, this may not be the right way to deal with the problem in all contexts. If, for example, a preprogrammed path was flown to record sensor data, the flight may have to be started over from the beginning.

### 5.4 Supervision

Task failures is only one half of a two-sided problem. In some situations, a task procedure may be acting inappropriately without realising it, thus *not* failing when in fact it should.

A way of dealing with this situation is *supervision*. Again, this involves two task procedures, a "caller" and a "callee". The callee, during execution, sends "status reports" to its caller, who through monitoring these and the state of the environment may detect when the callee is responding incorrectly. The caller may then interrupt the callee, or send to it some corrective commands.

The `FindTrack` task described above frequently acts as callee in this kind of pattern. When it is invoked, it is for a purpose, *e.g.* to find and track a particular vehicle, but the `FindTrack` task can not discriminate the vehicle of interest from others that may be found, or even discriminate vehicles from other moving things. Therefore, the task reports to its caller whenever it changes from searching to tracking the identity of the tracked object. The calling task may then retrieve information about this object and apply more sophisticated reasoning to determine if it is one that should actually be tracked (*e.g.* matching the movement of the object with information about the road network in the area to determine if it follows the road or not). If it is not, the `FindTrack` task can be commanded to drop the object and switch back to searching.

The same result could, of course, be obtained by implementing the `FindTrack` functionality in its caller, adapted to the task that the caller performs, but separating the two confers several advantages: It avoids code duplication (*i.e.* writing and debugging the same program twice), since the `FindTrack` task procedure can be used by many different tasks. It keeps the calling task procedure simpler, since it does not have to handle the idiosyncracies of interfacing to the image processing module and since the tracking of an object operates concurrently with the (possibly time-consuming) reasoning performed by the calling task, without the need to write this concurrent handling into the calling task explicitly.

### 5.5 Higher-Order Task Procedures

In more complex task procedures, there is often a heirarchical structure: the task decomposes into a series (or set) of subtasks, with some coordinating or "bridging" activity between them. Sometimes, for a group of tasks this bridging part may be the similar, or even identical, even though the tasks in the group are applicationwise unrelated. For example, two potential tasks for the WITAS UAV are surveying a collection of buildings (or other structures of interest) scattered throughout an area, and searching an area for a particular vehicle. Both these tasks involve navigating the UAV to a series of positions in turn (positions of the buildings in the first case, positions where the sought vehicle is likely to appear in the second), and performing some data-gathering activity at each position (taking photographs from different angles in the first case, image/video analysis in the second).

A single task procedure could be written to handle both tasks (as well as other tasks with similar structure), by using enough parameters to define the data-gathering activity that has to be done at each position. However, this procedure will grow very complex and difficult to maintain as the set of possible data-gathering subtasks grows. But, since the navigation part of the overall task is (mostly) independent of the activity carried out at each position, an alternative is to write a "higher-order" task procedure, `DoAtPositions`, which takes as argument a set of positions and an arbitrary task to carry out at each position[7]. Again, this both simplifies the writing of the task procedures involved and improves the potential for reuse.

## 6 Conclusions

These ideas, although grounded in some experience, are speculative. Here are some objections that can reasonably be made:

### 6.1 "This is all very interesting, but hardly new."

Although the concept of design patterns in software was introduced not so many years ago [7], there has been an almost explosive development in "pattern recognition" since[8], and there are even collections of patterns specifically aimed at the kind of concurrent, distributed programming that is typical of MTA task implementations [16, 12]. Shouldn't the simple observations found in the preceeding section already have been made, many times over? Indeed, they have. But this lack of novelty is not a fault, since one of the hallmarks of a good pattern is recurrence, in varying contexts.

Also, a few AI researchers have discussed reactive programming practice: Firby [6] describes a collection of RAPS task procedures written for an in-door mobile service robot, and atempts to structure it into modular, reusable subtasks. His conclusion is that heirarchical task decomposition, while a powerful structuring principle, alone is not enough. Some tasks need to spawn subtasks whose execution is tied to a condition on the state of the robot or its environment, and thus streches beyond that of the spawning task (this is perhaps also a candidate for a pattern). Beetz [3] analyzes reactive plans (programs) for mobile robot navigation tasks and designs a representation language specific to this application by introducing constructs that match patterns of use. Examples of at least two of the patterns discussed in the preceeding section can be found among those: supervision (expressed as augmenting default plans with context-triggered subplans) and higher-order tasks (expressed by the "at location" macro, which specifies that a particular part of the plan needs to be executed at a certain position, abstracting the details of how to get the robot to the position from the task to be carried out there). Beetz also introduces the notion of a task procedure being "embeddable" (meaning it can be safely run concurrently with other tasks, even in the presence of conflicting resource needs), "interruptible" (meaning it can be interrupted/resumed at arbitrary points without compromising the procedures ability to complete its task) and "transparent" (meaning the procedure accomplishes one and only one goal, and that this goal is explicitly indicated), and argues that these are all desirable properites to form a library of reusable task procedures.

### 6.2 "This is all very interesting, but what's the point?"

There are several:

[7] The mechanism used for passing subtasks as arguments depends on the language used to implement the task procedure. In the MTA framework, a task can be passed as a CORBA object, or a specification of the task (name and arguments) can be passed as a data structure definable in IDL.

[8] As evidenced by pattern catalogs, *e.g.* at `http://hillside.net/patterns/`.

Patterns suggest good practices: A task procedure that provides information on the reason for failures is more reusable than one that does not, since it can be combined with other tasks in a catch-and-retry fashion. Likewise, a mode-switching task procedure that provides meaningful status reports and "hooks" to force mode changes can be supervised, and therefore more useful as a subtask. A common theme in all the three design-oriented patterns in the preceeding section is that they aim towards increasing the potential for *reuse*, which is a cornerstone of efficient (or economic) software construction.

Patterns of use motivate features of task procedure languages in existence, and suggest potentially useful features missing from languages of today. Beetz design of a representation for robot navigation tasks [3] is an example of this approach. In creating the reactive layer of a layered architecture, looking for patterns in the intended application domain(s) may also guide the choice of implementation technology. In the WITAS UAV project, we have found a mode-switching structure to be more common than a script-like one, at least among basic tasks having to do with control of the UAV platform and its sensors. In an early phase of the project, the reactive layer was implemented using the RAPS language, and one of the lessons learned from this was that although it is certainly *possible* to implement a mode switching task using the constructs of this language, it is not very convenient.

### 6.3 "This is all very interesting, but it should be formalized."

Probably the most important function served by patterns is as an educational resource: They communicate experience, insight, and sometimes inspiration, between people faced with similar problems (programmers, in the case of software patterns). Thus, it is more important for a pattern description to be human-readable than machine-readable. However, recurrent patterns in reactive programs possibly also point towards mechanisms for automatically synthesizing such programs, for example in the form of search spaces or search control knowledge for automated planners (this is also advocated by Beetz [3]).

## Acknowledgements

Thanks to the reviewers for pointing out Firby's work, and for raising some of the objections. The WITAS project is, of course, a team effort, but I would especially like to mention Per Nyblom, who is the author of the TSL translator, and who has also contributed much to the development of the library of task procedures for the WITAS UAV. This work is partially supported by research grants from the Wallenberg Foundation, Sweden and NFFP-539 COMPAS.

## References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand, 'An architecture for autonomy', *International Journal of Robotics Research*, **Special Issue on "Integrated Architectures for Robot Control and Programming"**, (1998).

[2] B. Appleton. Patterns and software: Essential concepts and terminology. `http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html`, 2000.

[3] M. Beetz, 'Plan representation for robotic agents', in *Proc. 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS'02)*, (2002).

[4] P. Doherty, G. Granlund, K. Kuchcinski, E. Sandewall, K. Nordberg, E. Skarman, and J. Wiklund, 'The WITAS unmanned aerial vehicle project', in *Proc. European Conference on Artificial Intelligence*, (2000).

[5] R. J. Firby, *Adaptive Execution in Dynamic Domains*, Ph.D. dissertation, Yale University, 1989.

[6] R. J. Firby, 'Modularity issues in reactive planning', in *Proc. International Conference on AI Planning Systems (AIPS'96)*, (1996).

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[8] M. P. Georgeff and A. L. Lansky, 'Procedural knowledge', in *Proc. IEEE Special Issue on Knowledge Representation*, volume 74, pp. 1383 – 1398, (1986).

[9] L. P. Kaelbling and S. J. Rosenschein, 'Action and planning in embedded agents', in *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, ed., P. Maes, MIT Press, (1990).

[10] K. Konolige, K. Myers, E. Ruspini, and A. Saffiotti, 'The Saphira architecture: A design for autonomy', *Journal of Experimental and Theoretical AI*, (1996).

[11] M. Lin, *Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective*, Ph.D. dissertation, Linköpings universitet, 1999.

[12] T. Mowbray and R. Malveau, *CORBA Design Patterns*, Wiley, 1997.

[13] N. Muscettola, P. Nayak, B. Pell, and B. C. Williams, 'Remote agent: To boldly go where no AI system has gone before', *Artificial Intelligence*, **103**, (1998).

[14] D.J. Musliner, E.H. Durfee, and K.G. Shin, 'World modeling for the dynamic construction of real-time control plans', *Artificial Intelligence*, **74**(1), 83 – 127, (1995).

[15] A. Saffiotti, K. Konolige, and E. H. Ruspini, 'A multivalued logic approach to integrating planning and control', *Artificial Intelligence*, **76**, 481 – 526, (1995).

[16] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

# Paper Session II

August 23, 12:00 - 13:30

- **Decision Theoretic Planning for Playing Table Soccer**, M. Tacke, T. Weigel, B. Nebel
- **On-line Decision Theoretic Golog for Unpredictable Domains**, A. Ferrein, C. Fritz, G. Lakemeyer
- **Learning Partially Observable Action Models**, E. Amir

*

# Decision-Theoretic Planning for Playing Table Soccer

**Moritz Tacke, Thilo Weigel and Bernhard Nebel**
**Institut für Informatik**
**Universität Freiburg**
**79110 Freiburg, Germany**
***take,weigel,nebel*@informatik.uni-freiburg.de**

**Abstract.** Table soccer (also called "foosball") is much simpler than real soccer. Nevertheless, one faces the same challenges as in all other robotics domains. Sensors are noisy, actions must be selected under time pressure and the execution of actions is often less than perfect. One approach to solve the action selection problem in such a context is decision-theoretic planning, i.e., identifying the action that gives the maximum expected utility. In this paper we present a decision-theoretic planning system suited for controlling the behavior of a table soccer robot. The system employs forward-simulation for estimating the expected utility of alternative action sequences. As demonstrated in experiments, this system outperforms a purely reactive approach in simulation. However, this superiority of the approach did not extend to the the real soccer table.

## 1 Introduction

Playing *table soccer* (also called "foosball") is a task that is much simpler than playing real soccer. Nevertheless, one is faced with all the challenges one usually has to deal with in robotics. One has to interpret sensor signals and to select actions based on this interpretation. All this has to be done while keeping in mind that the sensor signals are noisy and the actuators are less than perfect.

One approach to solve the *action selection* problem is to use purely *reactive* methods. These are methods that select actions based on the current sensor input with only a minimum amount of computation. The selection of actions can be based on layered finite state automatons [2] or even simpler by using a simple decision tree. However, these purely reactive approaches have the disadvantage that they cannot anticipate changes in the environment caused by its own actions or by exogenous actions and for this reason might act sub-optimally.

Approaches such *behaviour networks* [3, 6] address this problem by modeling all possible actions, their consequences, and something similar to success likelihood. Based on this model, actions that promises to achieve the goals best are selected. As demonstrated by different robotic soccer teams [4, 7], this approach can be quite successful. However, it lacks theoretical foundation and, in fact, it is not clear under what circumstances the approach provably achieves its goals.

*Decision-theoretic planning* in contrast addresses the action selection problem by explicit deliberation about possible actions and aims at generating plans which promise to yield the *maximum expected utility* for an agent. This is achieved by explicitly considering the uncertain effects of the actions, the incomplete knowledge about the world and the possibly limited resources for carrying out a plan.

Decision theoretic planning can be implemented in various fashions. Using a classical refinement planner, it is possible to calculate the plan with the maximum expected utility by keeping ranges of possible utility values for partial plans [5]. A very popular way to realize decision-theoretic planning is the modeling of the planning problem as a Markov decision process [1].

The main challenge in using such an approach is to simplify the model of the domain such that the computational costs are not prohibitive. For this reason, we do not consider e.g. all possible ways an action can fail, but distinguish only between successful execution and failure. Furthermore, we do not consider responses to successful ball interceptions but consider the plan as failed once the opponent has intercepted the ball. Finally, planning is carried out only to a limited depth and the utility of the resulting state is assessed using a heuristic measure.

The rest of the paper is structured as follows. In Section 2, the KiRo system is presented. Section 3 describes the implementation of a decision theoretic planning algorithm for KiRo. Experimental results are presented in Section 4 and a short conclusion and outlook is given in Section 5.

## 2 KiRo

KiRo is a table soccer robot, i.e., an automated table soccer table [8]. Its hardware consists of the following components (see Figure 1):

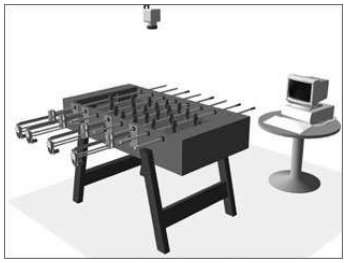


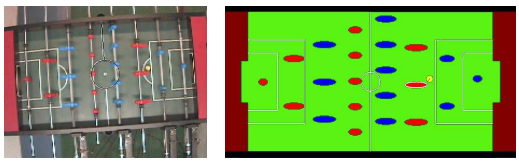
**Figure 1.** The hardware setup

- a standard table soccer table, where all rods of one player are equipped with electro motors strong enough to shift and turn the

rods fast,
- an overhead camera, and
- a standard PC, on which the control software runs.

The software executes a control cycle, consisting of the following four steps:

1. During the *vision analysis phase*, the positions of the various items on the field are estimated (see Figure 2(a)).
2. These positions are combined with knowledge about former ones in order to build the new *world model*. The world model, shown in Figure 2(b), contains information about the positions and movements of all items on field. The field is represented by a coordinate system where the origin is in the middle of the field and the $x$-axis connects both goals.
3. Based on this world model, the best actions are chosen in the *action selection* phase.
4. *Action execution* translates the chosen actions into steering commands. These commands are sent to the actuators.



(a) (b)

**Figure 2.** (a) The camera picture and (b) the generated world state

Since table soccer is a fast-paced game, the cycle duration has to be as short as possible in order to be able to react in time. KiRo works with a cycle time of 20 msec, which leads to strict bounds for the time available to select appropriate actions.

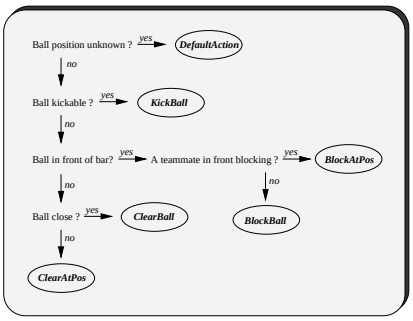



**Figure 3.** The original action selection procedure

The first approach to action selection has been a purely reactive decision tree. This essence of this approach is depicted in Figure 3. Although very crude, this approach was able to beat 75% of a random sample of human opponents [8].

In contrast, the system presented in this paper selects the action that promises the best consequences. To identify this action, it is necessary to plan ahead and to simulate the change in the world state caused by the different actions. As this approach needed a different kind of action model, the action control had to be completely rewritten.

## 3 Decision-Theoretic Planning for Table Soccer Playing

Table soccer does not seem to be well suited for decision-theoretic planning because it involves an opponent, which apparently means that we have to use game-solving methods, e.g., minimax search, instead of planning. However, the game is highly asymmetric. Only the player in possession of the ball, the *attacking player*, is able to decide on the future development of the game. The other player, the *defending player*, has to wait until the ball can be intercepted. For this reason, we can focus on a sub-game that is much easier to tackle.

We will consider the sub-game where the attacking player has continuous control of the ball. That means that this sub-game ends when the attacking player either scores a goal or looses control of the ball. This implies that the game tree can be pruned at nodes where the defending player intercepts the ball. Furthermore, all opponent actions that are unsuccessful in intercepting the ball do not influence the game at all. In summary, the defending player never needs to look ahead and just tries to intercept the current ball while the attacking player considers different possibilities of shooting the ball and takes the opponent into account only as a threat to the next action. So, we can indeed use decision-theoretic planning techniques to address the table soccer playing problem.

As a general strategy, KiRo uses a reactive positioning scheme when it is the defending player and employs decision-theoretic planning in the role of the attacking player. In what follows, we only consider the situation when KiRo is the attacking player.

### 3.1 Action and Forward Simulation

When planning, the attacking player can act all the time and the state changes continuously, because the ball is in motion most of time. Planning in such a setting is, of course, computationally infeasible. However, we do not have to consider all possible actions and all movements of the ball:

1. Most of the time, the movement of the ball and the rods is predictable. The ball moves according to its inertia, the rods move in the way which is specified by the employed actions.
2. The movement of the ball and of the rods are fully independent apart from one case: One figure touching the ball. In this situation, the movement of the ball is influenced.
3. Whenever the movement of the ball changes, the actions of the rods are likely to change as both players react on the new situation.

For this reason, we can interleave actions and physical simulations in a regular manner. Given a world state, an action as well as a reaction of the opponent, we simulate the evolution of the world until the point of time at which one of both players can manipulate the ball again. At this moment is it necessary to stop the simulation and to evaluate the reactions of both players in the new situation.

### 3.2 One Iteration of Planning

For a given world state $s$, in which KiRo is playing the ball, all possible consequences of KiRo's actions as well as all possible reactions of the opponent are considered. For each combination among these, a new world state is constructed which is used as a base for further

planning – provided that the opponent is not playing the ball in this state.

The search tree consists of three different kind of nodes which alternate in a given sequence. Each of theses types corresponds to a different planning step. The starting point of plan iteration is always a *state node* $\mathbf{s}$ corresponding to a game state $s$. The first step is to select a set of applicable actions $(a_1, \ldots, a_n)$. For each $a_i \in (a_1, \ldots, a_n)$ one *action node* $\mathbf{a}_i$ is created. As these nodes represent the different choices in state $s$, the utility value of the state node $\mathbf{s}$ is the maximum utility among its successors (once these are known):

$$utility(\mathbf{s}) = \max_{0<i\leq n} utility(\mathbf{a}_i).$$

Figure 4 illustrates the generation of the action nodes.

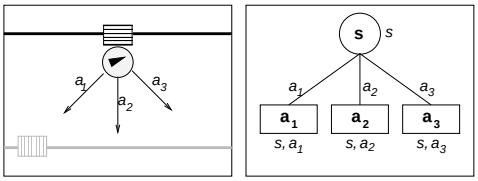



**Figure 4.** The action choice and its representation within the tree

The next step is the estimation of the opponent's reactions. Based on the world state $s$ a set $\{(o_1, p(o_1)), \ldots, (o_m, p(o_m))\}$ of hypotheses is created where the $o_j$ classify the reactions and the $p(o_j)$ the associated probabilities. Note that we assume that these reactions depend only on $s$ and are independent from the chosen action $\mathbf{a}_i$, which in fact is true in table soccer. There is usually no way an opponent can react to an the action of an attacking player.

For each action node $\mathbf{a}_i$, a set of successor *opponent nodes* $\{\mathbf{o}_{i1}, \ldots, \mathbf{o}_{im}\}$ is created. The value of $\mathbf{a}_i$ is the expected value over its successors:

$$utility(\mathbf{a}_i) = \sum_{j=1}^{m} p(o_j) \cdot utility(\mathbf{o}_{ij}).$$

In Figure 5 two possible reactions of the opponent and the formalization of this fact in the tree is depicted.

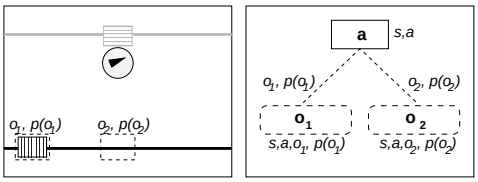



**Figure 5.** The formalization of the opponent

Every opponent node $\mathbf{o}_{ij}$ contains informations about the world state $s$ as well as one specific action $a_i$ and reaction $(o_j, p(o_j))$. To finish the planning step, the consequences $\{c_1, \ldots, c_q\}$ of $a_i$ are estimated along with their probabilities $p(c_1), \ldots, p(c_q)$. The now gathered information $\langle s, a_i, o_j, c_k \rangle$ is used to estimate new world states $s_{ijk}$ by means of a simulator. These states are used to build a new layer of *state nodes* $\mathbf{s}_{ijk}$.

The utility value of the precedent opponent node $\mathbf{o}_{ij}$ is calculated by

$$utility(\mathbf{o}_{ij}) = \sum_{k=1}^{q} p(c_k) \cdot utility(\mathbf{s}_{ijk}).$$

Figure 6 shows two different outcomes of an action and the use of the simulator to create a new world state based on the collected data.

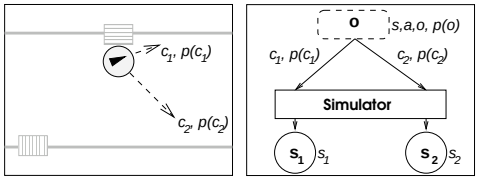



**Figure 6.** The possible outcomes of an action

After one iteration of planning, there exists a number of new state nodes containing the world state resulting from every possible action $a_i$, every possible reaction $o_j$ and every possible consequence $c_k$ of the action $a_i$. After the search tree has been built up to the leafs, those get evaluated using the utility function. These evaluations propagate backward through the tree until the root state node $\mathbf{s}$ is reached. The utility of $\mathbf{s}$ is the maximum expected utility among all selectable actions in world state $s$; the action $a_i$ yielding this value is the one to be selected in $s$.

## 3.3 Choosing an Applicable Action

Currently, KiRo's capabilities comprise the following actions for operating each of the four rods under his control:

- *KickBall*: Rotate the rod by 90° in order to kick the ball forward or diagonally to the left or right.
- *BlockBall*: Move the rod so that a figure intercepts the ball.
- *ClearBall*: Move to the same position as *BlockBall* but turn the rod to let the ball pass from behind.
- *StopBall*: Pen the ball in between figure and field.

Altogether, it is possible to assign a single rod one out of 6 different actions (three of them being kicks in different directions). Taking all four rods into account is it possible to create 24 different assignments of actions. In other words, our search tree would have a branching factor of 24. Fortunately, one can easily reduce this factor because only the rod close the ball can kick and the others have a choice between the remaining three actions. We decided to assign statically *BlockBall* to all rods between the ball and the own goal in order to have a defence even when the ball is accidentally lost or reflected. Rods between the opponent's goal and the ball should not handicap the chances to score a goal. For that reason, they are assigned *ClearBall*.

With these static assignments, the branching factor is reduced to six while KiRo is playing the ball. If the opponent possesses the ball, a static defensive allocation of actions without any planning is employed. This reflects the already mentioned observation that planning in these situations is useless.

In some situations, performing a certain action might be useless (e.g. trying to stop an already stopped ball) or the chance of a successful carrying out of an action might become too low. In these

cases, such actions are not evaluated in order to reduce the complexity further.

As the means the opponent is going to use in order to protect his goal are unknown, it is necessary to guess what his reactions will be. Each of these guesses is weighted by the probability that this reaction will occur. Currently, the opponent is always expected to either let his rods unmoved or to protect his goal according to the scheme applied in the *BlockBall*-action. Each of these alternatives is weighted with a probability of 0.5. These probabilities are, of course, only a crude approximation and it is planned to replace this simple opponent model by a more sophisticated, experience-based one.

### 3.4 Calculating the Successor State

The last – and most important – step is to estimate the probability of a success of KiRo's actions and to create a new world state based on the data collected yet. The actions of KiRo can have two possible outcomes, success and failure. The probabilities of these depend on the encountered world state, e.g. a quickly moving ball is more difficult to kick than a static one. Based on the world state and on data about the effectiveness of the actions collected during earlier games, the success probability is estimated.

The means to create a new world state based on the informations about KiRo's actions, their success or failure and the opponent's reactions is a simple simulator. This simulator models the world based on the following principles:

- The angle of incidence and the angle of reflection are equal.
- Friction is ignored.

Additionally, the simulator has a model that allows to interprete the steering commands issued by both players. Of course, the real world is only very coarsely modeled by a simulator based on these principles. Further, since the input data is imperfect, the simulation is accumulating errors. The simulated span of time, however, is very short, so that the errors are still acceptable.

Simulation is performed in two steps: In the first one, the game is simulated until the point of time in which KiRo's action takes place. Afterwards, two successor states are generated: One of the resulting world states is a state according to the known consequences of the success of KiRo's action. The other state reflects the failure of the action. In this case, we have a new problem: The consequences of a successful action are known – in case of failure, anything can happen. The most frequent kind of failure is the inability of KiRo to hit the ball. For this reason, the failing case is simulated by letting the ball pass the failing rod without changing its movement parameters. The resulting world state is not used as a starting point for a new planning step; it is directly evaluated using the heuristic utility function (see Subsection 3.6).

### 3.5 Estimating the Success Probabilities

In order to estimate the success probability for a given action on a certain rod in a given world state, a Bayesian approach is employed. The first step is to classify the ball movements by a 4-tuple $\langle d_x, d_y, v_x, v_y \rangle$, where

- $d_x$ is the distance in $x$-direction to the rod.
- $d_y$ is the minimal distance in y-direction to a figure on this rod .
- $v_x$ is the relative velocity in $x$-direction. "Relative" means in this context e.g. "approaching" or "departing"
- $v_y$ is the relative velocity in $y$-direction.

Each of these values is discretized according to a seven step scale.

The task is now to calculate the success probability $P(S|d_x, d_y, v_x, v_y)$ of the action given this tuple. Using Bayes' rule yields

$$P(S|d_x, d_y, v_x, v_y) = \frac{P(d_x, d_y, v_x, v_y|S) \cdot P(S)}{P(d_x, d_y, v_x, v_y)}$$

In order to simplify the computation of the conditional probability, two independence assumptions are made:

1. $P(d_x, d_y, v_x, v_y) = P(d_x) \cdot P(d_y) \cdot P(v_x) \cdot P(v_y)$
2. $P(d_x, d_y, v_x, v_y|S) = P(d_x|S) \cdot P(d_y|S) \cdot P(v_x|S) \cdot P(v_y|S)$.

This "naive Bayes" assumption is clearly not met; this, however, is usually the case in naive Bayes approaches (otherwise they would not be called "naive"). Using this assumption, we are able to give an easy to compute estimate for the success probability:

$$P_{NB}(S|d_x, d_y, v_x, v_y) \hat{=} \frac{P(d_x|S) \cdot P(d_y|S) \cdot P(v_x|S) \cdot P(v_y|S) \cdot P(S)}{P(d_x) \cdot P(d_y) \cdot P(v_x) \cdot P(v_y)}$$

### 3.6 The Utility Function

Table soccer poses a highly dynamic environment where only little time is available for selecting the most appropriate action. Due to the high uncertainties in sensing and acting, it is infeasible to create and carry out a complete plan for reaching the final aim of scoring a goal. It is necessary to plan with a limited horizon and to use an *utility function* for evaluating world states. Planning in this fashion is similar to depth-limited minimax search with a heuristic evaluation of the leafs of the game tree. The utility of inner nodes is estimated using a *rollback procedure* [1].

The heuristic utility function is used to estimate the world states contained in the leafs of the search tree – provided one has not reached a scored goal yet. The principles underlying this function are:

- If a goal is either scored or going to be scored (i.e. ball behind the keeper, moving towards the goal), a value of 100 is returned if it is the opponent's goal, otherwise 0.
- The closer the ball is to the opponent's goal wall, the better.
- If the distance between the ball and both front walls is equal, it is neither important who controls the ball nor whether the ball is on the left or right of the field. The closer it gets to one of the walls, the bigger the importance of these facts.

## 4 Results

The decision-theoretic planner has been fully implemented in the KiRo system and tested using a simulator and the real table soccer system.

### 4.1 Computational Costs

The implemented system runs on a 1.7 GHz AMD processor. On this processor, the vision analysis phase, the world modeling step and action execution (see Section 2) require together roughly 5 msec. With a cycle time of 20 msec, this gives us approximately 15 msec per cycle for planning.

Table 1 shows the worst case runtimes for different search depths. The search depth is in this case defined as the number of plan iterations. One planning step constructs a subtree of the depth 3; a

search depth of $x$ therefore corresponds to a tree depth of $3x$. The table shows that search depth values over 3 are clearly not feasible for KiRo with the current processor speeds. However, with newer, faster CPUs we might even be able to go to search depth of 3.

| Search depth | Runtime |
|---|---|
| 1 | 0.3 msec |
| 2 | 10 msec |
| 3 | 23 msec |
| 4 | 45 msec |
| 5 | 80 msec |

**Table 1.** Worst case runtimes

## 4.2 Performance Experiments

Two kinds of performance experiments have been conducted. The simulator has been used to compare the reactive and the decision-theoretic planning action selection directly by playing against each other. On the real table, games against human adversaries were performed to test both approaches indirectly.

## 4.3 Results on the Simulator

Two kinds of experiments were conducted on the simulator: In a first run, the decision-theoretic planning procedure had a fixed search depth of two. Using this setting, a number of games has been played. While the program using the reactive action selection scheme shot in average one goal in 10 minutes, the decision-theoretic approach scored once in 1.5 minutes.

As many of these goals were own goals by the reactive control system, a second criteria was employed: Field superiority. In this context, a team is called field superior if it is capable of keeping the ball in the opponent's half most of the time. The field superiority value of a team is therefore the percentage of time during which the ball was in the opponent's half of the field. Table 2 shows the results for the decision-theoretic planning approach, ordered by the employed search depth.

| Search depth | Field superiority value |
|---|---|
| 1 | 64% |
| 2 | 72% |
| 3 | 74% |
| 4 | 57% |

**Table 2.** Field superiority values for the decision-theoretic planning approach

The decision theoretic planning approach is field superior. The field superiority increases with the search depth until the efficiency gets decreased due to the excessive time consumption.

## 4.4 Results on Physical Table Soccer System

The good results from the simulation experiments could not be replicated on the real table soccer system. Since we do not have a table with robot control for both sides, we had to conduct the tests indirectly by playing against humans. The setting of these experiments was as follows. Every opponent team consisted of two players. The teams were not allowed to switch their positions, and every team had to play four matches against the robot. During two of these, the reactive action selection was controlling the robot; the other two matches were performed by the planning approach. The order of the matches was randomly drawn and the human opponents did not know against which action selection they were playing.

In 56 Matches, the reactive approach was on average able to shoot a goal in 0.6 minutes, while it took the human opponents 1.56 minutes to score. The planning approach hit the goal once in 0.84 minutes and admitted goals by the human teams on average every 1.19 minutes.

## 5 Conclusion

We presented a decision-theoretic planning approach to play table soccer. The presented approach used a forward-simulation scheme as well as an opponent model and a naive Bayesian approach to estimate the success probabilities of its own actions.

This approach was able to dominate a reactive action selection mechanism in direct matches performed on a simulator, but proved to be inferior in indirect comparisons playing on the real table soccer table against human teams. While the result on the real table soccer table is disappointing, the simulation results have shown that the approach has potential. In particular, there are a number of parameters that appear to be worthwhile to be experimented with. The opponent model, for example, is currently very simple and could, e.g., be trained by recording real games. Furthermore, the success probability should also be adapted to the real table. Finally, the execution of actions themselves might be able to be enhanced. Summarizing, the decision-theoretic planning approach has shown promise but still has to live up to its expectations.

## REFERENCES

[1] C. Boutilier, T. Dean, and S. Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1), 1986.

[3] K. Dorer. Behavior networks for continuous domains using situation-dependent motivations. In *Proc. 16th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1233–1238, Stockholm, Sweden, 1999.

[4] K. Dorer. The magmaFreiburg Soccer Team. In M. Veloso, E. Pagello, and H. Kitano, editors, *RoboCup-99: Robot Soccer World Cup III*, Lecture Notes in Artifi cial Intelligence, pages 600–603. Springer-Verlag, Berlin, Heidelberg, New York, 2000.

[5] P. Haddawy and M. Suwandi. Decision-theoretic refi nement planning using inheritance abstraction. In K. Hammond, editor, *Proc. Second International Conference on Artificial Intelligence*. University of Chicago, Illinois, AAAI Press, 1994.

[6] P. Maes. Situated agents can have goals. In P. Maes, editor, *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 49–70. MIT Press, Cambridge, MA, 1990.

[7] T. Weigel, J.-S. Gutmann, A. Kleiner, M. Dietl, and B. Nebel. CS-Freiburg: Coordinating Robots for Successful Soccer Playing. *IEEE Transactions on Robotics and Automation*, 18(5):685–699, October 2002.

[8] T. Weigel and B. Nebel. KiRo – An Autonomous Table Soccer Player. In *Proc. Int. RoboCup Symposium '02*, pages 119 – 127. Springer-Verlag, Fukuoka, Japan, 2002.

# On-line Decision-Theoretic Golog for Unpredictable Domains

**Alexander Ferrein** and **Christian Fritz** and **Gerhard Lakemeyer** [1]

**Abstract.** DTGolog was proposed by Boutilier et al. as an integration of decision-theoretic (DT) planning and the programming language Golog. Advantages include the ability to handle large state spaces and to limit the search space during planning with explicit programming. Soutchanski developed a version of DTGolog, where a program is executed on-line and DT planning can be applied to parts of a program only. One of the limitations is that DT planning generally cannot be applied to programs containing sensing actions. In order to deal with robotic scenarios in unpredictable domains, where certain kinds of sensing like measuring one's own position are ubiquitous, we propose a strategy where sensing during deliberation is replaced by suitable models like computed trajectories so that DT planning remains applicable. In the paper we discuss the necessary changes to DTGolog entailed by this strategy and an application of our approach in the ROBOCUP domain.

## 1 Introduction

Boutilier et al (2001) proposed DTGolog, an integration of Markov Decision Processes (MDPs) [12] and the programming language Golog [8], which is based on Reiter's variant of the situation calculus [13]. Golog is equipped with familiar control structures like sequence and while-loops, but also nondeterminism, which allow for complex combinations of actions operating on fluents (predicates and functions changing over time). DTGolog extends Golog by adding familiar MDP notions like stochastic actions and rewards. Moreover, decision-theoretic planning is incorporated in the form of an MDP-style optimization method, which takes a program $\rho$ and computes a policy (another program), which follows the controls of $\rho$ except that it chooses among nondeterministic actions in order to maximize expected utility up to a given horizon of actions. The advantage over traditional MDP's is that the state space need not be represented explicitly and that the search space can be narrowed effectively by Golog's control structures.

One serious limitation of DTGolog is that it does not account for sensing actions.[2] The reason for this limitation is that DTGolog operates in an off-line modus, that is, it computes a policy for the whole program, which is then handed to an execution module. When the program contains actions sensing fluents that can take on a large, perhaps infinite number of values, finding a policy quickly becomes infeasible, if not impossible. For this reason Soutchanski [15] introduced an on-line version of DTGolog, which interleaves policy optimization and execution. The main idea is that a user can specify for which parts of the program an MDP-style policy is to be computed. As an example, consider the program $optimize(\rho_1); sense(\phi)$; if $\phi$ then $\rho_2$ else $\rho_3$. The idea is, roughly, that first a policy is computed for the subprogram $\rho_1$, which is then executed, followed by an action sensing the truth value of $\phi$. Finally, depending on the outcome either $\rho_2$ or $\rho_3$ is executed, both of which may themselves contain further occurrences of $optimize$.

In order to see that Soutchanski's approach is problematic for decision making in highly dynamic domains, it is useful to distinguish two very different forms of sensing, which we refer to as *active* and *passive* sensing. An example of active sensing is an automatic taxi driver asking a customer for her destination. Typically, this form of sensing happens only occasionally and should be part of the robot's control program. An example of passive sensing is keeping track of one's own position, which happens frequently, often in the order of tens of milliseconds. It would make little sense to explicitly represent such passive sensing actions in the robot's control program, for these would make up the bulk of the program and render reasoning about the program all but impossible. While Soutchanski does not say so explicitly, he clearly is concerned only with active sensing actions, as all his sensing actions are part of the control program.

In highly dynamic domains, passive sensing is ubiquitous as a robot has to constantly monitor its own position and its environment. The aim of this paper is to show how decision-theoretic planning can be adapted to account for this form of sensing. The starting point for our investigations is the work by Grosskreutz and Lakemeyer [5], who integrated passive sensing into Golog. The idea is, roughly, that when reasoning about a program (e.g. projecting its outcome) one uses *models* of how fluents like the robot's position change. (To model the movement of a robot they use simple linear functions of time to approximate the robot's trajectories.) During actual execution these models are replaced by passive sensing actions which are represented as so-called exogenous actions, which periodically update fluents like the position of the robot and which are inserted by the interpreter of the program.

Assuming we have appropriate models of how the relevant fluents change during deliberation, could we then simply adopt Soutchanski's approach or even the original DTGolog if we ignore active sensing? The answer, in short, is No. What is missing in both cases is that, after a policy has been computed, its execution must be carefully *monitored.* This is because the model of the world used during deliberation is only a rough approximation of the real world and things may very well turn out differently and may even result in aborting the current policy. For example, when a driver initiates passing a car and another vehicle suddenly appears speeding from behind, it may be advisable to let the other car pass first. Monitoring then means to compare assumptions made by the model of the world (such as com-

[1] Computer Science Department, Knowledge-Based Systems Group RWTH Aachen, D-52056 Aachen {gerhard,fritz, ferrein}@cs.rwth-aachen.de

[2] The only exception are sensing actions which are introduced by the optimizer to determine the state after a stochastic action.

puted agent trajectories) with the actual values obtained by sensing during execution. As we will see, this can be achieved by annotating the policy with appropriate information.

Given that we are motivated by robots operating in highly dynamic and unpredictable domains, deliberation and decision making should happen quickly, preferably in less than a second. For arbitrary Golog programs this clearly cannot be guaranteed.[3] Here we are concerned with control programs for robots that operate continuously over longer periods of time. In such scenarios it makes little sense to find optimal policies for the robot's actions from start to finish, since it is impossible to predict what the world will be like after even a few seconds. Instead one is content to peek into the future to plan perhaps only a handful of actions with highest utility, like passing another car. As we will demonstrate at the end of the paper, under these assumptions, efficient decision-theoretic planning is achievable and can lead to overall good performance.

Since an application like an automatic taxi driver is currently still out of reach, we have chosen robotic soccer, in particular, the ROBOCUP MIDDLE SIZE LEAGUE as a benchmark. While the environment is still fairly controlled (a fixed playing field with four mobile robots on each team), game situations are nevertheless challenging due to their dynamics and unpredictability. To keep things simple, we only consider the case of passive sensors, that is, Golog programs as supplied by a user do not contain explicit sensing actions.

In related work, Poole [11] incorporates a form of decision-theoretic planning into his independent choice logic. While he also distinguishes passive from active sensing, he does not consider the issue of on-line DT planning. Other action logics addressing uncertainty include [14], where abduction is the focus, and [6], which addresses symbolic dynamic programming and which itself is based on [1]. Finally, [7] also discuss ways of replacing sensing by models of the environment during deliberation.

The rest of the paper is organized as follows. First, we give a brief overview of DTGolog and its underlying semantics. Then we sketch out our approach to on-line decision-theoretic planning, followed by a discussion of applying decision-theoretic to ROBOCUP's MIDDLE SIZE LEAGUE and some concluding remarks.

## 2 The Situation Calculus and DTGolog

### 2.1 The Situation Calculus

Golog is based on Reiter's variant of the Situation Calculus [13, 10], a second-order language for reasoning about actions and their effects. Changes in the world are only due to actions so that a situation is completely described by the history of actions since the initial situation $S_0$. Properties of the world are described by *fluents*, which are predicates and functions with a situation term as their last argument. For each fluent the user defines a successor state axiom describing precisely when a fluent value changes or does not change after performing an action. These, together with precondition axioms for each action, axioms for the initial situation, and foundational axioms as well as unique names and domain closure assumption, form a so-called *basic action theory* [13].

### 2.2 Off-line DTGolog

DTGolog uses basic action theories to give meaning to primitive actions and it inherits Golog's programming constructs such as sequence, if-then-else, while-loops, and procedures, as well as nondeterministic actions. From MDPs DTGolog borrows the notion of *reward*, which is a real number assigned to situations indicating the desirability of reaching that situation, and *stochastic actions*. To see what is behind the latter, consider the action of intercepting a ball in robotic soccer. Such an action routinely fails and we assign a low probability (0.2) to its success. To model this in DTGolog, we define a stochastic action $intercept$. It is associated with two non-stochastic or deterministic actions $interceptS$ and $interceptF$ for a successful and failed intercept, respectively. Instead of executing $intercept$ directly, nature chooses to execute $interceptS$ with probability $0.2$ and $interceptF$ with probability $0.8$. The effect of $interceptS$ can be as simple as setting the robot's position to the position of the ball. The effect of $interceptF$ can be to teleport the ball to some arbitrary other position and setting the robot's position to the old ball position.[4]

While the original Golog merely looks for any sequence of primitive actions that corresponds to a successful execution of a program, DTGolog takes a program and converts it into another simplified program, called a policy, which is a tree of conditional actions. This policy, roughly, follows the advice of the original program in case of deterministic actions and settles on those choices among nondeterministic actions which maximize expected utility. The search for the right choices is very similar to the search for an optimal policy in an MDP. One advantage of using Golog compared to a regular MDP is that the search can be arbitrarily constrained by restricting the number of nondeterministic actions.

DTGolog is defined in terms of a macro $BestDo(p, s, h, \pi, v, pr)$, which ultimately translates into a situation calculus expression. Given a program $p$ and a starting situation $s$, $BestDo$ computes a policy $\pi$ with expected utility $v$ and probability $pr$ for a successful execution. $h$ denotes a finite horizon, which provides a bound on the maximal depth of any branch in the policy. For space reasons we only consider the definition of $BestDo$ for nondeterministic choice and stochastic actions. (See [2] for more details.)

Suppose a program starts with a nondeterministic choice between two programs $p_1$ and $p_2$, written as $(p_1|p_2)$. Then

$$\begin{aligned} &BestDo((p_1|p_2); p, s, h, \pi, v, pr) \stackrel{def}{=} \\ &\quad \exists \pi_1, v_1, pr_1. BestDo(p_1; p, s, h, \pi_1, v_1, pr_1) \wedge \\ &\quad \exists \pi_2, v_2, pr_2. BestDo(p_2; p, s, h, \pi_2, v_2, pr_2) \wedge \\ &\quad ((v_1, p_1) \geq (v_2, p_2) \wedge \pi = \pi_1 \wedge pr = pr_1 \wedge v = v_1) \vee \\ &\quad (v_1, p_1) < (v_2, p_2) \wedge \pi = \pi_2 \wedge pr = pr_2 \wedge v = v_2) \end{aligned}$$

Here $BestDo$ commits the policy to the best choice among the two alternatives, where "best" is defined in terms of a multi-objective optimization of expected value and success probability. See [2] for an example of how $(v_i, p_i) \geq (v_j, p_j)$ can be defined.

Now suppose that $a$ is a stochastic action with nature's choices $n_1, n_2, \ldots, n_k$.

$$\begin{aligned} &BestDo(a; p, s, h, \pi, v, pr) \stackrel{def}{=} \\ &\quad \exists \pi'. BestDoAux(\{n_1, \ldots, n_k\}, p, s, h, \pi', v, pr) \\ &\quad \pi = a; senseEffect(a); \pi'. \end{aligned}$$

Here the policy is $a; senseEffect(a); \pi'$ where $\pi'$ is computed by $BestDoAux$ below. The action $senseEffect(a)$ is inserted in order to maintain the MDP assumption of full observability. Its job

[3] In the coffee-delivery example in [2], the robot needed several seconds or even minutes to find a policy.

[4] While this model is certainly simplistic, it suffices in most real game situations, since all that matters is that the ball is not in the robot's possession after a failed intercept.

is to make sure that after performing $a$ the robot gathers enough information to distinguish between the outcomes $n_i$. In the case of $intercept$, the sensing would involve finding out whether the robot has the ball denoted by the fluent $haveBall(s)$.

$$BestDoAux(\{n_1,\ldots,n_k\},p,s,h,\pi,v,pr) \stackrel{def}{=}$$
$$\neg Poss(n_1,s) \wedge BestDoAux(\{n_2,\ldots,n_k\},p,s,h,\pi,v,pr) \vee$$
$$Poss(n_1,s) \wedge$$
$$\exists \pi', v', pr'.BestDoAux(\{n_2,\ldots,n_k\},p,s,h,\pi',v',pr') \wedge$$
$$\exists \pi_1, v_1, pr_1.BestDo(n_1, do(n_1,s), h-1, \pi_1, v_1, pr_1) \wedge$$
$$\pi = if(\varphi_1, \pi_1, \pi') \wedge$$
$$v = v' + v_1 \cdot prob(n_1,s) \wedge$$
$$pr = pr' + pr_1 \cdot prob(n_1,s)$$

$$BestDoAux(\{\},p,s,h,\pi,v,pr) \stackrel{def}{=} \pi = Stop \wedge v = 0 \wedge pr = 0$$

Note that $BestDoAux$ produces a policy of the form $if(\varphi_1, \pi_1, if(\varphi_2, \pi_1, \ldots))$ accounting for all outcomes of nature's choices. The $\varphi_i$ are user-defined tests which allow the robot to distinguish between them. In our intercept example, these could be $haveBall(s)$ for $interceptS$ and $\neg haveBall(s)$ for $interceptF$. A policy contains $Stop$ if in the respective branch no further actions can be executed, i.e. an action was not possible.

## 2.3 On-line DTGolog

The original version of DTGolog, which we just described, operates in an off-line modus, that is, it first computes a policy for the whole program and only then initiates execution. As was observed already in [4], this is not practical for large programs and certainly not for applications with tight real-time constraints such as ROBOCUP. In the extreme one would only want to reason about the next action of a program, execute it and then continue with the rest of the program. This is the basic idea of an on-line interpretation of a Golog program [4]. To make this work, a so-called transition semantics is needed, which takes a configuration consisting of a program and a situation and turns it into another configuration. Formally, one introduces a predicate $Trans(\delta, s, \delta', s')$, which first appeared in [3], expressing a possible transition of program $\delta$ in situation $s$ to the program $\delta'$ leading to situation $s'$ by performing an action. For space reasons we only consider the case of while-loops.

$$Trans(while(\varphi,p),s,\delta',s') \equiv$$
$$\exists \delta''.Trans(p,s,\delta'',s') \wedge \varphi[s] \wedge \delta' = \delta''; while(\varphi,p)^5$$

Given such definitions for all constructs, the execution of a complete program can be defined in terms of the reflexive and transitive closure of $Trans$.[6]

A nice feature of on-line interpretation is that the step-wise execution of a program can easily be interleaved with other exogenous actions or events, which are supplied from outside. This is how we handle periodic sensor updates for position estimation, for example. (See [5] for details of how this can be done in Golog.)

With the basic transition mechanism in hand, it is, in principle, not hard to reintroduce off-line reasoning for parts of the program. In the case of DTGolog, Soutchanski proposed for that purpose an interleaving of off-line planning and on-line execution. We show an excerpt of his interpreter implemented in Prolog. We only consider the case of executing deterministic and sensing actions, leaving out stochastic actions:

```
online(E,S,H,Pol,U) :-
  incrBestDo(E, S, ER, H, Pol1, U1, Prob1),
  ( final(ER, S, H, Pol1, U1), Pol=Pol1, U=U1  ;
    reward(R, S), Pol1 = (A : Rest),
      %% deterministic action
    ( agentAction(A), doReally(A), !,
      online(ER, do(A,S), H, PolFut, UFut),
      Pol = (A : PolFut), U is R + UFut ;
      %% sensing action
      senseAction(A), doReally(A), !,
      online(ER, do(A,S), H, PolFut, UFut),
      Pol=(A: PolFut), U is R + UFut ;
       ...
    )
  ).
```

Roughly, the interpreter $online$ calculates a policy $\pi$ for a given program $e$ up to a given horizon $h$, executes its first action ($doReally(a)$) and recursively calls the interpreter with the remaining program again.

To control the search while optimizing Soutchanski proposes an operator $optimize$ defined by the following macro:

$$IncrBestDo(optimize(p_1);p_2,s,p_r,h,\pi,u,pr) \stackrel{def}{=}$$
$$\exists p'.IncrBestDo(p_1;Nil,s,p',h,\pi,u,pr) \wedge$$
$$(p' \neq Nil \wedge p_r = (optimize(p');p_2) \vee$$
$$p' = Nil \wedge p_r = p_2).$$

This has the effect that $p_1$ is optimized and the resulting policy is executed before $p_2$ is even considered. As mentioned already in the introduction, one advantage is that a user can deal with explicit (active) sensing actions by restricting $optimize$ to never go beyond the next sensing action.

Nevertheless the approach has a number of shortcomings. First note that, in the definition of the interpreter $online$, after executing only one action of a computed policy, the optimizer is called again. This means that large parts of the program are re-optimized over and over again, which is computationally too expensive for real-time decision making. Also note that it is only checked during the optimization phase whether an action is executable. Hence the interpreter ignores the possibility that an action, which was possible at planning time, becomes impossible to execute due to changes in the environment.

While there are easy fixes to these drawbacks, the fundamental problem of this approach is that it is not possible to do optimization ahead of sensing actions. Consider the program $p_1; sense(\varphi); if\ \varphi\ then\ p_2\ else\ p_3$. The condition must be evaluated before one is able to decide whether to execute $p_2$ or $p_3$, i.e. the sensing action must be executed online to evaluate the value of $\varphi$. The forementioned operator $optimize$ allows for limiting the search for an optimal policy for that case. Using $optimze$ the program can be rewritten as $optimize(p_1); (sense(\varphi)); if\ \varphi\ then\ p_2\ else\ p_3$. This program instructs the interpreter to first optimize $p_1$ without regarding the rest of the program. Then, the sensing action is executed to get the needed value from the environment. Afterwards, the conditional can be optimized and executed, resp.

In real-time domains sensor updates arrive with high frequency. The proposed active sensing in Soutchanski's interpreter is not feasible as is renders the applicability of the decision-theoretic planning approach impossible.

We propose a different kind of online interpreting decision-theoretic plans in Golog which is feasible for real-time domains. Our approach differs mainly in that we use passive sensing instead of the active sensing proposed by Soutchanski. We therefore do not have

[5] $\varphi[s]$ denotes the situation calculus formula obtained from $\varphi$ by restoring situation variable $s$ as the suppressed situation argument for all fluent names mentioned in $\varphi$. Also note that free variables are universally quantified in the following formulas.

[6] One also needs the notion of a final configuration, an issue we ignore here for simplicity.

any restrictions with sensing actions. Instead we use models of the world in the planning phase. To be able to validate if the model assumptions hold while executing a plan we annotate the policies in a special way desribed in Section 3.1. In Section 3.2 we show how annotated policies are executed and how invalid policies are detected. We deployed our approach in the RoboCup domain and show some of the results in Section 4.

## 3 On-line DTGolog for Passive Sensing

As the re-optimization of a remaining program is generally not feasible in real-time environments, our first modification of on-line DTGolog is to make sure that the whole policy and not just the first action is executed. For this purpose we introduce the following operator $solve(p, h)$ for a program $p$ and a fixed horizon $h$.

$$\begin{aligned} &Trans(solve(p,h), s, \delta', s') \equiv \\ &\quad \exists \pi, v, pr. BestDo(p, s, h, \pi, v, pr) \\ &\quad \wedge \delta' = applyPol(\pi) \wedge s' = s. \end{aligned}$$

The predicate $BestDo$ first calculates the policy for the whole program $p$. For now the reader may assume the definition of the previous section, but we will see below that it needs to be modified. This policy is then scheduled for execution as the remaining program. However, as discussed in the introduction, the policy is generated using an abstract model of the world to avoid sensing, and we need to monitor whether $\pi$ remains valid during execution. To allow for this special treatment, we use the special construct $applyPol$, whose definition is deferred until later.

### 3.1 Annotated Policies

In order to see why we need to modify the original definition of $BestDo$ and, for that matter, the one used by Soutchanski, we need to consider, in a little more detail, the idea of using a model of the world when planning vs. using sensor data during execution. The following fragment of the control program of our soccer robots might help to illustrate the problem:

$$\begin{aligned} &\mathbf{while}\ game\_on\ \mathbf{do} \ldots; \\ &\quad solve(\ldots; \\ &\qquad \mathbf{if}\ \exists x, y(ball\_pos(x,y) \wedge reachable(x,y)) \\ &\qquad \mathbf{then}\ intercept \\ &\qquad \mathbf{else} \ldots; \ldots, h) \\ &\mathbf{endwhile} \end{aligned}$$

While the game is still on, the robots execute a loop where they determine an optimal policy for the next few (typically less than six) actions, execute the policy and then continue the loop. One of the choices is intercepting the ball which requires that the ball is reachable, which can be defined as a clear trajectory between the robot and the ball. Now suppose $BestDo$ determines that the if-condition is true and that $intercept$ has the highest utility. In that case, since $intercept$ is a stochastic action, the resulting policy $\pi$ contains $\ldots intercept; senseEffect(intercept); \ldots$. Note, in particular, that the if-condition of the original program is not part of the policy. And this is where the problem lies. For during execution of the policy it may well be the case that the ball is no longer reachable because an opponent is blocking the way. In that case $intercept$ will fail and it makes sense to abort the policy and start planning for the next moves. For that, the if-condition should be re-evaluated using the most up-to-date information about the world provided by the sensors and compared to the old value. Hence we need to make sure that the if-condition and the old truth value are remembered in the policy.

In general, this means we need to modify the definition of $BestDo$ for those cases involving the evaluation of logical formulas. Here we consider if-then-else and test actions. While-loops are treated in a similar way.

$$\begin{aligned} &BestDo(if(\varphi, p_1, p_2); p, s, h, \pi, v, pr) \stackrel{def}{=} \\ &\quad \varphi[s] \wedge \exists \pi_1. BestDo(p_1; p, s, h, \pi_1, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, true); \pi_1 \vee \\ &\quad \neg\varphi[s] \wedge \exists \pi_2. BestDo(p_2; p, s, h, \pi_2, v, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, false); \pi_2 \end{aligned}$$

The only difference compared to the original $BestDo$ is that we prefix the generated policy with a marker $\mathfrak{M}(\varphi, true)$ in case the $\varphi$ turned out to be true in $s$ and $\mathfrak{M}(\varphi, false)$ if it is false. The treatment of a test action $?(\varphi)$ is even simpler, since only the case where $\varphi$ is true matters. If $\varphi$ is false, the current branch of the policy is terminated, which is indicated by the $Stop$ action.

$$\begin{aligned} &BestDo(?(\varphi); p, s, h, \pi, v, pr) \stackrel{def}{=} \\ &\quad \varphi[s] \wedge \exists \pi'. BestDo(p, s, h, \pi', v, pr) \wedge \\ &\quad \pi = \mathfrak{M}(\varphi, true); \pi' \vee \\ &\quad \neg\varphi[s] \wedge \pi = Stop \wedge pr = 0 \wedge v = reward(s) \end{aligned}$$

In the next subsection, we will see how our annotations will allow us to check at execution time whether the truth value of conditions in the program at planning time are still the same and what to do about it when they are not. Before that, however, it should be mentioned that explicit tests are not the only reason for a possible mismatch between planning and execution. To see that note that when a primitive action is entered into a policy, its executability has been determined by $BestDo$. Of course, it could happen that the same action is no longer possible at execution time. It turns out that this case can be handled without any special annotation.

### 3.2 Execution and Monitoring

Now that we have modified $BestDo$ so that we can discover problems at execution time, all that is left to do is to define the actual execution of a policy. Given our initial definition of $Trans(solve(p, h), s, \delta', s')$, this means that we need to define $Trans$ for the different cases of $applyPol(\pi)$. To keep the definitions simple, let us assume that every branch of a policy ends with $Stop$ or $nil$, where $nil$ represents the empty program.

$$\begin{aligned} Trans(applyPol(Nil), s, \delta', s') &\equiv s = s' \wedge \delta' = nil \\ Trans(applyPol(Stop), s, \delta', s') &\equiv s = s' \wedge \delta' = nil \end{aligned}$$

Given the fact that configurations with $nil$ as the program are always final, that is, execution may legally terminate, this simply means that nothing needs to be done after $Stop$ or $nil$.

In case a marker was inserted into the policy we have to check the test performed at planning time still yields the same result. If this is the case we are happy and continue executing the policy, that is, $applyPol$ remains in effect in the successor configuration. But what should we do if the test turns out different? We have chosen to simply

abort the policy, that is, the successor configuration has $nil$ as its program. While this may seem simplistic, it seems the right approach for applications like RoboCup. For consider the case of an intercept. If we find out that the path is blocked, the intercept will likely fail and all subsequent actions in the policy become meaningless. Moreover, a quick abort will enable immediate replanning according to the control program, which is not a bad idea under the circumstances.

$$\begin{aligned} Trans(applyPol(\mathfrak{M}(\varphi, v); \pi), s, \delta', s') \equiv s = s' \wedge \\ (v = true \wedge \varphi[s] \wedge \delta' = applyPol(\pi) \vee \\ v = false \wedge \neg\varphi[s] \wedge \delta' = applyPol(\pi) \vee \\ v = true \wedge \neg\varphi[s] \wedge \delta' = nil \vee \\ v = false \wedge \varphi[s] \wedge \delta' = nil) \end{aligned}$$

If the next construct in the policy is a primitive action other than a stochastic action or a $senseEffect$, then we execute the action and continue executing the rest of the policy. As discussed above, due to changes in the world it may be the case that $a$ has become impossible to execute. In this case we again abort the rest of the policy with the successor configuration $\langle nil, s\rangle$.

$$\begin{aligned} Trans(applyPol(a; \pi), s, \delta', s') \equiv \\ \exists \delta''. Trans(a; \pi, s, \delta'', s') \wedge \delta' = applyPol(\delta'') \vee \\ \neg Poss(a[s], s) \wedge \delta' = nil \wedge s' = s \end{aligned}$$

If $a$ is a stochastic action, we obtain

$$\begin{aligned} Trans(applyPol(a; senseEffect(a); \pi), s, \delta', s') \equiv \\ \exists \delta''. Trans(senseEffect(a); \pi, s, \delta'', s') \wedge \\ \delta' = applyPol(\delta'')) \end{aligned}$$

Note the subtlety that $a$ is ignored by $Trans$. This has to do with the fact that stochastic actions have no direct effects according to the way they are modeled in DTGolog. Instead one needs to perform $senseEffect$ to find out about the actual effects. Of course, even though $Trans$ ignores $a$, care must be taken by the implementation that it is executed in the real world.[7] As in the original DTGolog we also assume that $senseEffect$ actions are always executable.

Finally, if we encounter an $if$-construct, which was inserted into the policy due to a stochastic action, we determine which branch of the policy to choose and go on with the execution of that branch.

$$\begin{aligned} Trans(applyPol(if(\varphi, \pi_1, \pi_2)), s, \delta', s') \equiv \\ \varphi[s] \wedge Trans(applyPol(\pi_1), s, \delta', s') \vee \\ \neg\varphi[s] \wedge Trans(applyPol(\pi_2), s, \delta', s') \end{aligned}$$

We end this section with a few notes about the implementation of our on-line decision-theoretic interpreter called Readylog.[8]

We begin with a (very) rough sketch of the main loop of the interpreter.

```
/******** Interpreter mainloop ******/
/* (1)- exogenous action occured */
icpgo(E,H) :- exog_occurs(Act,H), exog_action(Act),!,
              icpgo(E,[Act|H]).

/* (2) - performing a step in program execution */
icpgo(E,H) :- trans(E,H,E1,H1),icpxeq(H,H1,H2),!,
              icpgo(E1,H2).

/* (3) - program is final -> execution finished */
icpgo(E,H) :- final(E,H).
```

[7] This can be done similar to Soutchanski's interpreter by inserting an appropriate $doReally(a)$ literal (see Section 2.3 or $icpxeq$ in our case (see above)).

[8] Readylog stands for "real-time dynamic Golog."

```
solve(nondet(
 [kick(ownNumber, 40),
  dribble_or_move_kick(ownNumber),
  dribble_to_points(ownNumber),
  if(isKickable(ownNumber),
     pickBest(var_turnAngle, [-3.1, -2.3, 2.3, 3.1],
       [turn_relative(ownNumber, var_turnAngle, 2),
        nondet([[intercept_ball(ownNumber, 1),
           dribble_or_move_kick(ownNumber)],
           [intercept_ball(no_ByRole(supporter), 1),
            dribble_or_move_kick(no_ByRole(supp.))]])]),
     nondet([[intercept_ball(ownNumber, 1),
           dribble_or_move_kick(ownNumber)],
           intercept_ball(ownNumber, 0.0, 1)]) ) ]), 4)
```

**Figure 1.** The *bestInterceptor* program performed by an offensive player. Here, $nondet(\Sigma)$ denotes the nondeterministic choice of actions.

```
/* (4) - waiting for an exogenous action to happen */
icpgo(E,H) :- wait_for_exog_occurs, !, icpgo(E,H).

/******** Executing actions ******/
/* (1) - No action was performed so
                              we don't execute anything */
icpxeq(H,H,H).

/* (2) - The action is not a sensing one:
                          exec. and ignore its sensing */
icpxeq(H,[Act|H],H1) :- not senses(Act,_),
                        execute(Act,_,H), H1=[Act|H]).

/* (3) - The action is a sensing one for F:
                                 execute sensing action*/
icpxeq(H,[Act|H],H1):-senses(Act,F),
                  execute(Act,Sr,H),H1=[e(F,Sr),Act|H].
```

First, it checks whether an exogenous event occurred and if so inserts it into the history. Next, it is checked if a transition to a new configuration can be made executing the next possible action. If there is no successor configuration reachable a test for a final configuration is conducted. In case (4) where none of the previous cases apply the interpreter waits until some exogenous event occurs, e.g. the robot has reaches a certain position.

For the execution the predicate $icpxeq$ exists checking whether the action to be executed is a sensing action or not. This is very similar to Soutchanski's $doReally$. Note that we still allow for passive sensing actions we only avoid them during plan generation.

We put a lot of effort into tuning the perfomance of the interpreter. One major speed-up was achieved by integrating a progression mechanism for the internal database in the spirit of Lin and Reiter [9] (step 5 in the loop). For space reasons we leave out all details except to say that this is indispensable for maintaining tractability because the action history would otherwise grow beyond control very quickly.

Additional speed-ups were obtained by using a Readylog preprocessor. It takes a complete domain axiomatization as input and generates optimized Prolog code, i.e. run-time invariants like static conditions are evaluated at compile-time to save the time of evaluating them many times at run-time.

## 4 Empirical Results in RoboCup

We used the described version of Readylog for our RoboCup Middle Size robot team at the world championships 2003 in Padua, Italy, and at the German Open 2004 in Paderborn. In this Section we show some details of the implementation of our soccer agent and present some results of the use of decision-theoretic planning in Golog in the soccer domain.

Among the basic actions we used the most important were $goto\_pos(x, y, \theta)$, $turn(\theta)$, $dribble\_to(x, y, \theta)$, $intercept$, $kick(power)$, and $move\_and\_kick(x, y, \theta)$.

While the goalie was controlled without Readylog in order to maintain the highest possible level of reactivity, all other players of our team had an individual Readylog procedure for playing. We assigned fixed roles to the three field players: defender, supporter, and

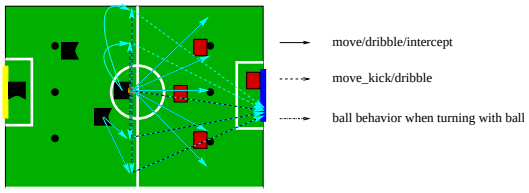



**Figure 2.** The set of alternatives for the attacker when it is in ball possession. The red boxes denote opponents, the black ones are teammates. Everything else are fi eld markings.

attacker. Only the best positioned player to the ball started to deliberate, i.e. solved the MDP given by the program in Fig. 1, otherwise it performed a program according to its role like defending the team's goal.

The set of alternatives made up from the *bestInterceptor* program is best described by Figure 2. In line 3 of Fig. 1 the choice is between a dribbling to the free goal corner or to dribble thereto but finishing the action with a shot as soon as the goal is straight ahead. In line 6 the agent decides among four angles to turn to in order to push the ball to either side where it can be intercepted by a teammate, using the $pickBest$ construct.

Figure 3 shows an example decision tree made up from this program. For readability we pruned some similar branches. The root node stands for the situation were the agent switched to off-line mode, i.e., the current situation. The boxes symbolize agent choices, i.e. the agent can decide which of the alternatives to take. The circles are nature's choices, denoting the possible outcomes of stochastic actions. Numbers of outgoing edges in these nodes are the probabilities for the possible outcomes. The numbers in the boxes are the rewards for the corresponding situation. The actually best policy in the situation of the example is marked by a thick line.

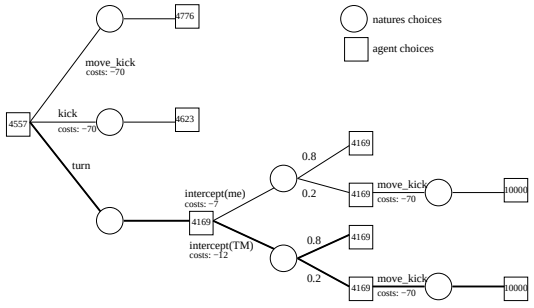



**Figure 3.** A (pruned) example decision tree for the *bestInterceptor* program.

Indispensable for a successful acting agent using decision-theoretic planning is a reasonable reward function. For this scenario, however, we used a rather primitive reward function based solely on the velocity, relative position and distance of the ball towards the opponents goal. In future work this could be refined for improving the overall play.

Naturally, the time a player spent deliberating depended highly on the number of alternatives that were possible. In this respect, whether or not the ball was kickable made the most difference (all times in seconds):

| | examples | min | avg | max |
|---|---|---|---|---|
| without ball | 698 | $< 0.01$ | 0.094 | 0.450 |
| with ball | 117 | 0.170 | 0.536 | 2.110 |

The hardware used was an on-board Pentium III-933.

With the described decision making method we shot 13 goals at the world championships and scratched the final round by one goal. In the end we placed 10th out of 24 and ended 5th (or 7th depending on ranking algorithm) out of 13.

## 5 Conclusion

In this paper, we proposed a novel method of on-line decision-theoretic planning and execution in Golog, which is particularly suited for robotic applications with frequent sensor updates like measuring one's own position and that of other agents and objects. We overcame the problem of Soutchanski's approach, which cannot plan past the next sensing action, by eliminating explicit sensor updates of the above kind from a robot control program altogether and replacing them instead with models that allow the approximate calculation of otherwise sensed values during planning. However, this also required annotating policies with information about the models so that discrepancies with the real world could be detected while executing the policy. Our approach was applied in the RoboCup domain with encouraging results.

One weakness of the current implementation is that rewards are assigned manually in a rather ad-hoc manner. In the future we hope to employ learning methods to improve the overall performance.

## REFERENCES

[1] C. Boutilier, R. Reiter, and B. Price, 'Symbolic dynamic programming for fi rst-order MDPs', in *IJCAI*, pp. 690–700, (2001).
[2] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *Proc. of AAAI-00*, pp. 355–362. AAAI Press, (2000).
[3] G. De Giacomo, Y. Lésperance, and H. Levesque, 'ConGolog, A concurrent programming language based on situation calculus', *Artificial Intelligence*, **121**(1–2), 109–169, (2000).
[4] G. De Giacomo and H. Levesque, 'An incremental interpreter for high-level programs with sensing', in *Logical Foundation for Cognitive Agents: Contributions in Honor of Ray Reiter*, eds., Hector J. Levesque and Fiora Pirri, 86–102, Springer, Berlin, (1999).
[5] Henrik G. and G. Lakemeyer, 'On-line execution of cc-Golog plans', in *Proc. of IJCAI-01*, (2001).
[6] A. Großmann, S. Hölldobler, and O. Skvortsova, 'Symbolic dynamic programming with the Fluent Calculus', in *Proceedings of the IASTED (ACI-2002)*, pp. 378–383, (2002).
[7] Y. Lespérance and H.-K. Ng, 'Integrating planning into reactive high-level robot programs', in *Proc. (CogRob-2000)*, pp. 49–54, (2000).
[8] H. Levesque, R. Reiter, Y. Lesperance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(1-3), 59–83, (1997).
[9] F. Lin and R. Reiter, 'How to progress a database', *Artificial Intelligence*, **92**(1-2), 131–167, (1997).
[10] J. McCarthy, 'Situations, actions and causal laws', Technical report, Stanford University, (1963).
[11] D. Poole, 'The independent choice logic for modelling multiple agents under uncertainty', *Artificial Intelligence*, **94**(1-2), 7–56, (1997).
[12] M. Puterman, *Markov Decision Processes: Discrete Dynamic Programming*, Wiley, New York, 1994.
[13] R. Reiter, *Knowledge in Action*, MIT Press, 2001.
[14] M. Shanahan, 'The event calculus explained', *Lecture Notes in Computer Science*, **1600**, (1999).
[15] M. Soutchanski, 'An on-line decision-theoretic golog interpreter', in *Proc. IJCAI-2001*, Seattle, Washington, (August 2001).

# Learning Partially Observable Action Models

**Eyal Amir**
**Computer Science Department**
**University of Illinois, Urbana-Champaign**
**Urbana, IL 61801, USA**
**eyal@cs.uiuc.edu**

**Abstract.** In this paper we present tractable algorithms for learning a logical model of actions' effects and preconditions in deterministic partially observable domains. These algorithms update a representation of the set of possible action models after every observation and action execution. We show that when actions are known to have no conditional effects, then the set of possible action models can be represented compactly indefinitely. We also show that certain desirable properties hold for actions that have conditional effects, and that sometimes those can be learned efficiently as well. Our approach takes time and space that are polynomial in the number of domain features, and it is the first exact solution that is tractable for a wide class of problems. It does so by representing the set of possible action models using propositional logic, while avoiding general-purpose logical inference. Learning in partially observable domains is difficult and intractable in general, but our results show that it can be solved exactly in large domains in which one can assume some structure for actions' effects and preconditions. These results are relevant for more general settings, such as learning HMMs, reinforcement learning, and learning in partially observable stochastic domains.

## 1 Introduction

Agents that act in complex domains usually have limited prior knowledge of their actions' preconditions and effects (the *transition model* of the world). Such agents need to learn about these action to act effectively, and they also need to track the state of the world, when their sensory information is limited. For example, a robot moving from room to room in a building can observe only its immediate environment. Upon discovering a switch in the wall, it may not know the consequences of flipping this switch. After flipping it, the agent may observe those effects that occur in its immediate environment, but not those outside the room. When it leaves the room and discovers some change in the world, it may want to ascribe this change to flipping the switch.

Learning transition models in partially observable domains is hard. In stochastic domains, learning transition models is central to learning Hidden Markov Models (HMMs) [17] and to reinforcement learning [8], both of which afford only solutions that are not guaranteed to approximate the optimal. In HMMs the transition model is learned using the Baum-Welch algorithm, which is a special case of EM. It is a hill-climbing algorithm which is only guaranteed to reach a local optima, and there is no time guarantee for convergence on this local optima. *Reinforcement learning* in partially observable domains [7] can be solved (approximately) by interleaving learning the POMDP with solving it (the learning and solving are both approximate because finite memory or finite granularity is always assumed) [3, 12, 13]. It is important to notice that this problem is harder than solving POMDPs. In some cases, one can solve the POMDP with some guarantee for relatively fast convergence and approximation, if one knows the underlying transition model [9, 14]. Also, in deterministic cases, computing the optimal undiscounted infinite horizon policy in (known) POMDPs is PSPACE-hard (NP-complete if a polynomial horizon) in the number of states [11], but reinforcement learning has no similar solution known to us.

In this paper we present a formal, exact, many times *tractable*, solution to the problem of *simultaneously learning and filtering (SLAF) preconditions and effects of actions* from experiences in partially observable domains. We put emphasis on the solution being tractable as a function of the *number of state features* rather than the (exponentially larger) number of states.

First, we present a formal system that captures this problem precisely for possibly nondeterministic actions. It maintains a set of pairs $\langle$*state,transition-relation*$\rangle$ that are consistent with the actions and observations collected so far (the *transition belief state*). Then, we present a generic algorithm that uses logical deduction and learns transition models in deterministic partially observable domains.

We present more tractable algorithms for special cases of SLAF. We examine actions that are (1) *always executable* or *sometimes inexecutable* (depending on deterministic preconditions), and (2) *conditional* or *nonconditional* (whenever executable, have the same effect). For the case of STRIPS actions (always executable, nonconditional) we show that our algorithm runs in time linear in the number of propositional domain features and the space taken to represent our transition belief state. We can maintain this transition belief state in polynomial space (in the number of features and actions available in our domain) under very relaxed conditions. We present a more general algorithm (than our STRIPS one) that treats other cases with a polynomial time per time step, when actions are known to act as 1:1 mapping on states, and they provide an approximation otherwise.

Our algorithms are the first to learn exact action models in partially observable domains. They are also first to find an action model at the same time that they determine the agent's knowledge about the state of the world. They draw on intuitions and results of [1] for known (nondeterministic) action models. If we assume that our transition model is fully known, then our results reduce to those of [1] for deterministic actions.

A wide range of virtual domains satisfy our assumptions of determinism and structured actions, and we are in the process of testing our algorithms in large domains, including over 1000 features (see [6] for current progress).

Previous work on learning action's effects and preconditions focused on fully observable domains. [5, 19] learn STRIPS actions with parameters by finding the most general and most specific in a version space of STRIPS operators. [15] uses a general-purpose classification system (in their case, MSDD) to learn the effects and preconditions of actions, identifying irrelevant variables. [2] presents an approach that is based on inductive logic programming. Most recently, [16] showed how to learn stochastic actions with no conditional effects (i.e., the same stochastic change occurs at every state in which the action is executable). The common theme among these approaches is their assumption that the state of the world is fully observed at any point in time. [18] is the only work that considers partial observability, and it does so by assuming that the world is fully observable, giving approximate computation in relevant domains.

## 2 Filtering Transition Relations

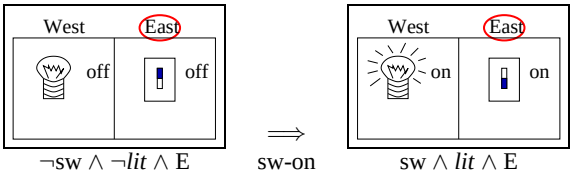



**Figure 1.** Two rooms and flipping the light switch

Consider a simple world with two rooms, one with a switch, and the other with a light bulb whose state can be observed only when the agent is in that room (see Figure 1). Assume that our agent initially knows nothing about the three actions go-E (go to the Eastern room), go-W (go to the Western room), and sw-on (flip the switch to *on*). Our agent's problem is to determine the effects of these actions (to the extent that it can, theoretically), while also tracking the world.

We describe the combined problem of filtering (updating the agent's belief state) and learning the transition model formally. A *transition system* is a tuple $\langle \mathcal{P}, \mathcal{S}, \mathcal{A}, \mathcal{R} \rangle$, where

- $\mathcal{P}$ is a finite set of propositional fluents;
- $\mathcal{S} \subseteq Pow(\mathcal{P})$ is the set of world states;
- $\mathcal{A}$ is a finite set of actions;
- $R \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the transition relation.

Here, a *world state*, $s \in \mathcal{S}$, is a subset of $\mathcal{P}$ that contains propositions true in this state, and $R(s, a, s')$ means that state $s'$ is a possible result of action $a$ in state $s$.

A *transition belief state* is a set of tuples $\langle s, R \rangle$ where $s$ is a state and $R$ a transition relation. Let $\mathfrak{R} = Pow(\mathcal{S} \times \mathcal{A} \times \mathcal{S})$ be the set of all possible transition relations on $\mathcal{S}, \mathcal{A}$. Let $\mathfrak{S} = \mathcal{S} \times \mathfrak{R}$. Every $\rho \subseteq \mathfrak{S}$ is a *transition belief state*. When we hold a a transition belief state $\rho$ we consider every tuple $\langle s, R \rangle \in \rho$ possible. With this formal system we assume that observations are given to us (if at all) as logical sentences after performing an action. They are either *state formulae* (propositional combinations of fluent names) or $OK$ or $\neg OK$ (observing the action is possible or impossible to execute). We denote the former kind of observation with $o$, and the latter with $OK$,$\neg OK$, respectively.

**Definition 1 (Transition Filtering Semantics)** *Let $\rho \subseteq \mathfrak{S}$ be a transition belief state. The* filtering *of $\rho$ with actions and observations $\langle a_1, o_1, \ldots, a_t, o_t \rangle$ is*

1. $Filter[\epsilon](\rho) = \rho$;
2. $Filter[a, OK](\rho) = \{\langle s', R \rangle \mid \langle s, a, s' \rangle \in R,\ \langle s, R \rangle \in \rho\}$;
3. $Filter[a, \neg OK](\rho) = \{\langle s, R \rangle \mid \langle s, R \rangle \in \rho,\ \forall s' \in \mathcal{S}\ \langle s, a, s' \rangle \notin R\}$;
4. $Filter[o](\rho) = \{\langle s, R \rangle \in \rho \mid o \text{ is true in } s\}$;
5. $Filter[\langle a_i, o_i, \ldots, a_t, o_t \rangle](\rho) = Filter[\langle a_{i+1}, o_{i+1}, \ldots, a_t, o_t \rangle](Filter[o_i](Filter[a_i](\rho)))$.

*We call Step 2* progression with $a$, *Step 3* disqualifying $a$, *and Step 4* filtering with $o$.

The intuition behind this definition is that every transition relation, $R$, and initial state, $s$, produce a set of state-relation pairs $\{\langle s_i, R \rangle\}_{i \in I}$ in the result of an action. If an observation discards some state $s_i$, the pair $\langle s_i, R \rangle$ is removed from this set. We conclude that $R$ is not possible when all pairs including it are removed from the set.

A nondeterministic domain description $D$ is a finite set of *transition rules* of the form "$a$ **causes** $F$ **if** $G$" which describe the effects of actions, for $F$ and $G$ propositional state formulae. We say that $F$ is the *head* and $G$ is the *tail* of those rules. When $G \equiv TRUE$ we write "$a$ **causes** $F$".

The semantics of a domain description that we choose is compatible with the *standard semantics* belief update operator of [20]. We define it below by first *completing* the description and then mapping the completed description to a transition relation.

For domain description $D$ we define a transition system with $\mathcal{P}_D$ and $\mathcal{A}_D$ the sets of propositional fluents and actions mentioned in $D$, respectively. For action $a$ and fluent $f$, let

$$G_D(a, f) = \bigvee \{G \mid \text{"}a \textbf{ causes } F \textbf{ if } G\text{"} \in D,\ f \in L(F)\},$$

a disjunction of the preconditions of rules that possibly affect fluent $f$ (an empty disjunction is equivalent to FALSE). We use "$a$ **keeps** $f$ **if** $G$" as a shorthand for the rules "$a$ **causes** $f$ **if** $f \wedge G$" and "$a$ **causes** $\neg f$ **if** $\neg f \wedge G$". It designates the *non-effects* of action $a$. Define $Comp(D)$, the *completion of* $D$

$$Comp(D) = D \cup \{\text{"}a \textbf{ keeps } f \textbf{ if } \neg G_D(a, f)\text{"} \mid a \in \mathcal{A}, f \in \mathcal{P},\ G_D(a, f) \not\equiv TRUE\}.$$

This definition is well behaved, in the sense that $Comp(D) = Comp(D \cup D')$, if $D' \subseteq Comp(D)$.

Let $F_D(a, s) = \{F \mid \text{"}a \textbf{ causes } F \textbf{ if } G\text{"} \in D,\ s \models G\}$, the set of effects of $a$ in $s$, according to $D$. $D$ defines a transition relation $R_D$ as follows

$$R_D = \{\langle s, a, s' \rangle \mid s, s' \in \mathcal{S},\ a \in \mathcal{A},\ s' \models F_D(a, s)\} \quad (1)$$

When there is no confusion, we write $R$ for $R_D$. We say that two domain descriptions $D_1, D_2$ are equivalent ($D_1 \equiv D_2$), if $R_{D_1} = R_{D_2}$. $D$ is a *complete domain description*, if $R_D = R_{Comp(D)}$. In that case we say that $R$ is *completely defined by* $D$.

| Time step | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Action | | go-W | | sw-on | | go-E | | sw-on | | go-W | | go-E | |
| Location | $E$ | | $\neg E$ | | $\neg E$ | | $E$ | | $E$ | | $\neg E$ | | $E$ |
| Bulb | ? | | $\neg lit$ | | $\neg lit$ | | ? | | ? | | $lit$ | | ? |
| Switch | $\neg sw$ | | ? | | ? | | $\neg sw$ | | $sw$ | | ? | | $sw$ |
| Possible | | $OK$ | | $\neg OK$ | | $OK$ | | $OK$ | | $OK$ | | $OK$ | |

**Figure 2.** An action-observation sequence (table entries are observations). Legend: $E$: east; $\neg E$: west; $lit$: light is on; $\neg lit$: light is off; $sw$: switch is on; $\neg sw$: switch is off; $OK$: action executable; $\neg OK$: action not executable.

**Example 2** *Consider the scenario of Figure 2 and assume that actions are deterministic, unconditional, and always executable (assuming no action was performed at step 2). Then, every action affects every fluent either negatively, positively, or not at all. Consequently, every transition relation $R$ is completely defined by some $D$ such that (viewing a tuple as a set of its elements)*

$$D \in \prod_{a \in \left\{ \begin{array}{l} \textit{go-W} \\ \textit{go-E} \\ \textit{sw-on} \end{array} \right\}} \left\{ \begin{array}{l} a \textbf{ causes } E, \\ a \textbf{ causes } \neg E \\ a \textbf{ keeps } E \end{array} \right\} \times \left\{ \begin{array}{l} a \textbf{ causes } sw, \\ a \textbf{ causes } \neg sw \\ a \textbf{ keeps } sw \end{array} \right\} \times \left\{ \begin{array}{l} a \textbf{ causes } \textit{lit}, \\ a \textbf{ causes } \neg \textit{lit} \\ a \textbf{ keeps } \textit{lit} \end{array} \right\}$$

*Say that initially we know the effects of go-E, go-W, but do not know what sw-on does. Then, transition filtering starts with the product set of $\mathcal{R}$ (of 27 possible relations) and all possible $2^3$ states. Also, at time step 4 we know that the world state is exactly $\{E, \neg lit, \neg sw\}$. We try sw-on and get that $Filter[\textit{sw-on}](\rho_4)$ includes the same set of transition relations but with each of those transition relations projecting the state $\{E, \neg lit, \neg sw\}$ to an appropriate choice from $\mathcal{S}$. When we receive the observations $o_5 = \neg E \wedge \neg sw$ of time step 5, $\rho_5 = Filter[o_5](Filter[\textit{sw-on}](\rho_4))$ removes from the transition belief state all the relations that gave rise to $\neg E$ or to $\neg sw$. We are left with transition relations satisfying one of the tuples in*

$$\left\{ \begin{array}{l} \textit{sw-on} \textbf{ causes } E, \\ \textit{sw-on} \textbf{ keeps } E \end{array} \right\} \times \{ \ \textit{sw-on} \textbf{ causes } sw \ \} \times \left\{ \begin{array}{l} \textit{sw-on} \textbf{ causes } \textit{lit} \\ \textit{sw-on} \textbf{ causes } \neg \textit{lit} \\ \textit{sw-on} \textbf{ keeps } \textit{lit} \end{array} \right\}$$

*Finally, when we perform action go-W, again we update the set of states associated with every transition relation in the set of pairs $\rho_5$. When we receive the observations of time step 6, we conclude $\rho_6 = Filter[o_6](Filter[\textit{go-W}](\rho_5)) =$*

$$\left\{ \left\langle \left\{ \begin{array}{l} \neg E \\ \textit{lit} \\ sw \end{array} \right\}, \left\{ \begin{array}{l} \textit{sw-on} \textbf{ causes } E, \\ \textit{sw-on} \textbf{ causes } sw, \\ \textit{sw-on} \textbf{ causes } \textit{lit}, \\ \textit{go-E}\ldots \end{array} \right\} \right\rangle, \left\langle \left\{ \begin{array}{l} \neg E \\ \textit{lit} \\ sw \end{array} \right\}, \left\{ \begin{array}{l} \textit{sw-on} \textbf{ keeps } E, \\ \textit{sw-on} \textbf{ causes } sw, \\ \textit{sw-on} \textbf{ causes } \textit{lit}, \\ \textit{go-E}\ldots \end{array} \right\} \right\rangle \right\} \quad (2)$$

SLAF reduces to filtering (updating the agent's belief state) [1, 20, 10] when the transition model is fully specified.

**Theorem 3** *Let $\rho = \sigma \times \{R\}$, where $\sigma \subseteq \mathcal{S}$ and $R \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, and let $\langle a_i, o_i \rangle_{i \leq t}$ be a sequence of actions and observations. If $Filter_R[\langle a_i, o_i \rangle_{i \leq t}](\sigma)$ is the belief-state filtering[1] of $\sigma$ with $\langle a_i, o_i \rangle_{i \leq t}$, then $Filter[\langle a_i, o_i \rangle_{i \leq t}](\rho) = Filter_R[\langle a_i, o_i \rangle_{i \leq t}](\sigma) \times \{R\}$.*

## 3 Logical Filtering of Transition Models

The example in the previous section illustrates how the explicit representation of transition belief states may be doubly exponential in the number of domain features and the number of actions. In this section we follow the intuition that propositional logic can serve to represents $\rho$ more compactly. *From here forth we assume that our actions are deterministic.*

In the following, for a set of propositional formulae, $\Psi$, $L(\Psi)$ is the signature of $\Psi$, i.e., the set of propositional symbols that appear in $\Psi$. $\mathcal{L}(\Psi)$ is the language of $\Psi$, i.e., the set of formulae built with $L(\Psi)$. Similarly, $\mathcal{L}(L)$ is the language of $L$, for a set of symbols $L$.

### 3.1 Representing Transition Belief States

We define a propositional logical language that allows us to represent sets of domain descriptions (thus, sets of transition relations). Let $\mathbb{P}_1, \mathbb{P}_2 \subseteq \mathcal{L}(\mathcal{P})$ be sets of state formulae such that $\mathbb{P}_1$ includes only literals or $FALSE$, $\mathbb{P}_2$ includes only terms (conjunctions of literals) that are not equivalent to FALSE, and for all $\varphi, \psi \in \mathbb{P}_1 \cup \mathbb{P}_2$, if $\varphi \equiv \psi$, then $\varphi = \psi$ or $\varphi \equiv TRUE$. We define a propositional vocabulary

$$L(\mathbb{P}_1, \mathbb{P}_2) = \{a_G^F \mid a \in \mathcal{A},\ F \in \mathbb{P}_1,\ G \in \mathbb{P}_2\}.$$

**Theorem 4** *For every rule $r =$"$a$ **causes** $F$ **if** $G$", for $F, G$ state formulae, there is a set of transition rules $TR = \{$"$a$ **causes** $l_i$ **if** $t_i$"$\}_{i \in I}$, with a set of indices $I$, terms $t_i$, and literals $l_i$, such that $r \equiv TR$ (i.e., we can exchange $r$ for $TR$, and get an equivalent domain description).*

In the rest of this paper we implicitly assume that $\mathbb{P}_1, \mathbb{P}_2 \subseteq \mathcal{L}(\mathcal{P})$ are sets of state formulae as above. Also, $D$ is a *complete* domain description with effects in $\mathbb{P}_1$ and preconditions in $\mathbb{P}_2$. We also assume that if $\neg \exists s' R_D(s, a, s')$ for some $s \in \mathcal{S}, a \in \mathcal{A}$, then there is a rule "$a$ **causes** FALSE **if** $G$" such that $s \models G$.

For set of formulae $\mathbb{P}_2$ we define $Bottom(\mathbb{P}_2) = \{G \in \mathbb{P}_2 \mid G \not\equiv FALSE,\ \forall G' \in \mathbb{P}_2\ [(G' \models G) \wedge (G' \not\equiv FALSE)] \Rightarrow G = G'\}$, i.e., $Bottom(\mathbb{P}_2)$ is the set of strongest preconditions in $\mathbb{P}_2$.

**Definition 5** *We define the theory*

$$T_D^L = \textit{rules}_D \wedge \textit{implied-weaker-rules} \wedge \textit{implied-stronger-rules} \wedge \textit{exec-preconds}_D \quad (3)$$

[1] Filtering semantics as defined in [1].

$\mathit{rules}_D = \{a^F_G \in L \mid$ "$a$ **causes** $F$ **if** $G$" $\in D\}$
$\mathit{implied\text{-}weaker\text{-}rules} = \bigwedge_{\substack{F \in \mathbb{P}_1 \\ G, G' \in \mathbb{P}_2 \\ a \in \mathcal{A}}} (a^F_{G'} \wedge (G \Rightarrow G') \Rightarrow a^F_G)$
$\mathit{implied\text{-}stronger\text{-}rules} = \bigwedge_{\substack{F \in \mathbb{P}_1 \\ G, G', G'' \in \mathbb{P}_2 \\ G'' \equiv G \vee G' \\ a \in \mathcal{A}}} (a^F_G \wedge a^F_{G'} \Rightarrow a^F_{G''}) \wedge \bigwedge_{\substack{F, F', F'' \in \mathbb{P}_1 \\ F'' \equiv F \wedge F' \\ G \in \mathbb{P}_2 \\ a \in \mathcal{A}}} (a^F_G \wedge a^{F'}_G \Rightarrow a^{F''}_G)$
$\mathit{exec\text{-}preconds}_D =$
$\{\neg a^{FALSE}_G \in L \mid G \in Bottom(\mathbb{P}_2),$
$\forall G' \in \mathbb{P}_2\ ((G \models G') \Rightarrow$ "$a$ **causes** *FALSE* **if** $G'$" $\notin D)\}$.

The intention is that $a^F_G \in L$ is true in $T^L_D$ exactly when "$a$ **causes** $F$ **if** $G$" is in $D$ or there is a stronger transition rule "$a$ **causes** $F'$ **if** $G'$" in $D$.

The following theorem shows how we can represent deterministic transition relations (with conditional effects) using only the positive causality statements[2].

**Theorem 6 (Representing Deterministic Actions)** *1. $T_D$ is a complete theory*
*2. If $T_D \models a^F_G$, then for every $s, s'$, if $R_D(s, a, s')$ and $s \models G$, then $s' \models F$.*
*3. If $T_D \models \neg a^F_G$, then there are $s, s'$ such that $R_D(s, a, s')$ and $s \models G$, $s' \models \neg F$.*

Consequently, for every pair of complete domain descriptions $D_1, D_2$, $D_1 \equiv D_2$ iff $T_{D_1} \equiv T_{D_2}$. Thus, every transition relation $R$ has a unique theory $T_D$ and every theory defines a unique transition relation. (We write $T_R$ for the theory representing transition relation $R$.)

**Corollary 7 (Always-Executable, Deterministic)** *If in $D$ all actions are always executable, then*[3] *exec-preconds $= \{\neg a^{FALSE}_G \mid a \in \mathcal{A},\ G \in Bottom(\mathbb{P}_2)\}$ and*

$$T_D \equiv \mathit{rules}_D \wedge \mathit{exec\text{-}preconds} \wedge \bigwedge_{\substack{F \in \mathbb{P}_1 \\ G, G' \in \mathbb{P}_2 \\ a \in \mathcal{A}}} (a^F_{G'} \wedge (G \Rightarrow G') \Rightarrow a^F_G) \wedge \bigwedge_{\substack{F \in \mathbb{P}_1 \\ G \in \mathbb{P}_2 \\ a \in \mathcal{A}}} \neg(a^F_G \wedge a^{\neg F}_G) \wedge \bigwedge_{\substack{F \in \mathbb{P}_1 \\ G, G', G'' \in \mathbb{P}_2 \\ G'' \equiv G \vee G' \\ a \in \mathcal{A}}} (a^F_G \wedge a^F_{G'} \Rightarrow a^F_{G''})$$

Define $a^{f\circ} = a^f_f \wedge a^{\neg f}_{\neg f}$.

**Corollary 8 (Unconditional, Always-Exec., Deterministic)** *Let $\mathbb{P}_2 = \{TRUE\}$, and assume that $D$ possibly includes sentences of the form "$a$ **keeps** $F$", and no sentences of the form "$a$ causes $FALSE$". Then,*

$$T_D \equiv rules_D \wedge \bigwedge_{f \in \mathcal{P}, a \in \mathcal{A}} (a^f \bar{\vee} a^{\neg f} \bar{\vee} a^{f\circ}).$$

We encode sets of domain descriptions as follows: For a set $\mathcal{R} \subset \mathbb{P}(\mathcal{S} \times \mathcal{A} \times \mathcal{S})$ let[4] $T_{\mathcal{R}} = \bigvee_{R \in \mathcal{R}} T_R$. For a tuple $\langle s, R\rangle$, $s \in \mathcal{S}$, we define $T_{\langle s,R\rangle} = T_R \wedge s$. Finally, for a transition belief state, $\rho$, we define $T_\rho = \bigvee_{\langle s,R\rangle \in \rho} T_{\langle s,R\rangle}$.

[2] At present it is not clear to the author how one can observe non-causality in nondeterministic settings.
[3] $D$ is omitted as a subscript because it is not relevant.
[4] We assume that the set of fluents $\mathcal{P}$ is finite.

**Example 9** *Consider $\rho_6$ from Example 2 (equation 2). There, we considered only deterministic, same-effect, always-executable actions. We take $\mathbb{P}_1$ to include only unit clauses, and $\mathbb{P}_2 = \{TRUE\}$. We can write $\rho_6$ using a logical formula that is satisfied only by the tuples in $\rho_6$:*

$$T_{\rho_6} \equiv \begin{pmatrix} \neg E \wedge \\ sw \wedge \\ lit \end{pmatrix} \wedge \begin{pmatrix} \textit{go-W}^{\neg E} \wedge \\ \textit{go-W}^{sw\circ} \wedge \\ \textit{go-W}^{lit\circ} \end{pmatrix} \wedge \begin{pmatrix} \textit{go-E}^{E} \wedge \\ \textit{go-E}^{sw\circ} \wedge \\ \textit{go-E}^{lit\circ} \end{pmatrix} \wedge \begin{Bmatrix} (\textit{sw-on}^{E} \vee \textit{sw-on}^{E\circ}) \wedge \\ \textit{sw-on}^{sw} \wedge \\ \textit{sw-on}^{lit} \end{Bmatrix} \wedge \bigwedge_{f \in \mathcal{P}, a \in \mathcal{A}} (a^f \bar{\vee} a^{\neg f} \bar{\vee} a^{f\circ})$$

*Notice that, e.g., $\neg$go-$E^{\neg E}$ and $\neg$go-$E^{E\circ}$ are logical consequences of $T_{\rho_6}$.*

*The same way allows us to represent a much larger set of transition relations. For example, in the previous section we avoided listing the contents of the set of tuples $\rho_0$ because it was too large (27 possible relations cross-combined with all $2^3$ world states). Now we can write it simply as follows:*

$$\begin{pmatrix} \textit{go-W}^{\neg E} \\ \textit{go-W}^{sw\circ} \\ \textit{go-W}^{lit\circ} \end{pmatrix} \wedge \begin{pmatrix} \textit{go-E}^{E} \wedge \\ \textit{go-E}^{sw\circ} \wedge \\ \textit{go-E}^{lit\circ} \end{pmatrix} \wedge \begin{pmatrix} (\textit{sw-on}^{E} \vee \textit{sw-on}^{\neg E} \vee \textit{sw-on}^{E\circ}) \wedge \\ (\textit{sw-on}^{sw} \vee \textit{sw-on}^{\neg sw} \vee \textit{sw-on}^{sw\circ}) \wedge \\ (\textit{sw-on}^{lit} \vee \textit{sw-on}^{\neg lit} \vee \textit{sw-on}^{lit\circ}) \end{pmatrix} \wedge \bigwedge_{f \in \mathcal{P}, a \in \mathcal{A}} (a^f \bar{\vee} a^{\neg f} \bar{\vee} a^{f\circ})$$

Thus, we can represent a transition belief state, $\rho$, with a logical formula, $\varphi$, over the propositional state fluents and the propositional symbols for effect sentences. $\varphi$ represents the set of tuples $\langle s, R\rangle$ that satisfy it. We call this logical representation a *transition belief formula*. It follows that $\varphi \models$ *base*, if we take *base* to be the subformula of $T_D$ that is not $\mathit{rules}_D$ in the appropriate case of Corollaries 7, 8 because *base* is independent of $D$ in those cases.

## 3.2 Filtering Logical Action Models

For a deterministic (possibly conditional) action, $a$, define the effect model of $a$ for time $t$ to be

$$T_{\mathit{eff}}(a,t) = \bigwedge_{l \in \mathbb{P}_1, G \in \mathbb{P}_2} ((a_t \wedge a^l_G \wedge G_t) \Rightarrow l_{t+1}) \wedge \bigwedge_{l \in \mathbb{P}_1} (l_{t+1} \wedge a_t \Rightarrow (\bigvee_{G \in \mathbb{P}_2} (a^l_G \wedge G_t))) \quad (4)$$

where $a_t$ is a propositional symbol asserting that action $a$ occurred at time $t$, and we use the convention that $\varphi_t = \varphi_{[\mathcal{P}/\mathcal{P}_t]}$, i.e., $\varphi_t$ is the result of replacing every propositional symbol of $\varphi$ with the same propositional symbol that now has an added subscript, $t$. The first part of the conjunction is the assertion that if $a$ executes at time $t$, and it causes $l$, if $G$ holds, and $G$ holds at time $t$, then $l$ holds at time $t+1$. The second part of the conjunction says that $l$ is true at time $t+1$ after $a$'s execution only if $a$ has an effect $l$ conditional on some $G$, and this $G$ is true at time $t$.

For an always-executable, non-conditional action, $a$, we get a simpler formula

$$T_{\mathit{eff}}(a,t) \equiv \bigwedge_{l \in \mathbb{P}_1} ((a_t \wedge (a^l \vee (a^l_l \wedge l_t)) \Rightarrow l_{t+1}) \wedge \bigwedge_{l \in \mathbb{P}_1} (l_{t+1} \wedge a_t \Rightarrow (a^l \vee (a^l_l \wedge l_t)))$$

**Definition 10 (Logical Transition Filtering)**
**Progression:** $Filter[a](\varphi) = Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{\mathit{eff}}(a,t))$
**Filtering:** $Filter[o](\varphi) = \varphi \wedge o$

Thus, $Filter[a](\varphi)$ is the set of consequences of $\varphi_t$ in the vocabulary $L_{t+1} = \mathcal{P}_{t+1} \cup L$, the vocabulary that includes only fluents of time $t+1$ and effect propositions from $L$. The following theorem shows that filtering a transition belief formula is equivalent to filtering a transition belief state.

**Theorem 11** *For $\varphi$ transition belief formula, $a$ action,*

$$Filter[a](\{\langle s, R\rangle \in \mathfrak{S} \mid \langle s, R\rangle \textit{ satisfies } \varphi\}) = \{\langle s, R\rangle \in \mathfrak{S} \mid \langle s, R\rangle \textit{ satisfies } Filter[a](\varphi)\}$$

## 3.3 Distribution Properties

Several distribution properties always hold for filtering of transition belief states (or formulae). The first one follows from set theoretical considerations. 11.

**Corollary 12** *For $\varphi, \psi$ transition belief formulae, $a$ action,*
1. $Filter[a](\varphi \vee \psi) \equiv Filter[a](\varphi) \vee Filter[a](\psi)$
2. $\models Filter[a](\varphi \wedge \psi) \Rightarrow Filter[a](\varphi) \wedge Filter[a](\psi)$

Stronger properties hold if filtering an action is a 1:1 mapping between state-transition-relation pairs.

**Corollary 13** *Let $a$ be an action, and $\varphi, \psi$ be transition belief formulae. Then, $Filter[a](\varphi \wedge \psi) \equiv Filter[a](\varphi) \wedge Filter[a](\psi)$, if*
1. *For every transition relation $R$ possible with $\varphi \vee \psi$, $a$ maps states in $\{s \mid \langle s, R\rangle \models \varphi\}$ 1:1 to states in $\mathcal{S}$, or*
2. *Whenever $\langle s_1, R\rangle \models \varphi \vee \psi$, $\langle s_2, R\rangle \models \varphi \vee \psi$, then $s_1 = s_2$.*

**Corollary 14** *For action $a$, state $s \in \mathcal{S}$, and $\varphi, \psi$ transition belief formulae,*

$$Filter[a](s \wedge \varphi \wedge \psi) \equiv Filter[a](s \wedge \varphi) \wedge Filter[a](s \wedge \psi)$$

This last corollary explains the relationship between learning in fully observable and partially observable worlds. Our algorithms for learning world models will be more tractable when our agent observes more of the environment. We see in Section 4 that polynomial-time algorithms exist for SLAF when filtering distributes over conjunctions.

Finally, when $T_{\mathit{eff}}(a,t) \equiv T^1 \wedge T^2$ and $\varphi \equiv \varphi^1 \wedge \varphi^2$, such that $L(T^1) \cap L(T^2) = \emptyset$ and $L(\varphi^i) \subseteq L(T^i)$, for $i \in \{1,2\}$, then the filtering factors into filtering of $\varphi^1, \varphi^2$ separately. More generally, the following holds.

**Theorem 15** *Let $a$ be an action, let $s \in \mathcal{S}$ be a state, let $\mathbb{P}_1$ include literals in $\mathcal{P}$ and FALSE, let $\mathbb{P}_2^i$ ($i \in \{1,2\}$) include clauses in $\mathcal{P}^i$ such that $L(\mathcal{P}) = L(\mathcal{P}^1) \dot{\cup} L(\mathcal{P}^2)$, and let $\varphi^i \in \mathcal{L}(L(\mathbb{P}_1, \mathbb{P}_2^i) \cup \mathcal{P})$ ($i \in \{1,2\}$) be transition belief formulae. Then,*

$$Filter[a](\varphi \wedge \psi) \equiv Filter[a](\varphi) \wedge Filter[a](\psi)$$

# 4 Factored Learning and Filtering

Learning world models is easier when filtering distributes over logical connectives. The computation becomes tractable, with the bottleneck being the time to filter each part separately. Figure 3 presents an algorithm for SLAF using this observation. Filtering of a single fluent (done in function *Fluent-SLAF*) and more efficient solutions are the focus of the rest of this section.

PROCEDURE Factored-SLAF($\langle a_i, o_i\rangle_{0<i\leq t}$,$\varphi$)
$\forall i$, $a_i$ action, $o_i$ observation, $\varphi$ transition belief formula.
1. For $i$ from 1 to $t$ do,
   (a) Set $\varphi \leftarrow$ Step-SLAF($o_i$,$a_i$,$\varphi$).
   (b) Eliminate subsumed clauses in $\varphi$.
2. Return $\varphi$.

PROCEDURE Step-SLAF($o$,$a$,$\varphi$)
$o$ an observation sentence (conjunction of literals), $a$ an action, $\varphi$ a transition belief formula.
1. If $\varphi$ is a literal, then return $o \wedge$Fluent-SLAF($o$,$a$,$\varphi$).
2. If $\varphi = \varphi_1 \wedge \varphi_2$, return Step-SLAF($o$,$a$,$\varphi_1$)$\wedge$Step-SLAF($o$,$a$,$\varphi_2$).
3. If $\varphi = \varphi_1 \vee \varphi_2$, return Step-SLAF($o$,$a$,$\varphi_1$)$\vee$Step-SLAF($o$,$a$,$\varphi_2$).

PROCEDURE Fluent-SLAF($o$,$a$,$\varphi$)
$o$ an observation sentence (conjunction of literals), $a$ an action, $\varphi$ a fluent.
1. Return $Cn^{L_{t+1}}(\varphi_t \wedge a_t \wedge T_{eff}(a,t))$.

**Figure 3.** SLAF using distribution over $\wedge, \vee$

## 4.1 Always-Executable STRIPS Actions

STRIPS actions [4] are deterministic and unconditional (but sometimes not executable). In this section we examine them with the assumption that our they always executable. We return to inexecutability in Section 4.2.

Let $L_f = \{f\} \cup \{a^f, a^{\neg f}, a^{f\circ} \mid a \in \mathcal{A}\}$ be the propositional vocabulary including only the propositional fluent symbol $f$ and effect propositions mentioning $f$. We say that $\varphi$ is a *fluent-factored transition belief formula*, if $\varphi = \mathit{base} \wedge \bigwedge_{f\in\mathcal{P}} \varphi_f$, with $L(\varphi_f) \subseteq L_f$. When a transition belief formula $\varphi$ is fluent-factored, then the result of filtering is also a fluent-factored formula.

**Theorem 16** *Let $\varphi = \mathit{base} \wedge \bigwedge_{f\in\mathcal{P}} \varphi_f$ be a fluent-factored transition belief formula, with $L(\varphi_f) \subseteq L_f$. Then,*

$$Filter[a](\varphi) \equiv \mathit{base} \wedge \bigwedge_{f\in\mathcal{P}} Filter[a](\varphi_f)$$

*and $L(Filter[a](\varphi_f)) \subseteq L_f$. Also, if $o$ is a conjunction of literals, then $Filter[o](\varphi)$ is fluent-factored.*

We are left with the problem of filtering each $\varphi_f$ with $a$ and $o$. Let $L_f^0 = L_f \setminus \{f\}$.

**Theorem 17** *Let $\varphi_f$ be a transition belief formula with $L(\varphi_f) \subseteq L_f$. Then,*

$$Filter[a](\varphi_f) \equiv (f \Rightarrow (a^f \vee ((\varphi_f \Rightarrow f) \wedge a^{f\circ}))) \wedge (\neg f \Rightarrow (a^{\neg f} \vee ((\varphi_f \Rightarrow \neg f) \wedge a^{f\circ}))) \wedge Cn(\varphi_f) \cap \mathcal{L}(L_f^0)$$

We can compute $Cn(\varphi_f) \cap \mathcal{L}(L_f^0)$ without general-purpose automated deduction, if we keep $\varphi_f$ in a the following form

$$(\neg f \vee \mathit{expl}_f) \wedge (f \vee \mathit{expl}_{\neg f}) \wedge \xi_f$$

where $\mathit{expl}_f$, $\mathit{expl}_{\neg f}$, and $\xi_f$ are in $\mathcal{L}(L_f^0)$. Every formula in $\mathcal{L}(L_f)$ is logically equivalent to a formula in this form,

PROCEDURE AE-STRIPS-SLAF($\langle a_i, o_i \rangle_{0<i\leq t}$,$\varphi$)
$\forall i$, $a_i$ an action, $o_i$ an observation, $\varphi = \bigwedge_{f\in\mathcal{P}} \varphi_f$ a fluent-factored transition belief formula.
1. For $i$ from 1 to $t$ do,
   (a) Set $\varphi \leftarrow \bigwedge_{f\in\mathcal{P}}$ AE-STRIPS-Fluent-SLAF($o_i$,$a_i$,$\varphi_f$).
   (b) Eliminate subsumed clauses in $\varphi$.
2. Return $\varphi$.

PROCEDURE AE-STRIPS-Fluent-SLAF($o$,$a$,$\varphi$)
$o$ conjunction of literals, $a$ action, $\varphi = (\neg f \vee \mathit{expl}_f) \wedge (f \vee \mathit{expl}_{\neg f}) \wedge \xi_f$ in *$f$-free form*.
1. Set $\mathit{expl}'_f = a^f \vee (a^{f\circ} \wedge \mathit{expl}_f))$.
2. Set $\mathit{expl}'_{\neg f} = a\neg f \vee (a^{f\circ} \wedge \mathit{expl}_{\neg f}))$.
3. If $f$ does not appear (positively or negatively) in $o$, then set $\xi'_f = \xi_f$.
4. Else, if $o \models f$ (we observed $f$), then
   (a) Set $\xi'_f \leftarrow \xi_f \wedge \mathit{expl}'_f$.
   (b) Set $\mathit{expl}'_f \leftarrow TRUE$ and $\mathit{expl}'_{\neg f} \leftarrow FALSE$.
5. Else (we observed $\neg f$),
   (a) Set $\xi'_f \leftarrow \xi_f \wedge \mathit{expl}'_{\neg f}$.
   (b) Set $\mathit{expl}'_f = FALSE$ and $\mathit{expl}'_{\neg f} = TRUE$.
6. Return $(\neg f \vee \mathit{expl}'_f) \wedge (f \vee \mathit{expl}'_{\neg f}) \wedge \xi'_f$

**Figure 4.** SLAF with always-executable STRIPS.

which we call *$f$-free form*. Figure 4 presents a complete algorithm for SLAF using this observation and form.

Now, we examine the size of the formula that results from filtering. A transition belief formula $\varphi$ in CNF is in *$f$-$k$-CNF* if every clause mentioning $f$ or $\neg f$ has at most $k$ literals. For example, $f \vee a^f$ is in $f$-2-CNF, but $a^f_1 \vee a^{\neg f}_2$ is in $f$-0-CNF. We also say that *$a, o$ determine $f$ in $\varphi$* if $\varphi \models a^f$ or $\varphi \models a^{\neg f}$ or $o \models f$ or $o \models \neg f$.

**Corollary 18** *Let $\varphi = \bigwedge_{f\in\mathcal{P}} \varphi_f$ be a fluent-factored transition belief formula, and $o$ a conjunction of literals. Then, Procedure AE-STRIPS-SLAF($\langle a, o\rangle$, $\varphi$) returns a fluent-factored transition belief formula $\varphi' \equiv Filter[o](Filter[a](\varphi))$ in time $O(|\varphi|)$. Further, if $\varphi$ is in $f$-$k$-CNF and $\varphi, a, o$ determine $f$, then $\varphi'$ is in $f$-1-CNF. Otherwise, $\varphi'$ is in $f$-$(k+1)$-CNF.*

Thus, our transition belief formula remains compact, if we know the effect of our action on $a$ in $\varphi$, or we observe $f$ frequently enough. For example, if we observe every fluent every 4 actions, then our transition belief state is always in 5-CNF, meaning that it is of size at most $O(n \cdot m^5)$ for $n$ fluents and $m$ actions (this is much better than the worst case which can be doubly-exponential in $n, m$).

## 4.2 STRIPS Actions

Assume that we allow actions to fail but we always observe such success and inexecutability. In both executable/inexecutable cases we learn something about the executability of the action under consideration. Unfortunately, this prevents factoring for the general case of actions, unless one of the conditions of section 3.3 holds. In the rest of this section we assume that either one of those conditions holds, or we accept the approximation offered by Corollary 12. Define

$$a^l_e \equiv \bigwedge_{G\in Bottom(\mathbb{P}_2)}(\neg a^{FALSE}_G \Rightarrow a^l_G)$$
$$a^{l\circ}_e \equiv \bigwedge_{G\in Bottom(\mathbb{P}_2)}(\neg a^{FALSE}_G \Rightarrow a^{l\circ}_G)$$

Let $\mathcal{B}(a) \equiv \bigwedge_{l\in\mathbb{P}_1}(a^l_e \Rightarrow l_{t+1}) \wedge \mathit{base}$.

**Corollary 19 (STRIPS-SLAF of a literal)** *Let $l$ be a literal in $L(\mathbb{P}_1, \mathbb{P}_2)$ and $a$ an action. If $l \in \mathbb{P}_1$, then*

$$Filter[a, OK](l) \equiv (l_{t+1} \Leftrightarrow (a^l_e \vee a^{l\circ}_e)) \wedge \neg a^{FALSE}_l \wedge \mathcal{B}(a)$$
$$Filter[a, \neg OK](l) \equiv l_{t+1} \wedge a^{failed}_l \wedge \mathit{base}$$

*If $l \notin \mathbb{P}_1$ (i.e., $l$ is an effect literal), then*

$$Filter[a, OK](l) \equiv l \wedge \neg a^{FALSE}_{TRUE} \wedge \mathcal{B}(a)$$
$$Filter[a, \neg OK](l) \equiv l \wedge a^{failed}_{TRUE} \wedge \mathit{base}$$

Now we replace Procedure Fluent-SLAF in Figure 3 with Procedure STRIPS-Fluent-SLAF of Figure 5.

PROCEDURE STRIPS-Fluent-SLAF($o$,$a$,$\varphi$)
$o$ conjunction of literals, $a$ action, $\varphi$ fluent.
1. If $l \in \mathbb{P}_1$, then
   (a) If $o \models OK$, then return
   $(l_{t+1} \iff (a^l_e \vee a^{l\circ}_e)) \wedge \neg a^{FALSE}_l \wedge \mathcal{B}(a)$.
   (b) ($o \models \neg OK$) Return $l_{t+1} \wedge a^{failed}_l \wedge \mathit{base}$.
2. ($l \in \mathbb{P}_1$) If $o \models OK$, then return $l \wedge \neg a^{FALSE}_{TRUE} \wedge \mathcal{B}(a)$.
3. Return $l \wedge a^{failed}_{TRUE} \wedge \mathit{base}$.

**Figure 5.** SLAF with STRIPS actions, observing success/failure.

## 4.3 Conditional Effects

A similar formula to the one above holds for the general case of deterministic actions (possibly conditional). We assume that $a$ has preconditions using the propositions in $\{l^1, ..., l^k\}$.

**Theorem 20** *Filtering for a literal $l \in \mathbb{P}_1$ satisfies*

$$Filter[a, OK](l) \equiv (l^e_{t+1} \Rightarrow$$
$$\bigvee_{\substack{G \in Bottom(\mathbb{P}_2, \{l^1, ..., l^k\}) \\ G \models l^p}} (a^{l^e}_G \wedge \bigwedge_{j\leq k}((a^{l^j}_G \Rightarrow l^j_{t+1}) \wedge (a^{\neg l^j}_G \Rightarrow \neg l^j_{t+1}))))$$

## 5 Conclusions

We presented general principles and algorithms for learning and filtering in partially observable domains. Some of our results guarantee polynomial-time filtering of transition belief states indefinitely. In particular, STRIPS domains in which actions are always executable (or when the preconditions for those actions are known) can be learned in polynomial time, if fluents are observed frequently enough.

We expect our algorithms to generalize to action schemas, where actions are parametrized in various ways (e.g., objects on which they operate, and numbers that modify the extent of the action). We plan to explore this direction in the future, as well as extending this work to agents that have a prior distribution, knowledge, or preference over the possible worlds or the actions' effects.

## REFERENCES


[1] Eyal Amir and Stuart Russell, 'Logical filtering', in *Proc. Eighteenth International Joint Conference on Artificial Intelligence (IJCAI '03)*, pp. 75–82. Morgan Kaufmann, (2003).

[2] Scott Benson, 'Inductive learning of reactive action models', in *Proceedings of the 12th International Conference on Machine Learning (ICML-94)*, (1995).

[3] Lonnie Chrisman, 'Abstract probabilistic modeling of action', in *Proc. National Conference on Artificial Intelligence (AAAI '92)*. AAAI Press, (1992).

[4] Richard Fikes, Peter Hart, and Nils Nilsson, 'Learning and executing generalized robot plans', *Artificial Intelligence*, **3**, 251–288, (1972).

[5] Yolanda Gil, 'Learning by experimentation: Incremental refinement of incomplete planning domains', in *Proceedings of the 11th International Conference on Machine Learning (ICML-94)*, pp. 10–13, (1994).

[6] Brian Hlubocky and Eyal Amir, 'Knowledge-gathering agents in adventure games', in *AAAI-04 Workshop on Challenges in Game AI*. AAAI Press, (2004).

[7] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra, 'Planning and acting in partially observable stochastic domains', *Artificial Intelligence*, **101**, 99–134, (1998).

[8] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore, 'Reinforcement learning: a survey', *Journal of Artificial Intelligence Research*, **4**, 237–285, (1996).

[9] Michael Kearns, Yishay Mansour, and Andrew Y. Ng, 'Approximate planning in large pomdps via reusable trajectories', in *Proceedings of the 12th Conference on Neural Information Processing Systems (NIPS'99)*, pp. 1001–1007. MIT Press, (2000).

[10] Fangzhen Lin and Ray Reiter, 'How to Progress a Database', *Artificial Intelligence*, **92**(1-2), 131–167, (1997).

[11] Michael L. Littman, *Algorithms for sequential decision making*, Ph.D. dissertation, Department of Computer Science, Brown University, 1996. Technical report CS-96-09.

[12] R. Andrew McCallum, 'Instance-based utile distinctions for reinforcement learning with hidden state', in *Proceedings of the 12th International Conference on Machine Learning (ICML-95)*. Morgan Kaufmann, (1995).

[13] Nicolas Meuleau, Leonid Peshkin, Kee-Eung Kim, and Leslie Pack Kaelbling, 'Learning finite-state controllers for partially observable environments', in *Proc. Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI '99)*. Morgan Kaufmann, (1999).

[14] Andrew Y. Ng and Michael Jordan, 'Pegasus: A policy search method for large mdps and pomdps', in *Proc. Sixteenth Conference on Uncertainty in Artificial Intelligence (UAI '00)*, pp. 406–415. Morgan Kaufmann, (2000).

[15] Tim Oates and Paul R. Cohen, 'Searching for planning operators with context-dependent and probabilistic effects', in *Proc. National Conference on Artificial Intelligence (AAAI '96)*, pp. 863–868. AAAI Press, (1996).

[16] Hanna M. Pasula, Luke S. Zettlemoyer, and Leslie Pack Kaelbling, 'Learning probabilistic relational planning rules'. AAAI Press, (2004).

[17] L. R. Rabiner, 'A tutorial on hidden Markov models and selected applications in speech recognition', *Proceedings of the IEEE*, **77**(2), 257–285, (February 1989).

[18] Matthew D. Schmill, Tim Oates, and Paul R. Cohen, 'Learning planning operators in real-world, partially observable environments', in *Proceedings of the 5th Int'l Conf. on AI Planning and Scheduling (AIPS'00)*, pp. 246–253. AAAI Press, (2000).

[19] Xuemei Wang, 'Learning by observation and practice: an incremental approach for planning operator acquisition', in *Proceedings of the 12th International Conference on Machine Learning (ICML-95)*, pp. 549–557. Morgan Kaufmann, (1995).

[20] Mary-Anne Winslett, *Updating Logical Databases*, Cambridge University Press, 1990.

# Paper Session III

August 23, 15:30 - 17:30

- **Building Polygonal Maps from Laser Range Data**, J. Latecki, R. Lakaemper, X. Sun, D. Wolter
- **Hierarchical Voronoi-based Route Graph Representations for Planning, Spatial Reasoning and Communication**, J. O. Wallgrün
- **Schematized Maps for Robot Guidance**, D. Wolter, K-F Richter
- **How can I, robot, pick up that object with my hand**, A. Morales, P.J. Sanz, A.P. del Pobil

*

# Building Polygonal Maps from Laser Range Data

**Longin Jan Latecki** [1] and **Rolf Lakaemper** [2] and **Xinyu Sun** [3] and **Diedrich Wolter**[4]

**Abstract.** This paper presents a new approach to the problem of building a global map from laser range data, utilizing shape based object recognition techniques originally developed for tasks in computer vision. In contrast to classical approaches, the perceived environment is represented by polygonal curves (polylines), possibly containing rich shape information yet consisting of a relatively small number of vertices. The main task, besides segmentation of the raw scan point data into polylines and denoising, is to find corresponding environmental features in consecutive scans to merge the polyline-data to a global map. The correspondence problem is solved using shape similarity between the polylines. The approach does not require any odometry data and is robust to discontinuities in robot position, e.g., when the robot slips. Since higher order objects in the form of polylines and their shape similarity are present in our approach, it provides a link between the necessary low-level and the desired high-level information in robot navigation. The presented integration of spatial arrangement information, illustrates the fact that high level spatial information can be easily integrated in our framework.

## 1 INTRODUCTION

The problems of self-localization and robot mapping are of high importance to the field of mobile robotics. Robot mapping describes the process of acquiring spatial models of physical environments through mobile robots. Self-localization is the method of determining the robot's position with the robot's internal spatial representation. The central method required is a matching of sensor data, which - in the typical case of a laser range finder as the robot's sensor - is called scan matching. Whenever a robot needs to cope with unknown or changing environments, localization and mapping have to be carried out simultaneously, this technique is called SLAM (Simultaneous Localization and Mapping). To attack the problem of mapping and/or localization, mainly statistical techniques are used (Thrun [15], Dissanayake et al. [3]), e.g., the extended Kalman filter, a linear recursive estimator for systems described by non-linear process models and/or observation models, are the basis for most current SLAM algorithms. Bayesian rules build the foundation of the models employed. For localization, often partially observable Markov decision processes (POMDP) are utilized.

The robot's internal geometric representation forms the basis for these techniques. It is build atop of the perceptual data read from the laser range finder (LRF). Typically, either the planar location of reflection points read from the LRF is used directly as the geometric representation, or simple features in the form of line segments or corner points are extracted (Cox [2]; Gutmann and Schlegel [5]; Gutmann [7]; Röfer [14]). Although robot mapping and localization techniques are very sophisticated they do not yield the desired performance. We observe that these systems use only a very primitive geometric representation. As the internal geometric representation is a foundation for the sophisticated techniques in localization and mapping, shortcomings on the level of the geometric representation affect the overall performance. The main goal of this paper is the introduction of an elaborate and cognitively motivated geometric representation and a reasoning formalism for robot mapping. A successful geometric representation must result in a much more compact representation than uninterrupted perceptual data, but must neither discard valuable information nor imply any loss of generality. We claim that the representation proposed in this paper, namely polygonal curves or polylines, representing parts of object surfaces being obtained from segmented scans, fulfills these demands. The relation among the objects is based on shape similarity and on qualitative arrangement information. Representing the passable space explicitly by means of shape is not only adequate for mapping applications but also helps to bridge the gap from metric information needed to topological knowledge due to the object centered perspective offered. Moreover, an object-centered representation is a crucial building block in dealing with changing environments, as such a representation allows us to separate the partial changes from the unchanged parts. In this paper we focus on incremental building of the object representation.

There exist approaches to map building that apply more sophisticated geometric method to scan data, e.g., Forsberg et al. [4] and Jensfelt and Christensen [9]. However, they focus on extraction of linear structures only, why we not only consider extraction of polygonal structures but also on similarity of polygonal structures. The similarity of polygonal structures is a driving force of our approach.

## 2 MAP BUILDING PROCESS

Map Building is the process of memorizing perceived objects and features the robot has passed by, merging corresponding objects in consecutive scans of the local environment. The robot's internal spatial representation is referred to as a map, in the case of a feature based spatial representation it is commonly referred to as a feature map [15]. A key challenge in map building is to match a local sensor reading against the global map. Multiple problems occur, e.g., the noise of the data perceived must be filtered in a way to obtain the required features, and the correspondence between perceived objects must be found on the basis of the filtered (visual) features, since additional information, i.e., odometry, has proven to be inaccurate. This excludes the possibility of simply superimposing consecutive scans on top of one another, as can be seen in the example shown in Figure 1(a). It shows the effects of accumulated errors in rotation and distance measure, because the geometrical properties of the envi-

[1] Temple University, Philadelphia, USA, email: latecki@temple.edu
[2] email: lakamper@temple.edu
[3] email: xysun@euclid.math.temple.edu
[4] University of Bremen, Bremen, Germany email: dwolter@informatik.uni-bremen.de

ronment indicated by the LRF purely interpreted in connection with odometry increasingly differ from reality, resulting in large displacements of the perceived objects and block-like representations in the global map. Odometry is designed to record the distance the robot has traveled and the rotational angle the robot has turned with respect to the starting position. However, odometry makes unreliable recording on a large scale.

Using alignment based on shape similarity, we can construct a significantly better map by correcting the errors of the odometry or even discarding the recording of the odometry completely. The alignment is computed using stable objects in the map to calculate the current position of the robot. Assuming that some objects in two consecutive scans are not moving, we align the stationary objects in the second scan to the corresponding objects in the first scan. The robot's movement and its position defining the global position of the scanned environment is achieved by the resulting movement and rotational angle computed by the alignment. An example global map computed using our alignment algorithm is shown in Figure 1(c). The scans are aligned iteratively, then superimposed on each other without any information from the odometry. The row scan data alignment based on robot's odometry is shown in Figure 1(a). The quality of the map in (c) is much better, and the objects can be easily identified. To have a fair comparison, the map in (c) should be compared to the map in (b). The map (b) is obtained from (a) with a simple algorithm that corrects significant rotation errors of the odometry by finding rotation angles that maximize the proximity of long lines and rotated according the scans accordingly. Although the improvement from (b) to (c) is significant, we can still find some fuzziness in certain areas of (c). This is due to the nature of the perceived data, which introduces noise in several ways:

1. Scanning an object with a fractal or nonrigid shape. An example of such can be a plant in any office building. The locations of the scan points on such objects are mostly random, and the recordings of the scan points are not going to provide us useful position information, even if the robot is *not* moving.
2. Scanning objects too far away. Scanning objects far away inevitably will create more error than scanning objects close by, due to uneven ground or other movement/vibration effects. Aligning all objects in the scans, the error introduced by this defective data will accumulate and propagate.
3. Scanning moving objects. Objects are aligned under the assumption that only the robot moves, being the only source of change of the distance between them. But once there is a moving object in the scan, such an assumption is no longer true, and the alignment will not be reliable, especially when the moving object is near the scanner.

In the following we will address these problems, and discuss their solutions in our framework.

One shortcoming of the recursive alignment process is that when errors occur, they will not be corrected nor eliminated. In fact, they are most likely to be accumulated and propagated as show in Figure 1(c). Another drawback is that each scan still exists independently of the others, i.e., we are not creating *a global map* composed of objects in the common sense of the word, but composed of a set of unrelated polylines. Hundreds of laser scans are printed on paper and human eyes can easily identify objects from the printout, but these scans cannot be used by robot localization because these objects are not present in the internal robot representation.

Hence a process is needed that can deal with these problems, and that can create a global map composed of just a few objects. We propose such a process that we call merging in this paper. In order to describe it, we need to introduce some notation.

A global map is built iteratively as the robot moves. We denote the scan and the global map at time $t$ by $S_t$ and $G_t$ respectively. Each scan $S_t$ and the global map $G_t$ is composed of polylines. We assume that the range data is mapped to locations of reflection points in the Euclidean plane, using a local coordinate system. These points are segmented into individual polylines with a simple heuristic: Traversing the reflection points following the order of the LRF, an object transition is said to be present wherever two consecutive points are further apart than a given distance threshold (20 cm in the case of our example map). The precise choice of the threshold is not crucial as subsequent processing accounts for differences. The obtained polygonal objects are further simplified to reduce the influence of noise (Section 3.1). We apply discrete curve evolution (DCE, described below) followed by least square fitting of line segments to obtain the simplified polyline objects.

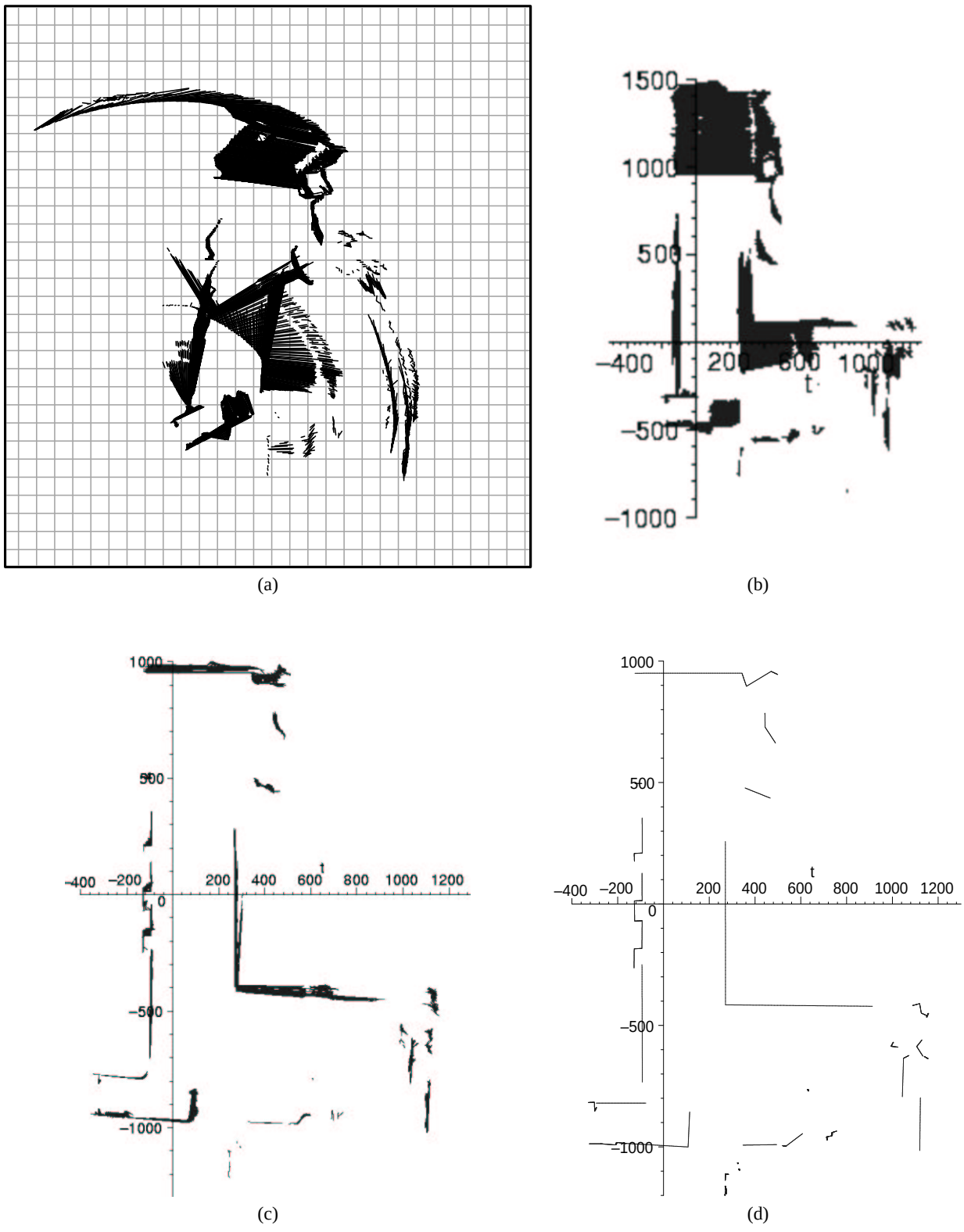
To create the global map $G$, we start with the first global map $G_1$ being equal to the first scan $S_1$. Assuming we have created the global map $G_{t-1}$ at time $t-1$, we use $G_{t-1}$ and $S_t$ to create $G_t$ in the following steps:

- We extract a virtual scan $\tilde{S}_{t-1}$ from the global map $G_{t-1}$. $\tilde{S}_{t-1}$ is the part of the global map to which the previous scan $S_{t-1}$ was merged. The virtual scan $\tilde{S}_{t-1}$ is a corrected version of the actual scan $S_{t-1}$. The noise correction was achieved by merging in the previous step of $S_{t-1}$ to the global map $G_{t-2}$.
- We use shape similarity (described in Section 3) to find the correspondence between objects in $S_t$ and $\tilde{S}_{t-1}$, which can be a many-to-many correspondence. Since $\tilde{S}_{t-1}$ is part of the global map $G_{t-1}$, we obtain the correspondence between $S_t$ and parts of the global map.
- We align polylines in $S_t$ to the corresponding polylines in $G_{t-1}$. We repeatedly apply the least squares method to find the optimal translation and rotation. The main difference in comparison to the standard approaches is that the corresponding points are limited to polylines with similar shape. Thus, we greatly reduce the problem of local minimum. This allows us to align $S_t$ to $G_{t-1}$ even if the robot displacment is large (see Figure 3).
- Finally, we merge the aligned scan $S_t$ to $G_{t-1}$ to create $G_t$. Merging an aligned new scan to the global map adds newly detected polylines in the surrounding area to the global map while not discarding the existing polylines no longer seen by the robot. The goal is to produce a global map composed of a few polylines. The details of merging are presented in Section 4.

Repeating this procedure iteratively for each new scan, we are able to create a global map that remembers all the objects and features along the path the robot has traveled. The result can be seen in Figure 1(d), which displays the final global map as a set of polylines. We used exactly the same raw input data for all four maps in Figure 1. This feature-based approach yields a compact representation, and most importantly, it represents the robot's surroundings by a single set of non-overlapping polylines that can be easily recognized and compared using shape similarity.

## 3 STRUCTURAL REPRESENTATION OF SHAPE

A first step in the presented approach is to extract shape information from data acquired by a laser range finder (LRF). Polygonal lines, termed polylines, serve in our approach as a fundamental building




(a) (b) (c) (d)

**Figure 1.** (a) A global map built using the information from odometry only. (b) A global map with significant rotation errors corrected. (c) A global map built using the proposed alignment only. (d) A real global map built using the proposed merging approach.

block. They already capture more context than other features typically employed in scan matching approaches (e.g., simple line segments or even uninterpreted data). The richness of perceivable shapes in a regular indoor scenario yields a more reliable matching than other feature-based approaches, as mixups in determining features are more unlikely to occur. At the same time, we are able to construct a compact representation for an arbitrary environment. However, we exploit even more context information than represented by a single polyline considering shape as a structure of polylines. This allows us with basically no extra effort to cope with environments displaying mostly simple shapes.

### 3.1 Grouping and Simplification of Polylines

Polylines extracted from raw scan data still carry all the information (and noise) retrieved by the sensor. To make the representation more compact and to cancel out noise, we employ a technique called Discrete Curve Evolution (DCE) introduced by Latecki & Lakämper [11, 12] which achieves these goals without losing valuable shape information. DCE is a context-sensitive process that proceeds iteratively. Though the process is context-sensitive, it is based on a local relevance measure for a vertex $v$ and its two neighbor vertices $u, w$[5]:

$$K(u, v, w) = |d(u, v) + d(v, w) - d(u, w)|$$

Hereby, $d$ denotes the Euclidean distance. The process of DCE is very simple and proceeds in a straightforward manner. The least relevant vertex is removed until this relevance measure exceeds a given threshold, thereby defining the level of polygon simplification. Consequently, as no relevance measure is assigned to end-points, they remain fixed. The choice of a specific simplification threshold is not crucial, refer to Figure 2 for results. Implementation of DCE can benefit from the observation that a polyline can be represented simultaneously as a double-linked list and a self-balancing tree which reflects the order of relevance measures. Thus, the overall complexity is $O(n \log n)$. Proceeding this way we obtain a cyclic, ordered vector of polylines.

### 3.2 Similarity of Polylines

Matching scans against the global map within the context of a shape based representation is naturally based on shape matching. This somehow revives a notion in Lu and Milos' fundamental work [13] *"scan matching is similar to model-based shape matching"* that so far has not received much attention. In the presented approach we adopt a shape matching originated from computer vision that has proven successful in the context of shape retrieval [12]. It may easily be adapted to our needs. The property of invariance to change of scale as often desired in computer vision approaches is not adequate in our domain and must be excluded.

To compute the similarity measure between two polygonal curves, we establish the best possible correspondence of maximal convex arcs, where a *convex arc* is a left- or right-arcuated arc. To achieve this, we first decompose the polygonal curves into maximal subarcs that are likewise bent. Since a simple one-to-one comparison of maximal arcs of two polylines is of little use, due to the fact that the curves may consist of a different number of such arcs and even similar shapes may have different small features, we allow for 1-to-1, 1-to-many, and many-to-1 correspondences of maximal arcs. The main idea here is that we have at least on one of the contours a maximal convex arc that corresponds to a part of the other contour composed of adjacent maximal arcs. The best correspondence, i.e., the one yielding the lowest similarity measure, can be computed using dynamic programming, where the similarity of the corresponding visual parts is as defined below. Using dynamic programming, the similarity between corresponding parts is computed and aggregated. The computation is described extensively in [12]. The similarity induced from the optimal correspondence of polylines $C$ and $D$ will be denoted $S(C, D)$.

Basic similarity of arcs is defined in tangent space, a multi-valued step function representing angular directions of line-segments only. This representation was previously used in computer vision, in particular in [1]. Denoting the mapping function by $T$, the similarity gets defined as follows:

$$S_a(C, D) = (1+(l(C)-l(D))^2) \int_0^1 (T_C(s)-T_D(s)+\Theta_{C,D})^2 ds$$

where $l(C)$ denotes the arc length of $C$, and the whole integral is over arc length. The constant $\Theta_{C,D}$ is chosen to minimize the integral (cp. [12]). Obviously, the similarity measure is a rather a dissimilarity measure as the identical curves yield 0, the lowest possible measure. This measure differs from the original work in that it is affected by an absolute change of size rather than a relative one (cp. [12]). It should be noted that this measure is based on shape information only, neither the arcs' position nor orientation are considered. This is possible due to the large context information of polylines; position and orientation will be accounted for when computing the actual matching.

### 3.3 Matching of Polylines

Computing the actual matching of two structural shape representations extracted from scan and map is performed by finding the *best* correspondence of polylines which respects the cyclic order. We must also take into account that (a) not all polylines may get matched as the features' visibility changes and (b) that due to segmentation noise (cp. section 3.1) it is not necessarily a one-to-one correspondence. Furthermore, any correspondence of polylines induces an alignment of the polylines which constraints the scan's origin. For example, matching a polyline perceived in front of the robot to a polyline that is based on the estimated position of the robot to its right, the robot must have turned right. Hence, we demand all induced alignments to be alike. To enable efficient computation of the matching, an estimation of the induced alignment is required. It can either be derived from odometry (if available) or simply reflect the assumption that the robot has not moved. We stress that we do not use any odometry data in our approach.

Let us assume that $\vec{B} = (B_1, B_2, \ldots, B_b)$ and $\vec{B'} = (B'_1, B'_2, \ldots, B'_{b'})$ are two cyclic ordered vectors of polylines. Denoting correspondence of $B_i$ and $B'_j$ by relation $\sim$, the task can be formulated as minimization as follows.

$$\sum_{(\vec{B}_i, \vec{B'}_j) \in \sim} S(\vec{B}_i, \vec{B'}_j) + C \cdot (2|\sim| - |\vec{B}| - |\vec{B'}|) \stackrel{!}{=} \min$$

Hereby, $C$ denotes a penalty for not matching a polyline. This is necessary, as not establishing any correspondence would otherwise yield the lowest possible similarity 0. The similarity measure $S$ is composed of the shape similarity measure presented in Section 3.2 and the alignment measure considering the difference between the alignment induced by the corresponding polylines' and the estimated one.

[5] Context is respected in the course of simplification as vertices' neighborhood changes.

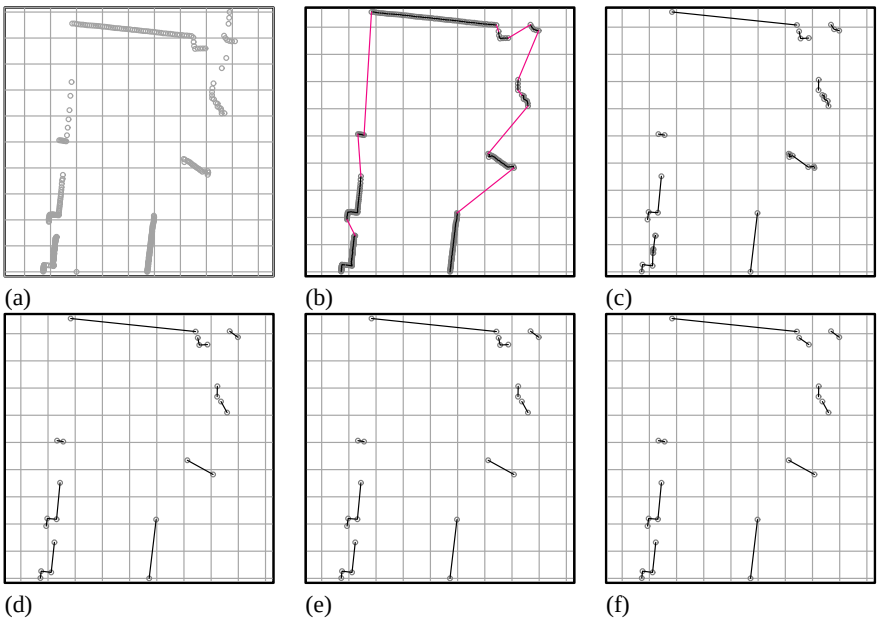



**Figure 2.** The process of extracting polygonal features from a scan. Raw scan points (a) are grouped to polylines (b), then simplification by means of DCE is performed. Figures (c) to (f) show various simplification levels (1,5,10, and 15 respectively) highlighting that the precise choice the simplification level is not critical for shape information obtained. The grid denotes 1 meter distance.

Considering alignments becomes necessary when many featureless shapes, e.g., chairs' legs, need to be tracked.

We use an adequate extension of the dynamic programming scheme to compute the best correspondence. The extension regards the ability to detect even 1-to-many and many-to-1 correspondences of polylines and results in a linear extra effort such that the overall complexity is $O(n^3)$. Observe that $n$ is very low, it was around 10 for our example map, since this is the number of polylines in a scan. The outlined matching is powerful enough to track shapes even if no odometry information is available and the robot has traveled a remarkable distance between two consecutive scans. Figure 3 shows the polyline correspondence computed by our method, where the corresponding polylines from two different scans are connected by dashed lines. As can be observed, approaches based on the nearest point rule fail in this case.

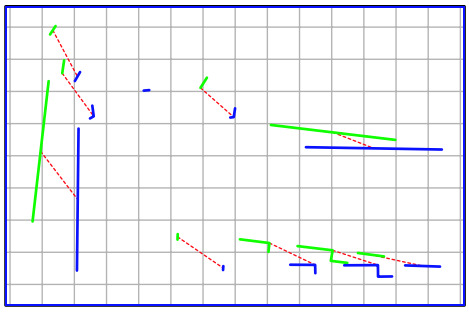


**Figure 3.** Exemplary results of the shape based matching for two scans (green and blue polylines; the grid denotes 1m distance).

## 4 MERGING

Merging an aligned new scan to the global map adds newly detected features in the surrounding area to the global map while not discarding the existing features no longer seen by the robot. It produces a global map composed of polylines.

It is common that portions of the same object may be perceived as several independent objects in a single scan. Such a phenomenon can happen when there is another object blocking the view, or simply because of the angle and the distance from which the object is viewed. During the merging process, we need to accurately identify objects in each scan, look for their corresponding objects in the existing global map, calculate the updated position of the object, remove moving objects, remove areas where there are objects with nonrigid shape, and more importantly, merge several objects into a single object whenever possible.

The main idea of the presented merging process is to simulate the robot laser scanner to merge the aligned scan $S_t$ to the global map $G_{t-1}$. As output we obtain an updated global map $G_t$. The simulated laser rays emanate from the current robot pose and follow the parameters of the robot scan device. This means for our data that the simulated rays move counter-clockwise in steps of $0.5^\circ$ and cover the angle of $180^\circ$. If a ray intersects a polyline in $S_t$, we select the closest intersection point to the robot pose, which is called a simulated scan point (ssp). We are creating at most 361 ssps. Furthermore, we add all end points of polylines in $S_t$ to the list of newly created scan points with proper ordering (since we know between which scan rays they lie). From each ssp $p_s$ in $S_t$, we find the closest point $p_g$ on $G_{t-1}$. If the distance from $p_s$ to $p_g$ is below a certain threshold, then we take a weighted average of the two points to create a new point for the new global map $G_t$. We use a larger weight for the point $p_g$ in $G_{t-1}$, since we have more confidence about its position. If the closet point cannot be found within a certain distance, the ssp $p_s$ will be a new point on the new global map.

Since we want every polyline vertex in a global map $G_{t-1}$ to have a corresponding point in the new global map, we have the following rule. If two consecutive ssps $p_{s_1}$ and $p_{s_2}$ have two corresponding closest points $p_{g_1}$ and $p_{g_2}$ in $G_{t-1}$, and there is a vertex $v$ between $p_{g_1}$ and $p_{g_2}$, we create a new ssp $p_{s_v}$ in $S_t$ that is the closest point

to $v$ in the part of $S_t$ between $p_{s_1}$ and $p_{s_2}$. Then we take a weighted average of the two points $p_{s_v}$ and $v$ to create a new point for the new global map $G_t$. This step is illustrated in Figure 4.

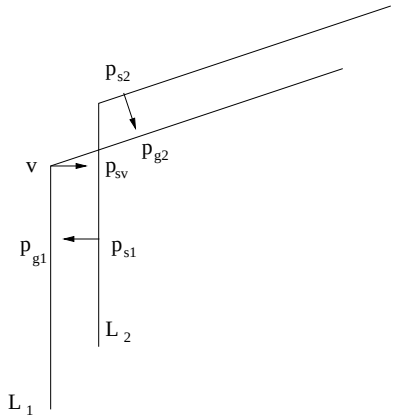



**Figure 4.** $v$ is a vertex of a polyline $L_1$ of the global map $G_{t-1}$. $v$ is between the closest points $p_{g_1}$ and $p_{g_2}$ to two consecutive simulated scan points (ssps) $p_{s_1}$ and $p_{s_2}$ on polyline $L_2$ in $S_t$. In such case, we create a new ssp $p_{s_v}$ in the scan $S_t$ as the closest point to $v$ on a polyline $L_2$ of the scan $S_t$. Then we take a weighted average of the two points $p_{s_v}$ and $v$ to create a new point for the new global map $G_t$.

The next step is to create a set of ordered polylines from the newly created points (ncps) in the global map $G_t$. The following facts guide this process:

1. All ncps in the global map $G_t$ are ordered. They either inherited their order from the simulated scan rays or where sorted in between two ssps (who inherited their order from the simulated scan rays).
2. All consecutive ncps whose predecessors belonged to the same polyline either in $G_{t-1}$ or in $S_t$ are classified as belonging to the same polyline.
3. The parts of polylines in $G_{t-1}$ that are not predecessors of any ncps are integrated in $G_t$ as separate polylines.

First we connect all consecutive (in scan order) ncps in the new global map that belong to the same polyline. We further merge consecutive polylines $P_1$ and $P_2$ to a single polyline $Q$ if the last vertex of $P_1$ and the first vertex of $P_2$ had predecessors in the same polyline either in $S_t$ or in $G_{t-1}$. The motivation for this step is that the polyline $Q$ is likely to represent a single object in $G_t$ (created from $S_t$ and $G_{t-1}$), since whenever two consecutive points are in the same object either in $S_t$ or in $G_{t-1}$, they can be classified as belonging to the same object. This rule implies that separate portions of the same object will be connected if they are ever detected as points of the same object. The only exception to this rule is a bifurcation, which may be caused by dynamic object, e.g., a moving door. To resolve bifurcations, we create two disjoint polylines in a global map.

The created polylines in the new global map $G_t$ may not be properly ordered. A polyline $P$ is *properly ordered* if the order of its vertices is constant with the arc length distance to its first vertex $a$ i.e., vertex $v$ proceeds $w$ if the arc length distances satisfy $d_P(a,v) < d_P(a,w)$. We apply recursively the following simple rule to obtain properly ordered polylines: Let $u, v, w$ be three consecutive vertices in a polyline $P$, if the inner angle at $v$ is less then a certain threshold (which is $20^o$ in our case), then vertex $v$ is removed. This rule not only makes polylines properly ordered, but also removes scan artifacts due to sensor noise or due to objects with fine shape features like plants. This rule is justified by the fact that sharp inner angles cannot be created by three consecutive laser rays unless they hit an object with fine shape.

For objects with fine shape, we obtain a dense sequence of sharp inner angles. Since this rule would remove them completely, these kind of objects requires a special treatment. Such objects have different reflections patters for different scans, thus, their shape may change from scan to scan. These object are not considered in the merging process, but they are placed on the global map after merging. To (temporarily) remove objects, we use a bounding box around them, called "eraser box", to erase the areas where the perceived data is highly unreliable. We create an eraser box in a scan by creating a bounding rectangle containing all consecutive vertices in each polyline of $S_t$ with inner angles that are less than $20^\circ$. If such a sharp angle is span by three consecutive scan points, it is very unlikely that these points result from scan readings of a real object with a stable shape. Thus, such sharp angles indicate either noisy readings or an object with fuzzy shape, like plants. The processing of eraser boxes is composed of three main steps: (1) for each scan we identify eraser boxes, (2) we transfer the eraser boxes to the global map, and (3) we merge overlapping eraser boxes in the global map to a single eraser box that contain them. We need these three steps, because the data in a small area may appear to be reliable in a given scan, but appears highly unreliable in most of the other scans. Thus, we mark the whole area unreliable regardless of the quality of any single scan. The data inside the eraser boxes on the global map is not used for further iterations of building the global map.

In order to create stable global maps, we can only merge perceived stationary objects, i.e., we need to detect moving objects and remove them from actual scans before merging. To remove moving objects, we use the result of shape matching to identify corresponding objects in scan $S_{t-1}$ and $S_t$, say $O_{t-1}$ is matched to $O_t$. We can also easily remember in the process of merging that $O_{t-1}$ is merged into the object $M_{t-1}$ in $G_{t-1}$. Since $O_t$ and $M_{t-1}$ represent the same real object, their distance after alignment should be very small. If this is not the case, the object $O_t$ is moving, in which case we need to backtrack to its first appearance in the scans, remove it, and redo the merging process again to remove any unwanted effects it may have caused.

Finally, we apply DCE and least square fitting to create a new set of simplified polylines. An example result is shown in Fig. 1(d).

## 5 CONCLUSIONS

We have presented a comprehensive geometric model for robot mapping based on shape information. Shape matching has been tailored to the domain of scan matching. The matching is powerful enough to disregard pose information and cope with significantly differing scans. This improves performance of today's scan matching approaches dramatically. In this paper we concentrate on solving geometric problems related to global map building, and do not include any statistical methods, to demonstrate the power of the proposed geometric approach. However, we are aware that statistical methods are needed to guarantee robust performance. A suitable extension of our approach to include statistical methods will be presented in a separate paper.

## ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grant INT-0331786 and the grant 16 1811 705 from Temple University Office of the Vice President for Research and Graduate Studies. It was carried out in collaboration with the SFB/TR 8 Spatial Cognition, project R3 [Q-Shape] support by the Deutsche Forschungsgemeinschaft (DFG). All support is gratefully acknowledged.

## REFERENCES

[1] M. Arkin, L. P. Chew, D. P. Huttenlocher, K. Kedem, and J. S. B. Mitchell (1991). An efficiently computable metric for comparing polygonal shapes. *IEEE Trans. PAMI*, 13:209–206.

[2] Cox, I.J., Blanche – An experiment in Guidance and Navigation of an Autonomous Robot Vehicle. *IEEE Transaction on Robotics and Automation* 7:2, 193–204, 1991.

[3] Dissanayake, G. ,Durrant-Whyte, H., and Bailey, T., A computationally efficient solution to the simultaneous localization and map building (SLAM) problem. *ICRA'2000 Workshop on Mobile Robot Navigation and Mapping*, 2000.

[4] J. Forsberg, U. Larsson, and A. Wernersson. Mobile Robot Navigation using the Range-Weighted Hough Transform. *IEEE Robotics and Automation Magazine* 21, pp. 18-26, 1995.

[5] Gutmann, J.-S., Schlegel, C., AMOS: Comparison of Scan Matching Approaches for Self-Localization in Indoor Environments. *1st Euromicro Workshop on Advanced Mobile Robots (Eurobot)*, 1996.

[6] Gutmann, J.-S. and Konolige, K., Incremental Mapping of Large Cyclic Environments. *Int. Symposium on Computational Intelligence in Robotics and Automation (CIRA'99)*, Monterey, 1999.

[7] Gutmann, J.-S., Robuste Navigation mobiler System, *PhD thesis*, University of Freiburg, Germany, 2000.

[8] D. Hähnel, D. Schulz, and W. Burgard. Map Building with Mobile Robots in Populated Environments, *Int. Conf. on Int. Robots and Systems (IROS)*, 2002.

[9] P. Jensfelt and H. I. Christensen. Pose Tracking Using Laser Scanning and Minimalistic Environmental Models, *IEEE Trans. on Robotics and Automation*, 2001.

[10] B. Kuipers. The Spatial Semantic Hierarchy, *Artificial Intelligence* 119, pp. 191–233, 2000.

[11] L. J. Latecki and R. Lakämper (1999). Convexity Rule for Shape Decomposition Based on Discrete Contour Evolution. *Computer Vision and Image Understanding* 73:441–454.

[12] L. J. Latecki and R. Lakämper (2000). Shape Similarity Measure Based on Correspondence of Visual Parts. *IEEE Trans. Pattern Analysis and Machine Intelligence* 22:1185-1190.

[13] Lu, F., Milios, E. (1997). Robot Pose Estimation in Unknown Environments by Matching 2D Range Scans. *Journal of Intelligent and Robotic Systems* 18:3 249–275

[14] Röfer, T., Using Histogram Correlation to Create Consistent Laser Scan Maps . *IEEE Int. Conf. on Robotics Systems (IROS)*. Lausanne, Switzerland, 625–630, 2002.

[15] Thrun, S. (2002). Robot Mapping: A Survey, In Lakemeyer, G. and Nebel, B. (eds.): *Exploring Artificial Intelligence in the New Millenium*, Morgan Kaufmann, 2002.

# Hierarchical Voronoi-based Route Graph Representations for Planning, Spatial Reasoning, and Communication

**Jan Oliver Wallgrün**[1]

**Abstract.** In this paper we propose a spatial representation approach for a mobile robot operating in an office-like indoor environment which is intended to provide an interface between low-level information required for navigation and abstract information required for high-level symbolic reasoning about routes. The representation is based on a route graph [16] that links navigational decision points via edges corresponding to route segments. We describe a particular route graph representation that is derived from the generalized Voronoi diagram of the environment and enables the robot to incrementally construct the representation autonomously. Since the Voronoi-based route graph still reflects irrelevant features of the environment, our proposed representation is a hierarchical structure consisting of route graph layers representing the environment at different levels of granularity. It is shown how the more abstract layers can be derived from the original route graph by using relevance measures to assess the significance of the vertices. We provide examples of how planning, spatial reasoning, and communication can benefit from this kind of representation.

## 1 Introduction

In the context of mobile robot control systems, a crucial step is to decide in which way spatial information about the robot's environment required to solve different subtasks like path planning, path execution, localization, spatial reasoning, etc. should be stored. Since any truly autonomous mobile robot will have to be able to construct and maintain its model of the environment on its own based on observations, the representation will have to bridge the gap from low-level sensor data to entities needed for high-level reasoning.

Representations used in current mobile robot systems can be grouped into two main classes: Metric approaches [12, 10, 14] rely on an absolute coordinate system superimposed onto the environment to specify position and orientation of spatial entities. Topological representations on the other hand represent the environment by a graph structure that explicitly stores spatial relations like adjacency or connectivity between the entities represented by the vertices [7, 3].

In this paper we describe a so-called route graph representation (a concept introduced in [16]) which is a special kind of topological map in which the graph structure describes qualitatively different routes through the environment. The vertices in this representation correspond to navigational decision points, while the edges correspond to route segments connecting the decision points. The representation and the involved procedures are intended to serve as a navigation and mapping module within a hybrid control architecture combining reactive and deliberative components.

Our particular route graph representation is based on the generalized Voronoi diagram (GVD) which is a retraction of the free parts of the working space onto a network of one-dimensional curves reflecting the connectivity of free space (see figure 1). The GVD allows us to derive a route network from information about the obstacle boundaries. The graph corresponding to the GVD, the generalized Voronoi graph (GVG), forms the core of our route graph representation and is annotated with additional information like relative positions of the vertices. While the resulting route graph can be used for navigation by applying a simple motion behavior to travel along edges until the next vertex is reached [3], it still contains parts that are caused by insignificant features of the environment like small niches or that result from noise in the sensor data and that are irrelevant for high-level reasoning. Therefore, we develop a way to deal with this problem by deriving more abstract route graphs from the original GVG employing measures to assess the relevance of the Voronoi vertices and the regions accessible by them. Based on the ability to abstract from the GVG, we propose a hierarchical organized multi-layer representation with the original GVG at the bottom level and layers containing route graphs representing the environment at different levels of granularity stacked on top of it. Corresponding features in adjacent layers are linked with each other allowing to switch to a finer or coarser level of granularity. We argue in favor of such a representation for a mobile robot system for application in office-like indoor scenarios showing how it is particularly well-suited to provide an interface between low-level navigational information and abstract information required for high-level planning, reasoning, and communication. Planning and spatial reasoning based on this representation can be performed in a hierarchical manner to make them more efficient.

The paper is structured as follows: Section 2 describes the Voronoi-based route graph representation scheme and briefly discusses advantages of the representation and important issues like incremental construction and localization. In section 3 the idea of a hierarchization of the Voronoi-based route graph is elaborated and relevance measures are proposed to derive such a representation from the original route graph. Section 4 presents planning and reasoning examples within this kind of representation and section 5 provides first results of the experimental evaluation of the described approach.

## 2 Voronoi-based route graph representation

The GVD is a generalization of standard Voronoi diagrams [1] that handles other geometric primitives, e. g. line segments [9, 6], instead of only point sites. It is also related to the idea of a shape's skeleton

[1] Department of Mathematics and Informatics, Universität Bremen, Germany email: wallgruen@informatik.uni-bremen.de

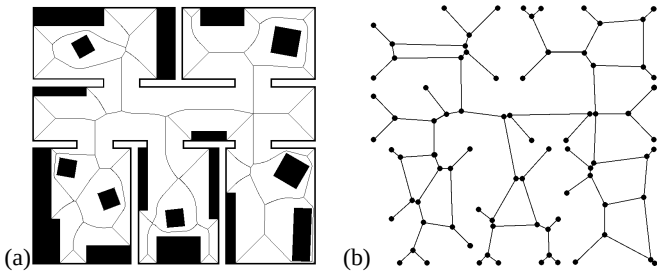



**Figure 1.** (a) The generalized Voronoi diagram (GVD) (fine lines) of a 2D environment, (b) the corresponding generalized Voronoi graph (GVG) with vertices placed at the position of the corresponding meet points for visualization.

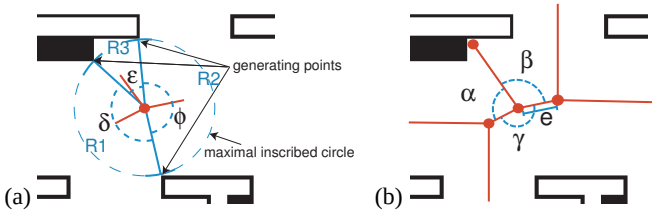



**Figure 2.** Different kinds of annotations to the GVG-based route graph: (a) Vertex signature containing the distance of the generating points (radius of the maximal inscribed circle) and angles between the connections to the generating points, (b) relative position information given by the angles between departing edges and the length of the edges.

introduced in [2] and has been first used in robotics as an intermediate representations to solve motion planning tasks given complete information about the working space of the robot (usually by providing a geometric description of the boundaries of the obstacles) [13, 8]. In the two-dimensional case it contains all points of free space that are center of maximal inscribed circles (circles that are maximally expanded without intersecting the obstacle boundaries) that touch the obstacle boundaries at at least two points. Figure 1a shows a simple two-dimensional environment and the corresponding GVD (fine lines) consisting of curves that intersect at meet points and end up in corners of the environment.

As described in [3] simple motion behaviors to follow a Voronoi curve from one meet point to the next or get from a point aside the GVD to one on the GVD can be defined. Therefore, the generalized Voronoi graph (GVG), which is the graph corresponding to the GVD (see figure 1b) with vertices corresponding to meet or corner points and edges connecting vertices joined by Voronoi curves, is well-suited to serve as a topological map [3, 17]. The robot can travel from one vertex of the GVG to any other by repeatedly applying the Follow-Voronoi-Curve behavior while keeping track of the robot's position within the graph structure. The only additional information required for this is the clockwise order of departing edges for each vertex. It is also possible to construct this kind of representation autonomously during an exploration by tracing the Voronoi curves with the same behavior and registering the meet points encountered together with their departing edges.

However, in real world applications noisy and discrete sensor data together with the instability of the underlying GVD, which may show additional or missing vertices and edges if the boundary information changes slightly, require to store more information about the environment and more complex procedures to make the approach robust enough to be applicable. In addition, to construct a complete GVG by tracing every single Voronoi curve of the GVD is quite costly and can be avoided by making better use of the sensor data.

To overcome these problems, we developed a representation that extends the GVG with additional annotations to vertices and edges together with procedures for localization and incremental construction for a robot equipped with a laser range finder [15]. We will briefly describe this approach in the following.

## 2.1 The GVG-based representation

Besides the graph structure and the clockwise order of edges the following information is contained in our representation:

1. Vertices are labeled with a *signature* (see figure 2a) that contains the angles and distance to the generating points (those points where the maximal inscribed circle centered on the Voronoi meet point touches the obstacle boundaries). This information can be used to distinguish vertices.
2. The approximate relative position of vertices is represented by annotating the edges with the approximate distances and vertices with the approximate angles between departing edges (see figure 2b).
3. Every edge is annotated with a description of the Voronoi curve corresponding to this edge since this curve may deviate from the direct connection between the two vertices.
4. Additional information about which edges are traversable (not too close to obstacles) for the robot and which edges lead to still unexplored areas is also annotated to the graph structure.

In the remainder of the text we will for simplicity still call this extended representation a GVG.

## 2.2 Path planning, localization, and incremental construction

Path planning in this extended GVG can still be done by applying standard graph search techniques which now might employ the annotations, for instance to plan shortest paths. For execution of a planned path and for incrementally constructing this representation during exploration a robust localization within the graph structure is required.

Therefore, we developed a localization scheme that compares a *local* GVG computed from a single $360^\circ$ scan taken from the robot's current position with the (partially constructed) *global* GVG to identify correspondences taking into account the vertex signature, the relative position information, and odometry information about the last movement. The local GVG describes how the GVG looks in the neighborhood of the robot's current position as far as this can be derived from the sensor data available from this point. The comparison scheme searches for a similar subgraph instead of an exact isomorphism. This way robust localization can be achieved despite the instability caused by the imperfect sensors.

Incremental construction of the global representation is achieved by sequentially merging local GVGs computed for different positions starting with the local GVG computed for the start position of the robot. Thus, it makes maximal use of the information available from each observation avoiding unnecessary exploration steps. It uses the results of the comparison scheme employed for localization to identify parts of the local GVG that can be used to complement the global GVG. The idea of merging local GVGs to construct the global representation is illustrated in figure 3 that shows how a global GVG

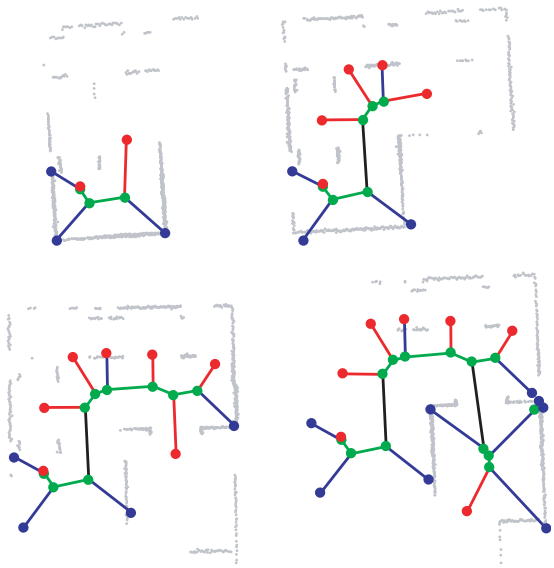


**Figure 3.** A sequence of growing global GVGs (starting with the local GVG of the robot's start position) constructed during the exploration of an unknown environment.

grows over time (see [15] for details on the localization and construction algorithms).

The GVG-based route graph is a rather compact representation of the essential spatial information required for navigation that can be augmented with additional (e. g. semantic) information if needed. The topological localization approach avoids special efforts required to keep the annotated metric information globally consistent since successful navigation is possible without globally consistent metric information. In addition, the Voronoi-based approach allows systematic exploration of an unknown environment by keeping track of which edges still need to be explored until the GVG is complete. Path planning within the GVG is more efficient than in most metric approaches because the representation only represents qualitative different routes.

## 3 Hierarchization of the Voronoi-based route graph

Voronoi-based route graphs as described in the previous section contain the information required for successful navigation. However, they also contain details not required for many tasks since not all meet points of the underlying GVD really correspond to decision points relevant for navigation. Some are caused by minor features of the environment like small dents or niches and some are merely the result of noise in the sensor data. For high-level reasoning, planning, and for communication issues a more abstract level of representation would be preferable if it is still linked to the detailed level required for actually acting within the environment. In addition, the relevant vertices are also those that are very stable and thus less likely to be missing in one of the local GVGs. Hence, localization can also benefit from a more abstract level of representation only containing the relevant vertices. Therefore, our goal is to construct a hierarchically structured multi-layer route graph representation that bridges from detailed navigational information to abstract high-level information about the environment and allows to efficiently reason in a hierarchical manner. Every layer of this representation consists of a route graph that models the environment at a certain level of granularity and its features are linked to those of the next higher and next lower layer in a way that allows to switch to a finer or coarser level. To derive more abstract layers from the original GVG a measure is required that assesses the relevance of individual Voronoi vertices for navigation and we will develop such a measure in the next section.

### 3.1 Assessing the relevance of Voronoi vertices

The GVG as described in section 2 is mainly an undirected Graph RG $= (V, E)$ (with additional annotations) containing only vertices of degree one (the corner vertices) or of degree three or higher (the inner vertices). As figure 2a illustrates, the lines connecting a Voronoi vertex $v$ with its generating points on the obstacle boundaries separate different parts of free space that are accessible via one of $v$'s departing edges. We will call each such area, that can be reached from $v$ without crossing one of the connecting lines again, a region $R_i^v$ of $v$. For each Voronoi vertex of degree $n$ there exist $n$ such regions. If $v$ is part of a cycle in the route graph, the regions corresponding to the two edges of $v$ that also belong to this cycle will be identical since the edges provide access to the same part of the environment, just from different directions.

How relevant a Voronoi vertex $v$ is for navigation depends directly on its regions. To be regarded as a decision point, at least three of $v$'s regions need to be significant enough to be judged as different continuations after arriving at this point. Otherwise, no real decision is to be made at this point. In addition, having two very significant regions can not make up for the third region being insignificant, e. g. a small niche in a corridor will not create a decision point in front of it irrespective of how long the corridor continues in both direction. Furthermore, having many insignificant regions will not make up for the third most significant one being still insignificant, e. g. two small niches on opposing sides of the corridor will not cause a decision point either.

Therefore, assuming we have a second measure called RSM (for region significance measure) that assesses the significance of each region of $v$, it makes sense to take the RSM value of $v$'s third most significant region as the relevance value of $v$. Using $\text{maxRSM}_3^v$ to denote the k-highest RSM value of a region of $v$, we thus define our Voronoi vertex relevance measure (VVRM) for all $v \in V$ with $\text{degree}(v) \geq 3$ as:

$$\text{VVRM}(v) = \text{maxRSM}_3^v.$$

We now need to define the RSM measure in a way that captures the notion of a significant region in the context of navigation in an indoor environment. The two major factors that we wanted to account for in our measure are the following: First, the distance from $v$ to the remotest goals belonging to the region should influence the significance of the region, since a region is clearly more significant if one can reach goals within it that are far from the current position. Second, we wanted to include the aspect of visibility to ensure that a region is assessed as less significant if most of it can be perceived from a larger area around $v$.

An additional constraint on our measure is that the significance values should be computable from the information contained in the GVG alone without referring to a geometric description of the boundaries of obstacles because this is the only information available in our mapping approach. Furthermore, cyclic regions should be treated as maximally significant so that cycles in the graph will never be split up when deriving a coarser route graph from the GVG (see section 4.1).

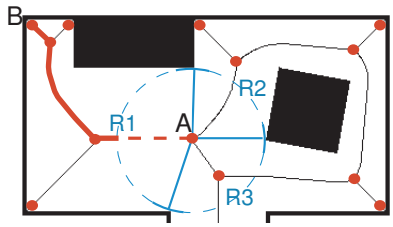



**Figure 4.** Computation of the RSM value for the region $R1$ to the left of vertex $A$: The length of the path to $B$ lying within the maximal inscribed circle (dashed line) is subtracted from the length of the complete path to $B$ yielding the length of the solid thick part of the path.

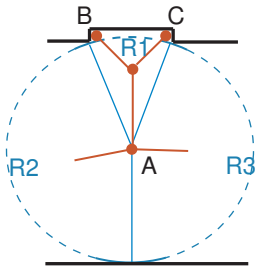



**Figure 5.** Region $R1$ of vertex $A$ is caused by a small niche in a wall. Since $A$ lies in a large area the distance along the GVD from $A$ to the corner vertices $B$ and $C$ is rather big. However, most part of the connection to these vertices lies within the maximal inscribed circle and will thus be subtracted during the RSM computation resulting in a small RSM value for region $R1$.

Hence, we define the RSM measure as follows:

1. $\text{RSM}(R_i^v) = \infty$, if $v$ and the departing edge corresponding to $R_i^v$ belong to a cycle in the route graph.
2. Otherwise, the shortest paths from $v$ to the corner vertices belonging to $R_i^v$ in terms of the distance along the GVD are considered. As illustrated in figure 4, it is determined for which corner vertex the length of this path minus the length of the part of this path lying within the maximal inscribed circle of $v$ is maximal, and this is returned as the value $\text{RSM}(R_i^v)$.

In the non-cyclic case the distance of the furthest corner vertex contained in the region is used to measure how far the robot could travel into this region. The subtraction of the length of the part that lies in the maximal inscribed circle of $v$ introduces the notion of visibility as mentioned above. For instance in figure 5 the small niche that causes Voronoi vertex $A$ in a wide hallway will be assessed as insignificant since the most part of the path to $B$ or $C$ is contained within the maximal inscribed circle centered on $A$ meaning most parts of the corresponding region are visible from every point within this circle.

Given the complete global GVG the individual RSM values for a vertex $v$ can be easily computed by applying a slightly modified version of Dijkstra's single source shortest path algorithm [5]. Picking the 3rd highest value then yields $\text{VVRM}(v)$. The modifications have to ensure that

- only the edge $e_{R_i^v}$ departing from $v$ into the region $R_i^v$ is considered connected to $v$ when relaxing the edges of the start vertex in the first step, and
- cycles leading back to $v$ via a different edge than $e_{R_i^v}$ will be detected and a value of $\infty$ will be returned.

With the worst-case time complexity of the dominating shortest path algorithm being $O(|E| + |V| \log |V|)$, we end up with a total time complexity for computing $\text{VVRM}(v)$ with $\text{degree}(v) = n$ of $O(n(|E| + |V| \log |V|))$.

The computation of the relevance values as described above assumes that a complete GVG is available. However, when we want to compute the relevance values for the vertices in a local GVG or for a still only partially constructed global GVG, we have to adapt this approach: We then treat all vertices that mark the boundary of the explored area like corner vertices and compute the relevance values as before. Doing this, all RSM values for regions which contain edges marked as unexplored and which do not correspond to a cycle in the graph will just be lower bounds on the true significance value of the region and will be marked as such. $\text{VVRM}(v)$ of any vertex $v$ only yields the exact relevance value if all RSM values for this vertex from the 3rd highest on are exact values and not just lower bounds. Otherwise, the fact that one of these RSM values could actually be higher could result in a higher 3rd highest RSM value for this vertex. Thus, $\text{VVRM}(v)$ in this case is also just a lower bound on the real relevance value of $v$. Lower bound estimates of relevance values are updated when more information allows to make an estimate closer to the true relevance value and such vertices are treated like vertices with an relevance value of $\infty$ when deriving a coarser route graph layer, as long as the exact value cannot be determined.

### 3.2 Coarser level route graphs and abstraction relation

Unfortunately we can only give a short description on how the simplification algorithm that derives a coarser route graph from the original route graph works here. The algorithm removes every vertex $v$ with an relevance value that is not higher than a given threshold $\theta$ together with the subgraphs that correspond to the regions of $v$ that are classified as insignificant by the RSM value. In certain cases it becomes necessary to replace $v$ by a new vertex of degree one to ensure that every relevant vertex has a departing edge for every significant region accessible from it. Replaced substructures of the original GVG will be represented either by a single vertex or a single edge in the coarser route graph.

Figure 6 shows the result of applying this algorithm to the GVG previously shown in figure 1. $\theta$ was set to 1000mm in this example, a value that already produces quite abstract representations since most of the vertices caused by small dents and niches are removed. Figure 7 illustrates how parts of the original GVG (shown at the bottom) are represented by a vertex or an edge in the coarser route graph (top). When we are building up the multi-layer representation corresponding features of adjacent layers are linked as indicated by the arrows. Thus, we have two kinds of edges in our hierarchical representation: route graph edges horizontally connecting vertices within the same route graph layer and abstraction edges vertically connecting vertices and edges in one layer with subsets in the layer below.

## 4 Path planning, reasoning, and communication with the hierarchical GVG-based route graph

In this section we point out on how we think path planning, spatial reasoning, and communication about spatial information can benefit from the hierarchical route graph representation described in the previous section.

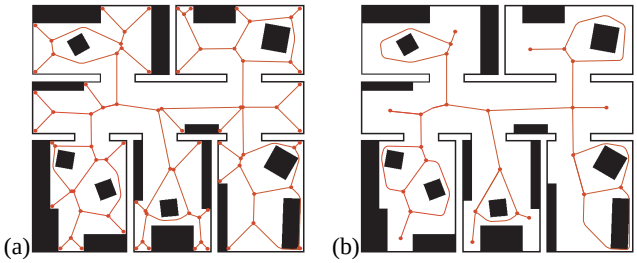



**Figure 6.** Results of the simplification algorithm: The original GVG (a) is transformed into a coarser route graph (b).

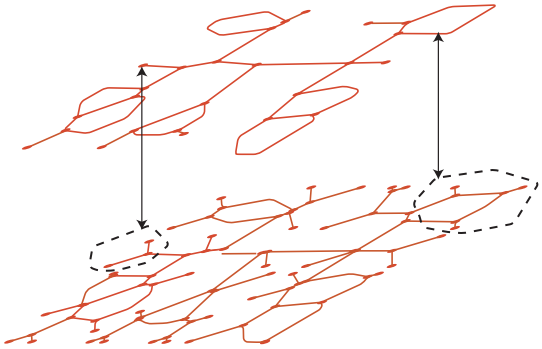


**Figure 7.** A two-layer hierarchical route graph representation with the original GVG at the bottom and a coarser route graph layer on top of it. Two examples of how parts of the detailed level are represented by a vertex or an edge of the coarser level are shown by the arrows.

### 4.1 Path planning

A hierarchical route graph representation like the two-layer example from the previous section can be employed for hierarchical path planning. The edges in the coarse layer in a way correspond to macro operations like driving from one door to the next along a corridor or passing an object on one side. Thus, planning on the high-level (e.g. by using graph search techniques) results in a plan that is not directly executable with the low-level navigation procedures of the robot. However, the abstraction relation allows to recursively break down more abstract operations into finer operations until a plan at the detailed level of the original GVG is reached.

The definition of the relevance measures and the simplification algorithm used assure that cycles in the original GVG are either retained at a coarser level or that a complete subgraph containing the cycle is replaced by a vertex or an edge. But a cycle will never split up when changing to a higher level of abstraction. This guarantees that a shortest path planned on a higher level will always result in the shortest path at the bottom level as well, when it is recursively transformed into an executable plan.

### 4.2 Spatial Reasoning

In [11] we described an approach to reason about the relative positions of the decision points within the low-level GVG-based route graph by propagating intervals for the distance and angles (called distance-orientation intervals (DOIs)) annotated to the graph structure along the sequence of edges connecting two vertices. This approach is similar to the composition of spatial relations in qualitative spatial reasoning [4]. The DOIs represent the uncertainty in the metric relative position information assuming certain maximal error boundaries. Reasoning about relative positions of the decision points in the route graph can for instance be applied to determine potential candidates for loops in the environment that need to be closed while constructing the representation during an exploration. Another application is judging if an unexplored junction in a partially constructed route graph might be a good shortcut to a place visited earlier.

This reasoning about routes can also benefit from the hierarchical organization of the route graph representation. Intermediate results from the low-level propagation can be stored as relative position information at the higher levels. This would allow to employ a hierarchical propagation scheme that uses the distance orientation intervals at the highest level if they are available or switches to a lower level whenever this is not the case, adding the result to the higher level after it has been computed. On the long run the higher level will be completely annotated speeding up the relative position computation significantly.

### 4.3 Communication

The most abstract route graph layer in our representation provides a compact description of the environment that is rather independent of the particular properties of the range sensor of the robot that constructed the representation. Therefore, this information is much better suited to be communicated to another spatial agent than the detailed description given by the original GVG. Scenarios that come to mind here are multi-robot exploration scenarios in which the individual robots exchange knowledge about parts of the environment they have explored so far. However, there are of course limits to the degree of difference in the sensors that can be dealt with without further developing reasoning mechanism to handle e. g. problems arising from differences in the individual GVGs caused by obstacles that can be seen by one robot but not the other due to different heights in which the range sensors are mounted.

Another application scenario in which the abstract route graph level can be employed beneficially is human-robot communication about routes. Augmenting the route graph with semantic information for instance stemming from door recognition modules will allow the robot to generate route instructions to guide a person to a certain goal. In addition, such a representation will make it easier to match a route description given by a human instructor to the robot's model of the environment and translate it into a detailed sequence of actions, since the abstract route graph with all irrelevant vertices and edges removed will be much closer to the route graph the instructor had in mind when generating the description.

## 5 Experimental results

In first experiments we tested the relevance measures and the simplification algorithm on real data collected with our Pioneer 2 robot while it was driving along a corridor in our office building. Figure 8a shows a section of the GVG constructed during this exploration run. 8b shows the route graph computed from this GVG (again for a threshold value of 1000mm). It demonstrates how the algorithm successfully removes vertices and edges caused by small dents or noise resulting in a route graph that only contains edges for traveling along the corridor and for entering the rooms on both sides.

In a second experiment we used a simulation to perform two exploration runs with different noise ratios in the range sensor data. Applying the simplification algorithm to both GVGs constructed

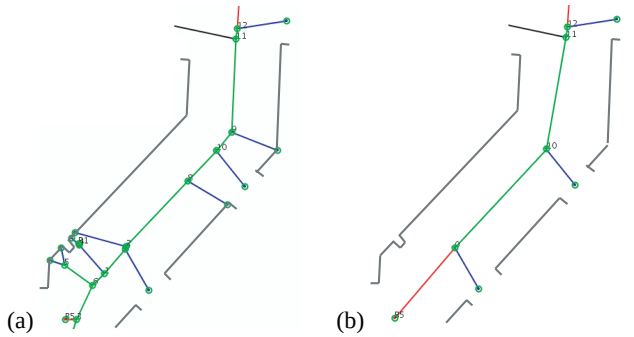


**Figure 8.** Example of a coarse route graph (b) computed from a GVG constructed with a real robot that drove down a corridor with flanking offices.

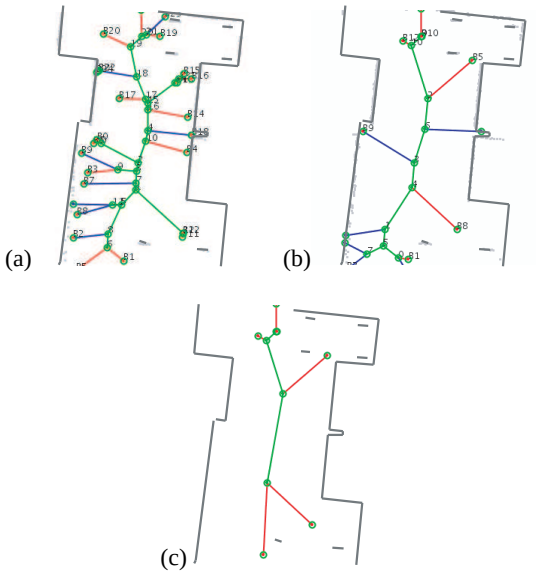


**Figure 9.** Simulation of different sensor properties: (a) shows the GVG of a robot with high and (b) of one with low sensor noise. Identical coarse route graphs are computed from both GVGs (c).

during those runs (shown in figure 9a and 9b) resulted in identical route graphs for both cases shown in 9c, only varying slightly in the exact positions of the vertices. This demonstrates that our coarser route graph representation is better suited to allow multiple robots equipped with different range sensors to exchange spatial knowledge than the original GVGs.

## 6 Conclusions

We have proposed a hierarchically organized Voronoi-based route graph representation for robot navigation and exploration tasks in office-like indoor scenarios. The representation bridges the gap between low-level spatial information for navigation and abstract route-based representations well-suited for high-level planning and spatial reasoning. We showed how such a representation can be constructed using a Voronoi vertex relevance measure and how it can be employed for hierarchical planning, spatial reasoning, and robot-robot or human-robot communication. We hope to further explore these applications in the future. In addition, we plan to address other issues involved in generating suitable abstract route graphs like the fact that multiple Voronoi vertices located close to each other may be treated more adequately as a single decision point, something that is important for human-robot communication.

## ACKNOWLEDGEMENTS

The author wishes to thank Christian Freksa, Reinhard Moratz, Frank Dylla, and Diedrich Wolter as well as two anonymous reviewers for valuable comments and suggestions. This research was supported by the German Research Foundation (DFG) grant SFB/TR 8 'Spatial Cognition'.

## REFERENCES

[1] F. Aurenhammer, 'Voronoi diagrams - a survey of a fundamental geometric data structure', *ACM Computing Surveys*, **23**(3), 345–405, (1991).
[2] H. Blum, 'A transformation for extracting new descriptors of shape', in *Models for the Perception of Speech and Visual Form*, ed., W. Wathen-Dunn, pp. 362–380, Cambridge, MA, (1967). M.I.T. Press.
[3] H. Choset and J. Burdick, 'Sensor based exploration: The hierarchical generalized Voronoi graph', *The International Journal of Robotics Research*, **19**(2), 96–125, (2000).
[4] A. G. Cohn, 'Qualitative spatial representation and reasoning techniques', in *KI97: Advances in Artificial Intelligence. Proceedings of the Twenty-first Annual German Conference on Artificial Intelligence*, eds., G. Brewka, C. Habel, and B. Nebel, pp. 1 – 30. Springer; Berlin, (1997).
[5] E. W. Dijkstra, 'A note on two problems in connection with graphs', *Numerische Mathematik*, **1**, 269–271, (1959).
[6] D. G. Kirkpatrick, 'Efficient computation of continuous skeletons', in *20th Annual Symposium on Foundations of Computer Science*, pp. 18–27. IEEE, (1979).
[7] B. J. Kuipers and Y.-T. Byun, 'A robust, qualitative method for robot spatial learning', in *AAAI 88. Seventh National Conference on Artificial Intelligence*, pp. 774–779, (1988).
[8] J.-C. Latombe, *Robot Motion Planning*, Kluwer Academic Publishers, 1991.
[9] D. T. Lee and Robert L. Drysdale III, 'Generalization of Voronoi diagrams in the plane', *SIAM Journal on Computing*, **10**(1), 269–271, (1981).
[10] F. Lu and E. Milios, 'Robot pose estimation in unknown environments by matching 2d scans', in *IEEE Computer Vision and Pattern Recognition Conference (CVPR)*, (1994).
[11] R. Moratz and J. O. Wallgrün, 'Spatial reasoning about relative orientation and distance for robot exploration', in *Spatial Information Theory: Foundations of Geographic Information Science. Conference on Spatial Information Theory (COSIT)*, eds., W. Kuhn, M.F. Worboys, and S. Timpf, Lecture Notes in Computer Science, pp. 61–74. Springer Heidelberg, (2003).
[12] H. Moravec and A. Elfes, 'High resolution maps from angle sonar', in *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 116–121, (1985).
[13] C. Ó'Dúnlaing and C. K. Yap, 'A retraction method for planning the motion of a disc', *Journal of Algorithms*, **6**, 104–111, (1982).
[14] S. Thrun, D. Fox, and W. Burgard, 'A probabilistic approach to concurrent mapping and localization for mobile robots', *Machine Learning*, **31**, 29–53, (1998). also appeared in Autonomous Robots 5, 253–271 (joint issue).
[15] J. O. Wallgrün, 'Exploration und Pfadplanung für mobile Roboter basierend auf Generalisierten Voronoi-Graphen', Diploma thesis, Fachbereich Informatik, Universität Hamburg, (2002).
[16] S. Werner, B. Krieg-Brückner, and T. Herrmann, 'Modelling navigational knowledge by route graphs', in *Spatial Cognition II*, volume 1849 of *LNCS,LNAI*, 295–316, Springer, (2000).
[17] D. Van Zwynsvoorde, T. Simeon, and R. Alami, 'Building topological models for navigation in large scale environments', in *2001 IEEE International Conference on Robotics and Automation (ICRA'2001)*, pp. 4256–4261, (2001).

# Schematized Aspect Maps for Robot Guidance

**Diedrich Wolter** and **Kai-Florian Richter** [1]

**Abstract.** We present an approach to mobile robot guidance. The proposed architecture grants an abstract interface to robot navigation allowing to bridge from perception to high level control. The approach is based upon a comprehensive map representing metric, configurational, and topological knowledge. Robot instruction and localization is dealt with by communicating and reasoning about cyclic ordering information of shape-features.

## 1 Motivation

Service robots are a growing field of interest and are higly relevant, not only for the research community but also for companies that try to take a share in an emerging market. Within near future, office complexes populated with various heterogenous service robots are a very probable scenario resulting in a range of different problems, for example coordination among the robots and interaction with human users.

A key point to instructing robots is to communicate spatial information, for example when commanding a robot to visit a certain place. To provide sufficient control over the various robots, a compatible method of instruction is required for all of them. As a consequence, the spatial representations of the individual robots would need to be compatible with each other. In the presence of different robot systems from various vendors, equipped with different sensors, this is a non-trivial task. Moreover, the chosen method of instruction needs to mediate between the technical abilities of a robot and its often data-driven spatial representation and features desired in a user interaction that are based on more abstract spatial information.

To consolidate these different demands we propose a central robot guidance system. A qualitative language, i.e. relational information, is applied for communication with the robots. Allowing for an easy formalization, qualitative information offers a good means for a specification of such a system, which also respects individual robots' capabilities. In particular, a system based on qualitative information can serve as an interface to a human user. Furthermore, with robot guidance it is possible to restrict the spatial representation of an individual robot to the needs of its specific task, like cleaning of the floor. There is no need to represent the complete working environment within each single robot. Especially within dynamic environments the management, i.e. construction and maintenance, of a spatial representation is a difficult task that has not been solved thoroughly yet [20]. Freeing individual service robots from this burden, our proposed architecture is a slender one.

The robot guidance outlined here does not rely upon a precise or even correct map of the environment. Typically, environments are subject to steady changes. Therefore, handling of unreliable, or even conflicting knowledge is of high importance. The proposed guidance system is well-suited to cope with uncertainty as robust qualitative ordering information gets used.

[1] Universität Bremen, Germany email: {dwolter,richter}@sfbtr8.uni-bremen.de

## 2 Related Work

Many robot architectures have been proposed that address the construction of a versatile mobile robot. Hereby, the robot's spatial representation is the key point. Mapping and localization issues have been covered by various authors (see [20] for an overview). It is generally agreed upon that a helpful robot map is not a single-layered representation, but provides different modalities of access, metric information and topological knowledge play key roles here [11, 18]. To respect uncertainty, successful robot architectures typically rely upon a stochastic modeling as a method of localization within a set of possible states [19].

The goal of all approaches mentioned is to gain complete metric knowledge of the environment the robot is located in. With increasing size of environments, the problem gets more difficult as compute-time increases and mix-ups in the localization occur. But not all this information may need to be acquired for the construction of a service robot. For example, a robot that is designed to collect garbage in a local surrounding like a single room needs a (spatial) representation that allows for a search strategy, but does not need to know where the room it cleans is located in a larger office complex. Local information is sufficient in guiding a robot from one usage site to another. Thus, with global guidance individual internal representations can be kept at a manageable size. Additionally, it allows to concentrate on a given task, i.e. to only use knowledge needed for the task at hand.

Multi-robot scenarios have been investigated and approaches to distributed robot systems have been proposed [5, 4]. However, these approaches rely upon a shared spatial representation that is communicated among the individual robots. Therefore, a compatibility on the lower levels of the internal spatial representation is necessary, which makes it difficult to combine different robot architectures. To obtain an open interface, a more abstract communication is advantageous. Similar to the localization based on regions of same configuration as presented within this approach, Schlieder [17] presents a qualitative approach to ordering of point-like landmarks; all landmarks are assumed visible. Another qualitative approach based on extended—but more abstract—uniquely identifiable features is presented by Barkowsky et al. [2]. In contrast to these approaches we use complex, extended features and only demand some features' visibility. Moreover, our shape-features need not be uniquely identifiable. Shape processing similar to the features employed in our approach has been proven feasible in the context of robot mapping [14]. Biologically inspired approaches similar to ours exist (e.g., [6]), too. Such view-based approaches typically employ a direct matching of sensor information perceived at certain view-points. Therefore, these approaches do not allow to mediate between different sensors.

Schematic maps are well-suited for communication [7]. As any service robot needs some kind of internal map, communication by means of pictorial information seems promising [8]. We apply a map-like spatial representation in our system to allow for this kind of interaction.

## 3 Robot Guidance System (RGS)

We propose a single central guidance system that manages several service robots that can be–to some degree–heterogenous. This robot guidance system (RGS) is built on the basis of a map of the working environment. The spatial representation derived from this map can be used for interaction with the robots (see section 6) and is suitable for the different tasks of the given robots as it allows for access in different aspects. With aspects we refer to different kinds of (spatial) information that is representable in a map. Depending on the task, it is possible to focus on certain aspects while ignoring others (cf. [3, 10]). The aspects represented include metric, configurational, and topological information. Accordingly, we term this representation *multi-aspect map*; its details are presented in section 5. Central to the multi-aspect map is the handling of polygonal shape-features extracted from range information. We cover issues related to them in the next section.

The configurational knowledge used in our approach is ordering information, which is a qualitative spatial representation. Qualitative representations summarize similar quantitative states into one qualitative characterization. From a practical viewpoint, a possibly infinite number of states is represented by means of equivalence classes. Therefore, this kind of representation is well suited to handle uncertainty.

## 4 Shape-Features

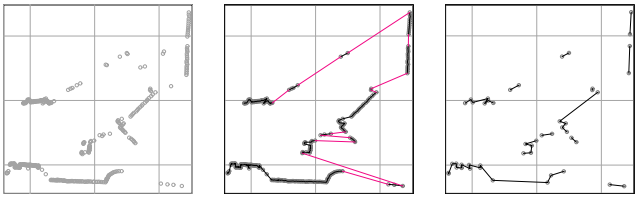
The spatial representation applied in the proposed architecture is based upon polygonal shape-features that represent boundaries of passable space. Using polygonal shape-features allows us to achieve the advantages in feature-based localization while avoiding its shortcomings: A feature-based representation is a compact one. Perceptual information gets interpreted and abstracted to form a feature. Moreover, features offer an object-based access to the information stored. Object-based access to a system's spatial representation is a fundamental prerequisite for interaction and communication.

The drawback of any feature-based approach is the necessity to reliably recognize features from perceptual data. The higher the number of features present in an environment, the more susceptible the recognition process is to any mix-up. Furthermore, if features are sparse (or even not present at all) correct localization is most likely to fail. Therefore, it is necessary to choose features that are distinctive and can be observed from any position within the environment. Shape information provides an excellent choice, as shape offers a great variety. Moreover, a direct link to pictorial information as represented by aspect maps is established (see Section 5).

Within a typical indoor scenario, a robot will be able to perceive sufficient information for reliable operation. Processing of polygonal shape features, which we call *polylines*, is necessary in perception and localization.

### 4.1 Processing Shape-Features

Let us assume that range information (typically acquired by a laser range finder) is mapped to the Euclidean plane. Reflection points are grouped to polylines. A simple heuristic may be used to implement this grouping: Whenever consecutive points are too far apart (a 20cm threshold has been used in our experiments), an object transition is assumed. Range finder information can be ordered in a counter-clockwise manner. We will denote the counter-clockwise ordering of objects $P$ before $Q$ as $P \prec Q$, meaning that $Q$ *directly* follows after $P$. Proceeding this way, we obtain an ordered list of polylines from a range finder scan. Figure 1 illustrates this. These polylines still represent all the information read form the range finder. However, this data contains some noise and is much more precise than actually required by the proposed system. Therefore, a generalization is applied that cancels noise as well as makes the data compact without loosing valuable shape information.



(a) (b) (c)

**Figure 1.** The process of extracting polygonal features from a scan consists of two steps: First, polygonal lines are set up from raw scanner data (a) (1 meter grid, the cross denotes the coordinate system's origin). The lines are split, wherever two adjacent vertices are too far apart (20 cm). The resulting set of polygonal lines (b) is then simplified by means of discrete curve evolution with a threshold of 2. The resulting set of polygonal lines consists of less data though still capturing the most significant shape information.

### 4.2 Discrete Curve Evolution (DCE)

The generalization process used for noise cancelation in our approach is called discrete curve evolution (DCE). It has been developed by Latecki & Lakämper [13, 14]. This process may be considered as an schematization (see 5.2). It describes a context sensitive process of simplifying a discrete curve by deletion of vertices and allows to reduce the influence of noise and to simplify a shape by removing *irrelevant* shape features. DCE proceeds in a straightforward manner: From a given polyline, the least *relevant* vertex is removed. This process is repeated until the least relevant vertex is more relevant than a given threshold. To determine a vertex' relevance, a measure is defined for a vertex $v$ and its left and right neighbor $u$ and $w$:

$$K(u, v, w) = |d(u, v) + d(v, w) - d(u, w)|$$

where $d$ denotes the Euclidean distance. For vertices that do not have two neighbors no relevance measure is defined. Consequently, endpoints remain fixed. An exemplary result is depicted in figure 1.

DCE may be implemented efficiently. As vertices can be represented within a double-linked polyline structure and a self-balancing tree (reflecting the order of relevance measures) simultaneously, the overall complexity is $O(n \log n)$. Since we apply DCE to segmented polylines, the number of a polyline's vertices is much smaller than the number of points read from the sensor.

### 4.3 Similarity of Polylines

A crucial method for localizing a robot is matching the robot's sensor readings against a map. As the spatial representation used here

relies upon shape features, detecting correspondences between polylines perceived and ones stored in the map is the solution at hand. The matching process relies on a similarity measure for polylines. To each pair of possibly corresponding polylines a measure of matching plausibility is assigned.

However, the proposed architecture does not rely exclusively on the correspondence of features, instead it is used alongside with configurational knowledge that poses constraints upon the matching. Therefore, discussion of the actual matching is delayed to section 4.4 and the following description focuses on computing the similarity of two given polylines.

The pair of polylines, whose similarity needs to be computed, is on the one hand perceived by the mobile robot and on the other hand extracted from the multi-aspect map. This map is specially designed to be adaptable to any robot's capabilities. Nevertheless, there may be remarkable differences between the two polylines (e.g., due to noisy perception). Therefore, we need a careful approach to determine similarity. We utilize a similarity measure of polylines originating from Computer Vision [12]. It can easily be adopted to polylines describing environmental features [21]. We briefly summarize the computation of similarity of polylines as presented in [12, 21].

The similarity measure is based on a matching of maximal left- or right-arcuated arcs. Any polyline's partioning into consecutive nonempty sequences of arcs is called a grouping, if consecutive groups overlap in exactly one line segment. This entails that any grouping covers the whole polyline.

Groupings $G, H$ are said to correspond (denoted $G \sim H$), if there exist a bijection $f_{G,H}$ between the two groupings such that on the level of maximum arcs only mappings of type 1-to-1, 1-to-many, or many-to-1 exist. Based on a similarity of arcs $S_{arcs}$ that is presented below, the similarity measure for polylines is defined.

$$S_{poly}(g,h) = \min_{G \sim H} \Sigma_{x \in G} S_{arcs}(x, f_{G,H}(x))$$

Similarity of arcs is defined in tangent space, a multi-valued step function mapping a curve into the interval $[0, 2\pi)$ by representing angular directions of line segments only. Furthermore, arc lengths are normalized. Denoting the mapping function by $T$, the similarity gets defined as follows:

$$S_{arcs}(c,d) = \int_0^1 (T_c(s) - T_d(s) + \Theta_{c,d})^2 ds \cdot (1 + (l(c) - l(d))^2)$$

where $l(c)$ denotes the arc length of $c$. The constant $\Theta_{c,d}$ is chosen to minimize the integral (it respects for different orientation of curves) and is given by $\Theta_{c,d} = \int_0^1 T_c(s) - T_d(s) ds$. In contrast to the original work, absolute instead of relative size is considered, since this is more adequate in our domain. The correspondence yielding the best similarity is computed using dynamic programming.

### 4.4 Similarity of Views

On the basis of an individual similarity of polylines we define a similarity of *views*, cyclic ordered sets of polylines. Whereas similarity of polylines respects only the spatial context of a single polyline, views account for a larger context. It is, thus, a much more distinctive measure. Similarity of views will be the fundamental building block in localizing the robot within the central map and also gets used in the construction process of the map itself.

The similarity is based on the individual similarities of corresponding features. Thus, the aim is to find a correspondence relation between features that optimizes the summed up similarity. However, we must also consider that (a) the cyclic ordering $\prec$ must not be violated and (b) not all features present in one view need to have a counterpart in the other. Whereas the latter may be viewed as a soft constraint, configurational knowledge is reliable and thus poses hard constraints upon the recognition process. By introducing a penalty $P$ for leaving a feature unmatched, we can formulate the computation of views' similarity by means of dynamic programming similar to aproximate string-matching. Therefore we must linearize the cyclic ordering. This can be done by selecting any feature of one view as the first one, and then considering every linearization of the second view. This yields an overall complexity of $O(n^3)$ where $n$ denotes the number of features.

To be more precise, let us assume that $F_1 \prec F_2 \prec \ldots \prec F_n$ and $\hat{F}_1 \prec \hat{F}_2 \prec \ldots \prec \hat{F}_m$ are two linearized views. A matrix $M$ of size $n \times m$ is set up and is initialized by setting $M_{1,1}$ to $S_{poly}(F_1, \hat{F}_1)$ leaving the remains empty. A cell $M_{i,j}$ may be computed when all cells $M_{i',j'}$ with $i' < i, j' < j$ have been computed. The cell's value is then given by:

$$\min \left\{ S(F_i, \hat{F}_j) + M_{i-1,j-1}, P(\hat{F}_j) + M_{i,j-1}, P(F_i) + M_{i-1,j} \right\}$$

The first term denotes a 1-to-1 matching of features $F_i$ and $\hat{F}_j$ extending the matching of previous features stored in $M_{i-1,j-1}$. The second and third term addresses the possibility that a feature is not matched at all. The penalty measure $P$ is chosen to scale linearly with the size of the feature, as it is much more likely to overlook a small object than a larger one.

## 5 Multi-Aspect Map

In this section we give further details on the spatial representation we use. As stated earlier, this representation, a map, is multi-aspect. It is suitable for navigational tasks, can be used on different levels of abstraction, and allows for different aspects in access. This multi-aspect map is the fundamental representation of our RGS; it is used for localization, path-planning, and interaction with the robots.

We termed the set of spatial representations that get used in our RGS *multi-aspect map* since the different kinds of information correspond to different *aspects* of the environment. The environmental representation can be accessed depending on a given task, for example topological information for path-planning or metric information for shape matching, and depending on the given context, i.e. the given robot. Our representation is, thus, customized for a specific situation while still originating from a single source. Use of such a multi-aspect map is comparable to the approach taken in the project 'Spatial Structures in Aspect Maps' [3, 10]; here, the idea is to extract from a given map-like representation all (spatial) information that is needed for a given task and, by focusing on the aspects relevant for the task, providing a cognitively adequate representation.

The basis for the multi-aspect map is a representation of the environment; its properties are described in section 5.1. There are three different kinds of information stored in the multi-aspect map: Metric information denotes the shape of the environmental features. The spatial configuration of the features, which corresponds to ordering information, is also explicitely stored in the map. Both kinds of information are needed for robot localization. Topological information is used for path-planning. It is stored as a graph. All kinds of information are present and accessible on different abstraction levels; these levels are adapted to the given robots and their respective sensors. The way we construct the multi-aspect map is detailed in section 5.2.

## 5.1 Properties of the Map

A map of the environment is the basic representation of the RGS. The map's structure is polygonal, i.e. its basic elements are polylines. Such a map can be obtained in different ways. One way is to use an existing map, i.e. an electronic version of a building's blueprint. This requires that all elements of the dataset can be addressed directly, i.e. it is possible to access objects individually, and that such an existing dataset is rich enough to contain all the information needed for the different tasks.

The map can also be obtained by using a robot that explores the environment and builds a map of it. Many approaches to robot mapping have been proposed [20, 9]. Even though they typically rely either on simple line segments or uninterpreted data, they can be extended to deal with polygonal lines, or polylines may be extracted from their output. An example of such map extracted from laser range finder data is depicted in figure 2.

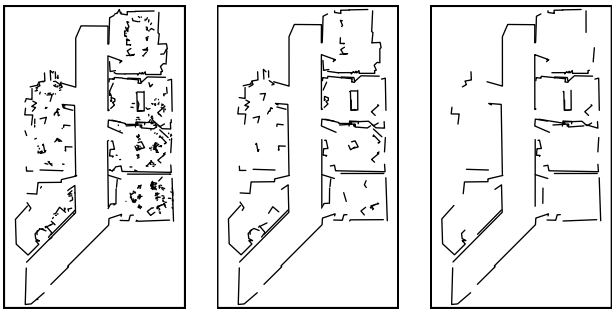
## 5.2 Constructing a Multi-Aspect Map

Construction of the multi-aspect map is a three-step process. First, we construct the maps designated for the different robots, i.e. schematizing the map of the environment to an adequate abstraction level. Next, we determine for each schematized map a graph reflecting the envionment's topological structure. Based on this graph we then calculate regions of similar order and, thus, partition the plane.

### *5.2.1 Schematizing maps*

Roughly speaking, map schematization describes a simplification—or even an elimination—of map objects, respecting essential spatial relations. Schematizing maps, hence, involves simplification of shape information. Complex polylines are simplified to obtain simpler ones that still show off the most important shape information while hiding the details. To simplify shapes, various techniques have been proposed. The DCE process as presented in section 4.2 is one promising approach to shape simplification. It has been successfully applied to shape simplification in map schematization [1]. Besides DCE, other approaches to shape simplification have been proposed, too. For example, the Curvature Scale Space proposed by Mokhatarian et al. is based on a simplification process' history [15, 16]; a Gaussian convolution filter is applied to accomplish the simplification. Simplification by means of smoothening changes shapes globally, whereas simplification by vertex removal like in DCE is composed out of local changes. Since any simplification must be checked for admissibility, e.g., to prevent violation of topological constraints (see below), DCE's discrete structure is advantageous here.

Due to the more complex structure of map schematization compared to simplification of a single polyline, the DCE process needs to be adapted, though. On the one hand, not every point of the structure can be removed, for example points that belong to multiple objects. They must be preserved to retain the fact that multiple objects meet at this point. On the other hand, as the spatial information has to remain correct for the different tasks to remain accomplishable, we take into account relational information of the map's entities. We need to take care that by removing vertices there does not occur a violation of any relational information. After an evolution step may, for example, (parts of) an entity be resided left to another entity while it was located right to it prior to this step, or two entities may now overlap. For further details of the necessary extensions to DCE for map schematization see [1].



(a) (b) (c)

**Figure 2.** The base map (a), and two different schematization levels: medium (b) and maximum (c)

Additionally, as small objects do not provide relevant features they get removed from the map. Just like the degree of schematization, the size threshold depends on the sensors used by the different robots. Figure 2 depicts some schematization levels as an example.

Theoretically, the maximum number of different abstraction levels corresponds to the number of inner points of all polygonal lines as the process of discrete curve evolution is stepwise and removes one vertex in each step. Practically, the actual number of different abstraction levels that get stored in the RGS is much smaller. Schematized maps are only needed on certain levels of abstraction; these levels are determined by the robots' perceptual abilities. These abilities are taken into account when setting the levels' DCE thresholds. Each marks a level of adequate abstraction. All schematization levels in-between are deemed qualitatively equal to either of them and are not considered. Thus, although quantitatively the number of possible maps is quite high, the number of maps that really get constructed is rather low due to the qualitative abstraction involved.

### *5.2.2 Regions of similar order*

Next, we determine a graph embedded in the map that reflects the environment's topological structure. We consider structure from a more abstract point of view, as we are only interested in noticeable differences, for example when moving through a door into another room. The graph is calculated based on the schematized map and gets used for path-planning (see 5.3). For example, Voronoi graphs are suitable for this purpose [18].

This graph is taken as the basis for determining *regions of similar order*: The graph is said to intersect with the boundary of a region whenever traversing its edges yields a high dissimilarity of views at nearby positions (see 4.4). Practically, computing the similarity for nearby views along the graph's edges is performed by a subsampling of similarity values at a given number of locations on the edges. Each time the value exceeds a given threshold a new region is generated. For each region a single, prototypical view is stored in the map, which gets used in the localization process.

## 5.3 Using the Multi-Aspect Map for Robot Navigation

Two tasks in robot navigation are carried out using the multi-aspect map, namely qualitative localization and topological path-planning. Prior to using the map for communicating with a robot, the adequate level of schematization needs to be chosen. We select the appropriate level from the multi-aspect map regarding the level of generalization

used by the robot for feature extraction. This ensures that features stored in the map are perceivable to the robot, i.e. they are not too small to be detected by the robot when not close-by.

The map's topological aspect is used to plan a robot's path from a given location to a goal region. We calculate a qualitative path by means of graph-search determining the regions the robot passes by. A prerequisite for successful navigation is that the robot can be localized; we describe this process in detail in the next section.

## 6 Instructing a Robot

To command a robot to a given area of the environment, the robot needs first of all to be localized within the multi-aspect map in order to plan a path. Localization combines the similarity of shape features with configurational knowledge. It is covered in section 6.1. Knowing the robot's position within the map, a path that leads the robot to its goal region can be computed. As will be presented in section 6.2 a single motion primitive is sufficient to guide the robot along this path.

### 6.1 Localization

For the RGS it is sufficient to localize a robot qualitatively. The term qualitative localization as opposed to metric localization is chosen to stress that only information required for the guidance task is used. The robot's position is represented by qualitative regions of similar order. Hence, localization means to recognize the region a robot currently visits. Similarity of views (see section 4.4) is the clue here. Therefore, the approach to localization taken here relates to view-based approaches (cf. [6]); however, the presented approach is more abstract, since sensor information is always interpreted first. Such a more abstract approach is advantageous here, as different robots utilizing different sensors (e.g., laser range finder mounted at differing heights) need to be localized.

To localize the robot, individual plausibilities are computed for the robot being in a particular region by determining the similarity between the prototypical view associated with the individual region and the view perceived by the robot. These plausibility values can easily be coupled with a stochastical approach to localization like Markov localization (cf. [19]). Plausibility values are therefore scaled such that the overall sum yields 1. The robot is said to visit the region which has currently the highest belief state.

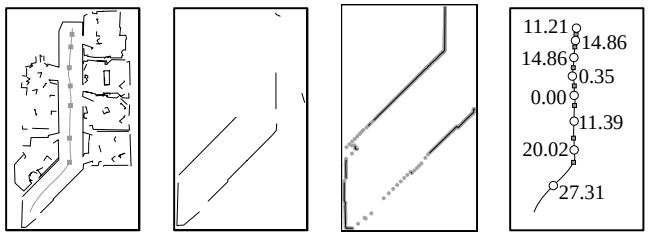



**Figure 3.** (a): An excerpt from the utilized path network shown on the medium level schematization. Blocks denote the borderline of regions of similar order. (b): The prototypical view associated with the network's lowest segment of (a). The shape-features extracted from a scan taken at the lower end of the hallway (c) are matched against the prototypical views stored for each region. The membership probability in % is given in (d). As can be observed, matching a single view already yields the right localization.

### 6.2 Instruction

Once the robot is localized, instruction can be realized with just one motion primitive that needs to be implemented into each mobile robot: moving the robot inbetween two features it has perceived. It moves along until the order of features changes, i.e. a feature becomes invisible or a new feature emerges and, thus, probably a new region is entered. This motion primitive is sufficient to guide the robot from cell to cell, i.e. along the different regions of similar order; its path is determined using the topological graph (see 5.2.2). For the purpose of localization the belief state is updated accordingly.

## 7 Conclusion

In this paper, we proposed a central system that can be used to guide various service robots acting in an environment. It allows to integrate different kinds of service robots within a larger context, while offering a single interface for the human user to all robots involved. This single interface is one of the system's main advantages, as a user does not need to remember and to switch to different interaction modes depending on the robot currently addressed. The specifics of the robot remain abstract to the user. Interaction is cognitively adequate since the user can concentrate on the task the robot is about to perform.

We use a single spatial representation—a *multi-aspect map*—in the system; this representation allows to access just the information needed for a given navigational task and robot, namely metric, ordering, or topological information on different abstraction levels. This is another main advantage of the proposed architecture: reasoning about the environment, i.e. localization and navigation, takes place on a qualitative level. We apply a partitioning of the environment in *regions of similar order*, which is a novel approach. It is robust but detailed. We employ these regions and a matching of *shape-features* in robot localization. We can, thus, perform this localization on a qualitative level, which is very robust, keeps the communication compact, and allows for an efficient path execution algorithm.

The prime focus of this paper has been to present the general structure of our proposed architecture and to point out its advantages. While individual aspects have already been implemented, future work comprises the integration of these parts and an evalutation of the whole system.

## Acknowledgements

This work is part of the projects R3-[Q-Shape] and I2-[MapSpace] of the Transregional Collaborative Research Center (SFB/TR 8) Spatial Cognition funded by the Deutsche Forschungsgemeinschaft (DFG). We would like to thank Prof. Latecki for discussion and comments.

## REFERENCES

[1] Barkowsky, T., Latecki, L.J.,& Richter, K.-F. (2000). Schematizing maps: Simplification of geographic shape by dicrete curve evolution. In: C. Freksa, W. Brauer, C. Habel, & K.F. Wender (eds.), Spatial Cognition II (pp. 41–53). Berlin: Springer.

[2] Barkowsky, T., Berendt, B., Egner, S., Freksa, C., Krink, T., Röhrig, T., & Wulf, A. (1994). The REALATOR: How to construct reality. In: ECAI'94 Workshop W12 Spatial and Temporal Reasoning.

[3] Berendt, B., Barkowsky, T., Freksa, C., & Kelter, S. (1998). Spatial representation with aspect maps. In: C. Freksa, C. Habel, & K.F. Wender, Spatial Cognition (pp. 157–175). Berlin: Springer.

[4] Burgard, W., Moors, M., Fox. D., Simmons, R., & Thrun, S. (2000). Collaborative multi-robot exploration. In: Proc. of the IEEE International Conference on Robotics & Automation (ICRA).

[5] Cohen, W. (1996). Adaptive mapping and navigation by teams of simple robots. Journal of Robotics & Autonomous Systems 18:411-434.
[6] Franz, M.O., B. Schölkopf, H.A. Mallot, &H.H. Bülthoff (1998). Learning view graphs for robot navigation. *Autonomous Robots*, 5, pp. 111–125
[7] Freksa, C. (1999). Spatial aspects of task-specific wayfinding maps – A representation-theoretic perspective. In: J.S. Gero & B. Tversky (eds.), Visual and Spatial Reasoning in Design (pp. 15–32). Key Centre of Design Computing and Cognition, University of Sidney.
[8] Freksa, C., Moratz, R., & Barkowsky, T. (2000). Schematic maps for robot navigation. In: C. Freksa, W. Brauer, C. Habel, & K.F. Wender (eds.), Spatial Cognition II (pp. 100–114). Berlin: Springer.
[9] Gutmann, J.-S. & Konolige, K. (1999). Incremental Mapping of Large Cyclic Environments. In: International Symposium on Computational Intelligence in Robotics and Automation (CIRA'99), Monterey.
[10] Klippel, A, Richter, K.-F., Barkowsky, T. & Freksa, C. The Cognitive Reality of Schematic Maps. In: A. Zipf, T., & L. Meng (eds.), Map-based Mobile Services – Theories, Methods and Implementations. Berlin: Springer. to appear.
[11] Kuipers, B. (2000). The Spatial Semantic Hierarchy. Artificial Intelligence 119:191–233.
[12] Latecki, L.J. & Lakämper, R. (2000): Shape Similarity Measure Based on Correspondence of Visual Parts. IEEE Trans. Pattern Analysis and Machine Intelligence (PAMI) 22(10):1185–1190.
[13] Latecki, L.J. & Lakämper, R. (1999). Convexity rule for shape decomposition based on discrete contour evolution. Computer Vision and Image Understanding 73:441–454.
[14] Latecki, L.J., Lakämper, R., & Wolter, D. (2003). Shape Similarity and Visual Parts. *Proceedings of Discrete Geometry for Computer Imagery*, Naples, Italy, November 2003.
[15] Mokhtarian, F. & Bober, M. (2003). *Curvature Scale Space Representation: Theory, Applications and MPEG-7 Standardization.* Kluwer Academic Press.
[16] Mokhtarian, F. & Mackworth, A. K. (1992). A theory of mulitscale, curvature-based shape representation for planar curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(8):789–805.
[17] Schlieder, C. (1995). Reasoning about ordering, In: A. Frank and W. Kuhn (eds.), Proceedings of the 3rd International Conference on Spatial Information Theory.
[18] Thrun, S. (1998). Learning Metric-Topological Maps for Indoor Mobile Robot Navigation. Artificial Intelligence, 99:21–71.
[19] Thrun, S. (2000). Probabilistic Algorithms in Robotics. Artifical Intelligence, 21(4): 93–109.
[20] Thrun, S. (2002). Robot Mapping: A Survey, In: G. Lakemeyer & B. Nebel (eds.), Exploring Artificial Intelligence in the New Millenium. Morgan Kaufmann.
[21] Wolter, D. and Latecki, L. J. (2004). Shape Matching for Robot Mapping, In: Zhang, C., Guesgen, H.W., and Yeap,W.K., Proceedings of 8th Pacific Rim International Conference on Artificial Intelligence, Auckland, New Zealand, to appear

# How can I, robot, pick up that object with my hand ?

**Antonio Morales** and **Pedro J. Sanz** and **Angel P. del Pobil** [1]

**Abstract.** This paper describes a practical approach to the robot grasping problem. An approach that is composed of two different parts. First, a vision-based grasp synthesis system implemented on a humanoid robot able to compute a set of feasible grasps and to execute any of them. This grasping system takes into account gripper kinematics constraints and uses little computational effort.

Second, a learning framework aimed at discovering the visual features that predict a reliable grasp. A grasp characterization scheme based on a set of visual features is developed in order to describe and compare grasps. In addition, a practical measure of grasp reliability is designed and implemented.

Moreover, an algorithm aimed at predicting the performance of an untested grasp using the results observed on previous similar attempts is presented. A second algorithm that actively selects the next grasp to be executed in order to improve the predictive quality of the accumulated experience is introduced, too.

An exhaustive database of experimental data is collected and used to test and validate both algorithms.

## 1 INTRODUCTION

The ability for manipulating and using objects are some of the most relevant skills that robots have to master in order to interact with its environment and constitute a key component for many robotic applications. Robotic manipulation can be studied at many levels, from the mechanical and physical interactions between different objects, through the proper design of mechanical robot hands, to the purposeful use of different objects. Traditionally, roboticist has focused on the former aspects, and for a good reason. Usually, complex manipulations , from the point of view of a robot, require a precise knowledge of the complex physics involved and the use of carefully designed hands. As a consequence, little attention has been paid on the, high-level, cognitive activities related with the purpose of manipulation and the nature of the manipulated objects.

This paper is the summary of a large project that has been focused on the improvement of the grasping capabilities of a robot in order to be able to grasp objects within unstructured environments. This unstructuredness is derived from the uncertain conditions of the objects to be grasped, and the little practical knowledge of the conditions that make a grasp stable.

We focus on the grasping problem, consisting of determining the kind grasp necessary to carry out certain manipulation tasks on an object. A grasp is defined both by the contacts on the objects surface and the hand and arm configuration necessary to reach them. Moreover, we focus on the pick up task. That is, we grasp the object in order to lift and transport it.

[1] Intelligent Robotics Lab., Univertitat Jaume I, Castellon, Spain. e-mail: {morales, sanzp}@icc.uji.es, pobil@ieee.org

Extensive research on this field during the last two decades has established a strong theoretical framework[15, 13, 2]. However, most of this research has been based on perfect models or ideal operational conditions. These assumptions often become unrealistic in real world applications.

Briefly, the principles of our approach are two: first, the use of sensorial, mainly visual, information to reduce the uncertainty in the environment; second, the development of a learning framework to apprehend the features of the environment that predict the outcome of the actions of the robot.

The development of this project yields two clearly separated parts: the development of a practical grasping system, and the design and implementation of a complete learning scheme.

The main features of the grasping in system (described in sec. 2), is that it makes use of sensorial inputs, mainly vision, to acquire relevant information for the grasping task, in particular the shape and location of the objects to grasp. In addition to this, we also develop a couple of grasp synthesis algorithm able to compute two and three finger grips from this information, using a small computational time, and meeting theoretical stability conditions. Finally, an algorithm to adapt the computed grips to the particular features of the gripper used is necessary, too.

Once this system is developed we face the problem of grip selection. Given a object, many different feasible grips can be performed on it, and it is thus critical to characterize the quality of candidate grips in order to execute the most reliable ones.

In this paper we introduce an ambitious approach that tries to use experience of real grasping actions to tune the behavior and the reliability assessment capabilities of the grasping system. More specifically we follow an active learning approach. According to this paradigm, the agent is allowed to interact with its environment. More specifically, it can execute actions which have an impact on the generation of training data. *Exploration* refers to the process of selecting actions in active learning. In the framework of our problem, the possible actions are the different candidate grips, at a given moment. Actions are selected by the agent in an ”intelligent” way in order to minimize the cost and duration of the learning process.

To reach this goal we develop a learning scheme that is composed of four main parts:

- A grasp characterization scheme that provides a unique description of any grasp (sec. 3). This characterization scheme is based on nine high-level vision-based descriptors. In this way, we represent each grip as a point in a multidimensional space.
- An experimental test (sec. 3.1) by means of which the robot can determine the reliability of a given grasp. This is achieved by executing the grasp and applying on it a set of practical tests to estimate the degree of stability.
- A set of techniques for predicting the reliability of a grasp from its similarity to other grasps (sec. 4). These techniques use the

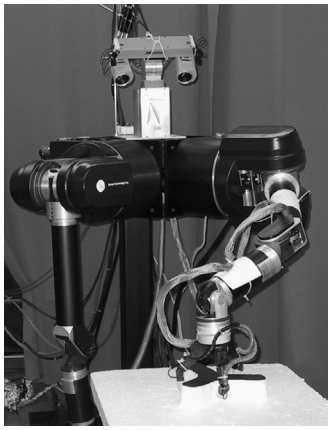
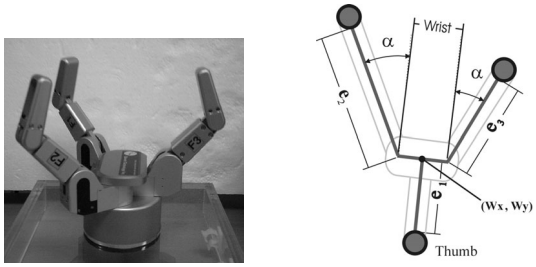


**Figure 1.** The UMASS Torso. A humanoid robotic system developed at the Laboratory for Perceptual Robotics in the University of Massachusetts[17].




(a) Photo (b) Kinematics

**Figure 2.** Barrett Hand, `http://www.barretttechnology.com`

characterization schema described in previous point, and are based on pattern classification and recognition techniques.

- An exploration algorithm (sec. 5) that makes use of the problem representation previously built to decide the next action, the grasp to be executed, in order to obtain a better knowledge of the environment with a lower cost, that is, with a minimum number of executions.

Finally, we carry out an experimental validation of these methods using real data from repeated grasping actions of the robot. We collect an extensive set of samples from real grasping executions (sec. 3.2), and use them to tune, test and validate our methods (secs. 4.3 and 5.1).

## 2 A PRACTICAL GRASPING SYSTEM

We have implemented a robotic grasping system on the UMass humanoid torso, at the Laboratory for Perceptual Robotics in the University of Massachusetts[17]. This humanoid robot consists of two Whole Arm Manipulators from Barrett Technologies, two Barrett hands with tactile sensors at the fingertips and a BiSight stereo head.

The stereo vision system estimates the two-dimensional location of the target object on the table, and provides a monocular image for surface curvature analysis (see [12] for more details). Once a grip is selected (consisting of contact locations and a hand posture), the hand is preshaped and positioned above the object. It moves down, closes the fingers so that the object is grasped, lifted and transported to a designated location.

The main modules/steps of the functioning of this robotic grasping system are the following:

1 **Image processing:** analyzes an image of an unknown planar object, extract its contour and identify triplets of grasping regions.
2 **Grip synthesis:** determines a number of feasible grasps selecting the grasping points for each region triplet; after that, generates finger configurations that could actually be applied to the object in order to perform a grip action.
3 **Grasp selection:** perform an 'intelligent' selection of the grip to execute.
4 **Execution:** execute the grip with support of visual and tactile feedback.

Details about the first, second, and fourth sections of a system of this kind, concerned with the generation of candidate grasping configurations, are fully described in [10, 11, 12], though in the next subsections we introduce the basic concepts.

### 2.1 Grasp synthesis

We define a *grasp* as the set of three contact points on an object contour, and the corresponding force directions, perpendicular to the contour, which meet in the grasp force focus. We call *hand configuration* each possible grip obtained applying the kinematics constraints of a robot hand to a grasp as defined above.

To avoid misunderstandings, in all this text when referring to grasps and configurations together, the term grip is used.

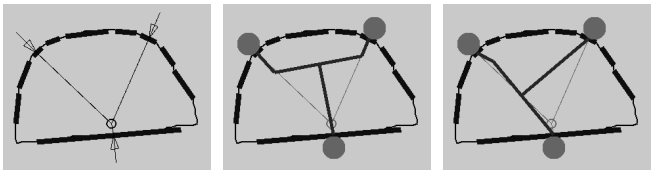
We assume a real-time system acting in an unstructured environment, which detects unknown objects and, through analysis of visual data, selects and executes a stable grip of such objects.

Fast computation is necessary in order to achieve a real-time interaction with the external world. The ability to cope with uncertainties, in terms of knowledge of friction coefficients or visual and positioning errors, is a must in an uncontrolled environment.

### 2.2 Configurations

With a perfectly homogeneous three-finger hand, for which the fingers are all the same, the three possible ways of combining fingers with contact points in a grasp are not distinguishable. This is not the case for the Barrett Hand, for which the kinematics of the thumb is different from that of the other two fingers. A photo of the hand is reproduced in Fig. 2(a). Its kinematics are depicted in Fig. 2(b). The hand has four degrees of freedom: the three finger extensions $e_1, e_2, e_3$ and the spread angle $\theta$.

For each grasp there are three possible positions of the thumb. After deciding where to place the thumb, there are still potentially infinite ways of making the hand touch the object at three contact points. However, when the action line of the thumb is fixed as well, only one solution is possible. A one-dimensional search along all possible thumb force directions gives the best Barrett Hand configuration for a grasp after the thumb position has been defined . Thus, every grasp ideally generates three different configurations, one for each thumb position. When no solutions are found for a thumb position within a grasp, due to the constraints deriving from the hand geometry and kinematics, no corresponding configurations are produced.



(a) Grasp (b) Config. 1 (c) Config. 2

**Figure 3.** Generating configurations from a grasp

Typically, dozens of configurations can be generated for an object, mostly depending on the number of regions found. In Fig. 3(b) and 3(c) two configurations generated from the grasp of Fig. 3(a) are depicted.

### 2.3 Two-finger grips

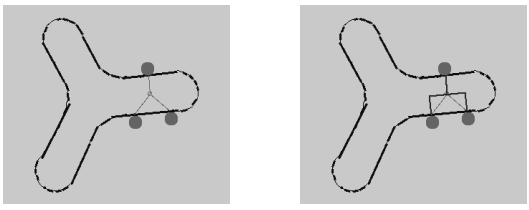
A particular kind of three-finger grasp is obtained as an extension of two-finger grasps. To generate a two-finger grasp, only two regions are needed, and they must be nearly parallel and facing each other (with friction, regions that are not perfectly parallel can also be used for two-finger grips).

Starting from a real two-finger grasp, if one of the regions is large enough to carry two Barrett Hand fingers, then a *virtual two-finger grasp* is generated. So, there is a special group of three-finger grasps that are computed in a completely different way, and thus have different properties and characteristics. From now on we will refer to them as *two-finger grasps*, meaning that two of the fingers are positioned on the same grasping region.

Each two-finger grasp can generate only one configuration, that is a *two-finger configuration*, as the thumb must be the finger opposed to the other two. An example of a two-finger grasp and its configuration are shown in Fig. 4 (a) and (b).

### 2.4 Implementation and results

The modules described in the previous sections have been implemented and tested. In a first stage they have been tested isolated, using as inputs images of different objects [10, 11]. These tests show that our implementation obtains the same results as do other classical works [5, 14] employing a few milliseconds on a common PC computer.



(a) Grasp (b) Configuration

**Figure 4.** Example of two-finger grip

On a second stage they have been embedded on the control system of the UMass humanoids torso for building a complete grasping system[12]. Nearly 70 real grasp executions have been performed using this system. These experiments have consisted in placing an object in front of the robot and grasping it by executing one of the hand configurations computed for the object. The selection of the configuration to execute have been done by a human operator.[2]

These experiments show the usefulness and validity of the developed algorithms. However, they also shown the limitations of the grasping system. The first main problem is that the grasp synthesis algorithms produce a large number of possible grips, and there is no clear rule for preferring one to the others. Regarding to this problem, we propose a set quality criteria [3] that gives a value for each grasp. However, this method is not satisfactory enough since it is purely a priory, with no feedback from reality.

A second main problem, is the unexpected bad performance of some *a priory* stable grasp. Though this can be caused by the inaccuracy of the sensor inputs and the execution controllers, it also strongly affected by risks not anticipated during the stability study used to design the grasp synthesis algorithms.

These limitations have motivated the development of the learning framework that uses experience for determining the features of grasps that asses its stability and reliability.

## 3 GRASP CHARACTERIZATION SCHEME AND RELIABILITY MEASUREMENT

A characterization scheme to provide a way to describe grasps so that they can be used by the learning procedures has been developed. We have opted for a scheme that measures a set of properties of each grasp. In this way a grasp will be represented by *n* measurements becoming a point in an *n-dimensional* space. This scheme consists of nine of these high-level features that have been designed in order to meet the next requirements:

**Vision-based computation.** The features are computed from visually-extracted information.

**Hand constraining.** Features take into account particular characteristics of the hand.

**Location and orientation invariance.** Displacements and rotations of the object do not affect the values of the features.

**Object independence.** Grasps with the same physical properties have the same characterization independently of the object for which they are computed.

**Physical meaning.** Features are computed to measure physical properties relevant to grasping.

**Stability and reliability.** Features consider stability and reliability hazards of a grasp.

To summarize, every grip is described by a nine-elements tuple, and therefore, can be abstracted as a point in a nine-dimensions space. This space would contain all the possible grip descriptors.

Due to the limitations of space, we only describe in detail one of the grasp descriptors, as an example of the kind how these requirements are actually applied in the design of the descriptors. For further

[2] In `http://www.robot.uji.es/people/morales/experiments` there is an exhaustive description, including video recordings, of all these experiments.

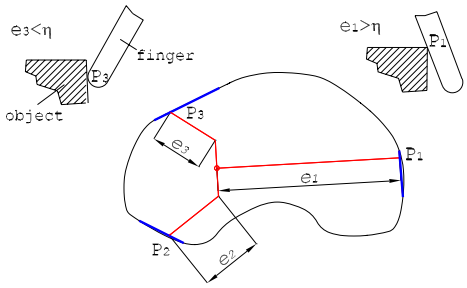



**Figure 5.** Geometrical representation of the Finger Limit Criterion.

details and a better explanation of all the descriptors the reader is referred to [3].

*An example of grasp descriptor: The Finger Limit criterion*

When trying to grip large objects, there is a limit in the extension of the fingers . Due to the way the Barrett Hand grips objects, there is a finger extension value that, if overcome, causes the grip to shift from a fingertip grip to a fingerside grip on the part edge, which is more risky and less stable although still possible (see Fig. 5). Therefore, a threshold on the maximum optimal finger extension $\eta$ has been set in order to avoid marginal contacts: $q_{FG} = \epsilon_1 + \epsilon_2 + \epsilon_3$ where $\epsilon_i = (\frac{e_i - \eta}{\lambda})^2$ if $e_i > \eta$, else 0. The threshold $\lambda$ is an estimation of the positioning error.

## 3.1 Experimental measurement of grasp reliability

A key issue in our experimental approach is the definition of a practical measurement of the reliability of a grasp. In order to do this a single object is placed on a table within the robot workspace. Using visual information the robot locates the object and computes a set of feasible grasp configurations. One of the configurations is selected, either manually by a human operator, or automatically by the robot, and executed.

If the robot has been able to lift the object safely, a set of stability tests are applied in sequence. These are aimed at measuring the stability of the current grasp. They consist of three consecutive shaking movements of the hand which are executed with an increasing acceleration. After each movement the tactile sensors are used to check whether the object has been dropped off.

This protocol provides us with a qualitative measure of the success of a grasp. Thus, an experiment may result in five different reliability classes: *E* indicates that the system was not able of lifting the object at all; *D, C, B* indicate that the object was dropped, respectively, during the first, second, or third series of shaking movements; finally *A* means the object did not fall and was returned successfully to its initial position on the table. Hence, we define $\Omega = \{A, B, C, D, E\}$ as the set of reliability classes.

## 3.2 Experimental sample dataset

To acquire a sample database large enough to validate the proposed methods, a series of exhaustive experiments have been carried out.

**Table 1.** SAMPLE DATASETS

| | E | D | C | B | A | Total |
|---|---|---|---|---|---|---|
| LIGHT | 102 | 84 | 33 | 27 | 18 | **264** |
| LOW | 38.6% | 31.8% | 12.5% | 10.2% | 6.8% | **(22)** |
| LIGHT | 51 | 97 | 56 | 38 | 118 | **360** |
| HIGH | 14.2% | 26.9% | 15.6% | 10.6% | 32.8% | **(34)** |
| HEAVY | 95 | 92 | 29 | 2 | 2 | **220** |
| HIGH | 43.1% | 41.8% | 13.2% | 0.9% | 0.9% | **(23)** |

Sample distributions among classes for the different data sets. The figures in brackets in the "Total" column indicates the number of different grip configurations really tested.

Four real objects has been built for this experiment: two with simple shapes and two with more complex shapes. In order to build the sample database the four objects are presented to the grasping system, and a sufficiently large number of grips are executed. The reliability of these grips is obtained applying the test described in section 3.1.

A particular execution of a grip configuration can be influenced by many unpredictable factors. To avoid this problem, each grip is executed a sufficiently large number of times, by varying the location and orientation in the presentation of the object.

The number of feasible grips that are computed for each single object is usually large, varying from several dozens to more than one hundred. The repetition above mentioned could lead to a non practical number of executions, so for each object only a few configuration grips are selected to be executed. This selection consists of the most representative configurations of each object. Each configuration grip is executed 12 times, 4 times for three different orientations of the object.

Since we are also interested in studying the grasping performances in different circumstances, several characteristics of the environment are tested. These are the weight of the objects and the friction coefficient. Two qualitative categories for each of both conditions are distinguished: heavy and light objects, and high and low friction. The different weight is obtained by making two different sets of objects similar in appearance, but made of different material. Different contact friction is achieved by using a latex fingertip to envelope the fingers.

A series of experiments where done following this experimental protocol. Three different combinations of physical properties were tested: light objects and low friction (light/low), heavy objects and high friction (heavy/high); and light objects and high friction (light/high). More than eight hundred samples were obtained from this exhaustive experimentation. Table 1 shows the number of different grips executed and the percentages of grips that resulted in each class of $\Omega$.

# 4 GRASP RELIABILITY PREDICTION

The learning methodology that we propose is composed of two main algorithmic components. First, a prediction scheme that computes the most likely reliability class of an untested grip, using previous experience as reference. This component assumes the existence of a set of previously executed grips having the values of the descriptors and their reliability class known.

The second component, that will be referred as exploration function, is responsible of building such set of previous attempts by successive selection of the most appropriate grip candidates. In this section we focus on the first component.

In theoretical terms a data set of previous experience is composed of $N$ executed triplets. Each grip $g_i, i = 1 \dots N$ is described by the nine visual features $q_1, \dots q_9$ introduced in subsection 3. The 9-dimensional space $G_S$ is formed by the ranges of the values of the features. Moreover, we have also recorded the performance of the grip and have assigned it to a class $\omega_i \in \Omega$ for each $g_i$.

A prediction function tries to assess the most likely reliability class for a candidate grasp $g_q \in G_S$ using as reference the previous experience. There exists a wide bibliography on the building of such functions based on the Bayesian decision theory and other non-statistical approaches. In this work we have studied three different approaches for the implementation of the prediction function.

### 4.1 Density estimation

The first one is a statistical parametric method[4]. It assumes that the samples that belong to every reliability category are distributed in the feature-space according to a particular density function. In our implementation this is a multivariate normal density. We use the existent datasets to estimate the parameters of this density functions, in our case, the *mean* $\mu_{\omega_i}$ and the *covariance matrix* $\Gamma_{\omega_i}$ where $\omega_i \in \Omega$. For our purposes we are interested of the posterior probability $p(\omega_i|g_q)$.

$$p(\omega_i|g_q) \approx \exp\left(-\frac{1}{2}(x-\mu_{\omega_i})^T\Gamma_{\omega_i}^{-1}(x-\mu_{\omega_i}) - \frac{1}{2}\log\det\Gamma_{\omega_i} + \log p(\omega_i)\right) \quad (1)$$

The most likely class is, then, the one with a higher conditional probability.

### 4.2 Voting KNN classification rule

A prediction function has the form $F(g) = \bar{\omega}$ where $g \in G_S$ and $\bar{\omega} \in \Omega$. There exists a wide bibliography on the building of such functions based on the Bayesian decision theory [4]. In this paper we have chosen the approach of the non-parametric techniques, in particular the *voting k-nearest neighbor (KNN) rule* [6, 4], for modeling this function. The non-parametric techniques do not assume any density distribution of the features and the classes. To predict the class of a *query* point $g_q$, the KNN rule counts the K-nearest neighbors and chooses the class that most often appears, the most voted.

In our implementation we have introduced some modifications to the basic schema. First we use the euclidean metric for measuring the distance between the points in the $G_S$. We weighted the contribution of each of the KNN points according to its distance to the query point. This gives more importance to the closer points. The kernel function used is $K(d) = \frac{1}{1+(d/T)}$, where T is an adjustable parameter, and $d$ is the distance.

We define $KNN(g_q) = \{(g_i, \omega_i), i = 1 \dots k, g_i \in G_S, \omega_i in \Omega\}$ as the *k* closest points to $g_q$ and $d_i$ their corresponding distances from $g_q$. The probability corresponding to a class $\bar{\omega}$ are computed using this expression:

$$p(\bar{\omega}, g_q) = \sum_{\substack{g_i \in KNN(g_q) \\ \omega_i = \bar{\omega}}} \frac{K(d_i)}{\sum_{g_j \in KNN(g_q)} K(d_j)} \quad (2)$$

Function P is also an expression of the posterior probability [6]. Our predictor would be defined as $F(g_q) = argmax_{\omega \in \Omega}\{p(\omega, g_q)\}$. That is, the class predicted $\omega$ is the one with the largest probability $p(\omega, g_q)$.

**Table 2.** COMPARISON USING THE LIGHT/HIGH SAMPLE DATASET

| | **0** | **1** | **2** | **3** | **4** | $\bar{e}$ |
|---|---|---|---|---|---|---|
| **random** | 23.5% | 26.2% | 20.3% | 20.7% | 9.3% | **0.415** |
| **density est.** | 35.0% | 20.3% | 15.6% | 17.2% | 11.9% | **0.365** |
| **knn** | 51.1% | 21.7% | 13.3% | 11.1% | 2.8% | **0.223** |

Percentages of misclassifications depending on the error distance. Distance 0 indicates successful classifications.

### 4.3 Validation and comparison of the methods

Three basic questions need to be answered about the prediction capabilities of the rules described in this section: first, are they able to predict anything at all?; second, are they able to generalize across different objects?; and third, did we have enough data to properly construct a risk function? To answer these questions we have developed a cross-validation method named *leave-one-grasp-out validation* similar to the well known *leave-one-out validation* and *n-fold cross-validation* [4]. This consists of the following steps: 1) given the whole data set, remove all the points of a particular grasp configuration and use this subset as validation set; 2) use the remaining samples for predicting the outcomes of the validation set and compute the mean error; 3) repeat steps 1) and 2) for all configurations. The validation error will be the mean error of the iterations of step 2). The goal of removing all the points of a configuration from the data set is to eliminate points similar to the query grasp in the experience dataset, thus testing generalization properties.

The error metric is based on the concept of *misclassification error distance*. The distance between two consecutive classes is defined as 1, that between A and C as 2, etc. In this way define the error distance $e(g_q) = \{0, \dots, 4\}$ for the prediction of a given query grip. Given a set of predictions $G = \{g_i, i = 1 \dots n\}$, we define the average error metric $\bar{e}(G) = \sum e(g_i)/4$.

Moreover, we compare these prediction methods against the theoretical results that would be obtained by a prediction method that would have chosen *randomly* the predicted class.

The performance of this methods is obtained using the validation procedure described above. Table tab:fullsize shows the results obtained for one the sample datasets (light objects and high friction). The results in the other two cases were similar. The figures obtained indicate the the KNN prediction function improves clearly the other prediction functions, moreover it obtains better results that the naive random prediction.

This results show its validity of KNN function for prediction within this problem. Finally, we also measure the evolution of the performance of the KNN prediction method with different sizes of the sample dataset (fig. 6) and we conclude that the performance improves when the available experience dataset is larger[8, 9].

## 5 ACTIVE LEARNING FOR EXPERIENCE ACQUISITION

The results of the analysis of prediction methods indicate that it is possible to predict reasonably well the reliability class of a grasp if enough previous experience is available. In this part of the project we question if it is possible to reach a similar degree of performance with less experience. In particular we aim at designing an exploration procedure that guides the continuous execution of grasps with the goal of acquiring the maximum performance possible with the minimum number of executed trials.

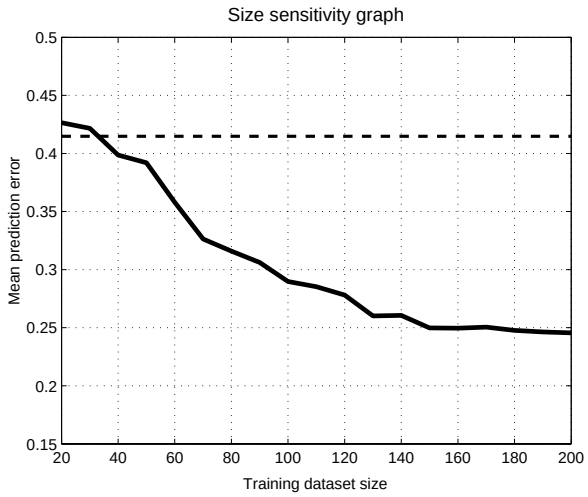



**Figure 6.** Evolution of the error when the size of the available data set varies. The Solid black line represents the errors obtained by the KNN prediction method, while the dashed line is the threshold of the random error.

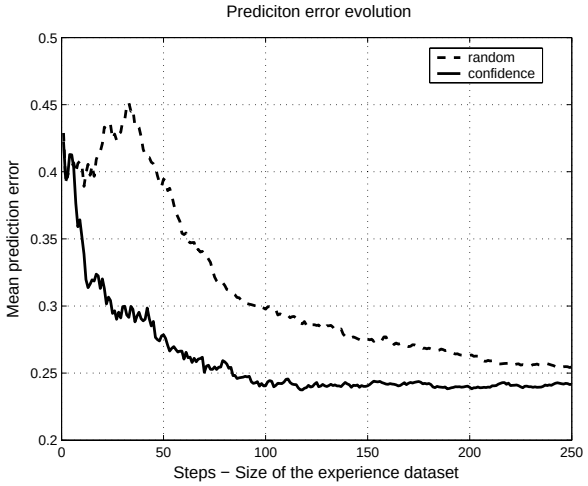



**Figure 7.** Evolution of the prediction error using the Light/High sample dataset.

In practice, the task of such exploration procedure is to select the next grasp to execute among a set of candidates. This selection must be done in order to improve the predictive capabilities of the stored experience, i.e., the set of already executed grasps.

The algorithm we propose assumes that at any point during the training of the grasping system a set of candidate grips $g_i \in G_S$ is proposed and the algorithm has to select the next grasp to be executed. To accomplish this task, it can take into account the results of previous experiments.

The approach we propose for the selection is inspired in the idea hinted by Thrun [16], "queries are favored that have the least predictable outcome". That is, those candidates which category is less predictable are preferred. This idea is based on the intuition that such candidates are located in areas where the implicit model represented by the experience data set is less clear.

We implement this idea by defining the term *prediction confidence*. For every grip candidate $g_i$, a class $\omega_i \in \Omega$ is computed using the KNN prediction scheme defined in the previous section. The confidence of that prediction is simply $p(\omega_i, g_i)$. In formal terms the prediction confidence for a grip $g_q$ is defined as $F_{conf}(g_q) = \max\{p(\omega|g_q)\}, \omega \in \Omega$. We use only the KNN prediction function since it proved to obtain the better results in the analyses described in previous section.

Once defined the notion of confidence, it is easy to describe the exploration function. It chooses the candidate with a minimum confidence value. Given a set of $m$ grasp candidates $G_q = \{g_1, \ldots, g_m\} \subset G_S$, the exploration function is defined as,

$$F_{exp}(G_q) = \underset{g_i \in G_q}{\operatorname{argmin}} F_{pred}(g_i) \tag{3}$$

Hereinafter, we will refer to this method as the *minimum confidence exploration*, or simply the risk exploration function.

Summarizing, this procedure procedure predicts a query point based on its similarity to its neighbors. This is a case of *instance-based* also known as *memory-based learning* [1], which is a numeric variant of the more symbolic *case-based reasoning* [18]. These approaches do not construct an explicit representation of the target function when training samples are provided, but simply store them.

## 5.1 Validation of the exploration procedure

The performance of the exploration/selection procedure is measured by the predictive capability of the set of samples selected/executed, which reliability class is known. This can be easily measured by using this dataset to predict the class of the samples contained in a secondary *validation test*. We have designed a validation framework that follows this principle. In its design we also take inspiration from the running of the robot in the training environment or in a learning experiment. In this situation the robot will execute a sequence of *selection-execution* actions. Each of these actions will follow the next steps:

1. One or more objects appear in the workspace of the robot. The grasps for them are computed. These are the grasp candidates
2. The robot selects one of them by using the exploration function.
3. The grasp is executed and the reliability test is applied.
4. The new grasp and the performance outcome are added to the experience dataset.

For the execution of the validation algorithm, we take the whole sample dataset available and extract a subset, *validation dataset* from it. The remaining is used as a *pool dataset*. In a sequence of selection steps, a small subset of *candidate samples* are extracted randomly from this pool. The exploration function, in our case, the *minimum confidence* rule, is applied to select one of these candidates. The selected candidate is added to the *experience dataset* and the discarded candidates are returned back to the pool. The performance measurement is done by using the samples in the *experience dataset* for predicting the samples in the *validation set*. The sequence is repeated until the pool dataset is emptied or it contains few samples.

This procedure is repeated a sufficiently large number of times varying the contents of the pool and validation datasets and the performance measurements for each size of the *experience dataset* are averaged.

Figure 7 presents the evolution of the prediction error for different sizes of the Light/High sample dataset, that is equivalent to the number of steps of the algorithm described in the above paragraphs. The graph in dashed lines shows the evolution of the prediction error when the sample to execute is selected randomly among the set

of candidates. This case would represent the evolution when no specific exploration rule is applied. From this graph, and similar ones obtained using the other sample datasets, we conclude that the proposed exploration procedure clearly improves the random selection function, and is able to reach maximum performance levels with less than a hundred trials.

## 6 CONCLUSION

This paper is the summary of large project [7] aimed at improving the grasping skills of a robot to work in the face of unknown conditions and uncertainty. We have approached this problem following two different ways.

The goal of the first part is to develop and to implement a grasping system able to use vision for extracting and using relevant information for grasp synthesis. The visual approach allows the system to deal with unknown objects. We have already emphasized the inclusion of the particular kinematics of the robotic hand within the grasp synthesis algorithms. As a result we have developed a couple of algorithms able to compute two and three-finger grasp for unknown objects using vision as only input, and a third algorithm that constrains their results to the hand geometry.

Moreover, these algorithms have resulted to be fast and suitable to use in real-time manipulation activities. Finally, a complete implementation on the UMass Torso has shown the strengths and limitations of the grasping system. This observations have motivated the approached followed in the next part of the project.

In this second part, we have presented the development of a learning framework for assessing robot grasp reliability. This framework is based on two learning algorithms and a representation of the data, built on a grasp characterization scheme composed of nine high level vision-based descriptors.

The first algorithm is aimed at predicting the reliability of an untested grip from its comparison to previous recorded attempts. The second algorithm, based on the idea of active learning, is an exploration rule that has to select among a set of candidate grips the next one to execute, having the goal of improving the predictive performance of the accumulated experience.

An experimental measurement of the reliability of a grasp have been developed and used to gather an exhaustive database of sample grips. Several validation frameworks that make use of this database, have been designed to test and validate the usefulness and properties of the proposed algorithms.

The results have proved that the algorithms proposed in this work are able to carry out the expected tasks with a reasonable level of performance, despite the complex and unpredictable nature of the task space.

Moreover, the experimental and practical approach followed indicates a possible path that service robotic applications willing to be used in every-day human environments could follow. The inclusion of active learning schemes in robot systems is an appropriate way to improve their adaptability to unmodeled or partially unknown environments and, thus, building real intelligent robot systems.

## ACKNOWLEDGMENTS

This work could not have been possible without the priceless help and collaboration of Andy Fagg and Eris Chinellato.

This work has been funded in part by the Ministerio de Ciencia y Tecnología under project DPI2001-3801, by the Generalitat Valenciana under projects GRUPOS 03/153, GV01-244, CTIDIA/2002/195, by the Fundació Caixa-Castelló under project P1-1B2001-28. The second author has been supported by the Ministerio de Ciencia y Tecnología under a FPI Program graduate fellowship.

This work is also partly funded by grants CISE/CDA-9703217 and IRI-9704530, DARPA MARS DABT63-99-1-0004, and NASA/RICIS. The authors wish to thank Roderic Grupen, David Wheeler, Robert Platt and Danny Radhakrishnan, who have provided much of the foundation which allowed the robot experiments.

## REFERENCES

[1] D.W. Aha, 'Lazy learning', *Artificial Intelligence Review*, **11**, 7–10, (1997).
[2] A. Bicchi and V. Kumar, 'Robotic grasping and contact: A review', in *IEEE Intl. Conf. on Robotics and Automation*, (April 2000).
[3] E. Chinellato, A. Morales, R.B. Fisher, and A.P. del Pobil, 'Visual features for characterizing robot grasp quality', *IEEE Transactions on Systems, Man and Cybernetics (Part C)*, (2004). In Press.
[4] R.O. Duda, P.E. Hart, and D.G. Stork, *Pattern Classification*, John Wiley & Sons, Inc., 2nd edn., 2001.
[5] B. Faverjon and J. Ponce, 'On computing two-finger force-closure grasps of curved 2D objects', in *IEEE Intl. Conf. on Robotics and Automation*, pp. 424–429, (1991).
[6] T. M. Mitchell, *Machine Learning*, McGraw Hill, 1997.
[7] A. Morales, *Learning to predict grasp reliability with a multifinger robot hand by using visual features*, Ph.D. dissertation, Departament of Computer and Engineering Science, Universitat Jaume I, Castellón, Spain, January 2004. `http://www.robot.uji.es/people/morales/thesis`.
[8] A. Morales, Chinellato E, A.H. Fagg, and A.P. del Pobil, 'Experimental prediction of the preformance of grasps tasks from visual fetures', in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pp. 3423–3428, Las Vegas, Nevada, (October 2003).
[9] A. Morales, Chinellato E, A.H. Fagg, and A.P. del Pobil, 'Using experience for assessing grasp reliability', in *International Conference on Humanoid Robots (Humanoids 2003)*, Karlsruhe, Germany, (October 2003). On CD-ROM.
[10] A. Morales, G. Recatalá, P.J. Sanz, and A.P del Pobil, 'Heuristic vision-based computation of planar antipodal grasps on unknown objects', in *IEEE Intl. Conf. on Robotics and Automation*, pp. 583–588, Seoul, Korea, (May 2001).
[11] A. Morales, P.J. Sanz, and A.P. del Pobil, 'Vision-based computation of three-finger grasps on unknown planar objects', in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pp. 1693–1698, Lausanne, Switzerland, (October 2002).
[12] A. Morales, P.J. Sanz, A.P del Pobil, and A.H. Fagg, 'An experiment in constraining vision-based finger contact selection with gripper geometry', in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, pp. 1711–1716, Lausanne, Switzerland, (October 2002).
[13] A.M. Okamura, N.S. Smaby, and M.R Cutkosky, 'An overview of dextereous manipulation', in *IEEE Intl. Conf. on Robotics and Automation*, pp. 255–260, San Francisco, California, (April 2000).
[14] J. Ponce and B. Faverjon, 'On computing three-finger force-closure grasps of polygonal objects', *IEEE Transactions on Robotics and Automation*, **11**(6), 868–881, (1995).
[15] K.B. Shimoga, 'Robot grasp synthesis: A survey', *International Journal of Robotics Research*, **3**(15), 230–266, (June 1996).
[16] S. Thrun, 'Exploration in active learning', in *The Handbook of Brain Theory and Neural Networks*, ed., Michael A. Arbib, 381–384, MIT Press, (1995).
[17] UMass humaniod torso. `http://www-robotics.cs.umass.edu`.
[18] D. L. Waltz, 'Memory-based reasoning', in *The Handbook of Brain Theory and Neural Networks*, ed., Michael A. Arbib, 661–662, MIT Press, (1995).

# Paper Session IV

August 24, 9:00 - 10:30

- **On Ability to Automatically Execute Agent Programs with Sensing**, S. Sardina, G. DeGiacomo, Y. Lesperance, H. Levesque
- **Have Another Look: On Failures and Recovery in Perceptual Anchoring**, M. Broxvall, S. Coradeschi, L. Karlsson, A. Saffiotti
- **Flexible Interval Planning in Concurrent Temporal Golog**, A. Finzi, F. Pirrri

*

# On Ability to Autonomously Execute Agent Programs with Sensing

**Sebastian Sardina**[1] and **Giuseppe De Giacomo**[2] and **Yves Lespérance**[3] and **Hector J. Levesque**[4]

**Abstract.** Most existing work in agent programming assumes an execution model where an agent has a knowledge base (KB) about the current state of the world, and makes decisions about what to do in terms of what is entailed or consistent with this KB. Planning then involves looking ahead and gauging what would be consistent or entailed at various stages by possible future KBs. We show that in the presence of sensing, such a model does not always work properly, and propose an alternative that does. We then discuss how this affects agent programming language design/semantics.

## 1 INTRODUCTION

There has been considerable work on formal models of deliberation/planning under incomplete information, where an agent can perform sensing actions to acquire additional information. This problem is very important in agent applications such as web information retrieval/management. However, much of the previous work on formal models of deliberation—i.e., models of knowing how, ability, epistemic feasibility, executabiliy, etc. such as [14, 4, 9, 11, 19]—has been set in epistemic logic-based frameworks and is hard to relate to work on agent programming languages (e.g. 3APL [8], AgentSpeak(L) [17]). In this paper, we develop new non-epistemic formalizations of deliberation that are much closer and easier to relate to standard agent programming language semantics based on transition systems.

When doing deliberation/planning under incomplete information, one typically searches over a set of states, each of which is associated with a knowledge base (KB) or theory that represents what is known in the state. To evaluate tests in the program and to determine what transitions/actions are possible, one looks at what is *entailed* by the current KB. To allow for future sensing results, one looks at which of these are *consistent* with the current KB. We call this type of approach to deliberation "entailment and consistency-based" (EC-based). In this paper, we argue that EC-based approaches do not always work, and propose an alternative. Our accounts are formalized within the situation calculus and use a simple programming language based on ConGolog [6] to specify agent programs as described in Section 2, but we claim that the results generalize to most proposed agent programming languages/frameworks. We point out that this paper is mainly concerned with the semantics of the deliberation process and not much with the actual algorithms implementing this process.

We initially focus on deterministic programs/plans and how to formalize when an agent knows how to execute them. For such deterministic programs, what this amounts to is ensuring that the agent will always know what the next step to perform is, and no matter what sensing results are obtained, the agent will eventually get to the point where it knows it can terminate. In Sections 3 and 4, we develop a simple EC-based account of knowing how (*KHow*$_{EC}$). We show that this account gives the wrong results on a simple example involving indefinite iteration. Then, we show that whenever this account says that a deliberation/planning problem is solvable, there is a *conditional plan* (a finite tree program without loops) that is a solution. It follows that this account is limited to problems where the total number of steps needed can be bounded in advance. We claim that this limitation is not specific to the simple account and applies to all EC-based accounts of deliberation.

The source of the problem with the EC-based account is the use of local consistency checks to determine which sensing results are possible. This does not correctly distinguish between the models that satisfy the overall domain specification (for which the plan must work) and those that do not. To get a correct account of deliberation, one must take into account what is true in different models of the domain together with what is true in all of them (what is entailed). In Section 5, we develop such an entailment and truth-based account (*KHow*$_{ET}$), argue that it intuitively does the right thing, and show how it correctly handles our test examples.

We end by reviewing the paper's contributions, discussing the lessons for agent programming language design, and sketching other related results that we have but were left out due to lack of space.

## 2 THE SITUATION CALCULUS AND INDIGOLOG

The technical machinery we use to define program execution in the presence of sensing is based on that of [7, 6]. The starting point in the definition is the situation calculus [12]. We will not go over the language here except to note the following components: there is a special constant $S_0$ used to denote the *initial situation*, namely that situation in which no actions have yet occurred; there is a distinguished binary function symbol $do$ where $do(a, s)$ denotes the successor situation to $s$ resulting from performing the action $a$; relations whose truth values vary from situation to situation are called (relational) *fluents*, and are denoted by predicate symbols taking a situation term as their last argument. There is a special predicate $Poss(a, s)$ used to state that action $a$ is executable in situation $s$. We assume that actions

[1] Dept. of Computer Science, University of Toronto, Toronto, Canada, email: ssardina@cs.toronto.edu
[2] Dip. Informatica e Sistemistica, Univer. di Roma "La Sapienza", Roma, Italy, email: degiacomo@dis.uniroma1.it
[3] Dept. of Computer Science, York University, Toronto, Canada, email: lesperan@cs.yorku.ca
[4] Dept. of Computer Science, University of Toronto, Toronto, Canada, email: hector@cs.toronto.edu

return binary sensing results, and we use the predicate $SF(a, s)$ to characterize what the action tells the agent about its environment. For example, the axiom

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

states that the action $senseDoor(d)$ tells the agent whether the door is open in situation $s$. For actions with no useful sensing information, we write $SF(a, s) \equiv True$.

Within this language, we can formulate domain theories which describe how the world changes as a result of the available actions. Here, we use basic action theories [18] of the following form:

- A set of foundational, domain independent axioms for situations $\Sigma$ as in [18].
- Axioms describing the initial situation, $S_0$.
- Action precondition axioms, one for each primitive action $a$, characterizing $Poss(a, s)$.
- Successor state axioms for fluents of the form
  $F(\vec{x}, do(a, s)) \equiv \gamma(\vec{x}, a, s)$
  providing the usual solution to the frame problem.
- Sensed fluent axioms, as described above, of the form
  $SF(A(\vec{x}), s) \equiv \phi(\vec{x}, s)$
- Unique names axioms for the primitive actions.

To describe a run of a program which includes both actions and their sensing results, we use the notion of a *history*, i.e., a sequence of pairs $(a, \mu)$ where $a$ is a primitive action and $\mu$ is 1 or 0, a sensing result. Intuitively, the history $\sigma = (a_1, \mu_1) \cdot \ldots \cdot (a_n, \mu_n)$ is one where actions $a_1, \ldots, a_n$ happen starting in some initial situation, and each action $a_i$ returns sensing value $\mu_i$. We use $end[\sigma]$ to denote the situation term corresponding to the history $\sigma$, and $Sensed[\sigma]$ to denote the formula of the situation calculus stating all sensing results of the history $\sigma$. Formally,

$end[\epsilon] = S_0$, where $\epsilon$ is the empty history; and
$end[\sigma \cdot (a, \mu)] = do(a, end[\sigma])$.

$Sensed[\epsilon] = True$;
$Sensed[\sigma \cdot (a, 1)] = Sensed[\sigma] \wedge SF(a, end[\sigma])$;
$Sensed[\sigma \cdot (a, 0)] = Sensed[\sigma] \wedge \neg SF(a, end[\sigma])$.

Next we turn to programs. We consider a very simple deterministic language with the following constructs:

| | |
|---|---|
| $a$ | primitive action |
| $\delta_1; \delta_2$ | sequence |
| **if** $\phi$ **then** $\delta_1$ **else** $\delta_2$ **endIf** | conditional |
| **while** $\phi$ **do** $\delta$ **endWhile** | while loop |

This is a small subset of ConGolog [6] and we use its single step transition semantics in the style of [16]. This semantics introduces two special predicates $Trans$ and $Final$: $Trans(\delta, s, \delta', s')$ means that by executing program $\delta$ in situation $s$, one can get to situation $s'$ in one elementary step with the program $\delta'$ remaining to be executed; $Final(\delta, s)$ means that program $\delta$ may successfully terminate in situation $s$.

*Offline executions* of programs, which are the kind of executions originally proposed for Golog and ConGolog [10, 6], are characterized using the $Do(\delta, s, s')$ predicate, which means that there is an execution of program $\delta$ that starts in situation $s$ and terminates in situation $s'$. This holds if there is a sequence of legal transitions from the initial configuration up to a final configuration:

$$Do(\delta, s, s') \stackrel{\text{def}}{=} \exists \delta'. Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s'),$$

where $Trans^*$ is the reflexive transitive closure of $Trans$. An offline execution of $\delta$ from $s$ is a sequence of actions $a, \ldots, a_n$ such that: $\mathcal{D} \cup \mathcal{C} \models Do(\delta, s, do(a_n, \ldots, do(a_1, s) \ldots))$, where $\mathcal{D}$ is an action theory as mentioned above, and $\mathcal{C}$ is a set of axioms defining the predicates $Trans$ and $Final$ and the encoding of programs as first-order terms [6].

Observe that an offline executor has no access to sensing results, available only at runtime. IndiGolog, an extension of ConGolog to deal with online executions with sensing, is proposed in [7]. The semantics defines an *online execution* of a program $\delta$ starting from a history $\sigma$. We say that a configuration $(\delta, \sigma)$ *may evolve* to configuration $(\delta', \sigma')$ *w.r.t. a model $M$* (relative to an underlying theory of action $\mathcal{D}$) iff [5] (i) $M$ is a model of $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\}$, and (ii)

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma_i]\} \models Trans(\delta, end[\sigma], \delta', end[\sigma'])$$

and (iii)

$$\sigma' = \begin{cases} \sigma \cdot (a, 1) & \text{if } end[\sigma'] = do(a, end[\sigma]) \\ & \text{and } M \models SF(a, end[\sigma]) \\ \sigma \cdot (a, 0) & \text{if } end[\sigma'] = do(a, end[\sigma]) \\ & \text{and } M \not\models SF(a, end[\sigma]). \\ \sigma & \text{if } end[\sigma'] = end[\sigma], \end{cases}$$

The model $M$ above is only used to represent a possible environment and, hence, it is just used to generate the sensing results of the corresponding environment. Finally, we say that a configuration $(\delta, \sigma)$ is *final* whenever

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models Final(\delta, end[\sigma]).$$

Using these two concepts of configuration evolution and final configurations, one can define various notions of online, incremental, executions of programs as a sequence of legal configuration evolutions, possibly terminating in a final configuration.

## 3 DELIBERATION: EC-BASED ACCOUNT

Perhaps the first approach to come to mind for defining when an agent knows how/is able to execute a deterministic program $\delta$ in a history $\sigma$ goes as follows: the agent must always know what the next action prescribed by the program is and be able to perform it such that no matter what sensing output is obtained as a result of doing the action, she can continue this process with what remains of the program and, eventually, reach a configuration where she knows she can legally terminate. We can formalize this idea as follows.

We define *KHow*$_{EC}(\delta, \sigma)$ to be the smallest relation $\mathcal{R}(\delta, \sigma)$ such that:

(E1) if $(\delta, \sigma)$ is *final*, then $\mathcal{R}(\delta, \sigma)$;
(E2) if there exists an action $a$ such that $(\delta, \sigma)$ *may evolve* to configuration $(\delta', \sigma \cdot (a, \mu))$ for some $\delta'$ and $\mu$ *w.r.t. some model* of theory $\mathcal{D}$, and $\mathcal{R}(\delta', \sigma \cdot (a, \mu_i))$ holds for *every* configuration $(\delta', \sigma \cdot (a, \mu_i))$ such that $(\delta, \sigma)$ *may evolve* to *w.r.t. some model* $M_i$ of theory $\mathcal{D}$, then $\mathcal{R}(\delta, \sigma)$.

The first condition states that every terminating configuration is in the relation *KHow*$_{EC}$. The second condition states that if a configuration performs an action transition and for every consistent sensing result, the resulting configuration is in *KHow*$_{EC}$, then this configuration is also in *KHow*$_{EC}$.

[5] This definition is more general than the one in [7], where the sensing results were assumed to come from the actual environment rather than from a model (a model can represent any possible environment).

Note that, here, the agent's lack of complete knowledge in a history $\sigma$ is modeled by the theory $\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\}$ being incomplete and having many different models. *KHow*$_{EC}$ uses entailment to ensure that the information available is sufficient to determine which transition should be performed next. *KHow*$_{EC}$ uses consistency to determine which sensing results can occur, for which the agent needs to have a subplan that leads to a final configuration. Due to this, we say that *KHow*$_{EC}$ is an *entailment and consistency-based* (EC-based) account of knowing how.

This EC-based account of knowing how seems quite intuitive and attractive. However it has a fundamental limitation: it fails on programs involving indefinite iteration. The following simple example from [9] shows the problem.

Consider a situation in which an agent wants to cut down a tree. Assume that the agent has a primitive action *chop* to chop at the tree, and also assume that she can always find out whether the tree is down by doing the (binary) sensing action $look$. If the sensing result is 1, then the tree is down; otherwise the tree remains up. There is also a fluent $RemainingChops(s)$, which we assume ranges over the natural numbers $\mathbb{N}$ and whose value is unknown to the agent, and which is meant to represent how many *chop* actions are still required in $s$ to bring the tree down. The agent's goal is to bring the tree down, i.e., bringing about a situation $s$ such that $Down(s)$ holds, where

$$Down(s) \stackrel{\text{def}}{=} RemainingChops(s) = 0$$

The action theory $\mathcal{D}_{tc}$ is the union of:

1. The foundational axioms for situations $\Sigma$.
2. $\mathcal{D}_{una} = \{chop \neq look\}$.
3. $\mathcal{D}_{ss}$ contains the following successor state axiom:

$$\begin{array}{l} RemainingChops(do(a,s)) = n \equiv \\ \quad (a = chop \wedge RemainingChops(s) = n+1) \vee \\ \quad (a \neq chop \wedge RemainingChops(s) = n). \end{array}$$

4. $\mathcal{D}_{ap}$ contains the following two precondition axioms:

$$\begin{array}{l} Poss(chop, s) \equiv (RemainingChops > 0), \\ Poss(look, s) \equiv True. \end{array}$$

5. $\mathcal{D}_{S_0} = \{RemainingChops(S_0) \neq 0\}$.
6. $\mathcal{D}_{sf}$ contains the following two sensing axioms:

$$\begin{array}{l} SF(chop, s) \equiv True, \\ SF(look, s) \equiv (RemainingChops(s) = 0). \end{array}$$

Notice that sentence $\exists n.RemainingChops(S_0) = n$ (where the variable $n$ ranges over $\mathbb{N}$) is trivially entailed by this theory so "infinitely" hard tree trunks are ruled out. Nonetheless, the theory *does not* entail the sentence $RemainingChops(S_0) < k$ for any constant $k \in \mathbb{N}$. Hence, there exists some $n \in \mathbb{N}$, though unknown and unbounded, such that the tree will fall after $n$ chops. Because of this, intuitively, we should have that the agent can bring the tree down, since if the agent keeps chopping, the tree will eventually come down, and the agent can find out whether it has come down by looking. Thus, for the program

$$\delta_{tc} = \textbf{while } \neg Down \textbf{ do } chop; look \textbf{ endWhile}$$

we should have that *KHow*$_{EC}(\delta_{tc}, \epsilon)$ (note that $\delta_{tc}$ is deterministic). However, this is not the case:

**Theorem 1** *Let $\delta_{tc}$ be the above program to bring the tree down. Then, for all $k \in \mathbb{N}$, KHow$_{EC}(\delta_{tc}, [(chop, 1) \cdot (look, 0)]^k)$ does not hold. In particular, when $k = 0$, KHow$_{EC}(\delta_{tc}, \epsilon)$ does not hold.*

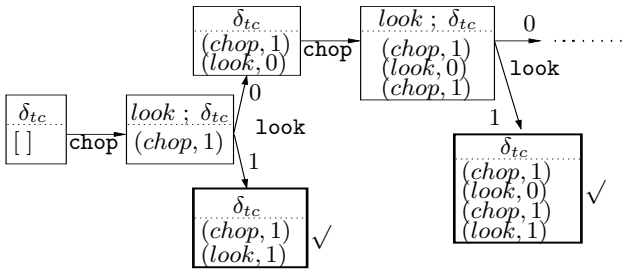



**Figure 1.** Online execution tree of program $\delta_{tc}$. Each box represents a configuration with the remaining program at the top and the current history at the bottom. Terminating configurations are marked with a check sign.

Thus, the simple EC-based formalization of knowing how gives the wrong result for this example. Why is this so? Intuitively, it is easy to check that if the agent knows how (to execute) the initial configuration, i.e., *KHow*$_{EC}(\delta_{tc}, \epsilon)$ holds, then she knows-how (to execute) *every* possible finite evolution of it, i.e., for all $j \in \mathbb{N}$, *KHow*$_{EC}(\delta_{tc}, [(chop, 1) \cdot (look, 0)]^j)$ and *KHow*$_{EC}((look; \delta_{tc}), [(chop, 1) \cdot (look, 0)]^j \cdot (chop, 1))$. Now consider the hypothetical scenario in which an agent keeps chopping and looking forever, always seeing that the tree is not down. There is no model of $\mathcal{D}_{tc}$ where $\delta_{tc}$ yields this scenario, as the tree is guaranteed to come down after a finite number of chops. However, by the above, we see that *KHow*$_{EC}$ is, in some way, taking this case into account in determining whether the agent knows how to execute $\delta_{tc}$ (see Figure 1). This happens because every finite prefix of this never-ending execution is indeed *consistent* with $\mathcal{D}_{tc}$. The problem is that the set of *all* of them together is not. This is why *KHow*$_{EC}$ fails, which can also be viewed as a lack of compactness issue. In the next section, we show that *KHow*$_{EC}$'s failure on the tree chopping example is due to a general limitation of the *KHow*$_{EC}$ formalization. Note that Moore's original account of ability [14] is closely related to *KHow*$_{EC}$ and also fails on the tree chopping example [9].

## 4 *KHow*$_{EC}$ ONLY HANDLES BOUNDED PROBLEMS

In this section, we show that whenever *KHow*$_{EC}(\delta, \sigma)$ holds for some program $\delta$ and history $\sigma$, there is simple kind of conditional plan, what we call a $TREE$ program, that can be followed to execute $\delta$ in $\sigma$. Since for $TREE$ programs (and conditional plans), the number of steps they perform can be bounded in advance (there are no loops), it follows that *KHow*$_{EC}$ will never be satisfied for programs whose execution cannot be bounded in advance. Since there are many such programs (for instance, the one for the tree chopping example), it follows that *KHow*$_{EC}$ is fundamentally limited as a formalization of knowing how and can only be used in contexts where attention can be restricted to bounded strategies. As in [19], we define the class of *(sense-branch) tree programs* $TREE$ with the following BNF rule:

$$dpt ::= nil \mid a; dpt_1 \mid sense_\phi; \textbf{if } \phi \textbf{ then } dpt_1 \textbf{ else } dpt_2$$

where $a$ is any non-sensing action, and $dpt_1$ and $dpt_2$ are tree programs.

This class includes conditional programs where one can only test a condition that has just been sensed. Thus as shown in [19], whenever a $TREE$ program is executable, it is also epistemically feasible, i.e.,

the agent can execute it without ever getting stuck not knowing what transition to perform next. $TREE$ programs are clearly deterministic.

Let us define a relation *KHowBy*$_{EC}$ : $Program \times History \times TREE$. The relation is intended to associate a program $\delta$ and history $\sigma$ for which *KHow*$_{EC}$ holds with some $TREE$ program(s) that can be used as a strategy for successfully executing $\delta$ in $\sigma$.

We define *KHowBy*$_{EC}(\delta, \sigma, \delta^{tp})$ to be the least relation $\mathcal{R}(\delta, \sigma, \delta^{tp})$ such that:

(A) if $(\delta, \sigma)$ is *final*, then $\mathcal{R}(\delta, \sigma, nil)$;
(B) if $(\delta, \sigma)$ *may evolve* to configurations $(\delta', \sigma \cdot (a, 1))$ and $(\delta', \sigma \cdot (a, 0))$ w.r.t. some models $M_1$ and $M_2$, respectively, of theory $\mathcal{D}$, and there exist $\delta_1^{tp}$ and $\delta_0^{tp}$ such that $\mathcal{R}(\delta', \sigma \cdot (a, 1), \delta_1^{tp})$ and $\mathcal{R}(\delta', \sigma \cdot (a, 0), \delta_0^{tp})$ hold, then $\mathcal{R}(\delta, \sigma, (a; \textbf{if } \phi \textbf{ then } \delta_1^{tp} \textbf{ else } \delta_0^{tp} \textbf{ endIf}))$ where $\phi$ is the condition on the right hand side of the sensed fluent axiom for $a$ (i.e., action $a$ senses the truth value of formula $\phi$).
(C) if there exists an action $a$ and a program $\delta'$ for which $(\delta, \sigma)$ *may evolve* to configuration $(\delta', \sigma \cdot (a, \mu))$ *only* for some *unique* sensing outcome $\mu$ and some model $M$ of theory $\mathcal{D}$, and there exist $\delta''$ such that $\mathcal{R}(\delta', \sigma \cdot (a, \mu), \delta'')$ holds, then $\mathcal{R}(\delta, \sigma, (a; \delta''))$.

Condition (A) deals with the simple case of a terminating configuration; condition (B) handles the case in which the current configuration can perform a step with some (sensing) action $a$ and where both sensing outcomes 1 and 0 are eventually possible/consistent; and condition (C) deals with the simpler cases of a non-sensing action step and a sensing action step for which there is only one consistent sensing outcome.

It is possible to show that whenever *KHowBy*$_{EC}(\delta, \sigma, \delta^{tp})$ holds, then *KHow*$_{EC}(\delta, \sigma)$ and *KHow*$_{EC}(\delta^{dp}, \sigma)$ hold, and the $TREE$ program $\delta^{tp}$ is guaranteed to terminate in a $Final$ situation of the given program $\delta$ (in all models).

**Theorem 2** *For all programs $\delta$, histories $\sigma$, and programs $\delta^{tp}$, if* *KHowBy*$_{EC}(\delta, \sigma, \delta^{tp})$ *then we have that*

- *KHow*$_{EC}(\delta, \sigma)$ *and* *KHow*$_{EC}(\delta^{dp}, \sigma)$ *hold; and*
- *There is a common execution for $\delta^{tp}$ and $\delta$ from $end[\sigma]$:*

$$\mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\} \models \exists s. Do(\delta^{tp}, end[\sigma], s) \wedge Do(\delta, end[\sigma], s).$$

In addition, every configuration captured in *KHow*$_{EC}$ can be executed using a $TREE$ program.

**Theorem 3** *For all programs $\delta$ and histories $\sigma$, if* *KHow*$_{EC}(\delta, \sigma)$, *then there exists a program $\delta^{tp}$ such that* *KHowBy*$_{EC}(\delta, \sigma, \delta^{tp})$.

Since the number of steps a $TREE$ program performs can be bounded in advance, it follows that *KHow*$_{EC}$ will never hold for programs/problems that are solvable, but whose execution requires a number of steps that cannot be bounded in advance, as it is the case with the program in the tree chopping example. Thus *KHow*$_{EC}$ is severely restricted as an account of knowing how; it can only be complete when all possible strategies are bounded.

## 5 DELIBERATION: ET-BASED ACCOUNT

We saw earlier that the reason *KHow*$_{EC}$ failed on the tree chopping example was that it required the agent to have a choice of action that guaranteed reaching a final configuration even for histories that were inconsistent with the domain specification such as the infinite history corresponding to the hypothetical scenario described in the previous paragraph. There was a branch in the configuration tree that corresponded to that history. This occurred because "local consistency" was used to construct the configuration tree. The consistency check kept switching which model of $\mathcal{D} \cup \mathcal{C}$ (which may be thought as representing the environment) was used to generate the next sensing result, postponing the observation that the tree had come down forever. But in the real world, sensing results come from a fixed environment (even if we don't know which environment this is). It seems reasonable that we could correct the problem by fixing the model of $\mathcal{D} \cup \mathcal{C}$ used in generating possible configurations in our formalization of knowing how. This is what we will now do.

We define when an agent knows how to execute a program $\delta$ in a history $\sigma$ and a model $M$ (which represents the environment), *KHowInM*$(\delta, \sigma, M)$, as the smallest relation $\mathcal{R}(\delta, \sigma)$ such that:

(T1) if $(\delta, \sigma)$ is *final*, then $\mathcal{R}(\delta, \sigma)$;
(T2) if $(\delta, \sigma)$ *may evolve* to $(\delta', \sigma \cdot (a, \mu))$ w.r.t. $M$ and $\mathcal{R}(\delta', \sigma \cdot (a, \mu))$, then $\mathcal{R}(\delta, \sigma)$;

The only difference between this and *KHow*$_{EC}$ is that the sensing results come from the fixed model $M$. Given this, we obtain the following formalization of when an agent knows how to execute a program $\delta$ in a history $\sigma$:

*KHow*$_{ET}(\delta, \sigma)$
iff
for every model $M$ such that $M \models \mathcal{D} \cup \mathcal{C} \cup \{Sensed[\sigma]\}$,
*KHowInM*$(\delta, \sigma, M)$ holds.

We call this type of formalization *entailment and truth-based*, since it uses entailment to ensure that the agent knows what transitions she can do, and *truth in a model* to obtain possible sensing results.

We claim that *KHow*$_{ET}$ is actually correct for programs $\delta$ that are deterministic. For instance, it handles the tree chopping example correctly:

**Proposition 4** *KHow*$_{ET}(\delta_{tc}, \epsilon)$ *holds w.r.t. theory $\mathcal{D}_{tc}$.*

Furthermore, *KHow*$_{ET}$ is strictly more general than *KHow*$_{EC}$. Formally,

**Theorem 5** *For any background theory $\mathcal{D}$ and any configuration $(\delta, \sigma)$, if* *KHow*$_{EC}(\delta, \sigma)$ *holds, then* *KHow*$_{ET}(\delta, \sigma)$. *Moreover, there is a background theory $\mathcal{D}^*$ and a configuration $(\delta^*, \sigma^*)$ such that* *KHow*$_{ET}(\delta^*, \sigma^*)$ *holds, but* *KHow*$_{EC}(\delta^*, \sigma^*)$ *does not.*

## 6 DISCUSSION AND CONCLUSION

In an extended version of this paper, we show how the notion of ability to achieve a goal can be defined in terms of our notions of knowing how to execute a deterministic program. We observe that an EC-based definition of ability inherits the limitations of the EC-based definition of knowing how. Then, we examine knowing how to execute a *nondeterministic* program. We consider two ways of interpreting this: one (angelic knowing how) where the agent does planning/lookahead to make the right choices, and another (demonic knowing how) where the agent makes choices arbitrarily. We discuss EC-based and ET-based formalizations of these notions. Finally, we show how angelic knowing how can be used to specify a powerful planning construct in the IndiGolog agent programming language.

In this paper, we have looked at how to formalize when an agent knows how to execute a program, which in the general case, when the program is nondeterministic and the agent does lookahead and reasons about possible execution strategies, subsumes ability to achieve a goal. First, we have shown that an intuitively reasonable entailment and consistency-based approach to formalizing knowing how, $KHow_{EC}$, fails on examples like our tree chopping case and that, in fact, $KHow_{EC}$ can only handle problems that can be solved in a bounded number of steps, i.e. without indefinite iteration.

The problems of accounts like $KHow_{EC}$ when they are formalized in epistemic logic, such as Moore's [14], had been pointed out before, for instance in [9]. However, the reasons for the problems were not well understood. The results we have presented clarify the source of the problems and show what is needed for their solution. A simple meta-theoretic approach to knowing how fails; one needs to take entailment and truth into account together. (Even if we use a more powerful logical language with an knowledge operator, knowledge and truth must be considered together.)

Our non-epistemic accounts of knowing how are easily related to models of agent programming language semantics and our results have important implications for this area. While most work on agent programming languages (e.g. 3APL [8], AgentSpeak(L) [17], etc.) has focused on reactive execution, sensing is acknowledged to be important and there has been interest in providing mechanisms for run-time planning/deliberation. The semantics of such languages are usually specified as a transition system. For instance in 3APL, configurations are pairs involving a program and a belief base, and a transition relation over such pairs is defined by a set of rules. Evaluating program tests is done by checking whether they are entailed by the belief base. Checking action preconditions is done by querying the agent's belief base update relation, which would typically involve determining entailments over the belief base — the 3APL semantics abstracts over the details of this. Sensing is not dealt with explicitly, although one can suppose that it could be handled by simply updating the belief base (AgentSpeak(L) has events for this kind of thing).

As mentioned, most work in the area only deals with on-line reactive execution, where no deliberation/lookahead is performed; this type of execution just involves repeatedly selecting some transition allowed in the current configuration and performing it. However, one natural view is that *deliberation can simply be taken as a different control regime involving search over the agent program's transition tree*. In this view, a deliberating interpreter could first lookahead and search the program's transition tree to find a sequence of transitions that leads to successful termination and later execute this sequence. This assumes that the agent can chose among all alternative transitions. Clearly, in the presence of sensing, this idea needs to be refined. One must find more than just a path to a final configuration in the transition tree; one needs to find some sort of conditional plan or subtree where the agent has chosen some transition among those allowed, but must have branches for all possible sensing results. The natural way of determining which sensing results are possible is checking their consistency with the current belief base. Thus, what is considered here is essentially an EC-based approach.

Also in work on planning under incomplete information, e.g. [3, 15, 5], a similar sort of setting is typically used, and finding a plan involves searching a (finite) space of knowledge states that are compatible with the planner's knowledge. The underlying models of all these planners are meant to represent only the *current* possible states of the environment, which, in turn, are updated upon the hypothetical execution of an action at planning time. We use models that are dynamic in the sense that they represent the potential responses of the environment for *any* future state. In that way, then, what the above planners are doing is deliberation in the style of $KHow_{EC}$. An interesting case arises with answer set planning/programming, e.g. [2, 20, 21]. There, plans are found by inspecting all models of an underlying logic program and, hence, they seem, in principle, to be more in the lines of our ET-based approach to deliberation. Nonetheless, all these approaches are eventually restricted to propositional languages and, as a result, only bounded problems can be expressed.

Our results show that the ET-based view of deliberation is fundamentally flawed when sensing is present. It produces an account that only handles problems that can be solved in a bounded number of actions. As an approach to implementing deliberation, this may be perfectly fine. But as a semantics or specification, it is wrong. What is required is a much different kind of account, like our ET-based one.

Finally, we point out that even though one might argue that results concerning the indistinguishability of unbounded nondeterminism [13, 1] (e.g., $a^*b$ being observationally indistinguishable from $a^*b+a^\omega$) are a problem for our approach, this is not the case because we are assuming that agents can reason about *all* possible program executions/futures.

## REFERENCES


[1] K.R. Apt and E.R. Olderog, *Verifi cation of Sequential and Concurrent Programs*, Springer-Verlag, 1997.

[2] Chitta Baral and Michael Gelfond, *Reasoning Agents in Dynamic Domains*, chapter 12, 257–275, Kluwer, 2000.

[3] P. Bertoli, A. Cimatti, M. Roveri, and P. Traverso, 'Planning in nondeterministic domains under partial observability via symbolic model checking', in *Proc. of IJCAI-01*, pp. 473–478, (2001).

[4] Ernest Davis, 'Knowledge preconditions for plans', *Journal of Logic and Computation*, **4**(5), 721–766, (1994).

[5] Giuseppe De Giaccomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati, 'Planning with sensing for a mobile robot', in *Proc, of ECP-97*, pp. 156–168, (1997).

[6] Giuseppe De Giacomo, Yves Lespérance, and Hector J. Levesque, 'ConGolog, a concurrent programming language based on the situation calculus', *Artifi cial Intelligence*, **121**, 109–169, (2000).

[7] Giuseppe De Giacomo and Hector J. Levesque, 'An incremental interpreter for high-level programs with sensing', in *Logical Foundations for Cognitive Agents*, eds., Hector J. Levesque and Fiora Pirri, 86–102, Springer-Verlag, (1999).

[8] K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. J. Ch. Meyer, 'A formal semantics for an abstract agent programming language', in *Proc. of ATAL-97*, pp. 215–229, (1998).

[9] Yves Lespérance, Hector J. Levesque, Fangzhen Lin, and Richard B. Scherl, 'Ability and knowing how in the situation calculus', *Studia Logica*, **66**(1), 165–186, (2000).

[10] H. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**, 59–84, (1997).

[11] Fangzhen Lin and Hector J. Levesque, 'What robots can do: Robot programs and effective achievability', *Artifi cial Intelligence*, **101**, 201–226, (1998).

[12] John McCarthy and Patrick Hayes, 'Some philosophical problems from the standpoint of artifi cial intellig ence', in *Machine Intelligence*, eds., B. Meltzer and D. Michie, volume 4, 463–502, Edinburgh University Press, (1979).

[13] Robin Milner, *Communication and Concurrency*, Prentice Hall, 1989.

[14] Robert C. Moore, 'A formal theory of knowledge and action', in *Formal Theories of the Common Sense World*, eds., J. R. Hobbs and Robert C. Moore, 319–358, (1985).

[15] Ron Petrick and Fahiem Bacchus, 'A knowledge-based approach to planning with incomplete information and sensing', in *Proc. of AIPS-02*, pp. 212–221, (2002).

[16] Gordon Plotkin, 'A structural approach to operational semantics', Technical Report DAIMI-FN-19, Computer Science Dept., Aarhus University, Denmark, (1981).

[17] Anand S. Rao, 'AgentSpeak(L): BDI agents speak out in a logica computable language', in *Agents Breaking Away (LNAI)*, eds., W. Vander Velde and J. W. Perram, volume 1038, 42–55, Springer-Verlag, (1996).
[18] Raymond Reiter, *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, MIT Press, 2001.
[19] Sebastian Sardina, Yves De Giacomo, Giuseppe Lespénce, and Hector Levesque, 'On the semantics of deliberation in IndiGolog – from theory to implementation', *Annals of Mathematics and Artifi cial Intelligence*, **41**(2–4), 259–299, (2004). Previous version appeared in Proc. of KR-2002.
[20] T. Son, C. Baral, and S. McIlraith, 'Extending answer set planning with sequence, conditional, loop, non-deterministic choice, and procedure constructs', in *Proceedings of the AAAI Spring Symposium on Answer Set Programming: Towards Effi cient and Scalable Knowledge Representation and Reasoning*.
[21] Tran Son, Phan Huy Tu, and Chitta Baral, 'Planning with sensing actions and incomplete information using logic programming', in *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2004*, Lecture Notes in Computer Science, pp. 261–274, Fort Lauderdale, FL, USA. Springer.

# Have another look On Failures and Recovery Planning in Perceptual Anchoring

**Mathias Broxvall** and **Silvia Coradeschi** and **Lars Karlsson** and **Alessandro Saf£otti**
**Applied Autonomous Sensor Systems, Örebro University, Sweden**
{**mathias.broxvall,silvia.coradeschi,lars.karlsson,alessandro.saf£otti**}**@aass.oru.se**

**Abstract.** An important requirement for autonomous systems is the ability to detect and recover from exceptional situations such as failures in observations. In this paper we demonstrate how techniques for planning with sensing under uncertainty can play a major role in solving the problem of recovering from such situations. In this £rst step we concentrate on failures in perceptual anchoring, that is how to connect a symbol representing an object to the percepts of that object. We provide a classi£cation of failures and present planning-based methods for recovering from them. We illustrate our approach by showing tests run on a mobile robot equipped with a color camera.

## 1 Introduction

There is an increasing demand for intelligent robots capable of robust operation in unconstrained environments. One of the great challenges for these robots is the need to cope autonomously with exceptional situations that arise during the execution of the assigned tasks. Explicit coding of all the possible exceptions is clearly unfeasible for environments and tasks of a realistic complexity. A more effective approch is to endow the system with the ability to use knowledge-based techniques to reason about the state of the execution, detect anomalies, and automatically generate a contingency plan.

Most existing systems that take this approach (e.g., [8, 2, 11, 17, 19]) focus on the *external* state of the world, looking for discrepancies between the observed state and the expected one. Discrepancies can originate in the failure of actions performed by the robot as well as in exogenous events. There is however another common cause of problems in the execution of robot plans, which is more related to the *internal* state of the robot: failures in perception, including the inability to acquire the perceptual data needed to perform the desired actions. The ability of the robot to detect perceptual failures and to recover from them is pivotal to its providing autonomous and robust operation. In this context, Murphy and Hershberger [16] have suggested a two-step approach: a generate and test strategy for classifying sensor failures, and a recovery strategy where the failing sensory process is replaced with an equivalent process.

Several works in the £eld have addressed the problem of planning for perceptual actions. Perception planning has been studied as a means for gathering better visual information [14, 1], for achieving safer landmark-based navigation [15, 9], for performing tasks that involve sensing actions [10, 13], and for generating image processing routines [3]. None of these works, however, deal with the problem of failures in the perceptual actions and of the automatic recovery from these failures.

We propose to use AI planning techniques to automatically generate a plan to recover from failures in the perceptual processes. We focus on one speci£c type of perceptual process: *perceptual anchoring*. Perceptual anchoring is the process of creating and maintaining the right correspondence between the symbols used by the planner to denote objects in the world and the perceptual data in the sensorimotoric system that refer to the same objects. In a previous paper [4] a simple case of anchoring failure due to the accumulation of uncertainty has been investigated. In this paper, we extend that investigation and analyze all the different cases of ambiguity that can make the anchoring process fail. For each case, we show how that situation can be automatically detected and isolated, and how we can use a planner to generate a sequence of actions to recover from the failure when possible.

In the next section, we give a brief reminder of perceptual anchoring. In section 3 we classify the different ways in which anchoring can fail, and explain how they can be detected. In section 4 we show how the failure situation can be modeled in a planner and a recovery plan generated automatically for those cases that can be £xed. Finally, we demonstrate our technique by presenting a series of experiments run on a mobile robot equipped with a color camera.

## 2 Perceptual Anchoring

Autonomous systems embedded in the physical world typically incorporate two different types of processes: high-level cognitive processes, that perform abstract reasoning and generate plans for actions; and sensory-motoric processes, that observe the physical world and act on it. These processes have different ways to refer to physical objects in the environment. Cognitive processes typically (although not necessarily) use symbols to denote objects, like ‘b1’. Sensory-motoric processes typically operate from sensor data that originate from observing these objects, like a region in a segmented image. If the overall system has to successfully perform its tasks, it needs to make sure that these processes “talk about” the same physical objects. This has been de£ned as the *anchoring problem* [7], illustrated in Fig. 1:

> Anchoring is the process of creating and maintaining the correspondence between symbols and sensor data that refer to the same physical objects.

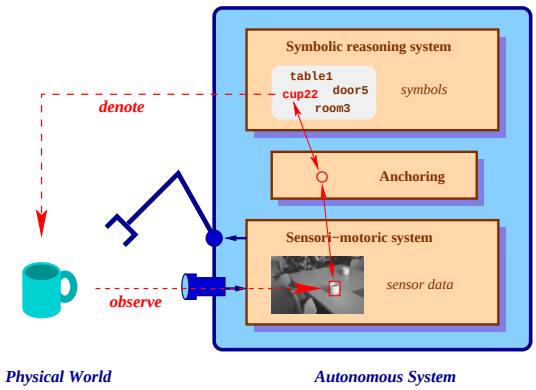



**Figure 1.** The perceptual anchoring problem.

In our work, we use the computational framework for anchoring de£ned in [5]. In that framework, the symbol-data correspondence for a speci£c object is represented by a data structure called an **anchor**. An anchor includes pointers to the symbol and sensor data being connected, together with a set of properties useful to re-identify the object, e.g., its color and position. These properties can also be used as input to the control routines.

Consider for concreteness a mobile robot equipped with a vision system and with a symbolic planner. Suppose that the planner has generated the action 'GoNear(b1)', where the symbol 'b1' denotes an object described in the planner as 'a tall green gas bottle'.[1] The 'GoNear' operator is implemented by a sensori-motor loop that controls the robot using the position parameters extracted from a region in the camera image. In order to execute the 'GoNear(b1)' action, the robot must make sure that the region used in the control loop is exactly the one generated by observing the object that the planner calls 'b1'. Thus, the robot uses a functionality called **Find** to link the symbol 'b1' to a region in the image that matches the description 'a tall green gas bottle'. The output of Find is an anchor that contains, among other properties, the $(x, y)$ position of the gas bottle, which is used by the 'GoNear' routine. While the robot is moving, a functionality called **Track** is used to update this position using new perceptual data. Should the gas bottle go temporarily out of view, e.g., because it is occluded by another object, the **Reacquire** functionality would be called to update the anchor as soon as the gas bottle is in view again. More details about perceptual anchoring can be found in [5, 6, 7].

A central ingredient in all the anchoring functionalities is the *matching* between the symbolic description given by the planner and the observed properties of the percepts generated by the sensor system. Matching is needed to decide which percepts to use to create or update the anchor for a given symbol. Matching between a symbolic description and a percept can be *partial* or *complete* [6]. It is complete if all the observed properties in the percept match the description and vice-versa. It is partial if all the observed properties in the percept match the description, but there is some property in the description that have not been observed (or not reliably).[2] For example, consider the description "a gas bottle with a yellow mark". A gas bottle in an image where no mark is visible provides a partial match, since the mark might not be visible from the current viewpoint.

[1] Throughout this paper, we use examples inspired by an ongoing rescue project where the robot is supposed to £nd overheated and dangerous gas bottles in a burnt building.

[2] Note that matching does not have to be a binary concept, but can be considered in degrees. In such cases, we use a threshold to determine what is partial and what is complete; the threshold can be raised if there are several matching candidates.

## 3 Anchoring with ambiguities

An important challenge in the anchoring process is how to treat ambiguous cases, that is cases for which it is not clear to what perceptual data the symbol should be associated. The £rst step to treat ambiguities is to detect that an ambiguity is actually present. The second step is then to try to resolve the ambiguity if possible or otherwise admit failure. In this section we concentrate on the detection of ambiguity, while in the next section we present how some classes of ambiguous situations can be recovered from using planning techniques.

To better characterize the ambiguous cases we need £rst to clarify the distinction between de£nite and inde£nite descriptions for an object. A description is *de£nite* when it denotes an unique object, for instance "the cup in my of£ce", supposing that I have just one cup in my of£ce. Linguistically one use in this case the article "the". An *inde£nite* description requires that the object corresponds to the description, but not that it is unique, for instance "a red cup". De£nite descriptions are especially challenging when an object is conceptually unique, but its perceptual properties do not characterize it unequivocally, for instance "the cup that I have seen before". This is a common event in the Reacquire functionality when more than one object matches the description of a previously seen object (in Reacquire descriptions are always de£nite). An example of this situation is shown later in this paper.

An important case in anchoring is when we have multiple candidate percepts matching the description and by consentrating on this we can indentify a number of different failures. The important variables when detecting and identifying an ambiguity in anchoring are the number of candidate percepts that match a description completely and partially, and whether the description involved is de£nite or inde£nite. In the following, we give a classi£cation of these ambiguities. We also describe what the Find and Reacquire functionalities return in each case, and what could constitute a recovery from the situation.

**Case 1: no candidates.** In this case no object matching the description is found. Therefore, both Find and Reacquire return a failure message. If the object might be somewhere else we generate a search-plan. If one has exhausted the search possibilities, there is a failure.

**Case 2: one or more partially matching candidates.** A partial matching indicates that one has inadequate information about some relevant properties of the perceived object(s). When this happens, both functionalities create temporary anchors for each of the candidates and return these anchors to be used by the recovery planner. Sometimes, one might still be able to determine whether this is the requested object - one might e.g. have prior information that there are no other similar objects around — and in those occasions, the case turns into one of a complete matching (see below). However, in most situations one will need to try to acquire more information about the object(s) in question, in order to get a complete match. Therefore, one needs to generate a recovery plan. The anchors created by the functionalities let the planner access to information about these objects while the recovery plan is constructed and executed. If the situation is successfully disambiguated, the planner informs the anchoring module which of the candidate perceived objects should be used for anchoring.

**Case 3: a single completely matching candidate.** This is the ideal

case: one just picks that candidate. Both functionalities create an anchor and return it to the planner.

**Case 4: one completely matching candidate, and some partially matching ones.** The inde£nite case is simple: one can just pick the completely matching candidate. For the de£nite case, that is also an option. However, if one is cautious and wants to ascertain that there is not an ambiguity hidden here, one might want to acquire more information to be able to rule out the candidates with incomplete matches. In our current implementation in the inde£nite case the Find functionality creates an anchor with the completely matching candidate and returns it to the planner. In the de£nite case both functionalities create anchors both for the complete matching candidate and the incomplete matching ones. They then return the anchors while making a distinction between the completely and partially matching ones.

**Case 5: multiple completely matching candidates.** Again, the inde£nite case is simple: just pick one of the candidates. The Find functionality creates an anchor with the completely matching candidate and returns it to the planner. In the de£nite case, however, this constitutes a serious problem: as the matchings are complete, the situation cannot be resolved by getting more perceptual information. Instead, the description has to be considered insuf£cient, and needs to be made more precise. (how to do that is not adressed in this paper).

Finally, we should point to some particular dif£cult situations: when the description is not only insuf£cient but also wrong; when important characteristics of the object have changed in a way we cannot predict (e.g. the shape has been deformed); and when our percepts are not just uncertain but misleading (e.g. a re¤ection is taken to be a color mark). In such cases, we might get mismatches that should have been matches, and vice versa, which in turn leads to an erroneous estimate of the situation and possibly also a misclassi£cation of what case we have.

## 4 Recovery planning for anchoring

In order to recover from cases 1, 2 and (optionally) 4 above, we encode the situations as planning problems for a conditional possibilistic/probabilistic planner called PTLplan [12]. The other cases either do not need to be solved (case 3) or cannot be solved (case 5).

PTLplan searches in a space of epistemic states, or e-states for short, where an e-state represents the agent's incomplete and uncertain knowledge about the world at some point in time. An e-state can be considered to represent a set of hypotheses about the actual state of the world, for instance that a certain gas bottle has a mark on it or has not a mark on it. The planner can reason about perceptive actions, such as looking at an object, and these actions have the effect that the agent makes observations that may help it to distinguish between the different hypotheses. Each different observation will result in a separate new and typically smaller e-state, and in each such e-state the agent will know more than before. For instance, looking at a gas bottle may result in two observations leading to two possible e-states: one where the agent knows there is a mark, and one where it knows there isn't a mark on that side.

A recovery situation in anchoring typically occurs when the robot is executing some higher-level plan and encounters one of the ambiguous but recoverable cases above. Such a situation is handled in £ve steps:

1. The problematic situation is detected and classi£ed as above, and the top-level plan is halted.
2. The planner automatically formulates an initial situation by considering the properties of the requested object and of the perceived objects, and generating different hypotheses for which of the objects corresponds to the requested object. It also formulates a goal that the requested object should be identi£ed if present.
3. The planner searches for a plan taking as parameters the e-state and the goal.
4. The plan is executed, and either the requested object is found and identi£ed and can be anchored, or it is established that it cannot be identi£ed.
5. If recovery was successful, the top-level plan is resumed.

The domain description used for anchoring recovery planning typically is not the same as is used for top-level plans (although in our case the planner is the same). Typically, the actions involved would be restricted to certain perceptual actions, and the description of the locality may be more detailed to facilitate search.

### 4.1 Formulating the initial situations and goals

In **case 1**, where no candidate for the requested object (say b1) has been found, a search needs to be performed. Therefore, the initial situation consists of a number of hypotheses of where the object can be found, including the hypothesis that it is nowhere around. For instance, if there are four places in the room of interest, and we have already searched at one of them, the hypotheses might be that b1 will be visible from one of the remaining places, or from none (f below). Note that the term following the "=" is the value of the property to the left of the "=", and the numbers are degrees of possibility associated with each hypothesis:

1.0 (visible-from b1 = r1_2)
1.0 (visible-from b1 = r1_3)
1.0 (visible-from b1 = r1_4)
0.5 (visible-from b1 = f)

To the above is added information about the topology of the room that is to be searched, and the description of the object to be anchored, e.g. (shape b1 = gasbottle). The goal is formulated as (exists (?x) (nec (visible-from b1 = ?x))), which means that the agent has determined from what place the object is visible.

In **case 2**, where there are one or more partially matching perceived objects, the agent needs to £gure out which of them actually matches the requested object b1. Thus, the hypotheses consists of the different ways b1 can be anchored, based on the known properties of b1 and the perceived properties of the perceived objects. Based on the descriptions $d$ for the requested object and $d_i$ for each perceived object $\mathsf{po}_i$, two extra descriptions are formulated for every $d_i$: £rst, a description $d_i^+$ which completely matches $d$; and second, a non-matching description $d_i^-$ which contains the different ways in which at least one incompletely speci£ed property in $d_i$ may not match with $d$. For instance, if $d$ = (mark b1 = t) and $d_1$ = (mark po1 = t f) (i.e. either true or false), then $d_i^+$ = (mark po1 = t) and $d_i^-$ = (mark po1 = f). Each hypothesis then consists of the conjunction of one $d_i^+$ for one of the $\mathsf{po}_i$ and $d_j^-$ for all remaining $j \neq i$. To each hypothesis is also added the statement (anchor b1 = $\mathsf{po}_i$) denoting that b1 should be anchored to the object anchored by $\mathsf{po}_i$. There is also one hypothesis that no object matches: $d_j^-$ for all $j$, and (anchor b1 = f). Finally, if the planner wishes to take a cautious approach and ascertain that no more than one object is matching, it might also add a number of hypotheses consisting of $d_i^+$, $d_j^+$ for two of the $\mathsf{po}_i$, $\mathsf{po}_j$ and $d_k$ for all remaining $k \neq i, j$, and (anchor b1 = f).

For instance, if b1 is known to be a green gas bottle with a mark on it — (mark b1 = t) — and we perceive two green gas bottles po1

and po2 but are not able to see any marks on them from the current perspective, the (incautious) hypotheses might be:

1.0 (mark po1 = t), (mark po2 = f), (anchor b1 = po1)
1.0 (mark po1 = f), (mark po2 = t), (anchor b1 = po2)
0.5 (mark po1 = f), (mark po2 = f), (anchor b1 = f)

In addition, each of the two hypotheses can be subdivided further into three different hypotheses regarding from where the mark can be detected: (mark-visible-from po1 = r1_1) and so on.

The goal is achieved once a speci£c action (anchor b1 *x*) has been performed. This action has as a precondition that *x* is the only remaining anchor for b1: (nec (anchor b1 = *x*)). Thus, all other candidate anchors have to be eliminated before anchor is applied.

**Case 4** is quite similar to case 2 above, but consists of one hypothesis where the completely matching percept is chosen for anchoring, and a number of hypotheses where there are other objects matching too.

## 4.2 Generating the recovery plan

After the initial situation and the goal have been established, plan generation starts, using the appropriate domain description. The following action, for instance, is for looking for marks (and other visual characteristics) on objects such as gas bottles.

```
(ptl-action
:name (look-at ?y)
:precond ( ((?p) (robot-at = ?p)) ((?y) (perceived-object ?y)) )
:results (cond
           ((and (mark ?y = t) (mark-visible-from ?y = ?p))
            (obs (mark! ?y = t)))
           ((not (and (mark ?y = t)
                      (mark-visible-from ?y = ?p)))
            (obs (mark! ?y = f))))
:execute  ((aiming-at me ?y)
           (anchor-£nd ?y :when (aiming-at me ?y))))
```

In short, the precond part states that the action requires a perceived object ?y and a current position ?p. The result part states that if ?y has a mark, and if the robot looks at ?y from the ?p from which the mark is visible, then the robot will observe the mark (and thus know that there is a mark), and otherwise it will not observe any mark. The obs form is the way to encode that the agent makes a speci£c observation.

The plans generated by PTLplan are conditional: after each action with observation effects (and with more than one alternative outcome), the plan branches. The plan below is generated for looking for marks on a single perceived object from three different positions, starting from a fourth position. Note how a conditional branching follows after each application of look-at: the £rst clause "(mark! po-4 = t/f)" of each branch is the observation one should have made in order to enter that branch, and the subsequent clauses are actions. The action (anchor b1 *x*) at the end of each branch represents the decision to anchor b1 to some speci£c perceived object (or to no object at all, if *x* = f).

```
((move r1_2) (look-at po-4)
 (cond
   ((mark! po-4 = f) (move r1_3) (look-at po-4)
    (cond
      ((mark! po-4 = f) (move r1_4) (look-at po-4)
       (cond
         ((mark! po-4 = t) (anchor b1 po-4) :success)
         ((mark! po-4 = f) (anchor b1 f) :success)))
      ((mark! po-4 = t) (anchor b1 po-4) :success))
   ((mark! po-4 = t) (anchor b1 po-4) :success)))
```

**Figure 2.** Our robot investigating two bottles

We omit the details of how the plan is generated here, as our approach is not dependent on the particular planning algorithm. Actually, another planner with corresponding expressive power could have been used instead.

## 4.3 Plan execution

The anchoring plan is then executed: the actions such as (look-at po-4) are translated into executable perceptive and movement tasks (see £eld :execute in the de£nition of look-at above). The anchor action has a special role: it causes the symbol of the requested object to be anchored to a speci£c perceived object. The robot can then continue performing the task in its top-level plan that was interrupted.

# 5 Tests on a robot

To be able to test the methods described above we have implemented and integrated them with a fuzzy behavior based system, the Thinking Cap [18], used for controlling a mobile robot. We have used this system to run a number of scenarios yielding different kinds of ambiguities. We give here a brief description of the system, the scenarios and the resulting executions.

The platform we have used is a Magellan Pro Research Robot. equipped with standard sonars, bumpers and IR sensors. In addition to the standard setup we have connected a camera and use a simple image recognition system to detect and extract information about objects matching a number of prede£ned patterns.

Apart from the anchoring, plan execution and planning modules described in the previous sections the complete system also consists of a number of other parts which allows the robot to navigate indoor environments safely and perceive the surroundings. Perception is accomplished by continuously receiving percepts from the vision system, associating them with earlier percepts and storing them for later use by the anchoring system.

In these test the robot operates in a room containing one or more gas bottles (Figure 2). These bottles can be of various colors and can optionally have a mark on some side. Typical tasks we have given the robot is to look for gas bottles matching a speci£c description, approaching them, moving around in or exiting the room and re-identifying previously found gas bottles. The actions available to the robot were to look for a speci£c object at a speci£c place, to look at a previously seen object, to move to different positions or near to an object, to select a speci£c object for anchoring, and to perform self-localization by moving to a £xed position.

**Scenario 1: No ambiguity.** The £rst and simplest scenario we have run is when we placed a green gas bottle in the room clearly visible from the robot's location and gave the planner the task to look for and approach b1 with the symbolic description ((color green) (shape gasbottle)). Initially, the plan executor called the Find functionality. Since there was only one completely matching percept (**case 3**) the system anchored b1 to this percept and continued with the plan. The position property of the b1 anchor was used to approach the gas bottle and £nish the original task.

**Scenario 2: Searching the room.** For the next tests we look at **case 1**, where we have no matching candidates to a Find. We set this up by partially obstructing the gas bottle so that it could be seen only from certain positions in the room. Next, we started the robot at a position where the gas bottle was not visible and gave it again the task to look for and approach b1 The £rst call to the Find functionality failed. This triggered the planner to generate a recovery plan from a description of the current world state, using the information that there should somewhere be a gas bottle. The result was a conditional plan that would navigate to different parts of the room, looking for the gas bottle, and announcing success when it was found. After this, the original task of approaching b1 could continue.

**Scenario 3: Partially matching objects.** In this scenario we choose to look at **case 2** were the system perceives one or more objects only partially matching the description. We did this by using a red gas bottle with a mark on it which was not visible from the initial position. We then asked the system to look for b1 matching ((color red) (mark t) (shape gasbottle)) and the Find functionality was called.

At this point in time the system perceived a red gas bottle but could not determine whether it was marked on some side. Thus we had only one partially matching candidate. The system now created a temporary anchor Anchor-1 for this object and the planner generated a recovery plan using the knowledge that Anchor-1 might be the same as b1 and then should have a mark visible from some side. The planner produced a conditional plan which would navigate through the room and observe Anchor-1 to see if a mark was visible from the different viewpoints and to halt when the mark was found. The robot navigated through the room, found the mark and concluded that the observed gas bottle was the right one.

We also successfully ran the same scenario with more advanced setups where we either had no mark on the gas bottle, or where we had two gas bottles of which only one was marked.

**Scenario 4: Planning to reacquire.** In order to test a reacquire ambiguity we had to setup a scenario where the position of an object could not be used to uniquely identify a previously acquired object. To do this we started with two gas bottles in front of the robot, one of the gas bottles had a mark on the side facing the robot. Next, we asked the robot to look for b1 with the inde£nite description ((marked yes) (shape gasbottle)); to exit to a corridor in the opposite side of the room; and £nally to again enter the room and reacquire b1.

In the initial Find we got one partial and one completely matching candidate (**case 4**) and the marked gas bottle was anchored to b1. After this the robot navigated to the opposite side of the room; entered the corridor and went back into the room again. The accumulated uncertainty in the robot's self localization was now so large that when the robot was doing the £nal reacquire, it failed to determine which percept corresponded to b1. Since the mark on the initially anchored gas bottle could not be seen from this position we had an ambiguity due to multiple partial matchings (**case 2**). Thus the planner was triggered to resolve the ambiguity and it generated a plan to investigate both gas bottles to see which one was marked. The result was that the robot reacquired the right gas bottle.

We also tested alternative versions of this setup where instead of failing due to bad self localization we either moved the gas bottles or introduced a new gas bottle before acquiring them again. Moving without observing, or introducing new bottles always gave ambiguities. Due to the implicit tracking done by the vision system, moving them while observed gave only ambiguities if the gasbottles overlapped from the cameras viewpoint during movement. In either case these version gave the same kind of ambiguities and was also solved correctly by observing the gas bottles from different positions until the mark was found.

**Scenario 5: Planning for relocalization.** Since our implemented system mainly uses odometry for localization the degree of uncertainties in the position of objects increases monotonically with movement, unless the objects are observed. This means that even though we have acquired an object and have a position property we may get only a partial matching during a later reacquire on the same object.

To see that this case could be handled, we setup a scenario with two identical gasbottles where we let the robot acquire one of them as b1. Next, we moved the robot (out of the room and back) and asked it to go near b1. Because of odometry errors the position property of b1 could not be used to acquire the right gasbottle and instead we got an ambiguity (**case 2**). The solution generated by the planner was to use a self localization action to remove the odometry error and acqurie the right gas bottle.

## 6 Conclusions

There are two main contributions in this paper. Firstly, we have highlighted the usefulness of knowledge-based planning in robotics in the context of autonomous recovery from perceptual errors. Our results indicate that this direction is very promising for what concerns recovery from anchoring failures, in particular as the complexity and variety of the problems involved motivates the use of on-line planning as opposed to a hard-coded approach.

Secondly, we have presented a classi£cation of different cases that can be the outcome when an embedded agent such as a robot is attempting to anchor symbols to percepts. We have also shown how to use planning techniques to automatically recover from some of these cases, and we have demonstrated our approach on a mobile robot confronted with a number of failure situations.

## 7 Acknowledgements

This work has been supported by The Swedish Research Council (Vetenskapsrådet) and by the Swedish KK foundation.

## REFERENCES

[1] C. Barrouil, C. Castel, P. Fabiani, R. Mampey, P. Secchi, and C. Tessier. Perception strategy for a surveillance system. In *Proc. of ECAI*, pages 627–631, 1998.

[2] M. Beetz and D. McDermott. Expressing transformations of structured reactive plans. In *Proc. of the European Conf. on Planning,*, pages 64–76. Springer, 1997.

[3] Michael Beetz, Tom Arbuckle, Armin B. Cremers, and Markus Mann. Transparent, ¤exible, and resource-adaptive image processing for autonomous service robots. In *Proc. of the 13th European Conference on Arti£cial Intelligence*, pages 158–170. John Wiley and Sons, 1998.

[4] M. Broxvall, L. Karlsson, and A. Saf£otti. Steps toward detecting and recovering from perceptual failures. In *Proc. of the 8th Int. Conf. on Intelligent Autonomous Systems (IAS)*, Amsterdam, NL, 2004.

[5] S. Coradeschi and A. Saf£otti. Anchoring symbols to sensor data: preliminary report. In *Proc. of the 17th AAAI Conf.*, pages 129–135, Menlo Park, CA, 2000. AAAI Press.

[6] S. Coradeschi and A. Saf£otti. Perceptual anchoring of symbols for action. In *Proc. of the 17th IJCAI Conf.*, pages 407–412.

[7] S. Coradeschi and A. Saf£otti. An introduction to the anchoring problem. *Robotics and Autonomous Systems*, 43(2-3):85–96, 2003. Special issue on perceptual anchoring.

[8] R.E. Fikes, P. Hart, and N.J. Nilsson. Learning and executing generalized robot plans. *Arti£cial Intelligence*, 3(4):251–288, 1972.

[9] J. Gancet and S. Lacroix. PG2P: A perception-guided path planning approach for long range autonomous navigation in unkown natural environments. In *Proc. of IROS*, Las Vegas, NV, 2003. To appear.

[10] G. De Giacomo, L. Iocchi, D. Nardi, and R. Rosati. Planning with sensing for a mobile robot. In *Proc. of the 4th European Conf. on Planning*, pages 158–170. Springer, 1997.

[11] K.Z. Haigh and M.M. Veloso. Interleaving planning and robot execution for asynchronous user requests. *Autonomous Robots*, 5(1):79–95, 1998.

[12] L. Karlsson. Conditional progressive planning under uncertainty. In *Proc. of the 17th IJCAI Conf.*, pages 431–438. AAAI Press, 2001.

[13] L. Karlsson and T. Schiavinotto. Progressive planning for mobile robots: a progress report. In M. Beetz, J. Hertzberg, M. Ghallab, and M. Pollack, editors, *Advances in Plan-Based Control of Robotic Agents*, pages 106–122. Springer, Berlin, DE, 2002.

[14] S. Kovacic, A. Leonardis, and F. Pernus. Planning sequences of views for 3-D object recognition and pose determination. *Pattern Recognition*, 31:1407–1417, 1998.

[15] A. Lazanas and J.C. Latombe. Motion planning with uncertainty: A landmark approach. *Arti£cial Intelligence*, 76(1-2):285–317, 1995.

[16] Robin R. Murphy and David Hershberger. Classifying and recovering from sensing failures in autonomous mobile robots. In *Proc. AAAI-96*, pages 922–929, 1996.

[17] B. Pell, D.E. Bernard, S.A. Chien, E. Gat, N. Muscettola, P.P. Nayak, M.D. Wagner, and B.C. Williams. An autonomous spacecraft agent prototype. *Autonomous Robots*, 5(1):1–27, 1998.

[18] A. Saf£otti, K. Konolige, and E.H. Ruspini. A multivalued-logic approach to integrating planning and control. *Arti£cial Intelligence*, 76(1–2):481–526, 1995.

[19] L. Seabra-Lopes. Failure recovery planning in assembly based on acquired experience: learning by analogy. In *Proc. IEEE Intl. Symp. on Assembly and Task Planning*, Porto, PT, 1999.

# Flexible Interval Planning in Concurrent Temporal Golog

**Alberto Finzi** and **Fiora Pirri**[1]

**Abstract.** In this paper we present an approach to flexible planning and scheduling based on a suitable mapping of the Constraint Based Interval Planning paradigm [7, 2] into the Situation Calculus. We show how this representation is particular suitable for executive control processes, and illustrate this with an example.

## 1 Introduction

A central feature in executive control is flexible tasks alternation, yielding switching-time criteria, based on tasks, goal and the current situation needs. In *in vitro* domains[2], robots are requested to performing multiple tasks either simultaneously or in rapid alternation, using diverse sensing and actuating tools, like navigation, visual exploration, mapping, perceptual analysis, etc. To guarantee that such multiple-task performance can be achieved, some approaches have proposed that executive control processes supervise the selection, initiation, execution, and termination of actions. In this sense, a well known approach to executive control is Constraint Based Interval Planning (CBIP), amalgamating planning, scheduling and resources optimization for reasoning about the competing activities involved in a flexible concurrent plan (see [7, 2, 4]). The CBIP approach, like similar ones, emerged from the planning community, and have shown a strong practical impact in executive control processes [10, 18].

From the vantage point of cognitive robotics a question to be addressed is how executive control processes interact with basic perceptual-motor and cognitive processes used for performing individual tasks, how priorities among individual processes can be established, and how resources can be allocated to them during multiple-task performance (see [5, 8, 3]).

In particular, when dealing with the executive control it seems that approaches (such as the CBIP) tailored to the practical needs of reactive planning are more suitable. In this paper we show that CBIP perspective, on executive control processes, with all its arsenal of specifications in terms of flexible time, alternation constraints, resources optimization, failure recovering, and tasks scheduling, can be easily imported into the framework of the Situation Calculus([16, 9]), exploiting already established temporal and concurrent extensions of basic theory of actions, as those provided by [12, 14, 13, 17, 5]. The resulting language is one naturally belonging to the Concurrent Temporal Golog (CTG) families of languages, and it offers the possibility of manipulating flexible plans on a multiple time line. The paper is just introductory, and several problems still need to be addressed and solved; among these we mention the need of indexing situations so as to ensure multiple independent timelines, a suitable formalization of forgetting executed processes, progressing to the current set of processes, side effects of processes, and failure management. The mapping proposed is somehow obvious due to the expressive power of the Situation Calculus, but here can be meaningful, as it would offer to the temporal planning community a way to compare the Golog specification environment with the CBIP modeling constructs, and it provides a new account for deploying Golog at the executive control. In fact, we show how it can be used to implement a parallel control system for a particularly difficult task such as the *robocup rescue*.

[1] University of Rome "La Sapienza"

[2] Domains requiring effective robots performance, even if suitably constrained so as to keep possible domino effects under control

## 2 Preliminaries

### 2.1 Situation Calculus and Golog

The Situation Calculus ($SC$) [9] is a sorted first order language representing dynamic domains by means of *actions*, *situations*, and *fluents*. *Actions* and *situation* are first order terms. A situation denotes a history of actions compound with the binary symbol $do$: $do(a, s)$ is the situation obtained by executing the action $a$ after the sequence $s$. The constant symbol $S_0$ stands for the initial situation (i.e. the empty sequence). In this work, we assume the Temporal Concurrent Situation Calculus presented in [12, 15, 14]. To represent time in the Situation Calculus, one of the arguments to the action function symbol is the time of the action's occurrence. For example, $startGoing(a, 12.01)$ is the action of starting to move toward $a$ at time 12.01. All actions are viewed as instantaneous. The function symbol $time(a)$ denotes the occurrence time of action $a$, while $start(s)$ denotes the start time of situation $s$. The latter is defined as: $start(do(a, s)) = time(a)$, and $time(a)$ is defined, for each action term $a$, to be its temporal argument, for example, $time(startGo(a, t)) = t$. In this paper, we will use the notation $s[\vec{t}]$ to represent the situation $s$ with $\vec{t}$ free temporal variables. Following [16], we represent concurrent actions as sets of primitive actions: the actions are sorted in simple action $a$ and concurrent $c$. $a \in c$ means that the simple action $a$ is one of the concurrent actions in $c$. We rely on the standard interpretation of sets and their operations and relations. Since in the concurrent $SC$, situations are lists of concurrent actions, we have situation terms like $do(\{a_1, a_2\}, s)$. A *fluent* is a predicate whose last argument is a situation, e.g. $at(hill, do(endGo(hill, 10), break, do(stratGo(hill, 2.1), S_0)))$. Fluents predicates denote properties that can change with the action execution.

In the $SC$ concurrent durative actions are considered as *processes* [12, 16], represented by fluents, and durationless actions are to start and terminate the processes. For example, $going(hill, s)$ is started by the action $startGo(hill, t)$ and it is ended by $endGo(hill, t')$.

**Domain Theory.** In the $SC$ a dynamic domain can be described by a *Basic Action Theory* ($BAT$) which is composed of the classes of axioms: $\Sigma \cup \mathcal{D}_{S_0} \cup \mathcal{D}_{ssa} \cup \mathcal{D}_{una} \cup \mathcal{D}_{ap}$. Here $\Sigma$ is the set of foundational axioms for situations. We refer to the version introduced in [14] in order to represent timed concurrent actions. For example, these axioms impose that, given a situation $do(c, s)$, all the actions $a \in c$ occur at the same time (i.e. $coherent(c)$).

$\mathcal{D}_{una}$ are uniqueness axioms for each action. $\mathcal{D}_{S_0}$ is a sets of sentences describing the initial state (i.e. $S_0$) of the domain.

$\mathcal{D}_{ssa}$ specify the *successor state axioms*, for each fluent $F(\vec{x}, s)$ . Here we assume the successor state axioms modified in order to represent concurrent actions. This slight modification can be illustrated by the following example:

$$pointingTo(x, do(c, s)) \equiv startPointingTo(x) \in c \vee \\ pointing(x, s) \wedge endPointingTo(x) \notin c.$$

Finally, $\mathcal{D}_{ap}$ represents the action precondition axioms. In order to extend the $Poss$ predicate to concurrent actions, the following axioms are introduced:

$$Poss(a, s) \supset Poss(\{a\}, s).$$
$$Poss(c, s) \supset (\exists a) a \in c \wedge (\forall a)[a \in c \supset Poss(a, s)].$$

In a concurrent domain, *action preconditions axioms* on simple actions are not sufficient: two simple actions may each be possible, but their concurrent execution should not be permitted. This problem is called *precondition interaction problem* [16] (see [11] for a discussion) and its solution requires some additional precondition axioms.

**Temporal Concurrent Golog.** Golog is a situation calculus-based programming language for denoting complex actions composed of the primitive (simple or concurrent) actions defined in the $BAT$. Golog programs are defined by means of standard (and not so-standard) Algol-like control constructs: i. action sequence: $p_1; p_2$, ii. test: $\phi?$, iii. nondeterministic action choice $p_1|p_2$, iv. conditionals, while loops, and procedure calls. An example of a Golog program is:

$$\textbf{while } \neg at(hill, 3) \textbf{ do} \\ \textbf{if } \neg(\exists x) going(x) \textbf{ do } \pi(t, (t < 3)? : startGo(hill, t))$$

The semantics of a Golog program $\delta$ is a $SC$ formula $Do(\delta, s, s')$ meaning that $s'$ is a possible situation reached by $\delta$ once executed from $s$. Some of the construct definitions are the following

$$Do(nil, s, s).$$
$$Do((p_1 : p_2) : p_3, s, s') \doteq Do(p_1 : (p2 : p_3), s, s')$$
$$Do(a : p, s, s') \doteq Do(p, do(a, s), s')$$
$$Do(p_1|p_2, s, s') \doteq Do(p_1, s, s') \vee Do(p_2, s, s')$$
$$Do(\pi(x, p(x)), s, s') \doteq (\exists x) Do(p(x), s, s')$$

In this paper we consider a Golog language version endowed with the parallel execution between processes. Analogously to [3] concurrency is modeled by interleaved processes and the parallel construct $\|$ is defined as follows:

1. $Do(p_1 \| p_2, s, s') \doteq Do(p_2 \| p_1, s, s')$.
2. $Do((p_1 : p_2) : p_3 \| p, s, s') \doteq Do(p_1 : (p2 : p_3) \| p, s, s')$.
3. $Do(p_1 \| nil, s, s') \doteq Do(p_1, s, s')$.
4. $Do(a_1 : p_1 \| a_2 : p_2, s, s') \doteq Do(a_1 : p_1 \| p_2, do(\{a_2\}, s), s') \vee$ $Do(p_1 \| a_2 : p_2, do(\{a_1\}, s), s') \vee Do(p_1 \| p_2, do(\{a_1, a_2\}, s), s')$
5. $Do((\phi)? : p_1 \| p, s, s') \doteq \phi[s] \wedge Do(p_1 \| p, s, s')$.
6. $Do((p_1|p_2) \| p_3, s, s') \doteq Do(p_1 \| p_2, s, s') \vee Do(p_1 \| p_3, s, s')$.
7. $Do(\pi(x, p(x)) \| p, s, s') \doteq (\exists x) Do(p(x) \| p, s, s')$.

## 2.2 Constraint Based Interval Planning

In this section we briefly introduce the ontology and the basic concepts of Constraints Based Interval Planning paradigm, whose primitives are illustrated in Figure 1 (note that $[d, D]$ denotes the interval where $d$ is the minimum time distance and $D$ the maximum).

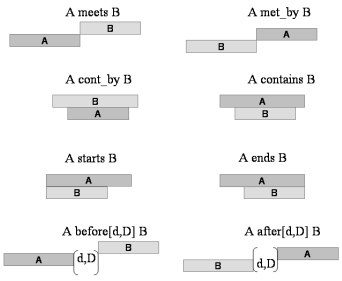



**Figure 1.** Interval relations in the underlying temporal model.

**Attributes and Intervals** The CBIP paradigm assumes a dynamic system modeled as a set of *attributes* whose state changes over time. Each attribute, called *state variable*, represents a concurrent thread, describing its history over time as a sequence of states and activities. Both states and activities are represented by temporal intervals called *tokens*. The history of states for a state variable over a period of time is called a *timeline*. For example, given a rover domain, $position$ is a possible attribute; $going(a, b)$ from time 1 to 3, and $at(b)$ from time 3 to 5 are intervals representing, respectively, an activity and a state. Each token can be described by the tuple $\langle v, p(\vec{x}), t_s, t_e \rangle$, where $v$ is the attribute (e.g. $position$), $p$ is the name of an activity, $\vec{x}$ are its parameters (e.g. $going(a, b)$), and $t_s$, $t_e$ are numerical variables indicating start and end times respectively.

To represent intervals on a timeline we will use the notation $[t_1, t_2]\ p\ [t_3, t_4]$, meaning that $p$ is s.t. $t_s \in [t_1, t_2]$ and $t_e \in [t_3, t_4]$ (e.g. given the timeline $pos$, $[0, 0]\ at(hill)\ [3, 4]$ ).

**Domain Constraints.** Given a set of attributes $A$ and a set of intervals $I$, a *CBI model* $M = (A, I, R)$ is specified by a set of constraints $R$, that is, each token $T = \langle v, p(\vec{x}_p), t_s, t_e \rangle$ has its own *configuration constraint* $G_T(v, \vec{x}_p, t_s, t_e)$, called *compatibility* (see [7]), representing all the possible legal relations with other intervals; for example compatibility establishes which token must proceed, follow, be co-temporal, etc. to others in a legal plan. These relations are, in turn, defined by equality constraints between parameter variables of different tokens, and by simple temporal constraints on the start and end variables. The latter are specified in terms of metric version of temporal relations *a la Allen* [1]. Here we restrict our attention to the following set of temporal relations: $meets$, $met\_by$, $contained\_by$, $contains$, $before[d, D]$, $after[d, D]$, $starts$, $ends$. For instance, $going(x, y)\ meets\ at(y)$, and $going(x, y)\ met\_by\ at(x)$ specifies that each $going$ interval is followed and preceded by a state $at$.

**Planning Problem.** Given the *CBI model* $M$ specifying the planning domain, a *planning problem* is defined by $P = (M, P_c)$, where $P_c$ is a *candidate plan*, representing an incomplete instance of a plan. The candidate plan consists of: i. a *planning horizon* specified by a pair of temporal values $(h_s, h_e)$, with $h_s < h_e$; ii. a timeline $\mathcal{T}_\sigma = (T_{\sigma_1}, \ldots, T_{\sigma_n})$ for each state variable $\sigma$, containing a set of tokens $T_i = \langle \sigma, P(\vec{x}), t_{s_i}, t_{e_i} \rangle$; iii. a set of ordering constraints among the token in the timeline: $h_s \leq t_{e_1} \leq t_{s_2} \leq \ldots$; iv. the set of constraints $\{C_1, \ldots, C_n\}$ associated with the tokens laying on the timelines. For instance, given the rover domain with the attributes $Location$ and $Navigation$, a candidate plan can be represented by a planning horizon $(0, 10)$, and by the two timelines for the $Location$ ($Lc$) and $Navigation$ ($Nv$) attributes, and the two associated incomplete sequence of tokens (together with their respective

constraints), e.g.

$$\begin{array}{ll} Lc: & [0,0]\ at(a)\ [3,3]\ \ [5,10]\ going(d,e)\ [6,10], \\ Nv: & [0,0]\ stop\ [3,5]\ \ [6,9]\ move\ [8,10], \end{array}$$

Notice that the *candidate plan* defines both the initial situation and the goals. A token $T_i$ in a *candidate plan* is said to be *fully supported* if its $G_{T_i}$ compatibility is satisfied, in a sense to be specified. For instance, $[5,10]\ going(d,e)$, in the example above is not fully supported, since $going(d,e)\ met\_by\ at(d)$ is to be satisfied. A candidate plan is called a *potential behavior* if: a. each token on each timeline is fully supported; b. all timelines fully cover the planning horizon; c. all timeline tokens are bound to a single value. In other worlds, a *possible behavior* represents a possible evolution of the dynamic system. A candidate plan can be seen as an incomplete specification of a possible behavior where gaps, unsupported tokens, and uninstantiated variables can be found (see example above). A candidate plan is said to be a *complete plan* if it satisfies both the properties a. and b. specified above. More generally, a candidate plan is a plan depending on a *plan identification function* (see [7] for further details).

Given the *planning problem* specified by the *CBI model* and the *candidate plan*, the planning task is to provide a *complete plan* with the maximum flexibility: the planner should minimally ground the (temporal and not) variables to allow for on-line binding of the values. For example, given the rover example, assuming the candidate plan presented above (assuming also that each activity takes at least one time unit), a *sufficient plan* could be

$$\begin{array}{l} Lc: [0,0]at(a)[3,3]going(a,d)[4,8]at(d)[6,9]\ going(d,e)\ [6,10], \\ Nv: [0,0]\ stop\ [3,3]move[4,8]stop[6,9]move[8,10]. \end{array}$$

Notice that the above specification completely fill the timelines till the end of the horizon and each token is fully supported.

## 3 Representing the Temporal Model in the Temporal Concurrent SC

In this section we show how to represent a CBI Model in the Temporal Concurrent $SC$ framework.

**Attributes and Intervals.** For each token $\langle v, p(\vec{x}), t_s, t_e\rangle$ we introduce a fluent $P_v(\vec{x}, t_s, s)$ and two actions $start\_p(\vec{x}, t)$ and $end\_p(\vec{x}, t)$ representing, respectively, the $p(\vec{x})$ process (here $t_s$ is the start time) starting and ending events.

The temporal model will be defined by a BAT. In particular the successor state axiom ($SSA$) is defined as follows:

$$\begin{array}{c} P_v(\vec{x}, t_s, do(c,s)) \equiv \exists a.P\_start(\vec{x}, a, s) \wedge a \in c \wedge time(a) = t_s \vee \\ (\exists t).P_v(\vec{x}, t, s) \wedge \neg(\exists a').P\_end(\vec{x}, a', s) \wedge a' \in c. \end{array}$$

Here $P\_start(\vec{x}, a, s)$ ($P\_end(\vec{x}, a, s)$) is true if $a$ starts (ends) $P_v(\vec{x}, t_s, s)$ in $s$. Continuing the previous example, $\langle Lc, going(x), t_s, t_e\rangle$ can be represented by the $going(x, t, s)$ fluent whose $SSA$ is represented as follows:

$$\begin{array}{c} going(x, t, do(c, s)) \equiv start\_going(x, t) \in c \vee \\ going(x, t, s) \wedge (\forall t')end\_going(x, t') \notin c. \end{array}$$

The $start\_p$ and $end\_p$ actions are specified by action preconditions axioms, where $\phi_p(\vec{x}, t, s)$ are sentences specifying the conditions under which the $start$ and $end$ actions can be executed:

$$\begin{array}{l} Poss(start\_p(\vec{x}, t), s) \equiv \phi_p(\vec{x}, t, s) \\ Poss(end\_p(\vec{x}, t), s) \equiv \phi_p(\vec{x}, t, s) \end{array}$$

**Domain Constraints.** Given the CBI model $M = (A, I, R)$, the $R$ constraints can be captured in the temporal BAT by exploiting the axiom preconditions needed in the concurrent action specification (to address the precondition interaction problem). For instance, the constraint on the token duration can be expressed by:

$$\begin{array}{c} Poss(c, s) \supset (\exists t)[A\_ends(\vec{x}, a, s) \wedge A(\vec{x}, t, s) \wedge a \in c \supset \\ d \leq time(a) - t \leq D] \end{array}$$

where $A\_end(\vec{x}, a, s)$ ($A\_start(\vec{x}, a, s)$) is true if $a$ ends (starts) $A(\vec{x})$ in $s$. Analogously the Allen-like temporal constraints introduced above (see Figure 1) can be easily represented in the concurrent temporal BAT (see Figure 1):

- $A(\vec{x})\ meets\ B(\vec{x})$:

$$\begin{array}{c} Poss(c, s) \supset \exists a.A\_end(\vec{x}, a, s) \wedge a \in c \supset \\ \exists a'.B\_start(\vec{x}, a', s) \wedge a' \in c. \end{array}$$

- $A(\vec{x})\ met\_by\ B(\vec{x})$:

$$\begin{array}{c} Poss(c, s) \supset \exists a.A\_start(\vec{x}, a, s) \wedge a \in c \supset \\ \exists a'.B\_end(\vec{x}, a', s) \wedge a' \in c. \end{array}$$

- $A(\vec{x})\ starts\ B(\vec{x})$:

$$\begin{array}{c} Poss(c, s) \supset \exists a.A\_start(\vec{x}, a, s) \wedge a \in c \supset \\ \exists a'.B\_start(\vec{x}, a', s) \wedge a' \in c. \end{array}$$

- $A(\vec{x})\ ends\ B(\vec{x})$:

$$\begin{array}{c} Poss(c, s) \supset \exists a.A\_end(\vec{x}, a, s) \wedge a \in c \supset \\ \exists a'.B\_end(\vec{x}, a', s) \wedge a' \in c. \end{array}$$

- $A(\vec{x})\ contained\_by\ B(\vec{x})$:

$$\begin{array}{c} Poss(c, s) \supset [\exists a.A\_start(\vec{x}, a, s) \wedge a \in c \supset B(\vec{x}, s) \wedge \\ \neg\exists a'.B\_end(\vec{x}, a', s) \wedge a' \in c] \wedge \\ [\exists a.B\_end(\vec{x}, a, s) \wedge a \in c \supset \neg A(\vec{x}, s) \vee \\ \exists a'.A\_end(\vec{x}, a', s) \wedge a' \in c]. \end{array}$$

- $A(\vec{x})\ contains\ B(\vec{x})$ is recursively defined once we introduce two auxiliary fluent/processes $AmeetsB(\vec{x})$ and $AendsB(\vec{x})$, s.t.:

$$\begin{array}{l} A(\vec{x})\ starts\ AmeetsB(\vec{x}),\ \ AmeetsB(\vec{x})\ cont\_by\ A(\vec{x}), \\ A(\vec{x})\ starts\ AendsB(\vec{x}),\ \ AendsB(\vec{x})\ cont\_by\ A(\vec{x}), \\ AmeetsB(\vec{x})\ meets\ B(\vec{x}),\ \ AmeetsB(\vec{x})\ cont\_by\ AendsB(\vec{x}), \\ AendsB(\vec{x})\ ends\ B(\vec{x}). \end{array}$$

- $A(\vec{x})\ before[d, D]\ B(\vec{x})$ is recursively defined by:

$$A(\vec{x})\ meets\ A\_bf\_B(\vec{x}, d, D),\ \ A\_bf\_B(\vec{x}, d, D)\ meets\ B(\vec{x}),$$

where $A\_bf\_B(\vec{x}, d, D, s)$ is an auxiliary fluent/process whose duration ranges over the interval $[d, D]$.

- $A(\vec{x})\ after[d, D]\ B(\vec{x})$ :

$$\begin{array}{c} Poss(c, s) \supset \exists a, t.A\_start(\vec{x}, a, s) \wedge a \in c \wedge time(c) = t \supset \\ after\_B(\vec{x}, t, d, D, s). \end{array}$$

where $after\_B(\vec{x}, t, d, D, s)$ is an auxiliary fluent which is true if there exists an action $a$ ending $B(\vec{x})$, with $d \leq time(s) - time(a) \leq D$.

Once all the temporal constraints are specified in this way, the $Poss(c, s)$ definition can be obtained by the closure of its necessary conditions.

**Planning Problem.** Once the domain temporal constraints have been represented in the Concurrent Temporal Situation Calculus, the planning problem is defined by a *candidate plan* representing both the initial situation and the system goals. We recall that the candidate plan is defined by: i. a planning horizon $(t_h, T_h)$; ii. timelines $\mathcal{T}_i$ for each state variable $\sigma_i$; iii. ordering constraints between tokens in the same timelines; iv. a set of constraints $C_i$ each of the kind: $[t_{s_i}, T_{s_i}]\ p_i(\vec{r})\ [t_{e_i}, T_{e_i}]$.

To represent the *candidate plan* as *golog program* in the concurrent temporal golog, we do not deploy occurencies and narratives [13] since it is not a domain constraint, as it defines the control knowledge (goals). We assume a complete specification of $\mathcal{D}_{S_0}$; the Golog scripting language is used to represent the $C_i$ constraints (in the $SC$ we have to distinguish among initial situation and goals). This is possible introducing, for each $C_i$, a procedure definition of the following kind:

$$\begin{aligned}\mathbf{proc}(c_i,\ (T_{e_i} > horizon)?\ |((T_{e_i} \leq horizon)\wedge\\ (\exists t,t').end\_P_i(\vec{r},t,t')\wedge\\ t_{s_i} \leq t' \leq T_{s_i} \wedge t_{e_i} \leq t \leq T_{e_i})?).\end{aligned}$$

This procedure is composed of only two tests: if the end time constraint is beyond the horizon the constraint is neglected, otherwise, the start and end timepoints have to satisfy the temporal constraints. Here $end\_P_i(\vec{r},t,t',s)$ is a fluent properties which is true iff $P_i(\vec{x},t,s)$ ends in $s$ at $t'$. For example, $[5,10]going(d,e)[6,10]$ can be represented as

$$\begin{aligned}\mathbf{proc}(c_2,\ (10 > horizon)?\ |(10 \leq horizon)\wedge\\ (\exists t,t').end\_going(d,e,t,t')\wedge\\ 5 \leq t' \leq 10 \wedge 6 \leq t \leq 10)?).\end{aligned}$$

Given these procedures, an incomplete plan over a timeline $\mathcal{T}_j$ can be define by the following procedure:

$$\begin{aligned}&\mathbf{proc}(plan\_\mathcal{T}_j,\\ &\quad \pi(n,\ (select(n))?\ :\ plan_j(n)\ :\ c_1)\ :\\ &\quad \pi(n,\ (select(n))?\ :\ plan_j(n)\ :\ c_2)\ :\ldots\ :\\ &\quad \pi(n,\ (select(n))?\ :\ plan_j(n)\ :\ c_k)),\end{aligned}$$

where $plan(n)$ is a planner whose depth is bounded by $n$ representing the maximal number of gaps (tokens) between $c_i$ and $c_{i+1}$. $plan(n)$ implements a simple planning algorithm, e.g. we can deploy the following straightforward algorithm:

$$\begin{aligned}&proc(plan_j(n),\\ &\quad true?\mid \pi(a,(primitive\_action(a,j))?\ :\ a)\ :\ plan(n-1))\end{aligned}$$

where $primitive\_action(a,j)$ is to select a primitive action belonging to the $\mathcal{T}_j$ timeline. For example, the $Navigation$ timeline introduced in Section 2.2, can be represented by a $plan\_\mathcal{T}_{Nav}$ with $c_2$ defined as before and $c_1$ as follows:

$$\mathbf{proc}(c_1,\ (10 > horizon)?\ |(10 \leq horizon \wedge end\_at(a,0,3))?).$$

Once an incomplete plan is over a timeline $\mathcal{T}_i$, given a set of timelines $\{\mathcal{T}_i\}$, a candidate plan becomes a parallel execution of its own procedure $plan\_\mathcal{T}_i$:

$$\mathbf{proc}(c\_plan,\ plan\_\mathcal{T}_1\ :nil \parallel\ \ldots\ \parallel plan\_\mathcal{T}_k\ :\ nil\ ).$$

Given a $BAT$ encoding the action theory, and the temporal constraints among the activities, any ground situation $\sigma$ s.t. $BAT \models Do(c\_plan, S_0, \sigma)$ represents a CBI *possible behavior*. More precisely, let $f(\sigma)$ be a behavior at a ground situation $\sigma$, $M = (I, A, R)$ a CBI model, defined in $BAT$, then for any *candidate plan* $P_c$ there is a $\mathcal{D}_{S_0}$ and a $c\_plan$ CTGolog procedure such that $p$ is a *possible behavior* of $(M, P_c)$, if there exists a $\sigma$ (ground) with $f(\sigma) = p$ and $BAT \models Do(c\_plan, S_0, \sigma)$.

A CBI *sufficient plan* is a complete CBI plan with maximal flexibility. In the $SC$ framework we can represent a sufficient plan as the couple $\langle s[\vec{x},\vec{t}], Constr(\vec{x},\vec{t})\rangle$ where $Constr(\vec{x},\vec{t})$ is a minimal set of constraints (among time variables $\vec{t}$ and argument variables $\vec{x}$) s.t.

$$BAT \cup \{Constr(\vec{x},\vec{t})\} \models Do(c\_plan, s_0, s[\vec{x},\vec{t}]).$$

Given this representation, it is possible to show that if $p_{sc} = \langle s[\vec{x},\vec{t}], Constr(\vec{x},\vec{t})\rangle$ is s.t. the previous property holds, then the associated $p_{CBI}$ CBI plan is a *sufficient plan*. Notice that the mapping does not work in the other direction, i.e. there exists a CBI flexible plan $p_{CBI}$ which cannot be captured by a $p_{sc}$. This is due to the fact that a complete CBI plan is identified with a situation $s[\vec{x}]$, where the order of two (starting or ending) events $a_1$ and $a_2$ belonging to two different timelines is already decided: at planning time the compiler decides if $a_1$ starts before, after, or concurrently with $a_2$. Instead, in a CBI sufficient plan this order between events can be defined at execution time. A complete $SC$ mapping of the sufficient CBI plan needs a more complex representation where each plan is associated to a set of flexible situations, we leave this issue to future work.

## 4 Example

We consider a rescue domain where a rover is to explore an unknown environment in order to map and localize victims. We assume the rover endowed with a pan-tilt and stero-cameras. Visual perception is exploited to detect interesting locations where it's worth to go and perform the observations. Basically the robot has to provide two main activities: exploring and mapping; search for victims. While the robot is in the exploration mode a rough visual perception ($vp\_monitor$) is always active in order to tag the map with salient regions. A more complex visual analysis is performed ($vp\_analsys$) in order to detect victims, during this activity the rover must be stopped while the pan-tilt and range-finder are coordinated in order to scan a salient portion of the visual space.

We consider the following state variables: *Pant-tilt*, *RangeFinder*, *LocMap*, *Navigation*, *VisualPerception*, *Mode*. Each state variable is associated with a set of processes/tokens. *Pan-tilt* can either be idling in pos $\vec{\theta}$ ($Pt\_idle(\vec{\theta})$), moving toward $\vec{\theta}$ ($Pt\_moving(\vec{\theta})$), or scanning ($Pt\_scanning(\vec{\theta})$); *Range finder*[3] states are $Rf\_idle$ and $Rf\_active$; *LocMap* maps and tracks the robot position via the $Lc\_at(\vec{x})$ and $Lc\_goTo(x)$ tokens; *Navigation* represents the navigation state through: $Nv\_stop$, $Nv\_movingTo(speed)$, $Nv\_wandering$; *Visual Perception* represents the state of the visual perception module: it can be either idle ($Vp\_idle$), activated to detect interesting objects in the environment $Vp\_monitor$, or analyzing an interesting region from $\vec{x}$ robot position with pant-tilt in $\theta$: $Vp\_analisys(\vec{x},\vec{\theta})$. *Mode* can be $Md\_map(st)$ or $Md\_search(st)$, where $st$ is $ok$ if the activity succeeds and $no$ if it fails.

Hard time constraints among the activities can be defined by a temporal model in CBI style. For example, $Vp\_monitor$ and $Vp\_observe(\vec{x},\vec{\theta})$ are respectively associated with the mapping and the search modes, hence we have $Vp\_monitor\ \ cont\_by\ \ Md\_map(st)$ and $Vp\_observe(\vec{x},\vec{\theta})\ \ cont\_by\ \ Md\_search$. The search mode can be started only if the local environment is mapped with success $Md\_search(st)\ \ met\_by\ \ Md\_map(ok)$. The victim detection requires the rover to be stopped $Vp\_analisys(\vec{x},\vec{\theta})\ \ cont\_by\ \ Nv\_stop$. The visual analysis needs

[3] It is actually a telemeter returning the precise distance of the point hit.

the pan-tilt scanning $Vp\_analisys(\vec{x},\vec{\theta})\ cont\ Pt\_scanning(\theta)$ and the $Pt\_scanning$ can start only if $met\_by\ Pt\_idle(\theta)$, etc.

Since these constraints are represented by the $\mathcal{D}_{AP}$ of the $BAT$, the embedded CBI temporal model is directly combined with the other dynamic properties specified in the $SC$ language. Now, given the $(0, 1000)$ plan horizon, the following partial plan should force the exploration of an unknown environment:

$$Md : [0,0]Md\_map(ok)[10,\ 100]Md\_search(ok)[11,\ 1000]$$
$$Nv : [0,0]Nv\_idle[0,1000]$$
$$\ldots$$
$$Pt : [0,0]Pt\_idle(\vec{\theta})[0,1000]$$

where, except for *Mode*, each timeline has only the initial activity defined. Following the approach presented in Section 3, this partial plan can be translated into the Golog procedure:

$$\mathbf{proc}(c\_plan,\ plan\_Md\ :nil \parallel plan\_Nv\ :\ nil \parallel \ldots \parallel plan\_Pt\ :\ nil),$$

where $plan\_Md$ is defined by the two time constraints $[0,0]Md\_map(ok)[10,\ 100]Md\_search(ok)[11,\ 1000]$, and the other procedures are encoded as generic planners. However, in order to make this planning activity feasible, the Golog scripting language can be exploited to directly encode some control knowledge. For example, the $Plan\_Pt$ procedure can be written as follows:

$$\mathbf{proc}(plan\_Pt,\ \pi(t,\pi(t',(\exists x.Md\_map(x,t))?|\ (\exists x.Md\_search(x,t))?\ :\ wait\_location\ :\ Ptscan\ :\ PtIdle : (time = t' \wedge t - t' \leq d)?)),$$

where $wait\_location\ :\ Ptscan\ :\ PtIdle$ are three Golog procedure defining the expected pant-tilt behavior during the search mode. The final test enforces a maximal $d$ time duration for the whole procedure execution.

## 5 Implementation

We provided a constraint logic programming (CLP) [6] implementation of the CTGolog based control system for the rescue domain. Since in this setting the CTGolog interpreter is to generate flexible temporal plans, it must be endowed with a constraint problem solver. Analogous to [14] we rely on a logic programming language with a built-in solver for linear constraints over the reals (CLP($\mathcal{R}$)). In this setting logical formulas, allowed for the definition of predicates, are restricted to be horn clauses of the form: $A \leftarrow c_1, \ldots, c_m | A_1, \ldots, A_n$, where $c_i$ are constraints and $A_j$ are atoms. Specifically, we appeal to the ECRC Common Logic Programming System ECLIPSE 5.7. In this way our planner and domain axioms make use of linear temporal relations like $2 * T_1 + T_2 = 5$ and $3 * T_2 - 5 \leq 2 * T_3$, and we rely on ECLIPSE to performing the reasoning in the temporal domain. The relations managed by the ECLIPSE built-in constraint solver have # as a prefix, for example, a temporal constraint represented in the Golog interpreter is:

```
do(C : A,S,S1) :- concurrent_action(C),
 poss(C,S), start(S,T1), time(C,T2), T1 #=< T2,
 do(A,do(C,S),S1).
```

Other temporal constraints are expressed in the action preconditions, for example, considering the pan-tilt processes:

```
poss(pt_pos_start(X,T),S) :-
  pt_idle(X,T1,S),T1 #< T,start(S,T2),T2 #>= T,
  nv_stop(T11,S),T11 #<T.
```

An example of the successor state axioms is the following.

```
pt_idle(X,T1,do(C,S)) :-
   pt_idle(X,T1,S) not member(pt_pos_start(_,T2),S);
   member(pt_pos_end(X,T1),S).
```

Given the BAT specification, for each timeline it is possible to specify a control procedure

```
proc(pt_go(X), pi(t1, [pt_pos_start(X,t1)]:
                pi(t2, [pt_pos_end(X,t2)] )) ).
```

Once the flexible temporal plan is compiled, it can be executed. We assume an execution monitor *cycleExec* which sends and receives commands at each time tick so that constraints can be, step by step solved and/or propagated. A dummy implementation of *cycleExec* is shown below.

```
planExec :-
      do(c-plan,s0,S),!,cycleExec(1,s0,S,S1).
cycleExec(T,S0,S0,S0) :- !.
cycleExec(T,S0,S,S1) :-
    checkMsg(T), exec(T,S0,S,S1), checkMsg(T),
        T1 is T+1,!, cycleExec(T1,S1,S,S2).
```

## 6 Summary and Outlook

We presented an approach to the embedding of the CBIP paradigm in the Golog framework. Several issues are left to future work, among them: progression and forgetting the past, parallel planning over independent timelines, failure management.

## REFERENCES


[1] J.F. Allen, 'An interval-based representation of temporal knowledge', in *IJCAI*, (1981).

[2] A.K. Jonsson D.E. Smith, J. Frank, 'Bridging the gap between planning and scheduling', *Knowledge Engineering Review*, **15**(1), (2000).

[3] Y. Lesperance G. De Giacomo and H. Levesque, 'Congolog, a concurrent programming language based on the situation calculus', **121**, (2000).

[4] Malik Ghallab and Herv Laruelle, 'Representation and control in ixtet, a temporal planner', in *AIPS 1994*, pp. 61–67.

[5] H. Grosskreutz and G. Lakemeyer, 'ccgolog – a logical language dealing with continuous change', *Logic Journal of the IGPL*, **11**(2), 179–221, (2003).

[6] Joxan Jaffar and Michael J. Maher, 'Constraint logic programming: A survey', *Journal of Logic Programming*, **19/20**, 503–581, (1994).

[7] Ari K. Jonsson, Paul H. Morris, Nicola Muscettola, Kanna Rajan, and Benjamin D. Smith, 'Planning in interplanetary space: Theory and practice', in *Artificial Intelligence Planning Systems*, pp. 177–186, (2000).

[8] Doherty P. Kvarnstrm, J. and P. Haslum, 'Extending talplanner with concurrency and resources'.

[9] J. McCarthy, 'Situations, actions and causal laws', Technical report, Stanford University, (1963). Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pp. 410-417.

[10] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams, 'Remote agent: To boldly go where no AI system has gone before', *Artificial Intelligence*, **103**(1-2), 5–47, (1998).

[11] J.A. Pinto, 'Integrating discrete and continuous change in a logical framework', *Computational Intelligence*, **14**(1), 39–88, (1998).

[12] J.A. Pinto and R. Reiter, 'Reasoning about time in the situation calculus', *Annals of Mathematics and Artificial Intelligence*, **14**(2-4), 251–268, (September 1995).

[13] Javier Pinto, 'Occurrences and narratives as constraints in the branching structure of the situation calculus', *Journal of Logic and Computation*, **8**(6), 777–808, (1998).

[14] Fiora Pirri and Raymond Reiter, 'Planning with natural actions in the situation calculus', 213–231, (2000).

[15] R. Reiter, 'Natural actions, concurrency and continuous time in the situation calculus', in *Proceedings of KR'96*, pp. 2–13, (1996).

[16] Raymond Reiter, *Knowledge in action : logical foundations for specifying and implementing dynamical systems*, MIT Press, 2001.

[17] Raymond Reiter and Zheng Yuhua, 'Scheduling in the situation calculus: A case study', *Annals of Mathematics and Artificial Intelligence*, **21**(2-4), 397–421, (1997).

[18] B. Williams, M. Ingham, S. Chung, P. Elliott, M. Hofbaur, and G. Sullivan, 'Model-based programming of fault-aware systems', *AI Magazine*, (Winter 2003).

# Paper Session V

August 24, 11:00 - 12:30

- **Imitation and Social Learning for Synthetic Characters**, D. Buchsbaum, B. Blumberg, C. Breazeaal
- **On Reasoning and Planning in Real-Time: An LDS-Based Approach**, M. Asker, J. Malec
- **Exploiting Qualitative Spatial Neighborhoods in the Situation Calculus**, F. Dylla, R. Moratz

*

# Imitation and Social Learning for Synthetic Characters

**Daphna Buchsbaum** and **Bruce Blumberg** and **Cynthia Breazeal**[1]

**Abstract.** An increasing amount of evidence suggests that in human infants the ability to learn by watching others, and in particular, the ability to imitate, could be crucial precursors to the development of appropriate social behavior, and ultimately the ability to reason about the thoughts, intents, beliefs, and desires of others [6].

We have created a number of imitative characters and robots [2], the latest of which is Max T. Mouse, an anthropomorphic animated mouse character who is able to observe the actions he sees his friend Morris Mouse performing, and compare them to the actions he knows how to perform himself. This matching process allows Max to accurately imitate Morris's gestures and actions, even when provided with limited synthetic visual input. Furthermore, by using his own perception, motor, and action systems as models for the behavioral and perceptual capabilities of others (a process known as Simulation Theory in the cognitive literature), Max can begin to identify simple goals and motivations for Morris's behavior, an important step towards developing characters with a full theory of mind.

## 1 INTRODUCTION

Humans (and many other animals), display a remarkably flexible and rich array of social competencies, demonstrating the ability to interpret, predict and react appropriately to the behavior of others, and to engage others in a variety of complex social interactions. We believe that developing systems that have these same sorts of social abilities is a critical step in designing robots, animated characters, and other computer agents, who appear intelligent and capable in their interactions with humans (and each other), and who are intuitive and engaging for humans to interact with.

Since humans provide our inspiration for designing socially intelligent artificial systems, we have approached the challenge by turning to theories of how the ability to interpret the actions and intentions of others, often called theory of mind (ToM), develops in humans. Research in the field of cognitive development suggests that the ability to learn by watching others, and in particular, the ability to imitate, are not only important components of learning new behaviors (or new contexts in which to perform existing behaviors), but could be crucial precursors to the development of appropriate social behavior, and ultimately, theory of mind. In particular, Meltzoff ( see [6], [7], [8]) presents a variety of evidence for the presence of imitative abilities in children from very early infancy, and proposes that this capacity could be foundational to more sophisticated social learning, and to ToM. The crux of his hypothesis is that infants' ability to translate the perception of anothers action into the production of their own action provides a basis for learning about self-other similarities, and the connection between behaviors and the mental states producing them.

[1] MIT Media Laboratory, Cambridge MA, USA email: daphna@media.mit.edu

In previous work, we began to explore this hypothesis by implementing a facial imitation architecture for an interactive humanoid robot [2]. In this paper, we present a system that expands upon our prior research, by providing a robust mechanism for observing and imitating whole gestures and movements. Furthermore, the characters presented in this paper are able to use their imitative abilities to bootstrap simple mechanisms for understanding each other's low-level goals and motivations, bringing us a step closer to our goal of creating socially intelligent artificial creatures.

In the next section we briefly explore the cognitive theories motivating our approach in a bit more detail. Subsequently, we describe our imitation architecture, and in particular, look at Max and Morris Mouse, two anthropomorphic animated mouse characters who are able to interact with each other, and observe each other's behavior. We will focus especially on Max's ability to imitate Morris, and on our ongoing research into giving these characters other social learning capabilities, including learning about their environment by observing each other's behavior, and gaining the knowledge necessary to engage in cooperative activities.

## 2 UNDERSTANDING OTHER'S MINDS

For artificial creatures to possess human-like social intelligence, they must be able to infer the mental states of others (e.g., their thoughts, intents, beliefs, desires, etc.) from observable behavior (e.g., their gestures, facial expressions, speech, actions, etc.). In humans, this competence is referred to as a theory of mind (ToM) [10], folk psychology [5], mindreading [12], or social commonsense [9].

In humans, this ability is accomplished in part by each participant treating the other as a conspecific—viewing the other as being like me. Perceiving similarities between self and other is an important part of the ability to take the role or perspective of another, allowing people to relate to and to empathize with their social partners. This sort of perspective shift may help us to predict and explain others emotions, behaviors and other mental states, and to formulate appropriate responses based on this understanding. For instance, it enables us to infer the intent or goal enacted by anothers behavior—an important skill for enabling richly cooperative behavior.

### 2.1 Simulation Theory

Simulation Theory (ST) is one of the dominant hypotheses about the nature of the cognitive mechanisms that underlie theory of mind [5], [4]. It can perhaps best be summarized by the clich to know a man is to walk a mile in his shoes. Simulation Theory posits that by simulating another persons actions and the stimuli they are experiencing using our own behavioral and stimulus processing mechanisms, humans can make predictions about the behaviors and mental states of others based on the mental states and behaviors that we would possess in their situation. In short, by thinking as if we were the other

person, we can use our own cognitive, behavioral, and motivational systems to understand what is going on in the heads of others.

From a design perspective, Simulation Theory is appealing because it suggests that instead of requiring a separate set of mechanisms for simulating other persons, we can make predictions about others by using our own cognitive mechanisms to recreate how we would think, feel, and act in their situation—thereby providing us some insight into their emotions, beliefs, desires, and intentions, etc. We argue that an ST-based mechanism could also be used by robots and animated characters to understand humans and each other in a similar way. Importantly, it is a strategy that naturally lends itself to representing the internal state of others and of the character itself in comparable terms. This would facilitate an artificial creature's ability to compare its own internal state to that of a person or character it is interacting with, in order to infer their mental states or to learn from observing their behavior. Such theories could provide a foothold for ultimately endowing machines with human-style social skills, learning abilities, and social understanding.

In the following section, we discuss our Simulation Theory-based imitation and movement recognition architecture, which we have developed using two 3D computer animated characters, Max and Morris Mouse.

## 3 MAX AND MORRIS

Max and Morris are the latest in long line of interactive animated characters developed by the Synthetic Characters Group at the MIT Media Lab [11], [1], [3]. They were built using the Synthetic Characters C5m toolkit, a specialized set of libraries for building autonomous, adaptive characters and robots. The toolkit contains a complete cognitive architecture for synthetic characters, including perception, action, belief, motor and navigation systems, as well as a new, high performance graphics layer for doing Java-based OpenGL 3D Graphics. A brief introduction to a few of these systems will be given here, but it is beyond the scope of this paper to discuss them all in detail (for more information please see [1], [3]).

### 3.1 The Motor System

For most character architectures, including the one implicit in this work, a creature consists broadly of two components: a behavior system and a motor system. Where the behavior system is responsible for working out what the creature ought to be doing, the motor system is responsible for carrying out the behavior systems requests. The primary task of the motor system for a conventional 3D virtual character is therefore to generate a coordinated series of animations that take the character from where his body is now to where the behavior system would like it to be.

To approach this problem, we have created multi-resolution, directed, weighted graphs, known as *posegraphs*. To create a character's posegraph, source animation material is broken up into *poses* corresponding to key-frames from the animation, and into collections of connected poses known as *movements*. Animations can be generated and played out on the character in real-time by interpolating down a path of connected pose nodes, with edges between nodes representing allowable transitions between poses. The graph represents the possible motion space of a character, and any motor action the character executes can be represented as a path through its posegraph.

Within the posegraph representation, *movements* are of particular importance to us here. Movements generally correspond to things we might intuitively think of as complete actions (e.g, sitting, jumping, waving), and therefore often match up closely with requests from the behavior system. While the pose representation provides us with greater motor knowledge and flexibility, the movement representation is often a more natural unit to work with. More critically, because movements correspond closely to motor primitives, or to simple behaviors, they also represent the level at which we would like to parse observed actions, in order to identify and imitate them. Therefore, inspired by Simulation Theory, our characters recognize and imitate actions they observe by comparing them with the movements they are capable of performing themselves, a process we will discuss in greater detail in the following section.

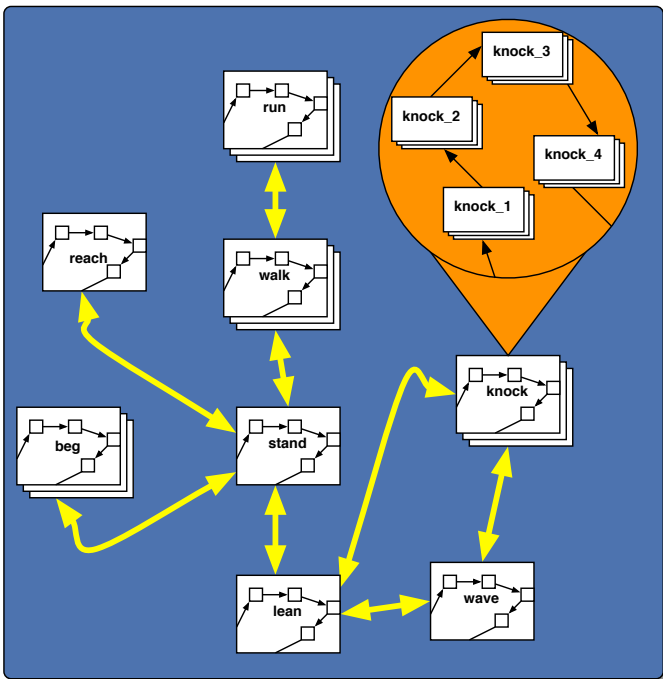



**Figure 1.** An example graph of movement nodes. Large rectangles represent movements, small squares represent poses. Stacks represent movements and poses created by blending multiple source animations together

### 3.2 Imitation and Movement Recognition

Max the Mouse is able to observe and imitate his friend Morris's movements, by comparing them to the movements he knows how to perform himself. Max watches Morris through a color-coded synthetic vision system, which uses a graphical camera mounted in Max's head to render the world from Max's perspective. The color-coding allows Max to visually locate and recognize a number of key end-effectors on Morris's body, such as his hands, nose and feet. Currently, Max is hard-wired to know the correspondence between his own effectors and Morris's (e.g. that his right hand is like Morris's right hand), but previous projects have featured characters using learned correspondences [2], and a similar extension is planned for this research.

As Max watches Morris, he roughly parses Morris's visible behavior into individual movements and gestures. Max locates places where Morris was momentarily still, or where he passed through a transitionary pose, such as standing, both of which could signal the beginning or end of an action. Max then tries to identify the observed

movement, by comparing it to all the movement representations contained within his own movement graph. To do this, Max compares the trajectories of Morris's effectors to the trajectories his own limbs would take while performing a given movement. This process allows Max to come up with the closest matching motion in his repertoire, using as few as seven visible effectors (as of writing, we have not tested the system using fewer than seven). By performing his best matching movement or gesture, Max can imitate Morris.

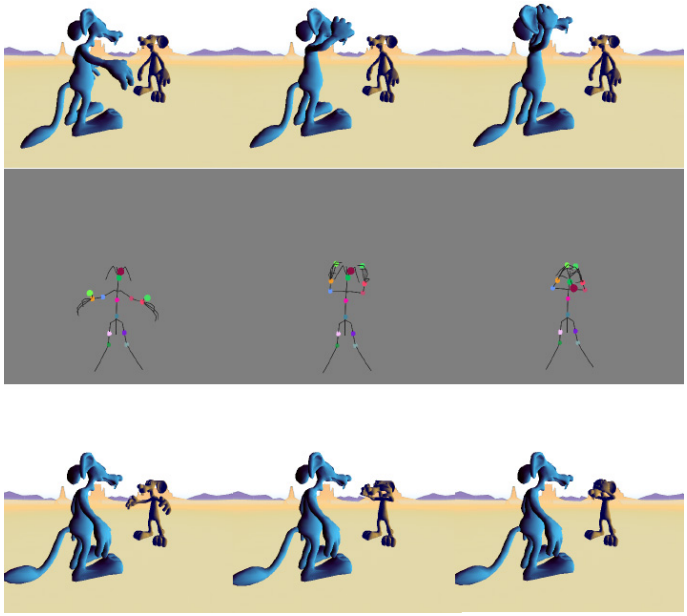


**Figure 2.** First row: Morris (blue) demonstrates an action (covering his eyes) while Max (brown) watches. Second row: Morris through Maxs eyes. The colored spheres represent key effectors. Third row: Max reproduces Morriss action, by performing the movements in his own repertoire that are closest to what he observed.

#### 3.2.1 *Matching Observed Gestures to Movements in the Graph*

As Max watches Morris demonstrate a gesture, he represents each frame of observed motion by noting the world-space positions of Morris's effectors relative to Morris's 'root-node' (the center of Morris's body). He then searches his posegraph for the poses (frames) closest to the beginning of the observed action (e.g. poses with similar hand, nose, and foot positions to those hes seen), using the Cartesian distance between corresponding effectors as his distance metric. Max uses these best-matching poses as starting places for searching his posegraph, exploring outward along the edges from these nodes, and discarding paths whose distance from the demonstrated gesture has become too high. Max can then look at the generated path through his graph and see whether it corresponds closely to any of his existing movements, or whether it represents a novel gesture.

One important benefit of using the posegraph to classify observed motion is that it simplifies the problem of dealing with partially observed (or poorly parsed) input. If Max watches Morris jump, but doesnt see the first part of the motion, he will still be able to classify the movement as jumping because the majority of the matching path in his posegraph will be contained within his own jump movement. Conversely, if Max has observed a bit of what Morris was doing before and after jumping, as well as the jump itself, he can use the fact that the entire jump movement was contained within the matching path in his graph to infer that this is the important portion of the observed motion. In general, this graph-based matching process allows observed behaviors to be classified amongst a characters own actions in real-time without needing any previous examples.

## 4 IDENTIFYING ACTIONS, MOTIVATIONS AND GOALS

Max and Morris both choose their actions using a hierarchically organized action system, composed of individual action units known as action tuples (detailed in [1]). Each action tuple contains an action to perform, trigger contexts in which to perform the action, an optional object to perform the action on, and do-until contexts indicating when the action has been completed. Within the each level of the action hierarchy, tuples compete probabilistically for expression, based on their action and trigger values.

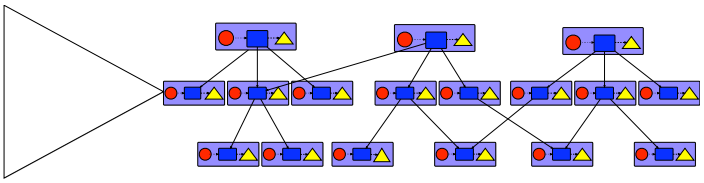


**Figure 3.** An example action system. Purple rectangles represent tuples. Red circles are trigger contexts, yellow triangles are objects, and blue rectangles are actions (do-until contexts not shown)

### 4.1 Action Identification

By matching observed gestures and movements to his own, Max is able to imitate Morris. Max can also use this same ability to try and identify which actions he believes Morris is currently performing. Max keeps a record of movement-action correspondences, that is, which action he is generally trying to carry out when he performs a particular movement (e.g. the 'reaching' gesture is most often performed during the 'getting' action). When he sees Morris perform a given movement, he identifies the action tuples it is most likely to be a part of. He then evaluates a subset of the trigger contexts, known as *can-I* triggers, to determine which of these actions was possible under the current circumstances. In this way, Max uses his own action selection and movement generation mechanisms to identify the action that Morris is currently performing.

### 4.2 Motivations and Goals

Another subset of trigger contexts, known as *should-I* triggers, can be viewed as simple motivationsfor example, a should-I trigger for Maxs eating action is hunger. Similarly, some do-until contexts, known as *success* contexts, can represent low-level goalsMaxs success context for reaching for an object is holding the object in his hands. By searching his own action system for the action that Morris is most likely to be performing, Max can identify likely *should-I* triggers and *success* do-untils for Morriss current actions. For example, if Max sees Morris eat, he can match this with his own eating action, which is triggered by hunger, and know that Morris is probably hungry. Similarly, Max can see Morris reaching for, or jumping to get, an

object, and know that Morris's goal is to hold the object in his hands, since that is the success context for Max's own 'get' action. Notice that in this second case, Max does not need to discern the purpose of jumping and reaching separately, since these are both subactions 'get' in his own hierarchy.

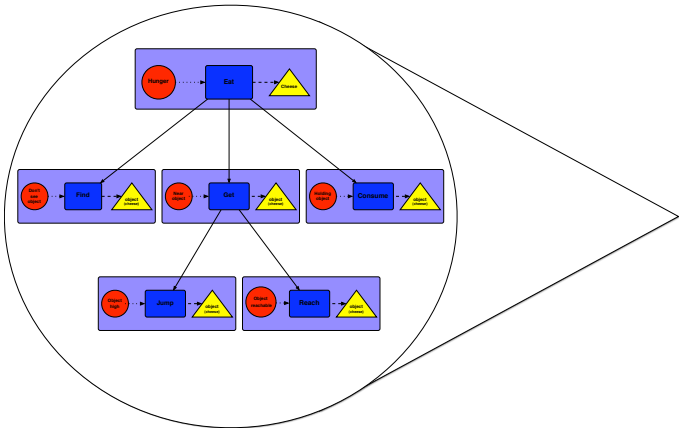



**Figure 4.** A close up of a motivational subsystem in the action system hierarchy (in this case the hunger subsystem)

We are currently developing mechanisms that allow Max to use the trigger and do-until information from his best matching action in order to interact with Morris in a more socially intelligent wayfor instance, Max might see Morris reaching and help him get the object he is reaching for, bringing him closer to more advanced social behavior such working on cooperative tasks.

## 5 CONCLUSION

We want to build animated characters and robots capable of rich social interactions with humans and each other, and who are able to learn by observing those around them. This paper presents an approach to creating imitative, interactive characters, inspired by the literature on infant development and by the Simulation Theory view of social cognition. Additionally, it introduces our ongoing work towards creating robots and animated characters who are able to understand simple motivations, goals and intentions, a critical step in creating artificial creatures who are able to interact with humans and each other as socially capable partners.

## ACKNOWLEDGEMENTS

We would like to thank the members of the Synthetic Characters and Robotic Life Groups of the MIT Media Lab for their help with, and contributions to, this project.

## REFERENCES

[1] B. Blumberg, M. Downie, Y. Ivanov, M. Berlin, M.P. Johnson, and B. Tomlinson, 'Integrated learning for synthetic characters', *ACM Transactions on Computer Graphics*, **21**, 417–426, (2002).
[2] C. Breazeal, D. Buchsbaum, J. Gray, D. Gatenby, and B. Blumberg, 'Learning from and about others: Towards using imitation to bootstrap social understanding of robots', *Artificial Life*, **forthcoming**, (2004).
[3] R. Burke, D. Isla, M. Downie, Y. Ivanov, and B. Blumberg. Creature smarts: The art and architecture of a virtual brain. Proceedings of the 2001 Computer Game Developers Conference, 2001.
[4] M. Davies and T. Stone, *Mental Simulation*, Blackwell Publishers, Oxford, 1995.
[5] R. Gordon, 'Folk psychology as simulation', *Mind and Language*, **1**, 158–171, (1993).
[6] A. Meltzoff, *The Human Infant as Imitative Generalist: A 20-Year progress report on infant imitation with implications for comparative psychology*, 347–370, Social Learnign in Animals: The Roots of Culture, Academic Press, New York, 1996.
[7] A. Meltzoff and J. Decety, 'What imitation tells us about social cognition: a rapproachment between developmental psychology and cognitive neuroscience', *Transactions of the Royal Society of London B*, **358**, 491–500, (2003).
[8] A. Meltzoff and A. Gopnik, 'The role of imitation in understanding persons and developing a theory of mind', *Developmental Psychology*, **24**, 470–476, (1993).
[9] M. Meltzoff and M.K. Moore, 'Explaining facial imitation: A theoretical model', *Early Development and Parenting*, **6**, 179–192, (1997).
[10] D. Premack and G. Woodruff, 'Does the chimpanzee have a theory of mind?', *Behavioral and Brain Sciences*, **1**, 515–526, (1978).
[11] B. Tomlinson, M. Downie, M. Berlin, J. Gray, D. Lyons, J. Cochran, and B. Blumberg. Leashing the alphawolves: Mixing user dfirection with autonomous emotion in a pack of semi-autonomous virtual characters. proceedings of the Symposium on Computer Animation, 2002.
[12] A. Whiten and W. Byrne, *Machiavellian Intelligence II: Extensions and Evaluations*, Cambridge University Press, 1997.

# On Reasoning and Planning in Real-Time: An LDS-Based Approach

**Mikael Asker** and **Jacek Malec**[1]

**Abstract.** Reasoning with limited computational resources (such as time or memory) is an important problem, in particular in cognitive embedded systems. Classical logic is usually considered inappropriate for this purpose as no guarantees regarding deadlines can be made. One of the more interesting approaches to address this problem is built around the concept of *active logics*. Although a step in the right direction, active logics still do not offer the ultimate solution.

Our work is based on the assumption that *Labeled Deductive Systems* offer appropriate metamathematical methodology to study the problem. As a first step, we have shown that the LDS-based approach is strictly more expressive than active logics. We have also implemented a prototype automatic theorem prover for LDS-based systems.

## 1 Introduction

Reasoning with limited computational resources (such as time or memory) is an important problem, in particular in cognitive embedded systems. Usually a decision, based on a principled reasoning process, needs to be taken within limited time and given constraints on the processing power and resources of the reasoning system. Therefore symbolic logic is often considered as an inadequate tool for implementing reasoning in such systems: classical logic does not guarantee that all relevant inferences will be made within prescribed time nor does it allow to limit the required memory resources satisfactorily. The paradigm shift that occured in Artificial Intelligence in the middle of 1980s can be attributed to increasing awareness of those limitations of the the predominant way of representing and using knowledge.

Since then there have been some attempts to constrain the inference process performed in a logical system in a principled way. One possibility is to limit the expressive power of the first-order logical calculus (as, e.g., in description logics) in order to guarantee polynomial-time computability. Another is to use polynomial approximations of the reasoning process. Yet another is to constrain the inference process in order to retain control over it. More details about each of those lines of research can be found in Section 6.

One of the more interesting lines of research in this area during 1990s has focused on logic as a model of an on-going reasoning process rather than as a static characterization of contents of a knowledge base. It begun with step-logic [7] and evolved into a family of *active logics*. The most recent focus of this research is on modeling dialog and discourse. However, other interesting applications like planning or multi-agent systems have also been investigated, while some other possibilities wait for analysis. In particular, the possibility of applying this framework to resource-bounded reasoning in cognitive robotic systems is in the focus of our interest.

Finally, one should name the relations to the large area of *belief revision* that also investigates the process of knowledge update rather than the static aspects of logical theories. However, there has been little attention paid to possibilities of using this approach in resource-bounded reasoning - the work has rather focused on the pure non-monotonicity aspect of knowledge revision process.

The rest of the paper is divided as follows. Section 2 presents the background of our investigation. Section 3 introduces the memory model being the foundation for active logics research. Then Section 4 presents an LDS formalization of the memory model. Section 5 discusses how the described approach could be used for planning in real-time for robotic applications. In Section 6 we briefly present related work. Finally the conclusions and some suggestion of further work are presented.

## 2 Background

The very first idea for this investigation has been born from the naive hypothesis that in order to be able to use symbolic logical reasoning in a real-time system context it would be sufficient to limit the depth of reasoning to a given, predefined level. This way one would be able to guarantee predictability of a system using this particular approach to reasoning. Unfortunately, such a modification performed on a classical logical system yields a formalism with a heavily modified and, in principle, unknown semantics [22]. It would be necessary to relate it to the classical one in a thorough manner. This task seems very hard and it is unclear for us what techniques should be used to proceed along this line. But the very basic idea of "modified provability": *A formula is a theorem iff it is provable within $n$ steps of reasoning*, is still appealing and will reappear in various disguises in our investigations.

The next observation made in the beginning of this work was that predictability (in the hard real-time sense) requires very tight control over the reasoning process. In the classical approach one specifies a number of axioms and a set of inference rules, and the entailed consequences are expected to "automagically" appear as results of an appropriate consequence relation. Unfortunately, this relation is very hard to compute and usually requires exponential algorithms. One possibility is to modify the consequence relation in such way that it becomes computable. However, the exact way of achieving that is far from obvious. We have investigated previous approaches (listed in Section 6) and concluded that a reasonable technique for doing this would be to introduce a mechanism that would allow one to control the inference process. One such mechanism is available in

[1] Department of Computer Science, Department of Computer Science, Lund University, Box 118, 221 00 Lund, Sweden, email: mikael.asker@fagotten.org, jacek@cs.lth.se

Labeled Deductive Systems [10].

In its most simple, somewhat trivialized, setting a labeled deductive system (LDS) attaches a *label* to every well-formed formula and allows the inference rules to analyze and modify labels, or even trigger on specific conditions defined on the labels. E.g., instead of the classical Modus Ponens rule $\frac{A, A\rightarrow B}{B}$ a labeled deduction system would use $\frac{\alpha:A,\ \beta:A\rightarrow B}{\gamma:B}$, where $\alpha, \beta, \gamma$ belong to a well-defined language (or, even better, algebra defined over this language) of labels, and where $\gamma$ would be an appropriate function of $\alpha$ and $\beta$. If we were to introduce our original idea of limited-depth inference, then $\gamma$ could be, e.g., $\max(\alpha, \beta) + 1$ provided that $\alpha$ and $\beta$ are smaller than some constant $N$.

A similar idea, although restricted to manipulation of labels which denote time points, has been introduced in *step-logic* [7] which later evolved into a family of *active logics* [9]. Such a restriction is actually a reasonable first step towards developing a formal system with provable computational properties. Active logics have been used so far to describe a variety of domains, like planning [21], epistemic reasoning [8], reasoning in the context of resource limitations [18] or modeling discourse. We are definitely interested in pursuing this line of investigations, however in a manner that is more amenable to metamathematical investigations. LDS seems to be a perfect technical choice for that purpose. In particular, various possibilities offered by the freedom of choice of the labeling algebras used to define the inference rules can be studied. Properties of the consequence relations defined this way are definitely worth analyzing in order to gather understanding of what can be achieved in the resource-limited setting, and what (semantical) price is paid for this.

## 3 Active Logics

Active logics originated from an attempt to formalize a memory model, inspired by cognitive psychology research, which was studied at the University of Maryland during the 1980s [5]. It has been first formalized by *step logic*. However, this formalization has left many of the interesting properties of the model outside its scope.

The memory model (MM later on) consists of five parts:

- LTM, the *long term memory*, which contains rules consisting of pairs of formulae: (trigger, contents). Semantic retrieval is associative based on trigger formulae.
- STM, the *short term memory*, which acts as the current focus of attention. All new inferences must include a formula from the STM.
- QTM, the *quick term memory*, which is a technical device for buffering the next cycle's STM content.
- RTM, the *relevant term memory*, which is the repository for default reasoning and relevance. It contains formulae which have recently been pushed out of the STM but still may be important for default resolution.
- ITM, the *intermediate term memory*, which contains all facts which have been pushed out of the STM. The contents of the ITM provides the history of the agents reasoning process. ITM may provide support for goal-directed behavior.

Three of the parts, LTM, STM and ITM, originate from cognitive psychology research. The other two, QTM and RTM, have been invented by Drapkin, Miller and Perlis, as an auxiliary technical device. Figure 1 shows how the parts are connected to each other.

## 4 Active logics as LDS-s

As the first step we have chosen the first active logic, namely the step logic $SL_7$ defined in [7]. It is, in its turn, a simplification of MM presented above. It appeared [2] that $SL_7$ can be rather straightforwardly formulated as an LDS. Below, we show how this formalization can be extended to the original MM. None of the active logic systems defined so far ([18], [17], and [13]) has been able to faithfully capture its full complexity. Therefore our first conclusion is that LDS offers a more expressive mechanism to control deduction. This chapter is based on MM presentation from [5] and $\mathbb{L}_{MM}$ from [1].

### 4.1 LDS

Traditionally a logic was perceived as a consequence relation on a *set* of formulae. Problems arising in some application areas have emphasized the need for consequence relations between *structures* of formulae, such as multisets, sequences or even richer structures. This finer-tuned approach to the notion of a logical system introduces new problems which call for an improved general framework in which many of the new logics arising from computer science applications can be presented and investigated. LDS, *labeled deductive systems*, was presented in [10] as such a unifying framework.

The first step in understanding LDS is to understand the intuitive message, which is very simple: Traditional logics manipulate formulae, while an LDS manipulates *declarative units*, i.e., pairs $formula : label$, of formulae and labels. The labels should be viewed as more information about the formulae, which is not encoded inside the formulae. E.g., they can contain reliability (in an expert system), where and how a formula was deduced, or time stamps.

A *logic* is here a pair $(\vdash, S_{\vdash})$ where $\vdash$ is a structured, possibly non-monotonic consequence relation on a language $L$ and $S_{\vdash}$ is an LDS. $\vdash$ is essentially required to satisfy no more than identity (i.e. $\{A\} \vdash A$) and a version of cut.

A simple form of LDS is the *algebraic LDS*. There are more advanced variants, *metabases*, in which the labels can be databases.

An *LDS proof system* is a triple $(\mathcal{A}, \mathbf{L}, \mathbb{R})$ where $\mathcal{A}$ is an algebra of labels (with some operations), $\mathbf{L}$ is a logical language and $\mathbb{R}$ is a discipline of labeling formulae of the logic (with labels from the algebra $A$), together with a notion of a *database* and a family of deduction rules and with agreed ways of propagating the labels via application of the deduction rules.

### 4.2 Elgot-Drapkin's Memory Model as an LDS

In our opinion the formalization of MM in step logic is an oversimplification. In particular, the STM size limit is omitted so that the number of formulae in each step may increase rapidly. This problem has also been recognized in [18], [17] and [13], which present other formal active logic systems. However, the major deficiency — the exponential growth of the number of formulae in each reasoning step — has not been satisfactorily solved by any of those approaches. In Section 5 we address this problem again, postulating a solution.

Below we present an LDS-based formulation of the Memory Model in order to show that LDS has substantially larger expressive power than any of the active logics studied so far.

The labeling algebra is based on the following structure:

$$S_{labels} \stackrel{df}{=} \{LTM, QTM, STM, ITM\} \times S_{wff} \times \{C, U\} \times \mathbb{N}^3 \tag{1}$$

where the interpretation of a tuple in $S_{labels}$ is the following. If $(loc, trigger, certainity, time, position,$ *time-left-in-rtm*$) \in S_{labels}$ is a label, then *loc* encodes the memory bank location of the formula (one of $LTM$, $QTM$, $STM$ or $ITM$), *trigger* is used for encoding

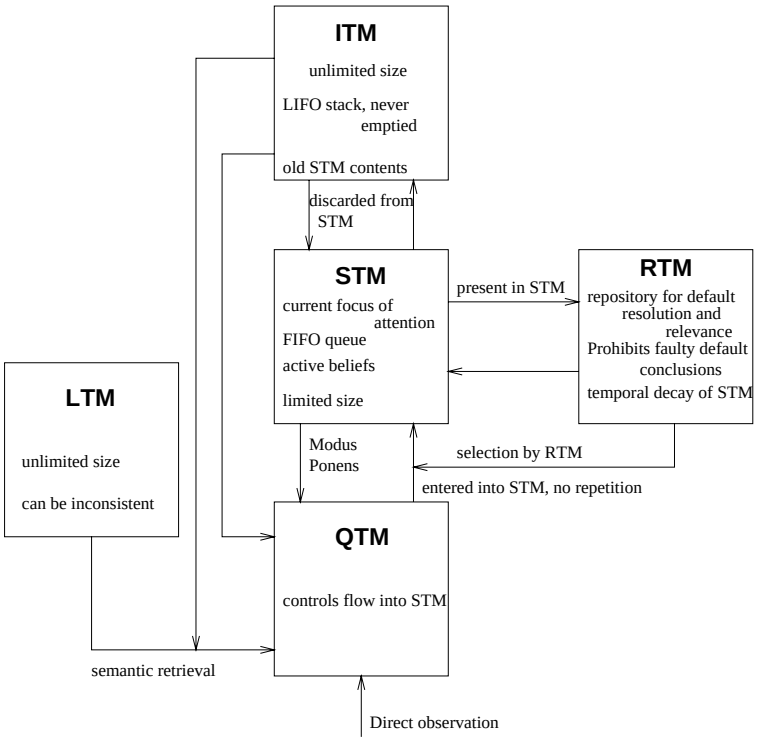



**Figure 1.** The memory model from [5].

the triggering formula for $LTM$ items (in particular, $\varepsilon$ is used to denote the empty triggering formula), *certainty* is used in case of defeasible reasoning to encode the status of the formula (certain or uncertain), *time* is the inference time, *position* denotes the formula's position in $STM$ or $ITM$, and, finally, *time-left-in-rtm* denotes the time the labeled formula should remain in the $RTM$. $R \in \mathbb{N}$ is a constant used to limit the time a formula remains in $RTM$ after it has left $STM$.

The set of axioms, $S_{axioms}$, is determined by the following three schemata:

| | | |
|---|---|---|
| (A1) | $(STM, \varepsilon, C, i, i, 0) : Now(i)$ | for all $i \in \mathbb{N}$ (CLOCK) |
| (A2) | $(QTM, \varepsilon, C, i, 0, 0) : \alpha$ | for all $\alpha \in OBS(i)$, $i \in \mathbb{N}$ (OBS) |
| (A3) | $(LTM, \gamma, C, 0, 0, 0) : \alpha$ | for all $(\gamma, \alpha) \in LTM$ (LTM) |

The first rule, SEMANTIC RETRIEVAL, describes retrieval from LTM into QTM:

$$\text{(SR)} \quad \frac{(STM, \varepsilon, c_1, i, p, R) : \alpha, (LTM, \beta, c_2, i, 0, 0) : \gamma, \alpha \; \mathcal{R}_{sr} \; \beta}{(QTM, \varepsilon, c_2, i, 0, 0) : \gamma}$$

The relation $\mathcal{R}_{sr}$ describes how the trigger formulae control the semantic retrieval.

The "real" inference using either MODUS PONENS or EXTENDED MODUS PONENS is performed from STM to QTM:

$$\text{(MP)} \quad \frac{(STM, \varepsilon, c_1, i, p_1, R) : \alpha, (STM, \varepsilon, c_2, i, p_2, R) : \alpha \rightarrow \beta}{(QTM, \varepsilon, min(c_1, c_2), i, 0, 0) : \beta}$$

(EMP)

$$\frac{\begin{array}{l}(STM, \varepsilon, c_1, i, p_1, R) : P_1 a \\ \ldots \\ (STM, \varepsilon, c_n, i, p_n, R) : P_n a \\ (STM, \varepsilon, c_{n+1}, i, p_{n+1}, R) : (\forall x)[(P_1 x \wedge \ldots \wedge P_n x) \rightarrow Qx]\end{array}}{(QTM, \varepsilon, min(c_1, \ldots, c_{n+1}), i, 0, 0) : Qa}$$

where function $min$ is defined over the set $\{U, C\}$ of certainty levels, with the natural ordering $U < C$. The idea behind it is that the status of a consequence should not be stronger than any of its premises.

The next rule, Negative Introspection, allows one to infer lack of knowledge of a particular formula at time $i$. In order to express that we need to define the set $S_{th}(i)$ of conclusions that can be drawn at time $i$. $S_{th}(i)$ can be computed by purely syntactical operations and it can be defined recursively using the inference rules. It is well-defined for every $i \in \mathbb{N}$ because the consequence relation is "directed" by the natural ordering of the set $\mathbb{N}$. Every inference rule necessarily increments the label. Therefore all the elements in $S_{th}(i)$ will be inferred from a finite number of instances of axiom (A1), namely those for which labels vary between 0 and $i - 1$, and from the finite amount of observations performed until the time $i$. As every inference rule increments the label, only a finite number of applications of every rule is possible before the label reaches $i$.

Given a finite set $S_{th}(i)$ of $i$-theorems, we can identify all closed subformulae occurring in them and not occuring as separate theorems (function $f_{csf}$). The process of finding all closed subformulae for a given finite set of formulae ($f_{formulae}$ yields unlabeled formulae) is computable.

We can now formulate the NEGATIVE INTROSPECTION rule:

$$\text{(NI)}\quad \frac{\alpha \in f_{csf}(S_{STM}(i)), \alpha \notin f_{formulae}(S_{STM}(i))}{(QTM, \varepsilon, C, i, 0, 0) : \neg K(i, \ulcorner \alpha \urcorner)}$$

where the set $S_{th}(i)$ described above is replaced by its memory-bank-specific counterparts, $S_{QTM}(i)$, $S_{new\text{-}STM}(i)$, $S_{STM}(i)$ and $S_{RTM}(i)$. Just like $S_{th}(i)$, they are computable by purely syntactic operations and can be defined recursively on $i$.

The (NI) rule involves the knowledge predicate $K$ that takes as one of its arguments a formula. Later rules will introduce predicates $Contra$ and $loses$ which behave similarly. In order to keep the language first-order we use the standard reification technique allowing us to treat formulae (or rather their names) as terms of the language. In order to make a distinction between formulae and their names, quoting (shown as $\ulcorner \alpha \urcorner$, for an arbitrary formula $\alpha$) is used.

MM in [5] and step logic use different methods to detect and handle contradictions. Step logic indicates detected contradictions with the $Contra$ predicate while MM uses instead certainty levels and the $loses$ predicate which is involved in the $RTM$ mechanism. We have allowed both possibilities, where CD1 handles the case of equal certainties while CD2 and CD2' deal with the case of different certainties:

$$\text{(CD1)}\quad \frac{(STM, \varepsilon, C, i, p_1, R) : \alpha \quad (STM, \varepsilon, C, i, p_2, R) : \neg\alpha}{(QTM, \varepsilon, C, i, 0, 0) : Contra(i, \ulcorner \alpha \urcorner, \ulcorner \neg\alpha \urcorner)}$$

$$\text{(CD2)}\quad \frac{(STM, \varepsilon, c_1, i, p_1, R) : \alpha \quad (STM, \varepsilon, c_2, i, p_2, R) : \neg\alpha \quad c_1 < c_2}{(QTM, \varepsilon, c_1, i, 0, 0) : loses(\ulcorner \alpha \urcorner)}$$

$$\text{(CD2')}\quad \frac{(STM, \varepsilon, c_1, i, p_1, R) : \neg\alpha \quad (STM, \varepsilon, c_2, i, p_2, R) : \alpha \quad c_1 < c_2}{(QTM, \varepsilon, c_1, i, 0, 0) : loses(\ulcorner \neg\alpha \urcorner)}$$

The next group of rules handles inheritance, i.e., governs the time a particular formula stays in a memory bank or is moved to another one. The first inheritance rule says that everything in $LTM$ stays in $LTM$ forever:

$$\text{(IL)}\quad \frac{(LTM, \alpha, c, i, 0, 0) : \beta}{(LTM, \alpha, c, i+1, 0, 0) : \beta}$$

The $STM$ is implemented as a FIFO queue of *sets* of declarative units, rather than as a FIFO queue of declarative units. This "lazy" implementation avoids selection among the $QTM$ contents.

One problem with the lazy $STM$ implementation is that limiting the number of sets in the $STM$ does not necessarily limit the total number of elements in those sets, which is the number of formulae in the $STM$. If many formulae are moved into $STM$ at the same time step, the sets will contain many elements, the $STM$ will contain many formulae and there will be more computation per inference step. The flow from $QTM$ to $STM$ must thus be controlled to limit the amount of computation to realistic levels. And because there is no selection among the $QTM$ contents, everything that enters $QTM$ also enters $STM$, so the flow into $QTM$ must be controlled as well.

Our $STM$ implementation uses the position field in the labels. The value of the position field should be zero, unless the associated formula is in $STM$ or $ITM$. In that case, it contains the time at which the formula was moved into $STM$ by the IQS rule. The position field then remains unchanged, while the IS rule propagates the formula forwards in time. A function $f_{min\text{-}STM\text{-}pos}(i)$ computes the minimum position value of all the declarative units in the $STM$ at time $i$. At time $i$, the declarative units in $STM$ can have position values $f_{min\text{-}STM\text{-}pos}(i), \ldots, i$, see Figure 2 below.

A simple way to define $f_{min\text{-}STM\text{-}pos}(i)$ would be to set it to $\max(0, i - S + 1)$, where $S$ is the intended maximum number of elements in $STM$. If a position field in a label is $f_{min\text{-}STM\text{-}pos}(i)$ at time $i$, then the associated formula can be moved to $ITM$ at time $i + 1$. The problem with this simple definition is that formulae will "time out" from $STM$ into $ITM$, even when no new formulae are entered into $STM$. That is not the FIFO behaviour described in[5].

Our solution to the "time out" problem is to interpret $S$ as the maximum number of *non-empty* sets in $STM$. We use a more complex definition of $f_{min\text{-}STM\text{-}pos}(i)$ and do not move anything out from $STM$ to $ITM$ if nothing is moved in from $QTM$ to $STM$. The exact $f_{min\text{-}STM\text{-}pos}(i)$ definition, rather cumbersome, is omitted but can be found in [1].

Useful formulae from $QTM$ are promoted to $STM$. Because of the "lazy" $STM$ implementation with sets of formulae in each position instead of single formulae we do not have to do much selection here. We just want to avoid multiple copies of the same formula in $STM$. We also make use of the $RTM$ content to avoid rework on contradictions which have already been resolved:

$$\text{(IQS)}\quad \frac{(QTM, \varepsilon, c, i, 0, 0) : \alpha \quad \alpha \notin f_{formulae}(S_{STM}(i)) \quad loses(\alpha) \notin f_{formulae}(S_{RTM}(i))}{(STM, \varepsilon, c, i+1, i+1, R) : \alpha}$$

When new formulae are entered into $STM$ from $QTM$, old formulae must be pushed out of $STM$ into $ITM$, to get a FIFO behaviour and to limit the $STM$ size to $S$. This is done by the (IS) and (ISI) rules which use the function $f_{min\text{-}STM\text{-}pos}$ mentioned above:

$$\text{(II)}\quad \frac{(ITM, \varepsilon, c, i, p, r) : \alpha}{(ITM, \varepsilon, c, i+1, p, \max(0, r-1)) : \alpha}$$

(IS)

$$\frac{\begin{array}{l}(STM, \varepsilon, c, i, p, R) : \alpha \\ (p > f_{min\text{-}STM\text{-}pos}(i)) \vee (S_{new\text{-}STM}(i+1) = \emptyset) \\ Contra(i-1, \ulcorner \alpha \urcorner, \ulcorner \beta \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ Contra(i-1, \ulcorner \beta \urcorner, \ulcorner \alpha \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ loses(\ulcorner \alpha \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ (\alpha \neq Now(i)) \wedge (\alpha \neq K(i-1, \beta)) \vee (K(i, \beta) \notin S_{QTM}(i)) \\ (\alpha \neq Contra(i-1, \beta, \gamma)) \vee (Contra(i, \beta, \gamma) \notin S_{QTM}(i))\end{array}}{(STM, \varepsilon, c, i+1, p, R) : \alpha}$$

(ISI)

$$\frac{\begin{array}{l}(STM, \varepsilon, c, i, p, R) : \alpha \\ (p = f_{min\text{-}STM\text{-}pos}(i)) \wedge (S_{new\text{-}STM}(i+1) \neq \emptyset) \\ Contra(i-1, \ulcorner \alpha \urcorner, \ulcorner \beta \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ Contra(i-1, \ulcorner \beta \urcorner, \ulcorner \alpha \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ loses(\ulcorner \alpha \urcorner) \notin f_{formulae}(S_{STM}(i)) \\ (\alpha \neq K(i-1, \beta)) \vee (K(i, \beta) \notin S_{QTM}(i)) \\ (\alpha \neq Contra(i-1, \beta, \gamma)) \vee (Contra(i, \beta, \gamma) \notin S_{QTM}(i))\end{array}}{(ITM, \varepsilon, c, i+1, p, R) : \alpha}$$

We can now define the LDS encoding the memory model:

**Definition 1 (Memory model LDS)** $\mathbb{L}_{MM} \stackrel{df}{=} (S_{labels}, \mathbf{L}, \mathbb{R}_{MM})$, *where the consequence relation $\mathbb{R}_{MM}$ is defined by the rules (SR), (MP), (EMP), (NI), (CD1), (CD2), (IL), (IQS), (IS), (ISI) and (II).*

$S_{labels}$ should be interpreted as an algebra.

The next step would be to show that the behaviour of $\mathbb{L}_{MM}$ is indeed as expected, namely faithfully implements the behaviour of MM. Unfortunately, it cannot be done in a formal way because the

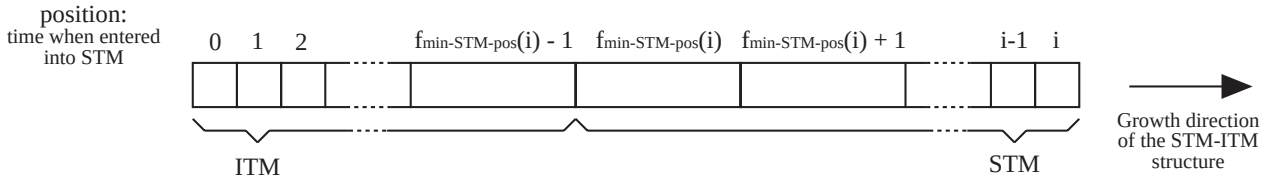



**Figure 2.** The structure of the $STM$ and $ITM$ buffer.

original memory model [5] was introduced only as an informal description of a cognitively plausible reasoning mechanism. Although this model, according to the authors, has been tested in practice, it has never been completely formalised. The subsequent formal systems, step logic and a number of active logics based on it, do not have all the control mechanisms present in the original MM. Therefore the correspondence could only be established against our interpretation of the behaviour as described in the literature. In order to achieve such result one can interpret the $\mathbb{L}_{MM}$ system using structures resembling the ones illustrated in Figure 1. This is the approach we have taken in our investigations. As usual, some more details can be found in [1].

One problem with $\mathbb{L}_{MM}$ is that the functions $S_{th}(i)$, $S_{QTM}(i)$, $S_{STM}(i)$, $S_{new\text{-}STM}(i)$ and $S_{RTM}(i)$ refer to subsets of $S_{theorems}$ which only sometimes agree with the contents of the current database. The sets are certainly computable, because one can compute them by starting from the axioms and apply the inference rules to all possible combinations of declarative units for $i$ time steps.

The sets are contained in the current database if the proof process is "completed" up to level $i$. In an implementation the time proceeds step by step and at each step the inference rules are applied to every possible combination of declarative units. So the "complete" sets above automatically become part of the current database. But when describing the system as an algebraic LDS we can't be sure of the "completeness level" of an arbitrary database. The requirement for completeness requires restrictions on the order in which inference rules are applied; some of the rules can't be applied to some of the declarative units until the database has reached a certain level of completeness.

One of the strengths of LDS is that it can handle features, which are normally at the meta-level, at the object level. It turns out that it can handle this ordering of the application of inference rules, too. The trick consists of including the whole database in the labeling of formulae. A sketch of such solution is presented in [1].

## 5 Planning in real-time

Some work on active logics has been devoted to application of this approach in real-time planning. In particular, Nirkhe [17, 18] has introduced an active logic for reasoning with limited resources and provided a modal semantics for it. However, the computational issues have not been addressed by this research.

A later formulation in [21] describes an active logic-based system, called Alma/Carne, designed for planning and plan execution in real-time. Alma is the reasoning part of the agent, based on active logic, and capable to deal with deadlines. Carne is the plan execution module, procedural in its nature, cooperating with Alma. However, the computational complexity issue inherited from the step logic, has not been addressed here either.

Although the idea behind Alma/Carne is appealing — in some sense it is rather obvious that a decent real-time cognitive system should have such a structure — the limitations of active logic, consisting of low granularity, limited to time points, of the reasoning process, are putting the possibility of practical applications of this approach into question.

As we have shown above, $\mathbb{L}_{MM}$ can offer exactly the same functionality as active logics, but with much richer structure of the labels attached to formulae. This way we can limit the number of formulae staying in focus to a small, manageable value. We can introduce a labeling in which not only time points are relevant for predicting the real-time behaviour of the system, but where the individual applications of inference rules can be counted, timed and predicted, if necessary. Therefore a solution similar to Alma/Carne, but based on $\mathbb{L}_{MM}$ (or some other suitable LDS) as the reasoning formalism, is envisioned as a possible breakthrough leading to the hard-real-time predictability of a reasoning system. The next step would be to perform the worst-case execution time analysis of the reasoning process, similarly to the one proposed in [16] for a different system.

As the first step in this direction we are developing a prototype implementation of a theorem prover for LDS-based systems, where the labeling policy and the "classical part" of an inference rule are handled in a modular way, allowing one to exchange the label processing policy while retaining those parts of inference rules (e.g., Modus Ponens or Inheritance) that deal with the actual formulae, intact. The system will provide a framework for experimenting with different LDS-s, analyzing their computational properties, and leading to a formalization that can survive the requirements of real-time. The prototype has been so far applied to simple problems, solvable in principle by hand. But already at this stage we see the benefit of the prover as a proof verifier.

## 6 Related work

The attempts to constrain in a principled way the inference process performed in a logical system have been done as long as one has used logic for knowledge representation and reasoning. One possibility is to limit the expressive power of the first-order logical calculus (as, e.g., in description logics) in order to guarantee polynomial-time computability. There is a number of theoretical results in this area (see, e.g., [6]) but we are rather interested in investigations aimed at practical computational complexity. One of the more popular approaches is to use a restricted language (like, e.g., description logics), see [12, 19, 20] for examples of this approach in practice.

Another possibility is to use polynomial approximations of the reasoning process. This approach is tightly coupled to the issue of theory compilation. The most important contributions in this area are [22, 3, 4, 14]. However, this approach, although it substantially reduces the computational complexity of the problem, still does not

provide tight bounds on the reasoning process.

Yet another possibility is to constrain the inference process in order to retain control over it. An early attempt has been reported in [15]. The next step in this direction was the step-logic [7] that evolved into a family of *active logics* [9]. Such a restriction is actually a reasonable first step towards developing a formal system with provable computational properties. Active logics have been used so far to describe a variety of domains, like planning [21], epistemic reasoning [8], reasoning in the context of resource limitations [18] or modeling discourse. However, none of the proposed systems has overcome the limitation of the exponential blow-up of the number of formulae produced in the inference process.

There is a growing insight that logic, if it is to be considered as a useful tool for building autonomous intelligent agents, has to be used in a substantially different way than before. Active logics are one example of this insight, while other important contributions might be found, e.g., in [11] or [23].

## 7 Conclusions and future work

We have presented an LDS-based formalization for the memory model entailing later formal active logic systems. This allows us to expect that even in the case of more complex, time-limited reasoning patterns, LDS will appear to be a useful and powerful tool. Actually, the technical problem with restricting the inference rule applications to a particular order in order to get hold of non-monotonic dependencies, can be solved satisfactorily by just extending the labeling algebra and then constraining the inference rule invocations by appropriately constructed predicates over these labels. LDS provides also far more sophisticated basis for defining semantics of such resource-limited reasoners, in particular, systems that reason in time and about time.

The technique described in this paper raises a number of interesting questions that we intend to investigate. First, what is the actual status of the consequence relation $\mathbb{R}_{MM}$ in the spectrum of algebraic consequence relations defined in [10]? Can this knowledge be used to better characterize the logic it captures? Is it possible to characterize the time-limited reasoning in such manner that the worst-case reasoning time (analogously to the worst-case execution time, known from the area of real-time systems) could be effectively computed? What would be then the semantical characterization of such a logic?

Another challenging problem is to practically realize a planning system based on this approach. We expect to be able to implement a $\mathbb{L}_{MM}$-based planner in the near future, and to experiment with physical robots in the next stage of the project.

Speaking slightly more generally, we hope that LDS may serve as a tool for specifying logics that artificial intelligence is looking for: formalisms describing the knowledge in flux (to quote the famous title of Peter Gärdenfors) that serve intelligent agents to reason about the world they are embedded in and about other agents, in real-time, without resorting to artificial, extra-logical mechanisms.

## ACKNOWLEDGEMENTS

The second author would like to thank Michael Fisher for pointing out LDS mechanism as a potential tool for implementing time-limited reasoning.

Sonia Fabre Escusa has made the preliminary implementation of a theorem prover for the $\mathbb{L}_{MM}$ LDS. It allowed us to find a number of inaccuracies in the text.

## REFERENCES

[1] M. Asker, *Logical Reasoning with Temporal Constraints*, Master's thesis, Department of Computer Science, Lund University, August 2003. Available at `http://ai.cs.lth.se/xj/MikaelAsker/exjobb0820.ps`.

[2] M. Asker and J. Malec, 'Reasoning with limited resources: An LDS-based approach', in *Proc. Eight Scandinavian Conference on Artificial Intelligence*, ed., et al. B. Tessem, pp. 13–24. IOS Press, (2003).

[3] M. Cadoli and F. Donini, 'A survey on knowledge compilation', *AI Communications*, (2001).

[4] M. Cadoli and M. Schaerf, 'Approximate reasoning and non-omniscient agents', in *Proc. TARK 92*, pp. 169–183, (1992).

[5] J. Drapkin, M. Miller, and D. Perlis, 'A memory model for real-time commonsense reasoning', Technical Report TR-86-21, Department of Computer Science, University of Maryland, (1986).

[6] H.-D. Ebbinghaus, 'Is there a logic for polynomial time?', *L. J. of the IGPL*, **7**(3), 359–374, (1999).

[7] J. Elgot-Drapkin, *Step Logic: Reasoning Situated in Time*, Ph.D. dissertation, Department of Computer Science, University of Maryland, 1988.

[8] J. Elgot-Drapkin, 'Step-logic and the three-wise-men problem', in *Proc. AAAI*, pp. 412–417, (1991).

[9] J. Elgot-Drapkin, S. Kraus, M. Miller, M. Nirkhe, and D. Perlis, 'Active logics: A unified formal approach to episodic reasoning', Technical report, Department of Computer Science, University of Maryland, (1996).

[10] D. Gabbay, *Labelled Deductive Systems, Vol. 1*, Oxford University Press, 1996.

[11] D. Gabbay and J. Woods, 'The new logic', *L. J. of the IGPL*, **9**(2), 141–174, (2001).

[12] G. De Giacomo, L. Iochhi, D. Nardi, and R. Rosati, 'A theory and implementation of cognitive mobile robots', *J. Logic Computation*, **9**(5), 759–785, (1999).

[13] A. Globerman, *A Modal Active Logic with Focus of Attention for Reasoning in Time*, Master's thesis, Department of Mathematics and Computer Science, Bar-Illan University, 1997.

[14] G. Gogic, C. Papadimitriou, and M. Sideri, 'Incremental recompilation of knowledge', *JAIR*, **8**, 23–37, (1998).

[15] H. Levesque, 'A logic of implicit and explicit belief', in *Proc. AAAI 84*, pp. 198–202, (1984).

[16] M. Lin and J. Malec, 'Timing analysis of RL programs', *Control Engineering Practice*, **6**, 403–408, (1998).

[17] M. Nirkhe, *Time-Situated Reasoning Within Tight Deadlines and Realistic Space and Computation Bounds*, Ph.D. dissertation, Department of Computer Science, University of Maryland, 1994.

[18] M. Nirkhe, S. Kraus, and D. Perlis, 'Situated reasoning within tight deadlines and realistic space and computation bounds', in *Proc. Common Sense 93*, (1993).

[19] P. F. Patel-Schneider, 'A decidable first-order logic for knowledge representation', in *Proc. IJCAI 85*, pp. 455–458, (1985).

[20] P. F. Patel-Schneider, 'A four-valued semantics for frame-based description languages', in *Proc. AAAI 86*, pp. 344–348, (1986).

[21] K. Purang, D. Purushothaman, D. Traum, C. Andersen, D. Traum, and D. Perlis, 'Practical reasoning and plan execution with active logic', in *Proceedings of the IJCAI'99 Workshop on Practical Reasoning and Rationality*, (1999).

[22] B. Selman and H. Kautz, 'Knowledge compilation and theory approximation', *JACM*, **43**(2), 193–224, (1996).

[23] M. Wooldridge and A. Lomuscio, 'A computationally grounded logic of visibility, perception, and knowledge', *L. J. of the IGPL*, **9**(2), 257–272, (2001).

# Exploiting Qualitative Spatial Neighborhoods in the Situation Calculus

**Frank Dylla and Reinhard Moratz**[1]

**Abstract.** We present first ideas on how results about qualitative spatial reasoning can be exploited in reasoning about action and change. Current work concentrates on a line segment based calculus, the dipole calculus and necessary extensions for representing navigational concepts like *turn right*. We investigate how its conceptual neighborhood structure can be applied in the situation calculus for reasoning qualitatively about relative positions in dynamic environments.

## 1 Introduction

Most current reliable robot systems are based on a completely determined geometrical world model. The applied metric methods are tending to fail in frequently changing spatial configurations and when accurate distance and orientation information is not obtainable. Qualitative representation of space abstracts from the physical world and enables computers to make predictions about spatial relations, even when precise quantitative information is not available [1]. Important aspects are topological and positional (orientation and distance) information about in most cases physically extended objects. Calculi dealing with such information have been well investigated over recent years and give general and sound reasoning strategies, e.g. about topology of regions as in RCC-8 [24, 25], about relative position orientation of three points as in Freksa's Double-Cross Calculus [6] or about orientation of two line segments as in the Dipole Calculus [22]. For reasoning with calculi as above standard constraint-based reasoning techniques can be applied. In this paper we investigate conceptual neighborhoods [5]. Two relations are conceptual neighbors if their spatial configurations can be continuously transformed into each other with only minimal change, e.g. in RCC-8 two disconnected regions (configuration 1) cannot overlap (3) without being externally connected (2) in between. Therefore 1 and 2 resp. 2 and 3 are conceptually neighboring relations but not 1 and 3. Expressing these connections between the relations leads to conceptual neighborhood graphs (CNH-graph). For further motivation for qualitative spatial reasoning we refer to [7] .

Frameworks for reasoning about action and change, e.g. the Situation Calculus [19] based programming language Golog [18], also provide facilities for representing and reasoning about sets of spatial locations. Current variants are able to deal with e.g. concurrency [3], continuous change [13] or decision theory [8].

[1] SFB/TR8 "Spatial Cognition" , University of Bremen, email:{dylla, moratz}@tzi.de

Unfortunately no formal spatial theory, e.g. for relative position terms like left, right or behind, is defined within for dealing with underspecified, coarse knowledge. Therefore every project modeling dynamic domains needs the naive formalization of such a theory by the developer again and again, although such concepts have been formally investigated.

We present first ideas about qualitative navigation on the bases of oriented line segments, which we consider as valuable starting point. In this context we will show one way how the results about conceptual neighborhood can be applied in the situation calculus resp. Golog. In the first stage of this work we will only look at simulated scenarios to omit additional complexity caused by real robot control.

The long term goal is a general representation and usage of qualitative spatial concepts about relative position like e.g. $left$, $right$, or $inFrontOf$ within the situation calculus resp. the programming language Golog, e.g. providing action facilities like $go(leftOf, exhibit_7)$. We do not only expect such formal qualitative concepts being useful in agent programming but also in human machine interaction [31, 20].

In this paper we will present several variants of the Dipole Calculus at different levels of granularity and their corresponding conceptual neighborhoods. We present necessary extensions for expressing robot navigation tasks more adequately. After a brief introduction of the Situation Calculus and the programming language Golog we will present a first approach combining the Dipole Calculus in the Situation Calculus resp. Golog.We will clarify our ideas with concrete examples and end with final conclusions.

## 2 Qualitative Spatial Reasoning

Qualitative Spatial Reasoning (QSR) is an abstraction that summarizes similar quantitative states into one qualitative characterization. From a cognitive perspective the qualitative method *compares* features of the domain rather than *measuring* them in terms of some artificial external scale [6]. The two main directions in QSR are topological reasoning about regions, e.g. the RCC-8, and positional reasoning about point configurations. An overview is given in [2].

Solving navigation tasks involves reasoning about paths as well as reasoning about configurations of objects or landmarks perceived along the way and thus requires the representation of orientation and distance information [28, 15]. Many approaches deal with global allocentric metrical data. For many navigational tasks we do not need absolute allocentric information about the position, instead we need relative egocentric

representations and a fast process for updating these relations during navigation [32, 33].

Several calculi dealing with relative positional information have been presented in recent years. Freksa's double cross calculus [6] deals with triples of points and can also be viewed as a positional binary relation between a dipole and a point. Schlieder proposed a calculus based on line segments with clock or counter-clock orientation of generating start and end points in [29]. He presented a CNH-graph of 14 basic relations. Isli and Cohn [14] presented a ternary algebra for reasoning about orientation. This algebra contains a tractable subset of base relations.

Moratz et al. [22] extend Schlieder's calculus. In a first variant ten additional relations are regarded, where the two dipoles meet in one point, resulting in a relation algebra in the sense of Tarski [16] with 24 basic relations. Also an extended version with 69 basic relations is introduced such that spatial configurations can be distinguished in a more fine grained fashion. An application oriented calculus dealing with ternary point configurations (TPCC) is presented in [21]. It is suited for both, human robot communication [20] and spatial reasoning in route graphs [21]. Even more fine grained calculi can be used to do path integration for mobile robots [23]. In [34] a line segment approach for shape matching in a robotic context is presented.

## 2.1 Neighborhood-based reasoning

Neighborhood-based reasoning describes whether two spatial configurations of objects can be transformed into each other by small changes [5]. The conceptual neighborhood of a qualitative spatial relation which holds for a spatial arrangement is the set of relations into which a relation can be changed with minimal transformations, e.g. by continuous deformation. Such a transformation can be a movement of one object of the configuration in a short period of time. On the discrete level of concepts, neighborhood corresponds to continuity on the geometric or physical level of description: continuous processes map onto identical or neighboring classes of descriptions [7]. Spatial neighborhoods are very natural perceptual and cognitive entities and other neighborhood structures can be derived from spatial neighborhoods, e.g. temporal neighborhoods.

A movement of an agent can then be modeled qualitatively as a sequence of neighboring spatial relations which hold for adjacent time intervals. Using this qualitative representation of trajectories neighborhood-based spatial reasoning can be used as a simple, abstract model of robot navigation and exploration. Neighborhoods can be formed recursively and represented by hierarchical tree or lattice structures.

Schlieder [29] mapped orientation onto ordering. He defined the orientation on triangles and for every set with more than three points recursively for every triangle. He extracted 14 basic relations to reason about ordering of line segments[2]. The conceptual neighborhood graph is shown in Fig. 3. The labels are illustrated in Fig. 4.

From the neighborhood graphs of the individual relations, the neighborhood graph of the overall configuration must be derived. In this global neighborhood graph, spatial transformations from a start state to a goal state can be determined. It has been investigated to use the neighborhood graph of two objects for spatial navigation [29]. It has not been investigated yet how a neighborhood graph for a configuration of more complex or even several objects can be constructed using efficient, qualitative methods based on local knowledge.

A problem for the efficient construction of neighborhood graphs for multiple objects is the combinatorial explosion due to the combined neighborhoods of all objects. A potential solution to this problem is to locally restrict the combination of transitions. If we partition the environment of the moving agent into small parts and then only the neighborhood transition graph for these smaller spatial configurations needs to be considered.

[2] 16 potential triangle configurations, but two configurations are geometrically impossible.

## 2.2 Dipole Relation Algebra

In [22] a qualitative spatial calculus dealing with two directed line segments, in the following also called *dipole*, as basic entities was presented. These dipoles are used for representing spatial objects with intrinsic orientation. A dipole $A$ is defined by two points, the start point $s_A$ and the end point $e_A$. The presented calculus deals with the orientation of two dipoles. An example of the relation lrrr is shown in Fig. 1. The four letters denote the relative position (e.g. *left* or *right*) of one of the points to the other dipole:

$$A \text{ lrrr } B := A \text{ l } s_B \wedge A \text{ r } e_B \wedge B \text{ r } s_A \wedge B \text{ r } s_A$$

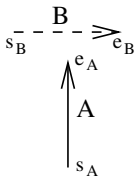



**Figure 1.** The lrrr orientation relation between two dipoles

Based on a two dimensional continuous space, $\mathbb{R}^2$, the location and orientation of two different dipoles can be distinguished by representing the relative position of start and end points. This means *left* or *right* and the same *start* or *end* point if no more than three points are allowed on a line, and without this restriction *back*, *interior* and *front* additionally (Fig. 2).

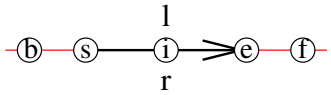



**Figure 2.** Extended dipole point relations

The first view leads to 24 *jointly exhaustive and pairwise disjoint* (jepd) basic relations, i.e. between any two dipoles exactly one relation holds at any time. Additionally they build

up a relation algebra ($\mathcal{DRA}_{24}$). A visualization is given in Fig. 4. Because of forming a relation algebra standard constraint-based reasoning techniques can be applied. The unrestricted version leads to a relation algebra with 69 basic relations ($\mathcal{DRA}_{69}$).

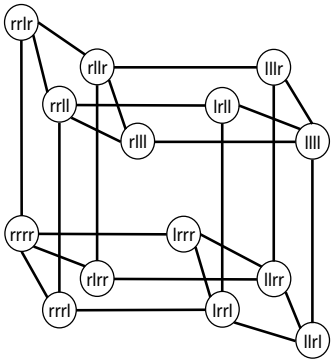



**Figure 3.** The conceptual neighborhood for the 14 basic relations by Schlieder

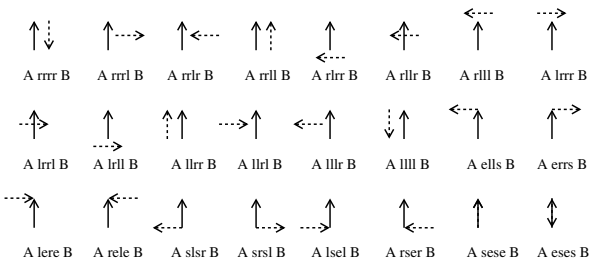



**Figure 4.** The 24 atomic relations of the dipole calculus. In the dipole calculus orthogonality is not defined, although the visual presentation might suggest.

## 2.3 Extended Dipole Relation Algebra

Unfortunately $\mathcal{DRA}_{69}$ may not be sufficient for robot navigation tasks, because many different dipole configurations are pooled in one relation. Thus we extend the representation with additional orientation knowledge and derive $\mathcal{DRA}_{77}$.

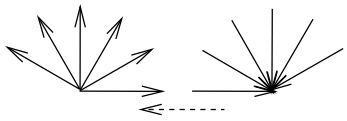


**Figure 5.** Pairs of dipoles (A: solid, B: dashed) subsumed by the same relation A(rrrr)B

Fig. 5 for example visualizes the large configuration space for the rrrr relation. This might lead to quite squiggly pathes if using these concepts for robot navigation. Other relations being extremely coarse are llrr, rrll and llll. We would expect a more goal directed behavior breaking up the relations by regarding the angle spanned by the two dipoles qualitatively. This gives us an important additional distinguishing feature with four distinctive values. These qualitative distinctions are parallelism (P) or anti-parallelism (A) and mathematically positive and negative angles between $A$ and $B$, leading to three refining relations for each of the four above mentioned relations (Fig. 6).

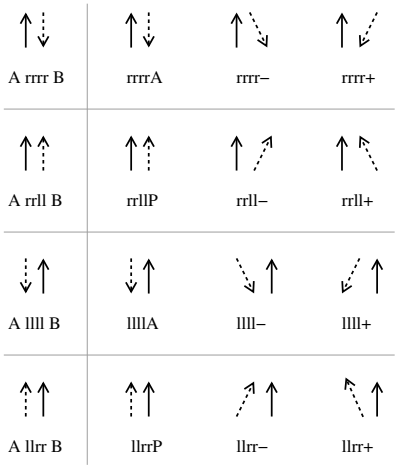



**Figure 6.** Refined base relations in $\mathcal{DRA}_{77}$

For the other relations a '+' or '−', 'P' or 'A' respectively, is already determined by the original base relation. We give a complete list of the resulting $\mathcal{DRA}_{77}$ algebra:

1. Original relations from$\mathcal{DRA}_{24}$:
   (a) Expanded relations (12): rrrr+, rrrrA, rrrr-, rrll+, rrllP, rrll-, llrr+, llrrP, llrr-, llll+, llllA, llll-
   (b) Unmodified relations (20): rrrl-, rrlr+, rlrr+, rllr+, rlll+, lrrr-, lrrl-, lrll-, llrl-, lllr+ ells+, errs-, lere-, rele+, slsr+, srsl-, lsel-, rser+, seseP, esesA
2. Additional cases on one line[3], seseP and esesA are already defined in 1.(b):
   (a) Basic Allen cases (12): ffbbP, efbsP, ifbiP, bfiiP, sfsiP, beieP, bbffP, bsefP, biifP, iibfP, sisfP, iebeP
   (b) Converse cases relative to Allen (9): ffffA, fefeA, fifiA, fbiiA, fseiA, ebisA, iifbA, eifsA, isebA
3. Other additional cases:
   (a) Without converse (12): lllb+, llfl-, llbr+, llrf-, lirl+, lfrr-, lril-, lrri+, blrr-, irrl-, frrr+, rbrr+
   (b) The converse (12): lbll-, flll+, brll-, rfll+, rlli-, rrlf+, illr+, rilr-, rrbl+, rlir+, rrfr-, rrrb-

[3] For a relation algebra about this subset of $\mathcal{DRA}_{69}$ see [27].

For lack of space we refer to our web page[4] for the CNH-graphs respectively CNH-tables for $\mathcal{DRA}_{24}$, $\mathcal{DRA}_{69}$ and $\mathcal{DRA}_{77}$. Restricting to relations suited for robotic navigational tasks where dipoles represent solid objects[5] we end up with only 39 base relations, thus giving us a condense CNH-graph.

## 3 The Situation Calculus

The situation calculus is a second order language for representing and reasoning about dynamic domains. Although many different variants have been developed from the original framework for dealing with e.g. concurrency [3], continuous change [11, 13] or uncertainty [10] all dialects are based on three sorts: *actions*, *situations* and *fluents*.

All changes in the world are caused by an action $a_i$ in the specific situation $s_i$ resulting in the successor situation $s_{i+1}$. The special constant $S_0$ denotes the *initial situation* where no action is performed. The binary function $s_{i+1} = do(a_i, s_i)$ starting from $S_0$ together with a sequence of actions forms a history. Actions are only applicable in the specific situation if preconditions hold which are axiomatized by the special predicate $Poss(a, s) \equiv preconditions$. Fluents are features of the world that might change from situation to situation, e.g. the agents *position* is changed by a *go*-action. Two fluent types can be distinguished. Relational fluents describe truth values while functional fluents hold general values and both might change over situations. They are denoted by predicate resp. function symbols holding the situation as last argument. The action effects on fluents are axiomatized in so called successor state axioms (SSA) [26]. The general form of a SSA for a relational fluent $F$ is

$$\begin{aligned} Poss(a, s) \quad \Rightarrow \quad & (F(.) = true \equiv \\ & a\, makes\, F(.)\, true \\ & \vee F(.) = true\, and\, a\, makes\, no\, change). \end{aligned}$$

With a basic action theory as presented in [17], namely the action precondition axioms, the successor state axioms, the initial situation and an additional unique names axiom a domain model can be formalized.

Golog [18] is a programming language based on the situation calculus for specifying complex tasks like those typically found in robotic scenarios. Golog offers programming constructs well known from imperative programming languages like *sequence*, *if-then-else*,, *while* and *recursive procedures*. Additionally, a *nondeterministic choice* operator is provided to choose from the given alternatives during runtime. Another important difference compared to most other programming languages is the notion of a *test condition*, which in general can be an arbitrary first order sentence.

Golog programs can be viewed as macros for complex actions which are mapped onto primitive actions in the situation calculus. With the above given features Golog serves as integrative framework for programming and planning in deterministic domains. Central for the semantics is the ternary relation $Do(\delta, s, s')$ which is a mapping onto a situation calculus formula. Roughly spoken $Do(\delta, s, s')$ means that given a program $\delta$ the situation $s'$ is reachable starting in $s$. Several extensions e.g. dealing with concurrency [3], sensing [9], continuous change [13], probabilistic projections [12] or decision theory [4, 30] have been presented.

[4] WWW.SFBTR8.UNI-BREMEN.DE/PROJECT/R3/CNH/

[5] Other non solid objects like doorways may also be represented by dipoles.

## 4 Examples

We have presented on the one hand the situation calculus as framework for reasoning about action and change, which spatial relations are build on an absolute geometrical coordinate system. On the other hand we presented the line segment based dipole calculus together with its conceptual neighborhood (CNH) graph for reasoning about relative position. The CNH-graph describes possible qualitative transitions between adjacent relative configurations by continuous motion.

Regarding only two dipoles (compare to Fig. 1 with the dashed dipole representing an agent and the solid dipole a static object) in $\mathcal{DRA}_{24}$ the term *behind* may be defined by relation rlrr and lrll resp. *front* by rlll and lrrr. In the following we will restrict dipoles to representing only solid objects.

### 4.1 General Assumptions and Definitions

Below we will use our newly developed dipole calculus $\mathcal{DRA}_{77}$, because we consider $\mathcal{DRA}_{69}$ not being fine grained enough, especially in the context of turning operations. As stated above the CNH-graph is presented on our web page[4]. We define the symmetric binary relation $cnh(p, q)$ holding if two relations $p$ and $q$ are conceptually neighboring. We denote the set of all defined dipoles in the domain with $D$.

A simple object is a single dipole. A complex object is a polygon, i.e. a sequence of $n$ dipoles $R_i \in D$ where two censecutive dipoles share a common point. For a closed complex object $R_0$ and $R_n$ must share a common point as well. How such representations can be efficiently and in a compact way be extracted is shown in [34].

Modeling a robot domain in the situation calculus at least one fluent $pos(s)$ for holding the recent position is needed: $pos(s) = \langle r_i, o \rangle$ with $r_i \in \mathcal{DRA}_{77}$ and $o \in O$. In our examples we consider only the basic navigational action $go(r_i, o)$. The precondition that the agent is not blocked holds at any time. Other actions dealing with relative positional information in a domain are e.g. transporting an object $R$ from current position to destination $\langle r_{dest}, O_{dest} \rangle$: $bring(R, r_{dest}, O_{dest})$ or informational questions about spatial configurations.

Because of restricting dipoles to representing only solid objects we can denote subsets (not nessecarily disjoint) of relations suitable for intra-object, agent-object and inter-object relations, regarding a dipole and an object. As defined above the subsequent dipoles of the intra-object description need to share a common point. Therefore only relations containing an $e$ or $s$ are suitable for object descriptions. For the purpose of simplicity we omit the case of an internal connection of two dipoles. Assuming the agent not being allowed to touch any other object only relations without sharing a start, end or internal point are applicable. Thus we can define a subset of relations $\mathcal{DRA}_{77}^{object}$ suitable for intra-object definition.

$$\mathcal{DRA}_{77} \supset \mathcal{DRA}_{77}^{object} = \{\text{ells+,errs-,lere-,rele+,slsr+,srsl-,lsel-,rser+}\}$$

For agent-object relations all other relations except relations containing an internal dipole connection are suitable, for inter-object relations all $\mathcal{DRA}_{77}$ relations may be used.

## 4.2 Naive implementation for two dipoles

In a first step we show how a CNH structure might be represented in the situation calculus for two dipoles representing an agent $A$ and an arbitrary object $R$. The successor state axiom for the *go*-action looks the same as in other domain models without a formal qualitative spatial theory:

$$Poss(a,s) \quad \Rightarrow \quad [pos(do(a,s)) = \langle r_j, o\rangle \equiv a = go(r_j,o) \vee [pos(s) = \langle r_j, o\rangle \wedge a \neq go(r_x, o_x)]]$$

But the graph structure of the dipole calculus helps us for the definition of the preconditions by exploiting the adjacency of the conceptual neighborhood structure. A movement of the agent to a relative position towards the object is only possible if he is already in a conceptually neighboring configuration. This results in:

$$Poss(go(r_j,o),s) \quad \Leftrightarrow \quad pos(s) = \langle r_i, o\rangle \wedge cnh(r_i, r_j).$$

Assuming an agent $A$ and an object $R$ being in relative position $A$(lrrr-)$B$ with the goal of being $A$(ffffA)$B$. The situation calculus and CNH-graph will give the same solution, namely two options to go around $R$. We sketched the action sequence resp. the transition through neighboring CNH-graph states in Fig. 7.

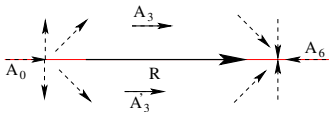



**Figure 7.** Simple example with two options for Agent $A$ going round object $R$

## 4.3 Complex objects (Going round the Kaaba)

Now we present an example for a complex object. One of the tasks during the hadsch (the great Muslim pilgrimage) is rounding the Kaaba (a cubic building in the main mosque in Mekka) seven times. The knowledge about the Kaaba $k$ (compare Fig. 8) can be represented as follows:

$$R_0(errs-)R_1 \wedge R_1(errs-)R_2 \wedge R_2(errs-)R_3 \wedge R_3(errs-)R_0$$

The agent $A$ starts in position $A_0$ with $A_0$(rrllP)$R_0$. At this time the other walls of the Kaaba are of no interest for determination of the relative position. Going round the corner of $R_0$ and $R_1$ we get the relations shown in Fig. 9.

Looking at all relations for a round trip this repeats for all corners while the other sides provide no useful knowledge.

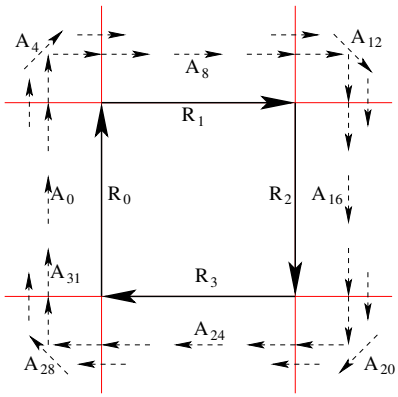



**Figure 8.** The 32 different qualitative positions of the agent $A$ relative to the kaaba $\{R_0 ... R_3\}$

| $R_x$ / $A_y$ | $A_0$ | $A_1$ | $A_2$ | $A_3$ | $A_4$ | $A_5$ | $A_6$ | $A_7$ | $A_8$ |
|---|---|---|---|---|---|---|---|---|---|
| $R_0$ | *rrllP* | *rrllP* | *rrllP* | *rrllP* | *rrll+* | *rrlf+* | *rrlr+* | *rrfr+* | *rrrr+* |
| $R_1$ | *rrrr−* | *rrrb−* | *rrrl−* | *rrbl−* | *rrll−* | *rrllP* | *rrllP* | *rrllP* | *rrllP* |

**Figure 9.** The first nine relations going round the Kaaba with $A_y(r_{y,x})R_x$

Thus in this example only two sides are sufficient for describing the relative position of an agent towards the complete object. We expect this being true for more complex, but convex objects, although we have no formal proof so far.

## 4.4 Going towards macro definitions

After extracting the neighborhoods for one complex action possibility like "turn right" we are now heading for some sort of macro definition such that an agent is able to perform a "turn right" on the basis of line segments and imprecise orientation information.

Imagine being blind standing at a wall with the task of turning right at the next corner with arbitrary angle and describing it to an external person. The only sensor is one's own right hand extended to the right front which can be seen as some sort of coarse "orientation sensor" transfering the task to a robot. One way describing the process of the first right turn in Fig. 8 might be:

1. Follow the wall until you feel an edge ($A_1$)
2. Go a little straight ahead so that the edge is to the right of you ($A_2$), i.e. the next wall comes just into reach on the right side.
3. Turn (in a bow) right around the corner until you are parallel to the next wall ($A_3$-$A_5$)
4. Go a little straight ahead until the first wall just gets out of reach ($A_6$)
5. Go straight ahead until the corner is right behind you ($A_7$)
6. Follow the wall ($A_8$)

All the named actions can be modeled as local behaviors and with the help of the base relations presented in the $\mathcal{DRA}_{77}$. If for example loosing parallelity ($A(rrllP)R_0$) to a wall while following it, we have to look whether we have a

mathematically positive or negative orientation towards the relating dipole and turn respectively. We will take such descriptions as a basis for our macro definitions. In the first sight the relations of the ($\mathcal{DRA}_{77}$) might seem to be too fine grained to represent a simple behavior like turning right adequately. But without the additional relations compared to the ($\mathcal{DRA}_{24}$) we have not found a way making the transition from one reference dipole to another (from $R_0$ to $R_1$) possible, which is necesarry to model going round a corner.

## 5 Conclusion and Outlook

We presented the design and implementation how the concept of conceptual neighborhood could be exploited for reasoning about relative positional information in the situation calculus in the absence of precise quantitative information. We introduced an extended dipole relation algebra $\mathcal{DRA}_{77}$ better suited for spatial navigation. We expect that every qualitative calculus with a conceptual neighborhood can be translated in a straightforward manner naively onto preconditions and successor state axioms. We have shown by example that not all dipoles of a complex object are necessary to determine the relative position towards the object. We expect the results for connected complex objects being applicable for several not connected dipoles. Additionally we extracted several subsets of the base relations for representing a complex object and dynamic agent behavior.

Future work will deal with the questions how to keep the position representation small for more than one dipole respectively object. A naive implementation would lead to a combinatorial explosion, because the relative position of the agent has to be traced for every single dipole. We will also look on the effects allowing dipoles to represent non-solid entities, e.g. doorways, and potentials to define some sort of general macro definitions for $turnLeft$ resp. $turnRight$ or $GoAround$ by paths in the conceptual neighborhood graph.

## 6 Acknowledgment

The authors like to thank Diedrich Wolter, Christian Freksa, Jochen Renz and Marco Ragni for fruitful discussions and impulses. And we would like to thank Marc-Björn Seidel for computing the neighborhood graph for different calculi. Our work was supported by the DFG Transregional Collaborative Research Center SFB/TR 8 "Spatial Cognition".

## References

[1] Anthony G. Cohn, 'Qualitative spatial representation and reasoning techniques', in *KI-97: Advances in Artificial Intelligence, 21st Annual German Conference on Artificial Intelligence, Freiburg, Germany, September 9-12, 1997, Proceedings*, eds., Gerhard Brewka, Christopher Habel, and Bernhard Nebel, volume 1303 of *Lecture Notes in Computer Science*, pp. 1 – 30, Berlin, (1997). Springer.

[2] Anthony G. Cohn and Shyamanta M. Hazarika, 'Qualitative spatial representation and reasoning: An overview', *Fundamenta Informaticae*, **46**(1-2), 1–29, (2001).

[3] Giuseppe De Giacomo, Yves Lésperance, and Hector J. Levesque, 'ConGolog, A concurrent programming language based on situation calculus', *Artificial Intelligence*, **121**(1–2), 109–169, (2000).

[4] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer, 'Extending DTGolog with Options', in *Proc of the 18th International Joint Conference on Artificial Intelligence*, (2003).

[5] Christian Freksa, 'Conceptual neighborhood and its role in temporal and spatial reasoning', in *Proceedings of the IMACS Workshop on Decision Support Systems and Qualitative Reasoning*, eds., Madan G. Singh and Luise Travé-Massuyès, pp. 181 – 187, North-Holland, Amsterdam, (1991). Elsevier.

[6] Christian Freksa, 'Using orientation information for qualitative spatial reasoning', in *Theories and methods of spatio-temporal reasoning in geographic space*, eds., A. U. Frank, I. Campari, and U. Formentini, 162 – 178, Springer, Berlin, (1992).

[7] Christian Freksa, 'Spatial cognition - an ai prespective', in *Proceedings of 16th European Conference on AI (ECAI 2004)*, (2004).

[8] Christian Fritz, *Integrating decision-theoretic planning and programming for robot control in highly dynamic domains*, Master's thesis, RWTH Aachen (Knowledge-based Systems Group), Aachen, Germany, 2003.

[9] G. De Giacomo and H. Levesque. An incremental interpreter for high-level programs with sensing.

[10] Henrik Grosskreutz, 'Probabilistic projection and belief update in the pGOLOG framework', in *CogRob-00 at ECAI-00*, (2000).

[11] Henrik Grosskreutz and Gerhard Lakemeyer, 'cc-Golog: Towards more realistic logic-based robot controllers', in *AAAI-00*, (2000).

[12] Henrik Grosskreutz and Gerhard Lakemeyer, 'Turning high-level plans into robot programs in uncertain domains', in *ECAI-00*, (2000).

[13] Henrik Grosskreutz and Gerhard Lakemeyer, 'On-line execution of cc-Golog plans', in *IJCAI-01*, (2001).

[14] Amar Isli and Anthony G. Cohn, 'An algebra for cyclic ordering of 2d orientations', in *AAAI/IAAI*, pp. 643–649, (1998).

[15] B. J. Kuipers, 'Representing knowledge of large-scale space', Technical Report 418, (1977).

[16] P Ladkin and R Maddux, 'On binary constraint problems', *Journal of the Association for Computing Machinery*, **41**(3), 435–469, (1994).

[17] H. Levesque, F. Pirri, and R. Reiter, 'Foundations for the situation calculus', in *Linköping Electronic Articles in Computer and Information Science*, volume 3, (1998).

[18] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl, 'GOLOG: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(1-3), 59–83, (1997).

[19] J. McCarthy, 'Situations, actions and causal laws', Technical report, Stanford University, (1963).

[20] R. Moratz, T. Tenbrink, J. Bateman, and K. Fischer, 'Spatial knowledge representation for human-robot interaction', in *Spatial Cognition III*, eds., Christian Freksa, W. Brauer, C. Habel, and K. F. Wender, Springer, Berlin, Heidelberg, (2002).

[21] Reinhard Moratz, Bernhard Nebel, and Christan Freksa, 'Qualitative spatial reasoning about relative position: The tradeoff between strong formal properties and successful reasoning about route graphs', in *Spatial Cognition III*, eds., Christian Freksa, Wilfried Brauer, Christopher Habel, and Karl Friedrich Wender, volume 2685 of *Lecture Notes in Artificial Intelligence*, 385–400, Springer, Berlin, Heidelberg, (2003).

[22] Reinhard Moratz, Jochen Renz, and Diedrich Wolter, 'Qualitative spatial reasoning about line segments', in *Proceedings of ECAI 2000*, pp. 234–238, (2000).

[23] Reinhard Moratz and Jan Oliver Wallgrün, 'Propagation of distance and orientation intervals', in *Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 3245–3250, (2003).

[24] D. A. Randell and A. G. Cohn, 'Modelling topological and metrical properties of physical processes', in *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning (KR'89)*, eds., R. J. Brachman, H. J. Levesque, and R. Reiter, pp. 357–368. Morgan

Kaufmann, (1989).
[25] David A. Randell, Zhan Cui, and Anthony Cohn, 'A spatial logic based on regions and connection', in *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, eds., Bernhard Nebel, Charles Rich, and William Swartout, 165–176, Morgan Kaufmann, San Mateo, California, (1992).
[26] R. Reiter, *Knowledge in Action*, MIT Press, 2001.
[27] Jochen Renz, 'A spatial odyssey of the interval algebra: 1. directed intervals', in *IJCAI*, pp. 51–56, (2001).
[28] Thomas Röfer, 'Route navigation using motion analysis', in *Cosit 1999*, eds., Christian Freksa and David M. Mark, pp. 21–36. Springer, Berlin, (1999).
[29] Christoph Schlieder, 'Reasoning about ordering', in *Spatial Information Theory*, pp. 341–349, (1995).
[30] Mikhail Soutchanski, 'An on-line decision-theoretic golog interpreter', in *IJCAI*, pp. 19–26, (2001).
[31] Thora Tenbrink, Kerstin Fischer, and Reinhard Moratz, 'Spatial strategies in linguistic human-robot communication', in *KI-Journal: Special Issue on Spatial Cognition*, ed., Christian Freksa, arenDTaP Verlag, (2002).
[32] R.F. Wang and E.S. Spelke, 'Updating egocentric representations in human navigation', *Cognition*, (2000).
[33] R.F. Wang and E.S. Spelke, 'Human spatial representation: Insights from animals', *Trends in Cognitive Sciences*, **6**(9), 376–382, (2002).
[34] Diedrich Wolter and Longin Jan Latecki, 'Shape matching for robot mapping', in *Proceedings of 8th Pacific Rim International Conference on Artificial Intelligence*, eds., Chengqi Zhang, Hans W. Guesgen, and Wai K. Yeap, Auckland, New Zealand, (August 2004).