Technik
Informatik & Medien
Hochschule Ulm
University of
Applied Sciences

University of Applied Sciences Ulm
Department of Computer Science
Master Course Information Systems

Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots

Master Thesis
by
Andreas Steck

November 14^{th}, 2010

Advisor: Prof. Dr. Christian Schlegel
Co-Advisor: Prof. Dr. Rüdiger Lunde

Declaration of Originality

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated.
I have only used the resources given in the list of references.

Ulm, November 14th 2010 Andreas Steck

Abstract

Autonomous mobile robots in general and service robots in particular are often required to operate in the same dynamically changing everyday environments as humans, sometimes even along with them. Such systems have to perceive their environment and react appropriately to the huge amount of changes and contingencies depending on situation and context. To achieve robustness in task execution, the goal is not to maximize efficiency but reliable and flexible execution of various tasks even in unknown and unpredictable environments. The skills, the components of the robotic system provide, have to be combined to form a variety of behaviors by setting the components into different configurations. Dynamically and flexibly managing the huge amount of different configurations is one of the major challenges tackled in this work. To further improve robustness in task execution and decision making at runtime symbolic and subsymbolic mechanisms have to be combined.

This is achieved by utilizing the *Three Layer Architecture*, where the sequencer mediates between the symbolic and subsymbolic mechanisms of the *skill layer* and the *deliberative layer*. The focus of this work is laid on the *sequencing layer*. Based on several use cases the requirements are derived a *sequencer* has to address. Especially, the situation and context dependent task execution and the handling and recovering from the huge amount of contingencies play a crucial role. This issue is addressed by dynamically generating a task net at runtime, where the task expansion depends on the current situation and can be modified at runtime. Plan steps can be removed or added at runtime to react on contingencies and to recover from failures.

The verification of the approach is achieved by a reference implementation of the utilized *Three Layer Architecture* focusing on the sequencing layer. The major contribution of this work to the overall architecture is the *Lisp*-based reference implementation of the *sequencing layer*, namely SMARTTCL. The system is able to perform various complex tasks including guiding the robot in a new environment and approaching the shown locations. Furthermore, mobile manipulation tasks, like cleaning up a table are performed.

Acknowledgements

It is my pleasure to thank everyone who directly or indirectly contributed to this work.

First of all, I gratefully thank my thesis adviser Prof. Christian Schlegel for the opportunity to work on service robots. He pointed me into the direction of *Three Layer Architectures* for Service Robots, which inspired me and provides a promising research direction. On the one hand, he helped me to avoid side-tracks and dead-end roads in my research work. On the other hand, he gave me the freedom to develop my own ideas, which was a driving factor in the success of this thesis.

I am also grateful to Prof. Rüdiger Lunde for his spontaneous agreement to act as co-adviser. In particular, I appreciate the preparedness to instantly read through a draft version of this thesis and providing some feedback.

Furthermore, I would like to express my thanks to my colleagues at ZAFH Servicerobotik at the University of Applied Sciences Ulm for many fruitful discussions. In particular, I want to thank Matthias Lutz for sharing several endless nights in the lab and the plenty fruitful discussions. Furthermore, I want to thank Alex Lotz for helping me to search and fix some bugs occurred by performing the real worls experiments. Finally, I want to thank everyone who was involved in developing SMART-SOFT components which I could use to perform my experiments.

Last but not least, I thank my girlfriend Lena for her patience in the last years, when I spend the nights and weekends either in the lab or at home with my laptop. Especially in the last month she relieved me of several of the everyday duties.

Contents

Chapter 1

Introduction

1.1 Introduction and Motivation

The development of *service robots* has gained more and more attention over the last years. In particular, the field of mobile manipulation raise the complexity. The execution of complex manipulation tasks, in unstructured and dynamic environments, require interdisciplinary scientific and engineering contributions which are currently beyond the state of the art in robotics. [2]

Nowadays, the algorithms and functionality of isolated and selected robots is developed enough to already build impressive robotic systems, but it is still challenging to integrate several such behaviors into one system. Today's robotic technologies support the development of complex systems with advanced perception, navigation and manipulation capabilities, but their effectiveness has been demonstrated in simple scenarios only (i.e. complex algorithms but low overall interaction of system components).

Potential applications require a robot to operate in the same environment as humans. In many situations the robots have even to work along with humans and to interact with them. Such systems have to be able to perceive their environment and to react appropriately to changes. The major goal is, that robots have to operate reliable and flexible in the highly dynamic everyday environments over long periods of time. Therefore the situation dependent usage of individual skills plays a crucial role. The robot control architecture should provide the flexibility, that the system can adapt itself to the current situation. Several decisions may require to consult simulators or analysis tools to check whether the desired configuration and parametrization of the skills is valid and reasonable.

In reference to the best available knowledge the robot has to decide at runtime for the most appropriate behavior to execute, to fulfill the demanded task. Otherwise the robot should state that it is not able to handle the current situation. This situation-driven task execution requires the integration of symbolic and subsymbolic mechanisms of information processing. Subsymbolic mechanisms ensure flexibility and reactivity since they typically allow short cycle times. But they are not able to look into the future and to reason about how to achieve higher-level goals. Symbolic mechanisms ensure goal oriented activities and are able to reason about the steps which are necessary to achieve higher-level goals, but typically require time-consuming calculations and a complete world model.

The state-of-the-art architecture to integrate symbolic and subsymbolic mechanisms is the *Three Layer Architecture* [15]. The concept behind the instantiation of the *Three Layer Architecture* used in this work is described in [37] [40] [42]. The lowest so-called *skill layer* comprises components mainly operating on the level of sensors an actuators. These components typically provide services for map building, path planning and collision free motion control. The medium *sequencing layer*,

which is the focus of this work, is responsible for the situation-driven task execution. Therefore the *sequencer* performs dynamic online reconfiguration of the components of the underlying *skill layer*. Finally, the uppermost *deliberative layer* processes time-consuming algorithms, like symbolic task planning, simulations (e.g. physics simulators) and system analysis (e.g. realtime schedulability analysis, performance analysis).

Furthermore a novel instantiation of the *Three Layer Architecture* needs to involve the novel design process for robotic system engineering described in [41]. This enables the development of complete robotic systems that are more complex and more robust than the robotic systems that can be developed with current technologies. In that approach both, the system engineer and the robot reason about the problem and the solution spaces. Instead of trying to find all-time optimal solutions at design-time the focus moves to making the best possible solution at run-time. Therefore, the robot developer has the ability to formally model and relate the different views of the robotic system design and reasons about their correctness by means of offline simulation techniques. At run-time the robot has the ability to reconfigure its internal structure and to adapt itself according to its understanding of the current situation.

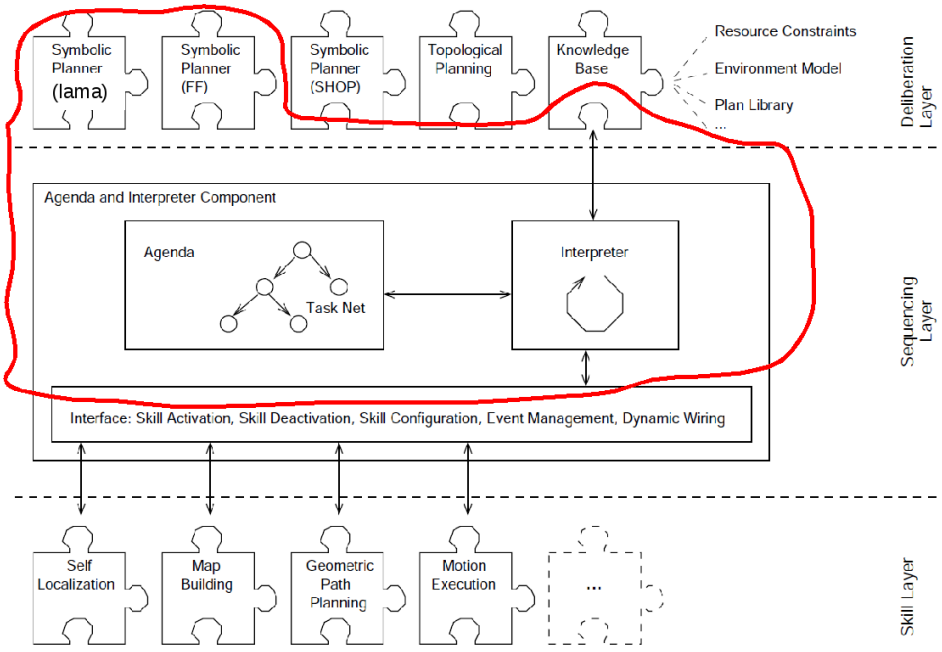

Figure 1.1: Three Layer Architecture. The red border marks the part this thesis contributes to.

The major contribution of this thesis is a novel instantiation of the *Three Layer Architecture* (fig. 1.1), focusing on the *sequencing layer*. Especially the situation-driven execution and the recovery from a variety of contingencies play a crucial role such a *sequencing layer* has to provide to ensure robust and reliable task execution over long periods of time. The focus of this thesis is laid on the concept and definition of a task coordination language[1] to implement the *sequencing layer*. This language is named SMARTTCL (*Smart Task Coordination Language*) and has to be seamless integrated in the *Three Layer Architecture*. It should provide easy to use and precisely defined interfaces towards

[1] In the literature, task coordination languages are often refered as execution languages. In this thesis these terms are used as synonyms.

the other layers. The at design-time left open variation point have to be bound at run-time. For this purpose, simulators, analysis tools, as well as symbolic task planners have to be integrated in the overall system.

1.2 Thesis Outline and Contributions

This thesis is concerned with aspects of instantiating a *Three layer Architecture* for autonomous mobile service robots. The major focus is laid on the *sequencing layer*. Furthermore, several aspects on how to interface to the *skill layer* and the *deliberative layer* are given.

Chapter 2: describes several use cases which are covering a range of different behaviors and situations that are of relevance for service robots.

Chapter 3: discusses related work with respect to control mobile service robots.

Chapter 4: presents the major contributions of this thesis. Out of the use cases described in chapter 2 several requirements are derived. The basic design decisions are motivated and compared to other approaches. The concept of the *sequencing layer* is presented.

Chapter 5: describes the *Lisp* based reference implementation of the *sequencing layer*, which is called SMARTTCL.

Chapter 6: describes the experiments performed with the instantiation of the *Three Layer Architecture* using SMARTTCL as the *sequencing layer*. The example scenarios cover the use cases and analysed requirements. The results of this thesis are summarized and discussed.

Chapter 7: draws a conclusion and presents future work.

Appendix A: illustrates the source files of the guiding tour scenario.

Chapter 2

Use Cases

This section describes several use-cases which are covering a range of different behaviors and situations that are of relevance for service robots. Each use case description contains a summary of the most important challenges occurring in the use case. Based on these challenges the requirements of this work are derived in section 4.1.

2.1 Navigation in Everyday Environments

A basic behavior each service robot has to manage is the navigation in everyday environments. Especially, the coordination and configuration of the skills which are necessary to approach arbitrarily given positions, cover several aspects an execution language (*sequencing layer*) has to support.

2.1.1 Navigation Basics

An example of the involved skill components is illustrated in fig 2.1. The deployment is specific to the applied robot, but can easily be adapted to any mobile robot platform. A higher-level view on each of the components is given below.

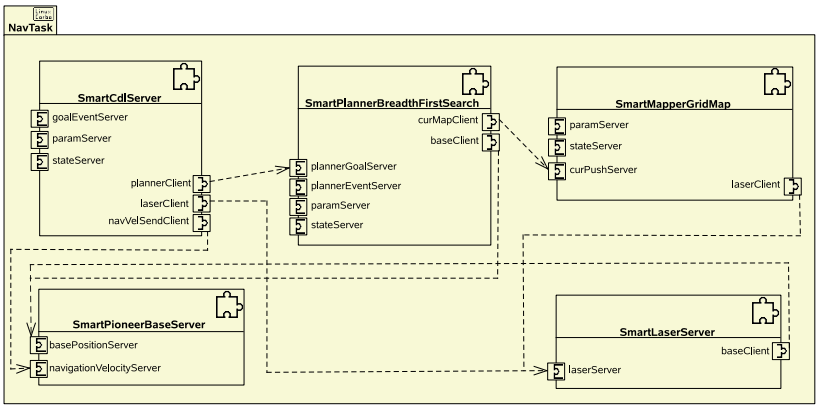

Figure 2.1: Navigation components: *Skill* components required for goal-directed navigation.

The *SmartPioneerBaseServer* abstracts the hardware of the mobile robot platform. It provides the position information of the robot and requires velocity commands (v,w) to control the robot.

The *SmartLaserServer* provides cyclic laser scans of the environment. These laser scans are, for example, used for obstacle avoidance, mapping and person following.

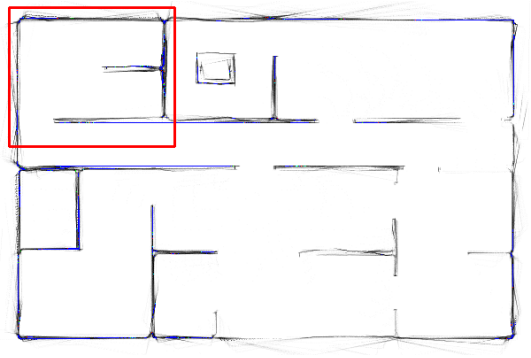

Figure 2.2: *Longterm map* showing a simple office building with a *current map* excerpt illustrated as red rectangle.

The *SmartMapperGridMap* component is based on gridmaps generated out of raw laser data and the robot's position information. The mapper builds a *longterm map* and a *current map* (fig. 2.2). The *longterm map* is a weighted occupancy gridmap which represents the robots environment perceived over a long period of time. The *current map* is a binary gridmap which acts as a short term memory. The preoccupation of the *current map* is usually generated out of the *longterm map*.

Using the *current map*, the *SmartPlannerBreadthFirstSearch* generates a path on that the robot can approach a goal region. The path is represented by intermediate waypoints and the goal waypoint. The *SmartPlannerBreadthFirstSearch* component comprises an event port to announce contingencies like, no path to the specified goal could be found, the specified goal is covered by obstacles or the goal has been reached by the robot. The waypoints calculated by the planner serve as goals for the *SmartCdlServer* component.

Finally, the *SmartCdlServer* component, which is based on the *CDL* [36] algorithm provides local motion planning. *CDL* is an improvement of the dynamic window approach. It considers the dynamics and kinematics of the robot, as well as its polygonal shape. It consumes raw laser scans or other sensor perceptions transformed into occupancy grids. The basic idea is that a robot moves along different curvatures (v,w combinations) which represent trajectories built up by circular arcs. The huge number of possible v,w combinations is reduced based on the observation that only a few curvatures are safely selectable given the current state and kinematics of the robot. High performance advantages are achieved by precalculating lookup tables. The final selection along the remaining admissible v,w combinations is done by an objective function which trades off speed, goaldirectedness and remaining distance until collision. Depending on the use case, different objective functions are applied. Thus, the *CDL* component can be used in an arbitrary number of scenarios. The component is, for example, used for approaching the waypoints send by a path planner and following a person.

To form navigation behaviors out of the above described skill components, the *sequencer* mainly orchestrates the components *SmartMapperGridMap*, *SmartPlannerBreadthFirstSearch* and *SmartCdlServer*.

2.1.2 Approaching a Position

To approach a desired position the *sequencer* has to set the above described components into an appropriate configuration. Furthermore, the goals have to be send to the path planning component. The communication between the mapper, path planner and cdl component are hidden from the *sequencing layer* to ensure high reactivity. Feedback from the *skill layer* to the *sequencer* is given by sending events. The path planner, for example, sends events to indicate that a goal has been reached or to indicate that any problem has occurred.

Depending on the event, different kinds of reaction are appropriate. An event that indicates, that no path could be found, for example, is typically handled by deleting the *current map* and preoccupying it with the cells of the *longterm map*. The configuration of the other components should not be affected by that reaction to the event. An event indicating that the goal has been reached, can be handled by two different strategies, depending on the configuration and purpose of the behavior. The first one is to stop the robot including to deactivate the mapper, path planner and cdl component. This deactivation is essential to save resources. For example, if the robot has to perform resource intensive mobile manipulation tasks at the desired goal location. The latter one is applied whether the goal is an intermediate goal and the robot has to continue driving without stopping. In such a situation the cdl component is switched into a strategy to drive reactive while avoiding obstacles. While the robot is continuing driving, the configuration of the new goal takes place. Afterwards, the cdl component is switched back to the strategy to approach the waypoints send by the path planner component.

Furthermore there is a distinction between driving to a goal region and approaching a goal point exactly. The first one is typically performed using a gridmap based path-planner which discretizes the goal according to the grid size of the map (e.g. 5cm). The latter is used to approach a goal more exactly and can only be used if no path-planning is necessary to reach the goal. In that case the goal is directly send to the *CDL* component. In both cases the *CDL* component is used to perform the local motion planning. But depending on the desired behavior the component is used in different strategies and with a different parametrization. Usually both strategies are combined, the first one to approach the goal region approximately by using the path-planner and the latter one to approach the goal position exactly.

2.1.3 Switching between Maps

The above described behavior, to continue driving while modifying the configuration of the mapper and path planner components, is especially essential to switch the robot between different maps (fig. 2.3). Typically the path planning is performed in the *current map*, which is an excerpt of the much larger *longterm map* (cf. fig. 2.2). The two reasons for that are, first to save resources by performing the planning algorithm in a much smaller search space. And second, to direct the wavefront of the path planner into a specific direction. This map building and path planning takes place at a discrete grid-based level. If the operating range of the robot gets larger there is a need for a topological representation, which is for example covered by different maps organized in a graph representation.

The dimensions of the *current maps* can be configured arbitrarily, but it is recommended to overlap the maps at least by a few meters. Often the *current maps* are mapped onto rooms in the real world. This ensures well-defined borders (walls) and crossing regions between the maps/rooms (doors). However, working with excerpts of a map raise the demand to switch between the maps – without stopping the robot and waiting until the new configuration is set correctly.

To switch between two *current maps*, the crossing regions to switch from the one map to the other are set as goals. These regions are typically available in both *current map* representations. If the robot

Figure 2.3: Switching between maps. The *current map* is marked with a green rectangle. The red smiley markes the robot; the yellow star marks the final goal region. Intermediate goals to switch between maps are marked with blue circles. Around the intermediate goal circles the wavefront algorithm used by the path planner is illustrated.

has reached the region, it is reconfigured to drive reactive. The mapper is deactivated, that the current map configuration can be modified. After changing the dimension of the *current map*, the mapper component is activated again and the new goal region is send to the path planner. To ensure that the current goal regions, the *current map* and the waypoint belong all to the same configuration, IDs are used for synchronisation. Furthermore, the path planner sends an event once it is synchronized correctly with the *current map* and is thus ready to send the waypoints to the *CDL* component. This event triggers to set the *CDL* component into the strategy to approach the waypoints. Again, in the timespan the configuration of the mapper and path planner components take place, the robot is driving reactive. As the timespan for the reconfiguration of the components is quiet small, the result is a goal-directed smooth motion of the robot while switching the maps.

So far, it is assumed that the maps overlap and have common crossing regions. But already the example depicted in figure 2.3 illustrates that it might be necessary to switch between several maps to reach first, the correct *current map* and afterwards the final goal region inside this *current map*. In such situations it would be necessary that the *sequencer* knows about all combinations of map switches to be able to drive to every desired goal independent of the current position of the robot. However, the different combinations grow with the number of maps and it will be almost impossible to encode and store them in advance.

A more adequate solution is it to call a symbolic task planner, providing the information in which map the robot is currently located, in which map the final goal region is located and which maps with their crossing regions are available. The symbolic task planner will generate an ordered sequence in that the maps have to be switched.

Summary of Challenges

Online reconfiguration of the Components: The different components have to be parametrized and activated/deactivated at runtime.

Handling Events: Events are used by the components to send any kind of feedback to the sequencing layer. These events have to be handled avoiding unwanted side effects.

Handling Contingencies: Different contingencies which occur during execution have to be handled to ensure robustness.

Situation-Driven Execution: The selection for the next steps to execute has to depend on the current situation and context. (configuration of the components)

Integration of a Symbolic Task Planner: To plan the correct ordering in that the different maps have to be switched to approach a desired location in the corresponding map.

Integration of a Knowledge Base: For example, the knowledge about the different *current map* properties and the mappings between the location names (“kitchen”, “dinner table”, “shelf”) and the position in world coordinates have to be stored and have to be easily accessible in a consistent way.

2.2 Handling User-Interaction

Interacting with persons, for example by speech or gestures, is another important behavior a service robot should support (fig. 2.4). This provides a simple and intuitive interface for humans to command the robot and to set the robot different tasks. Furthermore, it is also an intuitive interface for the robot to communicate with its users. Typical components supporting speech interaction are illustrated in fig. 2.4. The *SmartSpeechLoquendoOutputServer* provides an interface based on a *string*, which is send to the component and spoken by the speech synthesis system. The *SmartSpeechLoquendoInputServer* provides a state port to activate and deactivate the speech recognition engine. Deactivating this component is, for example, reasonable if the robot speaks a text, which should not be recognized by the robot itself. Another reason for deactivating speech is to save resources in situations where it is obvious that no speech interactions will occur. User interaction depends typically highly on the current situation and context. Usually, speech recognition engines have to be parametrized according to the words and sentences they can process using a grammar specification. Such a specification has to be restricted to the current situation. Only the vocabulary which is supposed to be spoken by the users should be included in the currently activated grammar specification. Not restricting the grammar will lead to large vocabularies, resulting in a significantly reduced speech recognition performance. Those grammar specifications do not only contain the pure vocabulary, they additionally contain the mappings between the phrases and the associated semantics. The semantics are used in the *sequencing layer*, as semantic is more general than the spoken text. The semantic expression “approach kitchen-table” corresponds, for example, to the spoken text “Go to the kitchen table”, “Please, go to the kitchen table”, “Go to the kitchen table, please”, “Approach the kitchen table” and so on.

The *SmartSpeechLoquendoInputServer* provides an event port to signal whether a desired phrase was recognized. Events are activated to only fire whether the desired semantic was recognized by the speech recognition engine. They contain the spoken text (as string) which was recognized, the associated semantic and the confidence as a quality measure of the recognition.

User interaction is mostly parallel to other activities the robot is performing. For example, the robot should be able to react to speech commands like “stop following” while it is following a person. Furthermore, the person following task is generally combined with other tasks, like memorizing locations. This is, for example, used to guide the robot through its new environment and to show it different locations. In that case the robot has to be able to cope with the speech interaction needed for person following (“stop following”, “wait”, “follow me”) as well as with the commands to memorize the locations (“This is the kitchen”, “The shelf is in front of you”, “The trash bin is on your left”). This

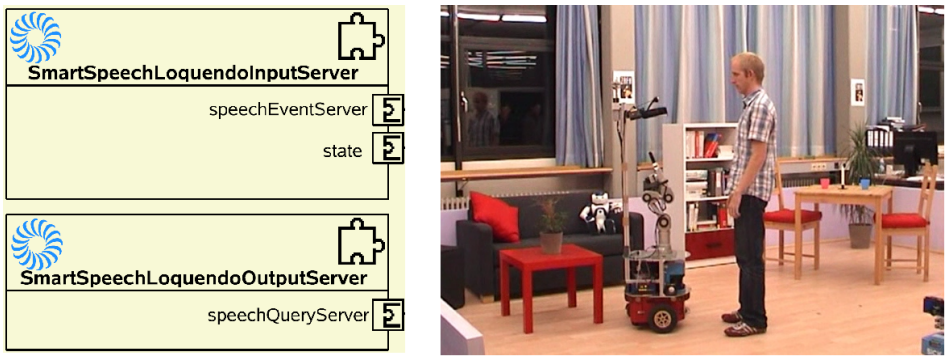

Figure 2.4: *Left:* Typical components for speech interaction. *Right:* The robot *Kate* interacts with a person by speech.

parallel and situation dependent usage of speech events raise the complexity an execution language has to be able to cope with.

Summary of Challenges

Online reconfiguration of the Components: Depending on the situation and context the parametrization of the speech components, for example, has to be modified.

Handling of Events: Handling multiple activations of the same event (e.g. speech-event) with different activations (parametrization) within different tasks.

Situation-Driven Execution: The execution and selection of the next task to execute highly depends on the user interaction (event).

2.3 Mobile Manipulation

Service robots managing domestic work, for example, have do be able to perform mobile manipulation. The mostly applied and common form of mobile manipulation behaviors are variations of the so-called pick-and-place tasks. This includes tasks like cleaning up a table (fig. 2.5) or fetching goods and bringing them to the operator. More complex manipulation behaviors are, for example, preparing a meal or doing the dishes. For all manipulation tasks it is of importance that the robot is able to detect and recognize objects. It should have the necessary information about the object to be able to handle them appropriately. An empty beverage can, for example, can be thrown into the trash bin, but empty cups should be taken to the kitchen sink or into a dishwasher. Several goods, like coke or beer are commonly stored in the refrigerator, whereas cups and plates are usually stored in a cupboard.

Mobile manipulation tasks often require complex behaviors and appropriate reactions to the huge amount of different contingencies. Complex tasks involve several steps until the goal is reached. For example, the command, that the robot should fetch a coke from the kitchen, bring it to the living room and serve it to a specific person has a huge potential for failures that endanger the success of the whole task. Depending on the failure, the task becomes either unsolvable or the contingency can be handled. Contingencies which can happen are, for example, that the path to the kitchen can not be found, there is no more coke in the kitchen or the person the coke should be delivered to can not be found. In case

Figure 2.5: The robot *Kate* cleans up a table. It stacks the cups into each other.

there is no more coke in the kitchen the robot might find some in the storage room, or it can drive to the living room and ask the person to order another drink. These contingencies are specific to the situation and the robot should be able to handle them.

Furthermore, while handling a delivering task as described above an event might occur which is of higher priority, like a ringing door-bell. In such a situation it is more important to open the door as to serve the coke. But after opening the door and receiving the guest the robot is in a completely different situation (not only different location). Furthermore, the abortion/interruption of the currently activated delivering task is not straightforward. The manipulator might be in a grasping configuration and has to be set into a configuration which is appropriate for driving. The situation is even more difficult if the robot has already grasped the coke, as it will need the gripper to open the door.

As the tasks are getting more complex it will not be possible to encode all behaviors and task expansions beforehand for all situations and world states. An example scenario, which is an extension of the already available cleanup table scenario [54]. The scenario comprises the three different objects (i) cups, (ii) beverage cans and (iii) chips cans. The objects are constraint, that two cups can be stacked into each other and two beverage cans can be stacked into one chips can. Furthermore, the beverage and chips cans are garbage if empty and should be taken to the trash bin, while the cups have to be taken to the kitchen sink to be washed. Different amounts of each object can be placed on the table. The robot has now to choose for the best, or at least a reasonable, way how to clean the table up. Therefore, a symbolic task planner can be considered giving it the objects and associated constraints. More details on this scenario are described in section 6.2.

Summary of Challenges

Online reconfiguration of the Components: The different components have to be parametrized and activated/deactivated at runtime.

Handling Events: Appropriate handling of the different events which are send from the skill components to the *sequencer*.

Handling Contingencies: The huge amount of contingencies has to be managed.

Situation-Driven Execution: The selection for the next steps to execute has to depend on the current situation and context.

Integration of a Symbolic Task Planner: To generate higher-level plans a symbolic task planner has to be considered.

Integration of a Knowledge Base: The knowledge about different locations, objects, persons, orders and goals has to be managed.

2.4 Composition of new Behaviors out of existing ones

New scenarios have to be composed out of existing scenarios or parts of them. Recurring behaviors should not be "reinvented" for every scenario, instead approved solution which provide flexibility, reasonable levels of abstraction and information hiding, have to be reused. Internal structures like task decomposition, events and housekeeping activities have to be hidden from the developer which reuses a behavior block.

A "new" behavior like *guiding the robot around*, for example, requires to compose the "existing" behaviors *follow person* and *memorize location* in a way that they run parallel. It should be able to tell the locations to the robot while it is following.

Furthermore, the developers of behaviors should be supported by tools. This includes, for example, debugging and monitoring the task execution. Which behavior was chosen in what situation and why.

Summary of Challenges

Parallel behaviors: Behaviors should be composable in a parallel way.

Reusability: Reusability of behaviors without detailed knowledge about internal structures.

Debugging/ Monitoring: Monitoring of the task execution and decision making process.

Verification/ Analysis: Verification and analysis of the behavior blocks.

2.5 Robustness by Online-Adaptation through Simulation and Analysis

The huge amount of different situations and contingencies is not known beforehand during the development of the robot. Several parameters are unknown during the design-time and have to be bound at runtime. These parameters can, for example, be bound by considering simulation tools at runtime. Using a physics simulator, for example, the maximum allowed velocities of the robot taking the current payload into account can be reasoned. As not all different situations are known at design-time, not all possible configurations of the robot system can be analysed beforehand. To achieve robustness, the analysis has to be performed at runtime before setting the configuration. Therefore, again simulators and analysis tools have to be considered. For the different configurations it can, for example, be chacked whether the resources (e.g. computational power, bandwidth of communication buses, ...) are available.

Summary of Challenges

Integration of simulators and analysis tools: To further improve the decision making of the robot simulators and analysis tools have to be considered at runtime.

Chapter 3

Related Work

This chapter presents work related to this thesis. At first, *Three Layer Architecture* is motivated. Afterwards, approaches focusing on robot control are discussed. Finally, task coordination languages, focusing on situation and context dependent execution are presented. These approaches are of special interest for this work and those having mostly influenced the design of SMARTTCL are further analysed in chapter 4.

3.1 Architecture

An architecture describes how the overall robot system is constructed from its components and how the components fit together to form the whole. In this section, the term architecture refers to the arrangement of the control mechanisms of a robot and how the components are organized in different layers. A very well overview on robot architectures is given in [21].

In [52] [53] a two layer architecture is described that integrates a state-of-the-art symbolic planner with a library of robot control programs. A hierarchical task network planner (HTN) is used to directly interact with the robot skills. Robot skills are implemented in control programs that directly access the sensors and actuators. To achieve high reusability the control programs are basic, which results in an increased complexity for the task planner. Complex robot tasks are accomplished by sequential execution of less complex actions that are triggered and configured by the *JShop2* [27] task planner. The *JShop2* planner performs task-decomposition according to hierarchical task-nets. The planner generates plans based on the current world state. The plan consists of skills represented in the skill library. The physical execution of a plan is done by control programs which are invoked. Executed control programs access and modify the current world state in symbolic terms as well as continuous representations. Several experiments performed by the authors of the paper [52] showed that failures occurred due to an inconsistent representation of the world model. The control program changes the world state and causes replanning with the updated state. To overcome the closed world assumption, which is necessary for HTN planning, they use special skills causing replanning explicitly. Aborting a plan and taking time for generating a new plan can take a lot of time. Procedural and deductive execution systems have complementary strength and weakness and are directly combined. [52] [53]

In contradiction to the above described approach, in this work a *Three Layer Architecture* [15] is used. The distinguishing feature of a three-layer architecture is that one can control real robots performing complex tasks even with a trivial deliberative layer and even with skills that cannot handle all situations. The sequencing component provides the necessary glue logic and is the place to store procedural knowledge that neither fits at the deliberative nor at the skill layer. "[...] a plan, generated

by most any current planner, still requires the help of an execution system to be useful for real-world execution." [51]. The *sequencer* bridges the gap between symbolic and subsymbolic representations.

An introduction into *Three Layer Architectures* can be found in [15]. The instantiations *Animate Agent Architecture* [10] [11], *3T* [7], *ATLANTIS* [12] and the instantiation based on SMARTSOFT [37] [40] [42] influenced this work. The different flavours of the instantiations are discussed in chapter 4. They all share the basic concept to divide the system complexity in the three layers of abstraction, namely the *skill layer*, *sequencing layer* and the *deliberative layer*. The major distinction is whether the goals are specified at the *deliberative layer* and then refined in a top-down manner, resulting in giving the overall control to the symbolic planner. Or specifing the goal at the *sequencing layer* which then orchestrates the two other layers and mediates between them.

3.2 Robot Control Mechanisms

Several Contributions exist, that focus on controlling robotic systems.

URBI

URBI (Universal Robotic Body Interface) [6], for example, includes a scripting language for controlling the low-level layer of a robot. It is based on a client/server architecture. The server is running on the robot and the client is sending commands to the server. Commands can be written directly in a telnet client or issued by a program using the *liburbi* library. *URBI* provides control constructs like `for`, `while`, `if then else` and `loop`. For event catching, control structures like `whenever`, `at` and `wait` are provided. *URBI* is, for example, used to demonstrate simple action/perception loops on *Aibo* [4], like walk pattern generation. The example depicted below shows a "Ball Tracking Head" behavior [6].

```
whenever (camera.ballx != -1) {
  headPan.val = headPan.val +
    camera.xfov * (0.5 - camera.ballx) &
  headTilt.val = headTilt.val +
    camera.yfov * (0.5 - camera.bally)
};

at & (camera.ballx == -1 ~ 500ms)
  scan : {
    headPan.valn = 0.5 sin:4000 ampli:0.5 &
    headTilt.valn= 0.5 cos:4000 ampli:0.5
  };

at (camera.ballx != -1) stop scan;
```

The focus of *URBI* is on controlling the joints of the robot or to access its sensors. It is supposed to be used together with other languages which do the cognitive part of the robot behavior.

Hybrid State Machines

In [9] [28] *hybrid state machines (HSM)* to program the robots behavior are proposed. [28] provides an implementation which is based on *Lua* [23]. It is proposed to bridge the gap between high-level strategic decision making and low-level actuator control. Examples demonstrate motion patterns like the standup skill with the humanoid standard platform *NAO* [26].

*S*tate *MACH*ine (*SMACH*)

SMACH [47] aims to rapidly compose complex robot behaviors out of primitive ones. It is based on the concept of hierarchical concurrent state machines. *SMACH* is implemented as a language extension of *Python* and integrated in *ROS* [32]. It is, for example, used in the *PR2* robot at *WillowGarage* as task coordinator for scenarios like playing pool, cleaning up the table with a cart and fetching beer from the refrigerator. However, the *SMACH* behavior of each scenario is developed from scratch, with almost no reuse. *SMACH* supports static composition of behaviors with little capabilities for situation dependent task execution.

3.3 Situation-Driven Task Coordination

The above mentioned approaches are not suited to be used as *sequencer* in a *Three Layer Architecture*. They provide almost no means to perform online reasoning and to interface with a *deliberative layer* as supported by SMARTTCL.

A comparison of languages which have already been used in *Three Layer Architectures* can be found in [22] [51]. These languages address aspects that are up-to-date but had been developed with limited robot platforms, capabilities and scenarios. The power of those concepts can now be exploited as robot platforms with advanced skills are available. The progress in robotics allows to proceed the development of those ideas. *ESL*, *RAP* and *SimpleAgenda* have strongly influenced the design of SMARTTCL. Specific aspects of these languages are analysed and compared in detail in chapter 4. In the following some languages are presented.

ESL

ESL (Execution Support Language) [13] is based on a set of macros that expand into *Lisp*. To support parallel activities the *Lisp multi-tasking* library is used. The *knowledge base* is a *Prolog*-based database also implemented in *Lisp*. *ESL* was designed to respond quickly to events while bringing together potentially large quantities of information to bear on its decisions. Contingency-handling is based on the concept of *cognizant failures* [29]. Failures are signaled when they occur and recovery procedures are used to recover from the failure. Recovery procedures are not bound to tasks. Conditional task execution is supported by alternative methods with conditions describing under which condition those methods are appropriate. Task synchronization is done by a data object called event. *ESL* tasks are inherited from events and thus tasks can also wait for other tasks to finish. They have a linear execution thread. The bodies of a task-net run all in parallel. The task-net itself blocks, until all children have finished. `ALLOW-FAILURES` will abort all sub-tasks if one of them fails. `OR-PARALLEL` finishes the sub-task if one of them finishes successfully or all of them fail. *ESL* provides a mechanism for solving a constrained version of controlling inter-task conflicts. Property locks are used to coordinate tasks so that they do not try to achieve different values for a single property at the same time. The example[1] depicted below gives an impression how *ESL* programs look like: [14]

```
(defvar *widget-status* :ok)

(defun operate-widget ()
  (if (eq *widget-status* :ok)
    (format t "~&OPERATING WIDGET SUCCESSFULLY.")
    (fail :widget-broken *widget-status*)))
```

[1] The example is part of the ESL user's guide [14]

```
(defun break-widget (&optional (state :broken))
  (setf *widget-status* state))

(defun attempt-widget-fix (from-state)
  (if (eq *widget-status* from-state)
    (setf *widget-status* :ok))
  (if (eq *widget-status* :ok)
    (format t "~&Widget is fixed.")
    (format t "~&Widget fix didn't work.")))

(defmacro esl-demo (&body body)
  `(progn
     ,@(mappend
        (fn (form)
          `((format t "~&~%Evaluating: ~S" ',form)
            (let ( (result (catch :abort ,form)) )
              (format t "~&~S returned ~S" ',form result))))
        body)
     (values)))
```

The example operates on a *WIDGET*, which responds to the following operations. `(operate-widget)` to print a message if the *WIDGET* status is `:OK`, otherwise `fail`. `(break-widget &optional (state :broken))` to force the widget into state `STATE`. (attempt-widget-fix from-state) to make the widget state `:OK` if and only if the current *WIDGET* state is `FROM-STATE`. [14]

Further aspects of *ESL* are analysed in chapter 4, how they influenced this thesis.

RAP

The *RAP* [10] system is designed to carrying out sketchy plans generated by planners. *RAP* uses a *Lisp*-based interpreter to manage a task-net and to interface to the *skill-level*. It takes task goals and breaks them down into steps that can be achieved by activating a set of skills. The final task selection is performed at runtime. It provides concepts for synchronization with the real world and to react to contingencies. Unfortunately, actions can just be executed in the leafs of the task tree. The success or failure of *RAP*s depend on the state of the *RAP World Model*. A *RAP* enables the appropriate skill and waits for it to generate a signal that indicates the success or failure of the skill (Atomic Actions). This tight integration severally restricts the usage of the *RAP*s, as each change in the world has to be synchronized with the world model. The following *RAP* example describes a task to place the pan-tilt unit to a specific position. [10]

```
(define-rap (primitive-pan-to ?angle)
  (succeed (and(head-pan-angle ?b)
                (within (- ?angle ?b) 0.02)))
  (method
    (primitive
      (enable (pan-to ?angle))
      (wait-for (:pan-at ?a) :SUCCEED (head-pan-angle ?a))
      (wait-for (:pan-stuck ?a) :FAIL (head-pan-angle ?a))
      (disable :above))))
```

If the current `head-pan-angle` of the pan-tilt unit in the *RAP World Model* is within 0.02 radians of the desired angle the task is complete and succeeds. If the current `head-pan-angle` cannot satisfy the *succeed* test, the method enables the `pan-to` skill and waits for it to generate a signal. In case of a `:pan-at` signal the method succeeds. The method also waits for the `:pan-stuck` signal

for the case the pan-tilt unit cannot move. In either case, when signals arrive, the method disables the `pan-to` skill it enabled and finishes. The final argument in the `wait-for` statement specifies a change to make in the *RAP World Model*. [10]

Several experiments are performed with the *RAP* system by the author of this thesis to gain experience with *sequencing layers*. Several of the limitations and problems occurred are described in chapter 4 and are taken into account for the design of SMARTTCL.

SimpleAgenda

The *SimpleAgenda* [34] provides simple task execution mechanisms. The *actions* are encoded in a construct called *operator*. The *operators* are managed within a stack in the *task*. The tasks are executed one after another. The operators are stored in the *KB* and instantiated at runtime. An operator is selected from the *KB* according to the match of the name and input variables specified in the task stack against the entries in the *KB*. The *SimpleAgenda* does not provide preconditions. Contingency handling is performed with rules. In case an operator execution fails the return value is matched against the rules stored in the *KB*. The rules can, for example, manipulate the stack of the task to recover from the contingency. Child tasks can be removed or added. The *SimpleAgenda* does not provide concurrency. It is implemented in *Lisp*. The *KB* which is used is the *SimpleKB* [35] which is also implemented in *Lisp*.

The example illustrated in figure 3.1 shows an execution trace of the person-following scenario. The steps of the example execution trace are described in the following:

1. The task execution is started with the three operators `op-init`, `op-followMe` and `op-exit` on the stack. Furthermore, three *rules* are assigned to the task.
2. The first operator is taken from the stack and executed. The initial configuration of the components is performed and the text that the robot is ready to follow a person is announced.
3. The next operator is responsible to wait for an event from the speech recognition component.
4. The event “follow-me” is fired by the speech recognition, implying to finish the operator with the return value `FOLLOWME`. That triggers the corresponding rule to set the robot into the configuration for person following and activating it (*skill-robot-move*). Furthermore, the operator `op-followMe` is pushed on the stack by the *rule* to further wait for events from the speech recognition component.
5. Intermediate state showing that the `op-followMe` is on top of the stack and will be executed next.
6. The operator `op-followMe` is executed and is again waiting for an event.
7. The event “stop” is fired by the speech recognition, implying that the operator finishes with the return value `STOP`. That triggers the corresponding *rule* to stop the robot. This *rule* does not push any operator on the stack.
8. The operator `op-exit` is executed to shut the robot down and announce the text “bye bye”.

Similar to the *RAP* system several experiments (including the mentioned one) are performed with the *SimpleAgenda* by the author of this work. The *SimpleAgenda* strongly influenced the design of `SmartTCL`. For example, the *SimpleKB* is used in the reference implementation of `SmartTCL`.

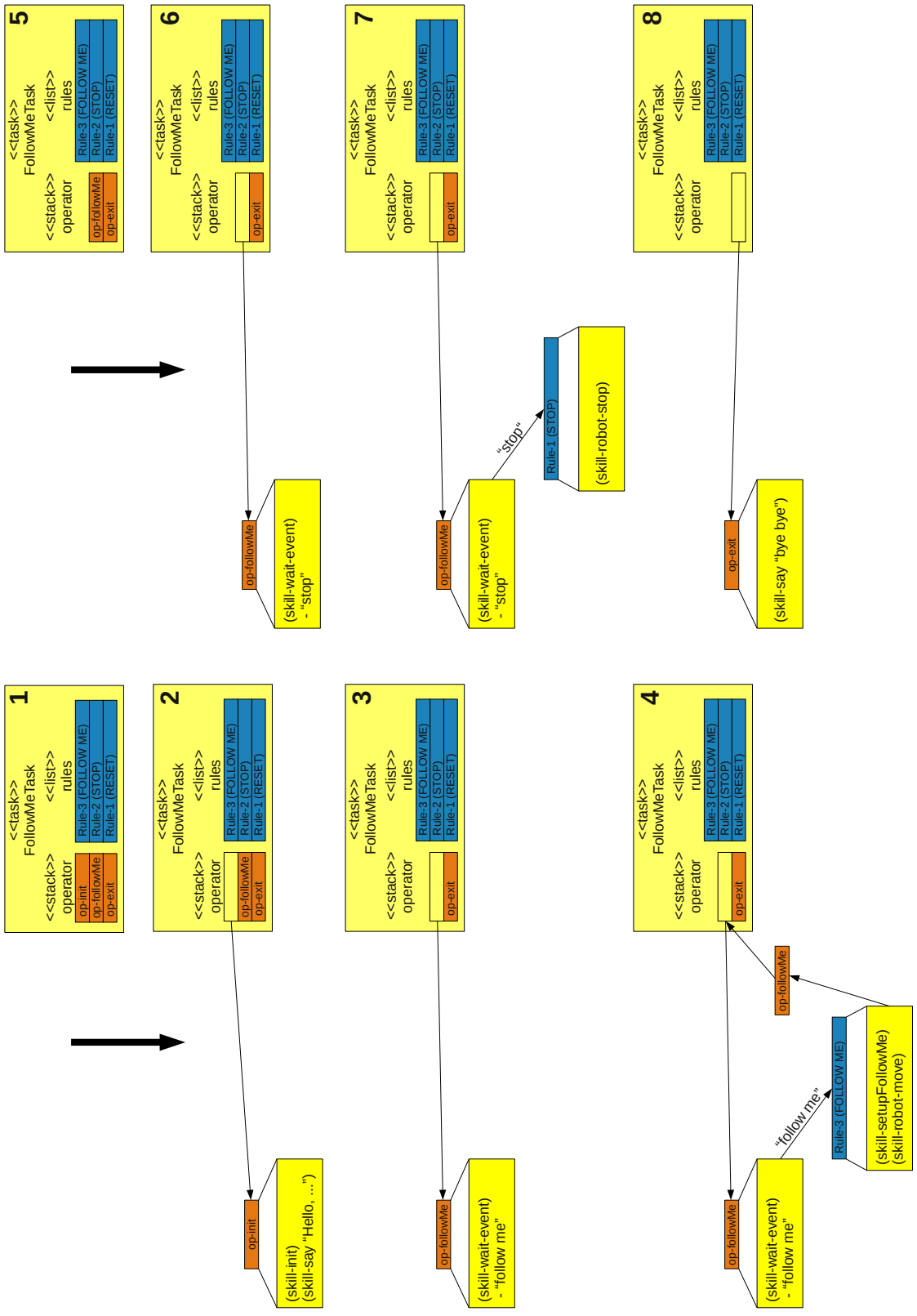

Figure 3.1: SimpleAgenda example

RPL

RPL [24] is a predecessor of *RAP*. Its syntax is more in the style of *Lisp*. It is implemented as a macro extension of *Lisp*. A plan consists of procedure calls glued together with syntactic constructs like `SEQ / IF`. For the execution of *RPL* plans a stack is used. The *RPL* interpreter does not attempt to maintain a world model that tracks the situation outside the robot. *RPL* supports (`true`) concurrency. Synchronisation is done by so-called valves. Tasks which should not run simultaneously compete for a valve. The winner is allowed to run, while the loser waits for the winner to finish and release the valve. Variables are lexically scoped (scope of an identifier limited to a block of source code), but global *Lisp* variables can also be used. The basic control concept of *RPL* is similar to task scheduling in operating systems. [24]

TDL

TDL [45] supports task decomposition, fine-grained synchronization of sub-tasks, execution monitoring and exception handling. *TDL* code is transformed into *C++* code that invokes the *Task Control Management (TCM)*. *TCM* is a reimplementation of the *TCA* task-level control mechanisms. In *TCM* there is no central server, instead it is a library that is linked into user code. [43] [44] [46]

TDL programs operate by creating and executing task trees. Each task tree node has an action associated with it. An action is a parametrized piece of code. It can contain arbitrary *C++* code, with certain restrictions. It can perform computations, dynamically add child nodes to the task tree[2] or perform some physical action in the world. Task trees have a parent/child relationship and synchronisation constraints. *TDL* provides two type of nodes: goal nodes are used to expand the task tree and represent higher-level tasks. The associated actions typically add children to that node. Command nodes are the leaves of the task tree and contain an action. The exception handling in *TDL* is similar to the "catch and throw" mechanisms in *C++, Java* and *Lisp*. The main difference is, that the control stack remains as it is when an exception handler is invoked. It is then up to the handler to manipulate the control stack by adding new nodes or terminating existing ones. Exception handlers are associated with a specific node in the task tree and contain a reason in form of a user defined string. [45]

[2] Task trees are specific task nets, where the tasks are stored in a tree structure. Generally, in a task net the tasks can be stored in an arbitrary way (e.g. task pool).

Chapter 4

Method

In this chapter the concepts of the instantiation of the *Three Layer Architecture* are discussed. The focus is laid on the *sequencing layer*. At first the requirements derived from the use cases and summarized challenges are analysed. Afterwards, the complex of problems is discussed. Finally, the concepts are presented.

4.1 Analysis of Requirements

In this section the requirements are analysed and weighted for their relevance for this thesis. The analysis is based on the use cases and summarized challenges depicted in chapter 2. The following list of requirements has no specific order.

I. Hierarchical Tasks

The tasks have to be organized in a hierarchical way. There needs to be a link between the task and its subtasks. The task description has to provide the possibility to express such a relationship.

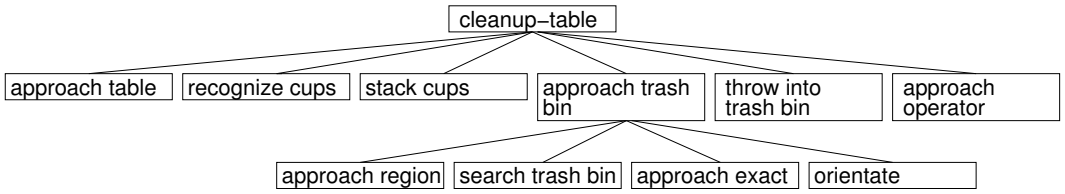

Figure 4.1: Example of a hierarchical task decomposition. The cleanup table task is expanded into detailed steps.

A hierarchical relationship between the tasks allows to describe a task as an abstract representation of the more detailed steps, encoded as subtasks, to achieve a goal. That further ensures reusability, as a higher level task can be composed out of already existing tasks. Figure 4.1 illustrates an excerpt of the task decomposition of the cleanup table scenario. The robot has to cleanup the cups from a table[1]. Therefore the robot hast to stack recognized cups into each other and throw them into the trash

[1] A video showing that task can be found on YouTube: `http://www.youtube.com/roboticsathsulm#p/u/5/40d4Dlk5LCQ`

bin. This task is composed out of the steps: approach the table, recognize cups, stack cups into each other, approach the trash bin, throw cups into trash bin and drive back to the operator. The task to approach the trash bin, again is further detailed into the steps: approach the region where the trash bin is expected, search for the trash bin and determine the exact position, approach the trash bin exactly and orientate the robot towards the trash bin.

II. Reuse of Tasks

Tasks have to be reusable to build different scenarios. The tasks have to be reusable as a kind of black box. That requires, that no detailed knowledge about the task internals should be necessary to know to be able to reuse the task.

Reusability addresses the aspects that behavior developers can compose new behaviors out of existing ones. The example given in the above described requirement I. illustrates, that several tasks are reused to create a new complex task. The results of this work should provide an example how tasks can be reused.

III. Parallel Execution

The underlying formalism of the interpreter has to support parallel execution of tasks. Different execution policies have to be supported.

- *all tasks have to finish their execution (parallel)*
- *if one of the parallel tasks has finished, the others are aborted (one-of)*

Parallel execution of tasks results from the use case, that a new task should be composable out of existing tasks. Requirement I. already addresses this issue, but does not address parallel execution. Several of the described use cases require to support the composition of existing tasks which run in parallel.

IV. Managing Events

Events or similar concepts supporting asynchronous notification have to be managed.

Events and other asynchronous notifications are the preferred way to interact with the robotic components. Notifications, that a specific goal has been reached are typically send as event, for example.

V. Handling Contingencies

The interpreter has to provide a mechanism to handle the huge amount of contingencies occurring at runtime.

The requirement to handle the huge amount of different contingencies occurs in almost each of the use cases. Especially the robust execution in real world raise the complexity.

VI. Plan Modification at Runtime

Plan modification provide the capability to change the execution steps of a task at runtime. The underlying formalism of the interpreter has to be organized in a way that allows to change the subtasks of a task.

Handling a contingency (requirement V.) includes to call an appropriate action depending on the currently executed task and kind of failure occurred. But requires also to recover from that contingency. Therefore it might be necessary to modify the plan described in the currently active task. That requirement, for example, is necessary to implement the use case to switch between different maps.

VII. Integration of Deliberative Tools

Tools for simulation, analysis as well as symbolic planners have to integrated.

The basic steps how to integrate, for example, a symbolic task planner have to be detailed. That includes information gathering in the *sequencing layer* and forwarding it to the symbolic task planner. Furthermore details how the generated plan is incorporated in the task execution in the sequencing layer has to be given. The integration of a symbolic task planner is required to implement the switching between maps use case and furthermore the use case to cleanup the table with several different objects.

VIII. Integration of a *Knowledge Base*

A Knowledge Base (KB) has to be integrated in the sequencing layer. It has to be accessible from the tasks to store and retrieve information occurring at runtime.

Especially scenarios including mobile manipulation require the integration of a *KB* to store, for example, information about objects.

IX. Information Exchange between Tasks

A mechanism to exchange information between tasks has to be supported. Information gathered or generated in a task has to be made available to other tasks.

X. Task Selection at Runtime

The final selection which task to execute has to be peformed at runtime. That includes, that several tasks for the same purpose can be described and stored.

XI. Debugging the Task Expansion

The task expansion at runtime has to be made available for the behavior developer to debug the execution steps and decision made at runtime.

Requirements Overview

The requirements analysed above are summarized in table 4.1. Each requirement is assigned with the relevance for this thesis.

4.2 Complex of Problems

This section focuses on the complex of problems arising in the definition of a task coordination language. Fundamental concepts and mechanisms are discussed. Several related approaches are taken into account how the issues are addressed there. Depending on that analysis, the concept of SMARTTCL is illustrated in section 4.3.

No.	Requirement	Relevance
I.	Hierarchical Tasks	very high
II.	Reuse of Tasks	very high
III.	Parallel Execution	very high
IV.	Managing Events	very high
V.	Handling Contingencies	high
VI.	Plan Modification at Runtime	medium
VII.	Integration of Deliberative Tools	medium
VIII.	Integration of a *Knowledge Base*	high
IX.	Information Exchange between Tasks	high
X.	Task Selection at Runtime	very high
XI.	Debugging the Task Expansion	low

Table 4.1: Requirements with assigned relevance.

4.2.1 Utilization of the *Three Layer Architecture*

This section illustrates how the *Three Layer Architecture* is utilized in other approaches. The different approaches are discussed and the pros and cons are analysed. The basic question is how the three layers interact with each other. The major decision is whether the *deliberative layer* generates a plan and forwards the plan to the *sequencer* (top-down) or the *sequencer* calls the *deliberative layer*.

The first approach proposes a top-down decomposition of the goal. The goal is specified at the *deliberative layer* which installs tasks in the *sequencing layer*. That approach is, for example, used in the *Animate Agent Architecture* [10] or in *3T* [7].

In the *Animate Agent Architecture* (fig. 4.2) the *RAP* system is used as *sequencer*. The overall goal the robot has to achieve is specified in the *deliberative layer*. The planner in that layer generates a plan including the steps how to achieve the goal. That plan is forwarded to the *RAP Executor* (*sequencing layer*). The *RAP Executor* is responsible for executing the plan by activating/deactivating the appropriate *skills* in the *skill layer*. Therefore, the steps generated by the planner in the *deliberative layer* are further detailed into substeps. While executing these substeps the current situation of the robot, which is represented in the *RAP World Model* is taken into account. To react on dynamic changes occurring in real world, both the *RAP Executor* and the *planner* interact with the *RAP World Model*. Small deviations in the plan execution can typically be handled in the *sequencing layer*. Otherwise, the *symbolic planner* generates a new plan which is then again forwarded to the *RAP Executor*. In such a situation, it has to be ensured, that the *RAP Executor* does not change the world state while the *symbolic planner* generates a new plan. This can result in a different initial state that was not assumed as the plan was generated. Furthermore, it is not obvious whether the *sequencing layer* or the *deliberative layer* is responsible to fix the current contingency. It could happen, that the *RAP Executor* has already solved the problem, but the *symbolic planner* has despite generated a new plan and forwarded it to the *RAP Executor*. This architecture is restricted to use exactly one symbolic planner. Other *deliberative* tools are not integrated in that architecture.

In contrast to that, *ATLANTIS* [12] and the SMARTSOFT based *Three Layer Architecture* described in [42], for example, proposes to specify the goal at the *sequencing layer*. The *sequencer* then can consider the symbolic planner in the *deliberative layer*. Based on the generated plan the *sequencer* can manage the further task expansion and execution, can react on contingencies and perform simple local plan repairs. As in that approach, the *deliberator* is not directly in the control loop the performance

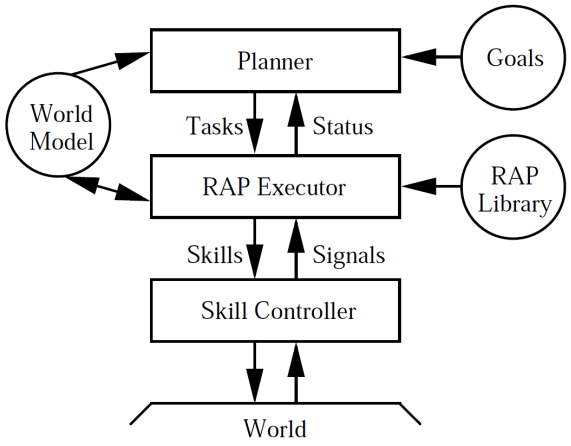

Figure 4.2: The *Animate Agent Architecture* [11].

is in no way affected by the *deliberator* to respond to contingencies. Calling the deliberative layer in specific situations gives the freedom to use different symbolic planners specific to the current problem domain. For example, calling *Metric-FF* [20] in case the domain requires to use metrics and numerical constraints (e.g. only three cups can be stacked into each other) or calling *LAMA* [31] which provides a better performance and seems[2] to find better solution with respect to the action costs. Furthermore, not only different symbolic planner systems can be called – that approach allows to consider several tools which can be incorporated in the decision making at runtime depending on situation and context. That includes tools providing system analysis, monitoring tools, physics simulators, plan verification and tools to perform risk analysis of plans or plan steps. Therefore, the latter described approach appears to provide the better solution. Giving the control to the *sequencer* ensures reactivity, but also gives the freedom to consider several time-consuming tools for decision making individually.

4.2.2 Agenda Structure

The structure of the agenda defines how the different tasks are related to each other and how they are managed. The analysed requirements already specify that the fundamental structure has to be hierarchical. Almost all execution languages support hierarchies, but with different aspects and capabilities.

The *SimpleAgenda*, for example, provides only a limited hierarchy. Two different structures are supported: tasks and operators. A task can contain any number of operators. But a task can not contain other tasks nor can an operator contain a task. That shallow hierarchy severally restrict the usage for building complex real world scenarios.

RAP and *TDL* are based on task-nets. Both distinguish leaf nodes from interior nodes of the task tree. That requires different structures for the different nodes in the task tree. In *TDL* the leaf nodes are called *commands*, the interior nodes are called *goals*. *Goal* nodes can contain *command* nodes as well as *goal* nodes. Each node in the task tree can contain an action. In *RAP*, for example, only the leaf nodes can execute actions. That restriction in *RAP* can not be applied to real robots. Several use cases require to execute an action during task expansion in each level of the expansion. For example, to activate a behavior that remains active while the task-net is further expanded and is deactivated on finishing the task. Otherwise dedicated leaf-nodes have to be inserted to activate/deactivate the desired

[2] Several experiments were performed where LAMA found the optimal solution and FF did not.

behaviors. Figure 4.3 illustrates these two variants. Furthermore, it seems to be a better solution to have only one structure to describe the tasks of the task-net.

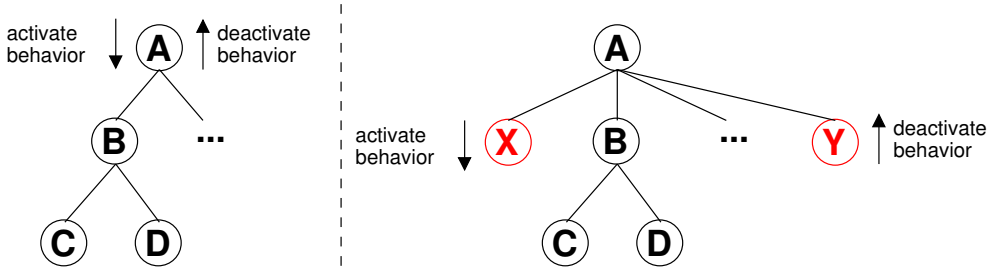

Figure 4.3: Activating/deactivating behaviors in nodes. *left:* Behaviors are activated and deactivated in inner nodes. *right:* Behaviors are activated and deactivated in artificially added dedicated leaf nodes.

4.2.3 Underlying Formalism of Interpreter

The underlying formalism of the task interpreter describes the mechanisms to execute a behavior. This includes managing hierarchies and parallelism, triggering and synchronizing the task execution and providing the flexibility for runtime decisions needed in robotics.

The analyzed requirements specify that the underlying mechanism has to support hierarchical as well as parallel task execution. Hierarchical task execution raises the questions how to describe the hierarchies, how to manage the current state of the task execution (currently active tasks), how to organize the child tasks and how to react on events activated within any of the tasks in the hierarchy. Parallelism raises the question how to synchronize parallel tasks, whether to use true concurrency and how to deal with parallel tasks in case one of them has finished execution – aborting the others?

Several languages support true concurrency. Actions described in the action blocks are executed within individual threads/tasks. In that formalism, actions are typically implemented as blocking calls. Events are difficult to handle, as the blocking calls can not be simply interrupted. Furthermore, action blocks should be abortable. That, as well requires that the blocking calls can be aborted.

The *Smach* [47] formalism, for example, is based on true concurrency. To be able to abort a blocking call the interactions provided by *ROS* [32] (where *Smach* is used) are extended with the *actionlib* [3]. The *actionlib* is used to send goals (request) to the *ROS* nodes (cf. components), requesting the current state about how the request is progressing and to abort a request. It is not obvious that this mechanism can be integrated in all algorithms and libraries used in the different nodes. Several libraries will not provide to cancel the computations as no direct access to the algorithm execution cycle is granted. Furthermore, using the *actionlib*, feedback can only be send to the requester of the action (*request*). In contradiction, configuring the components and sending the feedback with events supports that several components can register to the event and receive the feedback.

The *SimpleAgenda* does not provide concurrency. Thus, each *operator* is executed until it finishes. After execution the result string is checked for errors and if necessary matched against the *rules* in the *KB* to recover from contingencies. As no concurrency is supported no synchronisation is required. In the *SimpleAgenda* the operators are stored in the *KB*. They are instantiated at runtime after they are selected. Once instantiated, the action described in the *operator* is executed until the action finishes. Thus, no special care has to be taken to decide whether an operator has finished or not. After execution,

the operator instance is removed from the task.

RPL, for example, also supports true concurrency. Synchronisation is done by so-called valves. Tasks which should not run simultaneously compete for a valve. The winner is allowed to run, while the loser waits for the winner to finish and release the valve. The basic control concept of *RPL* is similar to task scheduling in operating systems.

In *ESL* true concurrency is achieved by using the *Lisp multi-tasking library*. Synchronization is based on *events* and *checkpoints*. *Events* are like condition variables except a task can wait on multiple events simultaneously. Events also pass data to the waiting tasks. *Checkpoints* are similar to events, but they persist once they are signalled. Tasks can be grouped with the keywords `or-parallel` and `and-parallel`. Using `or-parallel` stops all tasks whether one of the tasks in the group has finished. The computations of the other tasks in that group are aborted. If all tasks fail, the whole group fails with `:ALL-BRANCHES-FAILED`. A `and-parallel` group is active until all tasks in that group have finished. Whether one of the tasks fail, the whole group fails with the result of the tasks which has failed.

In the *RAP* system the tasks are instances of *RAPs*. True concurrency is not supported. True concurrency can be achieved in the *skill layer*. The lifecycle of a *RAP* is as follows: The instances of *RAPs* are created whether a *method* of a task has been chosen that contains a network of subtasks described in form of a task-net. These subtasks are instantiated and added to the *RAP* agenda. Before execution, each task's *SUCCESS* clause is evaluated in the *RAP World Model*. If the *SUCCESS* clause is evaluated to true the task has finished execution and is removed from the agenda. The evaluation of the *SUCCESS* clause is tightly integrated with the *RAP World Model*. In several situations it is not applicable to depend on the *RAP World Model* to evaluate that a task has finished. For example, a task responsible for speech output depends on a *RAP World Model* entry specifying that the given text was spoken by the system. That is not reasonable – the task has to update the *RAP World Model* according to the spoken text, but "somehow" that entry has to be removed after the task has finished. That is necessary that the next time the same text has to be spoken the *SUCCESS* clause of the task will not be evaluated to true because of an old entry in the *RAP World Model* and the task will be finished without execution. Of course, to overcome that a timestamp could be added to the *RAP World Model* entry. But then a timespan has to be defined in that the *SUCCESS* clause is evaluated to true whether the entry exists in the *RAP World Model*. Another example is given by a task that is responsible to drive to a specific location (coordinates given as ?x ?y). In *RAP* the *SUCCESS* clause would look like:

```
(SUCCESS (and
           (= robot-pos-x ?x)
           (= robot-pos-y ?y)))
```

But this *SUCCESS* clause only checks for the exact coordinates. A real robot is never able to approach a position as exactly as it is required in that clause. Thus, with real robots a goal circle would have to be specified. Such a goal circle is hard to include in the *SUCCESS* clause.

Not relying on true concurrency can be achieved by utilizing state chart as, for example, described in [39]. That approach uses the state chart implementation *Visual State* [48]. Only the *entry* and *exit* actions of the states are used. The *do* action is never used in that approach as it is not obvious how the semantic of the *do* action is defined. It is not clear whether the *do* action is executed once or repeatedly until the state is finished. For the computation in the *entry/exit* action only non-blocking calls or blocking calls with a fast response time (several milliseconds) are allowed. Parallel execution is supported by parallel regions. The *entry* actions of the parallel states are executed one after another. As the computations take just very little time, quasi-parallelism is achieved. Typically, the components are configured and activated within the *entry* action and the feedback of the execution is send by events

from the components to the state chart. Thus, the configuration of the components and the feedback send back is decoupled.

Therefore, the aspect of parallel execution seems to be best addressed by a quasi-parallel formalism as, for example, supported by *Visual State*.

4.2.4 Representation of Action Blocks

This section discusses the structure of the action blocks.

The basic idea in *RAP* is that – the task decomposition, providing different tactics to achieve a goal (*methods*), monitoring, error recovery and checking of pre- and post-conditions should be represented in the same "package" (RAP). Thus, all methods within one package are restricted to the same set of input/output variables (signature). Furthermore, as the *RAPs* are stored in a self-contained way, reusing contingency handling strategies (*xRAP* `with-repairs`), for example, in other *RAP* definitions is not supported.

In *RAP* the variants are stored in two levels. It is distinguished between tasks (*RAP* instances) and *methods* inside the *RAPs*. This distinction is a design aspect, that is the tasks are stored in the described structures. As a result, adding a new method requires to modify the whole *RAP* description. Adding variants will be more comfortable whether the different variants of the same task are not wrapped by one entity, but are stored as standalone entities.

In the *SimpleAgenda*, for example, in contrast to *RAP* the operators and rules are stored in the *KB* as single entities. The rules are assigned to the operators.

4.2.5 Action Block Selection at Runtime

Action block selection at runtime is, for example, supported by *SimpleAgenda, RAP, RPL, ESL, PLEXIL* and *TDL*. Different strategies how to select the most appropriate task are used.

In the *SimpleAgenda* the operators are stored in the *KB*. At runtime the operator described in the current active task is matched against the operator stored in the *KB*. For that matching the name of the operator and the binding of the input variables are taken into account. The *SimpleAgenda* does not provide a way to describe a *precondition* for an operator.

One question concerning *preconditions* is whether the precondition has to remain true during the execution of the task or has just to be true at the moment of the selection of the task and can change during execution. In the *RAP* system that is distinguished by providing a *precondition* clause that specifies a condition that has to be true during task selection (before execution) and a *constraints* clause that has to be true during execution. *Constraints* are inherited to the subtasks. Unfortunately it is not obvious how to monitor the *constraints* during task execution. In the *RAP* system that is done by using the *RAP World Model*. As soon as the state of the world changes, this has to be synchronized with the world model. This tight integration is one of the major limitations of the *RAP* system.

Furthermore, in *RAP* the different variants of a task are encoded within different methods of the same *RAP* description, as described in the following:

```
(DEFINE-RAP
  (INDEX (load-into-truck ?object))
  (SUCCESS (location ?object in-truck))
  (METHOD
    (CONTEXT (<world-model-query>))
    (TASK-NET
      (t1 (...))))
  (METHOD
```

```
(CONTEXT (<world-model-query>))
(TASK-NET
  (t1 (...)))))
```

The above *RAP* example shows a *RAP* containing two *methods*. The *context* specifies those situations in which the methods can be applied. A *method context* needs only to be true when the method is chosen and not during execution (cf. *RAP precondition*). If several *methods* are applicable in a given situation the *RAP* system chooses one of them at random.

In the *RAP* system the task is selected by the interpreter according to the following algorithm [10]:

- Choose a task (*RAP* instance) from the agenda.
- Check the task against the *RAP World Model* to see if it is finished.
- If the task is not finished, check its methods and choose one that is appropriate in the current situation.
- If the method is a primitive (leaf node) execute the action.
- If the method is a network of subtasks, put the subtasks on the agenda.

To select a task in the *RAP* system, the tasks in the agenda are checked if they are "eligible". A task is ineligible when any of the following situations hold true:

- Any task constraining the task has not finished execution.
- All of the assigned start times have not yet passed.
- The task has a constraint to wait.

To select among the eligible tasks the following heuristics are used [10]:

- Prefer higher priority task.
- Prefer tasks with deadlines that are closer.
- Prefer tasks that have never failed.
- Prefer tasks from the same family as the last task chosen.
- If there are still several tasks in contention, choose one at random

The task selection is based on the following constraints:

- Ordering constraints between tasks.
- Temporal constraints.
- Memory content constraints.

The *RAP* system does not provide already bound input variables. The distinction between tasks (*RAP* instances) and *methods* inside the *RAPs* make things more complicated, as the selection is organized in two steps. Two different clauses (*precondition* and *context*) are used for the same purpose, but at different levels. As mention above, the task selection is restricted to the queries applicable to the RAP *World Model*.

PLEXIL/DL [25] is an extension of the *PLEXIL* language. *Description Logic* (*DL*) [5] queries are used to describe complex context expressions. To specify *DL* queries only a decideable fragment of *First-Order-Logic (FOL)* is used. The *DL* queries can, for example, be used in lookup expression (c.f. preconditions) to make the execution of the tasks depending on the query result.

4.3 SMARTTCL Concept

In this section the concept of *SmartTCL* is presented. It is discussed how the complex of problemes is addressed by SMARTTCL. The different aspects are presented without any implementation details in mind. Further details on aspects of the reference implementation are presented in chapter 5.

4.3.1 The System Architecture

The basic idea how the *Three Layer Architecture* (fig. 4.4) is applied in this work is described in [40] [42]. The major difference to the *Animate Agent Architecture* is that the *sequencer* is seen as the determining mechanism to achieve a goal. Goals are specified in the *sequencing layer*. They provide the place to store procedural knowledge on how to configure the skills to behaviors, when to use the *deliberative layer* and what kind of action plots are suitable to achieve certain goals. The *deliberative layer* is considered by a node in the *sequencing layer*, that gathers the information necessary to process the request and passes this information to the appropriate tool in the *deliberative layer*.

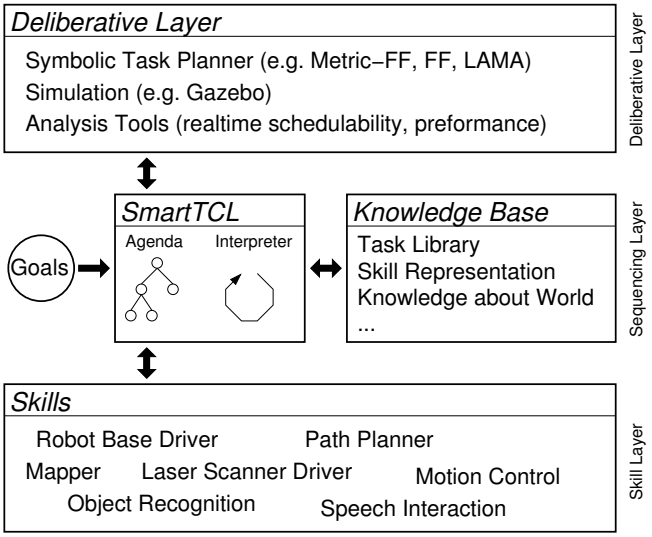

Figure 4.4: Instantiation of the *Three Layer Architecture* how it is used in this work.

That design decisions result in a *Three Layer Architecture* where the so-called *skill layer* comprises components mainly operating on the level of sensors and actuators. These components typically provide services for map building, path planning, speech interaction, object recognition and motion control. The *sequencing layer*, which is the focus of this work, is responsible for the situation-driven task execution. Therefor the sequencer performs dynamic online reconfiguration of the components. Finally, the *deliberative layer* processes time-consuming algorithms, like symbolic task planning, simulations (e.g. physics simulators), system analysis (e.g. realtime schedulability analysis, performance analysis) and risk analysis of plans or plan steps.

Further details on the concrete implementation of the *Three Layer Architecture* how it is applied together with the reference implementation of SMARTTCL are illustrated in chapter 5.

4.3.2 Interfacing between the Layers

This subsection discusses how the three layers communicate with each other. It focuses on describing which kind of interaction is required by SMARTTCL. The communication between the different layers is based on the concepts and mechanisms provided by SMARTSOFT (fig. 4.5). The *sequencer*, as well as the tools used in the *deliberative layer* are wrapped by SMARTSOFT components. Thus, from a communication point of view there is no difference whether the *sequencer* interacts with the components in the *skill layer* or the *deliberative layer*. Using only one communication infrastructure eases the overall architecture.

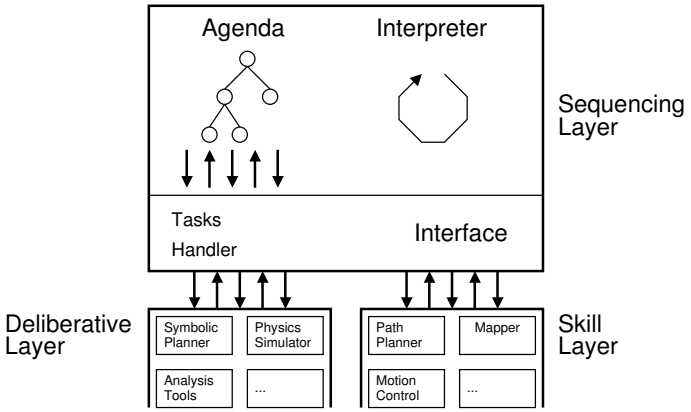

Figure 4.5: Interface between *sequencing layer* and *skill/deliberative layer*. The same concepts provided by SMARTSOFT are used.

The interaction is based on a subset of the communication mechanisms provided by SMARTSOFT. They include:

Dynamic Online Reconfiguration: The *sequencer* uses the `param` port to send parameters to the components, the `state` port to activate/deactivate the components and the `wiring` port to change the wiring between the components at runtime.

Event Pattern: Events are used as asynchronous notification to send feedback from the components to the *sequencer*. Using events, blocking calls can be omitted. Typically, the components are set into the corresponding configuration, process their algorithm according to the concept of local responsibility and send an event either if the goal has been achieved or any contingency occurred.

Query Pattern: Request/Response interaction is used to gather information from the components. That kind of interaction is required, for example, to request information about the recognized objects from a object recognition component.

Details, how the interfacing is realized in the reference implementation are given in chapter 5. Further hints, how vision based behaviors can be interfaced with SMARTSOFT can be found in [40].

4.3.3 Agenda Structure

SmartTCL is based on task-nets. Actions can be executed in each level of the task expansion. A node in the SMARTTCL task tree is called *Task Coordination Blocks (TCB)*. It is the only node type in the

task tree. Supporting only one structure for all nodes defines a clear overall structure and eases the usage for the behavior developers. The *TCB* structure is described in section 4.3.5.

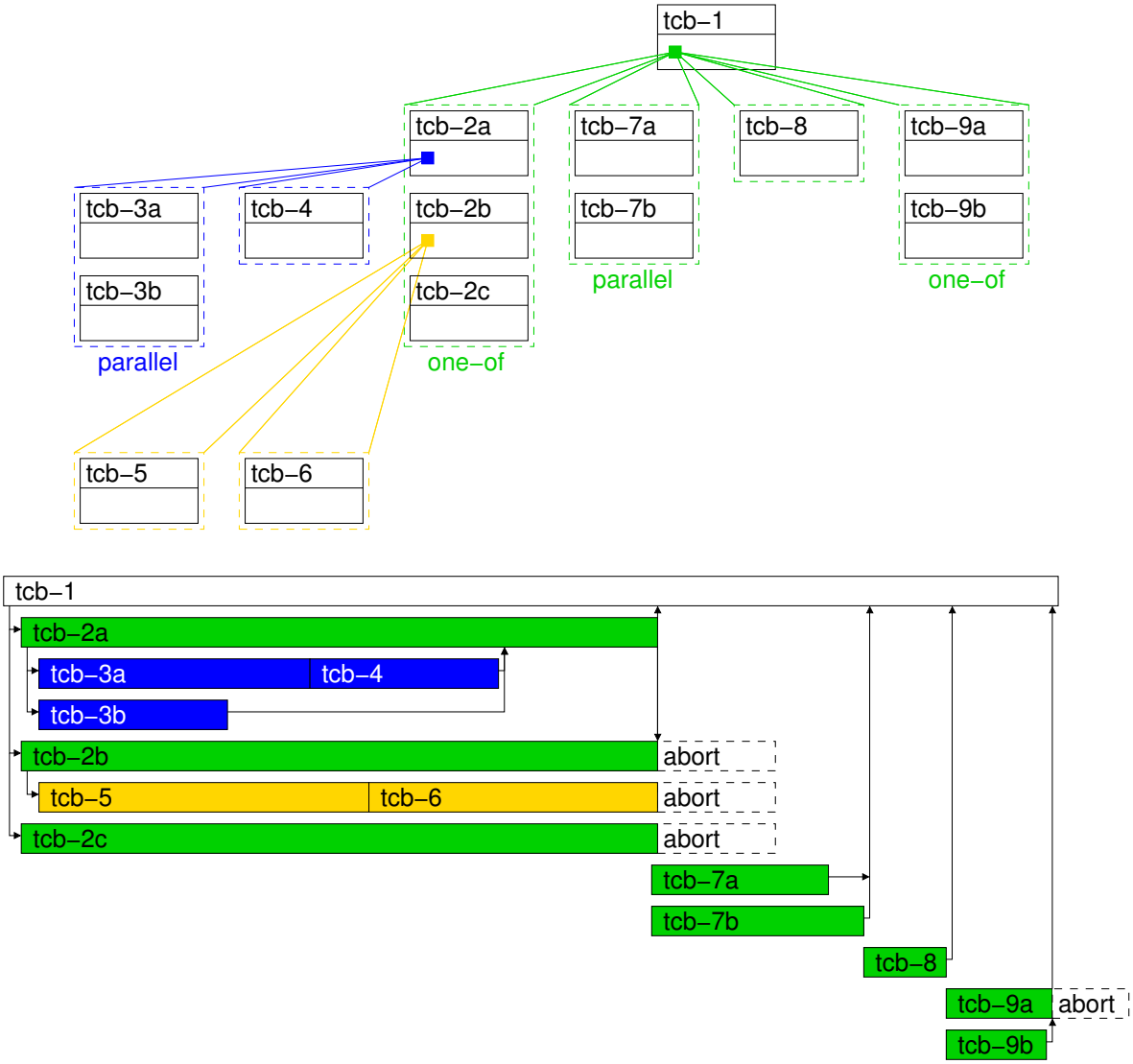

Figure 4.6: *top:* Task-net expansion example. *bottom:* Shows an exemplary execution of the above illustrated task-net.

Figure 4.6 shows an example of a task net. On the top, the task net structure is shown. On the bottom, an exemplary execution sequence of the task net is illustrated. The task `tcb-1` first expands the three parallel nodes `tcb-2a, tcb-2b` and `tcb-2c`. They are marked with the *one-of* label, indicating that the other tasks will be aborted whether one of the tasks has finished. The tasks `tcb-2a` and `tcb-2b` are further expanded. `Tcb-3a` and `tcb-3b` run in parallel. After both have finished execution the task `tcb-4` is executed. Parallel to that execution `tcb-5` and `tcb-6` are executed in sequence. After `tcb-4` has finished `tcb-2a` is still active (waiting for an event). As the three tasks are marked as *one-of*, the tasks `tcb-2b` and `tcb-2c` are aborted after `tcb-2a` has finished. Aborting `tcb-2b` includes aborting `tcb-6` which is a child and was still active. Afterwards the

parallel tasks `tcb-7a` and `tcb-7b` are executed. Both have to be finished before the next step (`tcb-8`) is expanded. Finally, after `tcb-8` has finished `tcb-9a` and `tcb-9b` are executed. The completion of `tcb-9b` results in aborting `tcb-9a` and also the completion of `tcb-1`.

4.3.4 Underlying Formalism of Interpreter

The execution cycle of the interpreter is triggered with events. They provide an asynchronous notification from the components of the *skill* and *deliberative layer*. Events decouple the configuration of the components from the feedback send by the components and thus prevent blocking calls. This is important to support parallel activities without the need for true concurrency (multiprocessing). Concurrency in the task execution is achieved by executing the *action* of the nodes one after another. That quasi-parallel (fig. 4.7) execution mechanism requires that the actions should not execute computa-

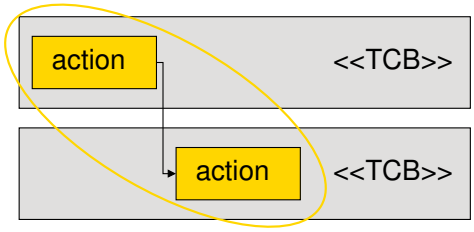

Figure 4.7: Quasi-parallel task execution.

tions or invoke blocking calls that take a long time relative to the time expected from the *sequencer*. True concurrency is achieved in the *skill layer*. Making the *actions* asynchronous furthermore provides to abort a task (node) independent of the *skill layer*. The computations in the *skill layer* can continue (if not possible to abort) without having any influence on the *sequencer*.

TCBs are stored in the *KB*. At runtime an appropriate task is selected and instantiated. A *TCB* has finished its execution if

- the *action* clause was executed,
- all children have finished
- and no more events are activated.

Finished tasks provide a return message to their parent *TCB* which is evaluated to react on failures. Therefore, *TCBs* can have associated *rules* to recover from situations by executing the *action* clause of the *rule*. *Rules* are stored in the *KB*. They are specific to a *TCB signature* and a *return-value* that specifies the reason for the failure.

After instantiating a *TCB* its *action* clause is executed. If no events are activated in the *action* clause and no *TCBs* are stored on the stack, the TCB is finished. If there is an entry on the stack, the included *TCB* or *TCBs* are instantiated and thus their *action* clauses are executed. In case that all child *TCBs* have finished their execution (the stack is empty) and an event is activated the *TCB* remains active, waiting for the event.

4.3.5 Task Coordination Block (*TCB*)

In SMARTTCL the *TCBs* are stored as standalone entities in the *KB*. Standalone entities can easier be handled at runtime. For example, modifing or adding TCBs within the *action* clauses of other *TCBs,*

rules or *event-handlers*.

The nodes of the task-tree are instances of a *Task Coordination Block (TCB)* (fig. 4.8). The *TCBs* are stored in the *KB* (*TCL-Library*) and instantiated at runtime. Several *TCBs* with the same purpose can exist with different signatures. The final selection which one of them will be executed is preformed at runtime. *TCBs* are identified by their signature (name, input/output variables) and *precondition* clause. *TCBs* with the same purpose (e.g. `drive-to`) can have different input/output variables (e.g. `drive-to ?location; drive-to ?x ?y`). The first one is used to drive to a location given by name. The latter is used if the location is given by coordinates (x, y). Input variables can already be bound in the *TCB* definition. The `drive-to ?location`, for example, can have variants with already bound locations (e.g. `drive-to sofa`; `drive-to table`). These already bound variables are taken into account in the task selection at runtime. Specifying already bound variables provides the ability to define further situation dependent *TCBs*.

Task Coordination Block (TCB)		
signature	tcb–name	: *<name>*
	input variables	: *<list of variables>*
	output variables	: *<list of variables>*
	precondition	: *<conditon>*
	priority	: *<integer>*
	rules	: *<list of rules (names)>*
	action	: *<action description>*
	abort–action	: *<action description>*
	plan	: *<child TCBs>*

Figure 4.8: Representation of a Task Coordination Block (TCB).

Furthermore, *TCBs* with the same name and input/output variables can have different preconditions. A *precondition* describes the condition that has to be true to execute the *TCB*.

TCBs have an associated *priotity* which is used for the final task selection at runtime. In case several *TCBs* with their *preconditions* match and could theoretically be executed, the final selection among them considers the priority.

TCBs have a parent/child relationship to support hierarchical task decomposition. A *TCB* can describe primitive, as well as complex behaviors. Primitive behaviors are encoded within the *action* clause. Complex behaviors are composed out of other *TCBs* (children). The children are defined as steps in the *plan* clause. The default execution policy of child *TCBs* is sequential. Otherwise, parallel execution of children is supported, for example, by the keywords *parallel* and *one-of*. A block of children marked as *parallel* is active until all children have finished their execution. If a *TCB* in a *one-of* block has finished, the other *TCBs* are aborted. Aborting a *TCB* involves the execution of the *abort-action* clause to support cleanup actions.

The synchronisation between the *TCBs* and the other layers is realized with events. Events are activated within an *action* clause. Each event can have several event-handlers which are stored in the *KB*. In the event activation one of the event-handlers which is appropriate is assigned to the event.

4.3.6 Information Exchange between *TCBs*

The different tasks have to exchange information during execution.

The *RAPs* provide input/output variables for the tasks. Furthermore, the *RAP World Model* can be used to store and retrieve information. As *ESL* is an extension to *Lisp*, the *Lisp* constructs for variable handling are used. Additionally, a special task group called *task-net* which is similar to the `and-parallel` group is provided. The difference is that *task-nets* in *ESL* create a lexical environment where all the input variables are bound to the subtasks.

The *SimpleAgenda* only supports input variables. Furthermore a *KB* is used to store and retrieve further information.

In *ESL* a *Prolog* based backchaining logical database is used. Local variables are designated by symbols whose print name begin with a question mark. To access the logical database, functions like `DB-ASSERT`, `DB-RETRACT` and `DB-QUERY` are supported.

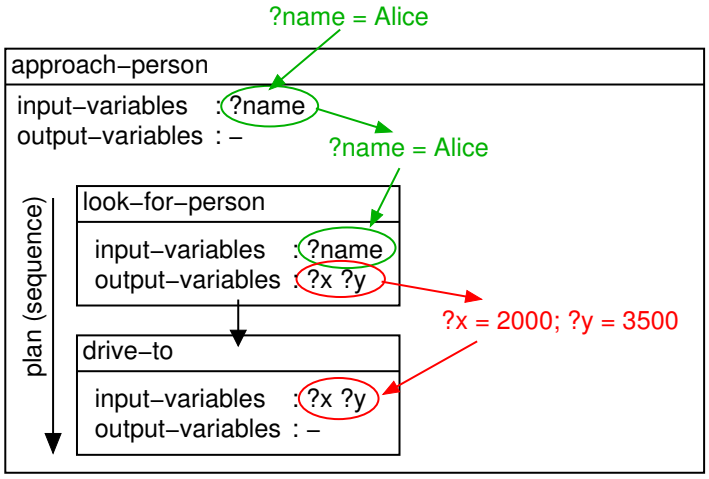

Figure 4.9: Example shows how variables are used to pass information from one *TCB* to another one and how to pass variables from the parent *TCB* to its children.

Information exchange between *TCBs* is mainly based on the concept of input/output variables (fig. 4.9) in a similar way it is, for example, done in *RAP*. Input variables have to be bound before the *TCB* is selected. It has to be ensured, that (i) one of the beforehand executed *TCBs* needs to specify the input variables as its output or (ii) the input variables are specified as input variables of the parent *TCB* and are thus already bound. That requires, that each *TCB* binds its output variables. If a *TCB* execution fails and thus the output variables could not be bound, the *rule* handling that contingency has to bind the output variables. More details on contingency handling with *rules* are described in section 4.3.8.

Additionally, information can be exchanged using a *KB*. The *KB* has at least to provide a simple tell/ask interface for frames. This functionality is required to store and retrieve the SMARTTCL structures (*TCBs*, *rules*, *event-handlers*). If further features are necessary more advanced *KB* systems can be integrated. They typically provide inference and reasoning capabilities. Recognized objects, for example, are stored in the *KB* and are enriched with further information which is already available due to the knowledge about "known" objects. This knowledge, for example, includes properties about an object, like can be stacked into another object and the exact mesh representation which will be required for collision free path planning and grasping calculations.

4.3.7 *TCB* selection at Runtime

The final selection which *TCB* is executed is performed at runtime. In the *plan* clause of the parent *TCB* just the name and input/output variables are specified. It is not specified exactly which *TCB* should be executed. In the *TCL-Library* several different variants of a *TCB* with the same name can exist for a huge variety of different situations. Figure 4.10, for example, shows some *TCBs* stored in the *TCL-Library*. For the `drive-to` *TCB* different variants are specified. That includes different input/output variables and also already bound input variables. In the example, the task-net asks for a *TCB* `drive-to sofa` which is successfully matched against the *TCL-Library*.

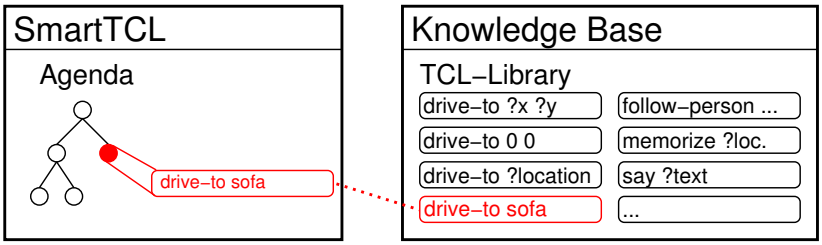

Figure 4.10: Selection of a *TCB*. A variety of different *TCBs* for the same purpose is stored in the *KB (TCL-Library)*.

The selection is performed according to the following algorithm:

- the number of input and output variables as well as the binding of the input variables are matched against the *TCBs* in the *KB*. *TCBs* can define already bound input variables.
- the *precondition* clause is evaluated if one is defined in the *TCB* description. Otherwise, the result is evaluated to *true*. In the *precondition*, for example, the current configuration of the components which are stored in the *KB* can be considered.
- out of the remaining *TCBs* the one with the highest priority is selected.

4.3.8 Feedback from *TCB* Execution – Contingency Handling

This section discusses how feedback is send from tasks that have finished their execution and how to react on that feeedback. In most languages the contingency handling is based on the philosophy of *cognizant failures* [29], stating that systems should be designed to detect failures when they occur. Thus, the system can react appropriately to these contingencies.

In the standard *RAP* system there was no specific construct to react on contingencies. The common way is, that a *RAP* is executed again and again as long it is not finished and the maximum number of retries is not exceeded. Thus, the same *RAP* is applied several times, hoping that another try will solve the problem[3]. Furthermore, constructs exist to start another *RAP* whether the currently active *RAP* has failed (*constraints*). In *xRAP*, the *RAP* definition is extended by the keywords `with-cleanup` and `with-repair` to react on contingencies. Inside these constructs, new tasks can be added to the agenda to react on the failure. There is no support to directly execute an action to react on contingencies.

[3] A futility threshold is supported to avoid futile loops.

Rule	
name	: *<name>*
tcb–name	: *<tcb–name>*
tcb output variables	: *<list of variables>*
tcb input variables	: *<list of variables>*
return–value	: *<return message>*
action	: *<action description>*

Figure 4.11: *TCB rule* to handle contingencies.

In the *SimpleAgenda* contingency handling is based on *rules*. *Rules* describe in their action clause how to recover from the contingency. *Rules* are specific to an operator. Several *rules* for the same operator can exist in parallel but with a different *return-value*. The *return-value* specifies the failure returned whether the execution of an operator fails. This *return-value* is matched against the *rules* which are assigned to the operator. All *rules* are stored in the *KB*.

In *ESL* a similar concept to the *SimpleAgenda rules* is proposed. The concept is called `with-recovery-procedures`. It provides to specify a cause for the failure which is matched against the return value of the task. An *action* is directly executed in case the recovery procedure is applied. Furthermore a *retries* statement is supported to describe how often the recovery procedure can be invoked during the current scope. Nevertheless, the recovery procedures in *ESL* are not directly assigned to a task. Thus they are used in a very generic way.

In SMARTTCL the reaction to failures is defined by *rules* (fig. 4.11) similar to the *SimpleAgenda*. Whenever a *TCB* returns from its execution, the return value is checked whether it contains an error. In this case the error message is matched against the rules stored in the *KB* which are activated within the parent *TCB*. The *rules action* clause is executed to recover from the situation. *Rules* are specific to the name and input/output variables of the *TCB*. That is important especially in case the *TCB* defines output variables that could not be bound during *TCB* execution because of the failure. These variables have then to be bound by the *rule*.

The *rules* for a *TCB* are assigned in the parent definition to provide more flexibility. The same *TCB* definition can be reused with a different set of *rules* depending on situation and context. That allows to handle the same failure occuring in the same *TCB* in different ways.

4.3.9 Component Representation in KB

Each component of the system is represented in the *KB* with the associated information about current state, parametrization, constraints and resource information. That representation can be used in the *precondition* clause to be able to define variants applicable, depending on different configurations of the components. The information about the components can, for example, be retrieved from the MDSD model created during development of the components. Further information on that aspect can be found in [41] [50]. Figure 4.12 shows an example how the components are represented in the *KB*.

In the example the state, parametrization and resources are shown. These properties can be used, for example, in the *precondition* clause.

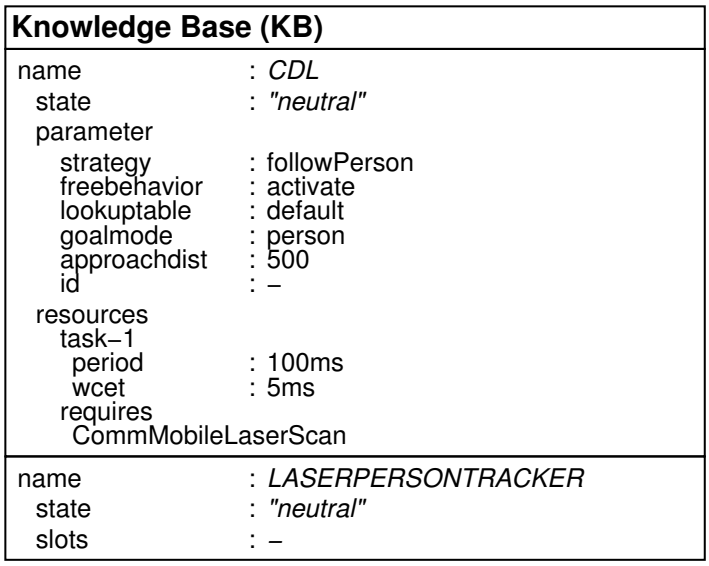

Figure 4.12: Example representation of components in the *KB*.

4.3.10 Event Management

Events are the basic mechanism to synchronize the task execution, decomposition and abortion. Events can be activated in the *action* clause of a *TCB*. The events are directly mapped onto the SMART-SOFT events.

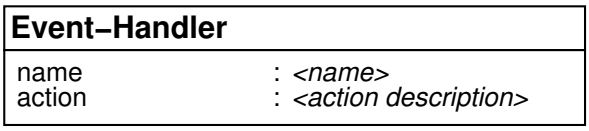

Figure 4.13: Event handler representation.

Each event in SMARTTCL has an associated event-handler (fig. 4.13) which is stored in the *KB*. An *event-handler* can be reused in several event activations. Different kinds of reactions are supported and can be described in the *action* clause of the event-handler to handle common situation, like: ① the path-planner sends an event that indicates that no path to the given goal could be found. The reaction will be to clear the current map and to preoccupy it with the longterm map. The corresponding *TCB* remains active. ② the path-planner sends an event that indicates that the desired goal has been reached. In the *action* clause the path-planner, cdl and mapper components are deactivated to save computational resources. Furthermore the corresponding *TCB* is finished and returns to the parent with *return-value* `SUCCESS`.

4.3.11 Calling the *Deliberative Layer*

The *sequencing layer* calls the *deliberative layer* in situations, where the sequencer is not able to decide about the further execution and expansion of the task-tree. The *deliberative layer* can be invoked from the *action* clause, either directly from a *TCB*, an *event-handler* or a *rule*. This allows,

for example, to call symbolic task planners (*FF, Metric-FF, LAMA*), simulators (*Gazebo* [17]) and analysis tools (*Cheddar* [8]). The required information to perform the request is gathered in the *action* clause and forwarded to the corresponding component in the *deliberative layer*. Based on the feedback, the current *plan* is modified. Examples how a symbolic planner is integrated are given in chapter 6.

Chapter 5

The *Lisp*-based Implementation of SMARTTCL

To gain more experience and to proof the concepts and mechanisms described in the previous section, the *Lisp* based reference implementation of SMARTTCL is developed. This section illustrates on some examples how the concepts are implemented in *Lisp*.

5.1 Instantiation of the Three Layer Architecture

The overall architecture (fig. 5.1) and the communication between the different layers is implemented using the concepts provided by the SMARTSOFT framework [38]. The mechanisms and tools of the *sequencing layer* and *deliberative layer* are made accessible by SMARTSOFT components and thus provide precisely defined interfaces.

The SMARTSOFT Framework and the SMARTSOFT MDSD TOOLCHAIN [50] ease the development process and ensure compatibility of the components. The SMARTSOFT METAMODEL and MDSD TOOLCHAIN ensure straightforward development of flexible and reusable components. Parameters explicated in a modeling level can be accessed during system development, deployment and at runtime.

The *knowledge base (KB)* provides a simple tell/ask interface. The currently used *KB* implementation is based on *Lisp*. The implementation is called *SimpleKB* [35] and is, for example, used by the *SimpleAgenda*.

5.2 Aspects of the SMARTSOFT Framework

The basic concept of SMARTSOFT describes loosely coupled components with local responsibilities and strictly enforced interaction patterns. These patterns provide a precisely defined semantic for communication between components. The set of interaction patterns cover request/response interaction as well as asynchronous notifications and push services. [38]

Some aspects of the patterns which are of importance for the interfacing between the layers are described below.

Dynamic online reconfiguration of the components is provided by (i) a *param* port to send name-value pairs to the components, (ii) a *state* port to activate/deactivate component services and (iii) *dynamic wiring* to change the connections between the components at runtime.

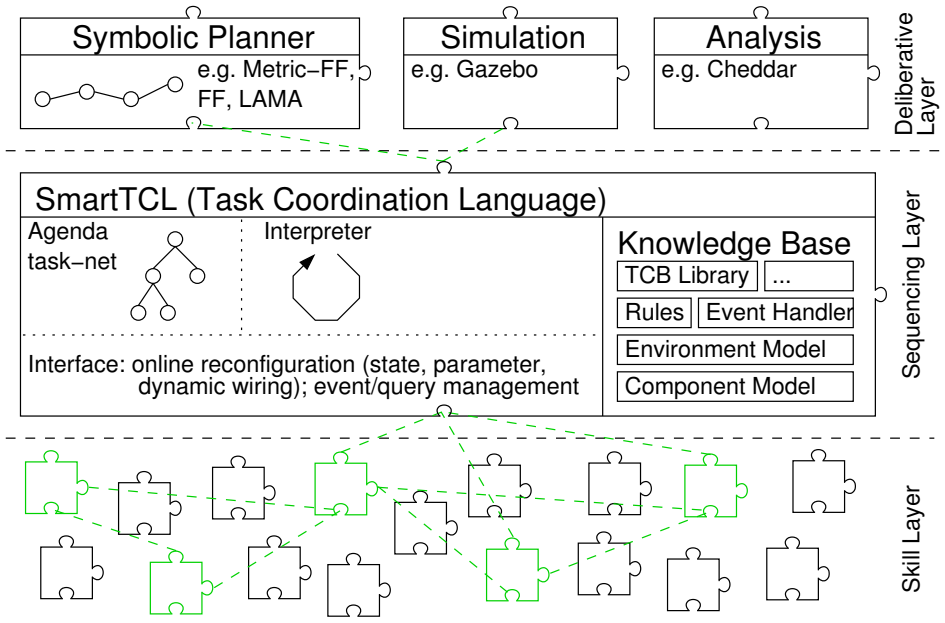

Figure 5.1: The SMARTSOFT based Three Layer Architecture.

Events are used to asynchronously send information from a server to subscribed clients to inform them that a specific condition became true. The event pattern is similar to common push services, but the filtering of the events takes place at the server side according to the parametrization of the event activation. Events are activated by sending a communication object containing the reference values needed for the test, whether the event should be fired or not. Every event can be activated several times simultaneously with individual parameters. It fires if the event condition is true considering the corresponding parameters. Therefore the event pattern provides an event-test-handler at the server side which has to be implemented by the component developer.

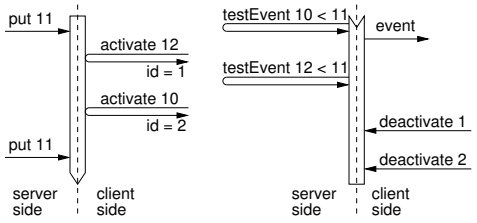

Figure 5.2: Event pattern example demonstrating the battery state event of the robot base component.

Fig. 5.2 demonstrates the usage of the event pattern by the example of the battery event of the robot base component. The event is fired in case the current voltage of the robot's battery is below the activation parameter. On the server side the event test handler are triggered by calling the *put* method with the current state of the event. In the example, the first call of *put* has no effect, since no events are activated. Afterwards two events with parameters 10 and 12 are activated. Each event is associated with an unique id. The next call to put triggers the event test handler, each for every activated event. If the condition is true, the event is fired.

Queries are used to gather specific non-recurring information from components. The query object has to contain all information needed by the server to process the request. The answer is sent back as an answer object. The server can handle the incoming queries either passive or active (queuing them and processing them in a separate thread). The client side supports asynchronous as well as blocking queries.

5.3 Language Extension vs. Standalone Language

SMARTTCL is a language extension to *Lisp*. Thus, basic mechanisms provided by *Lisp* can be used within SMARTTCL.

Language extensions have the advantage, that basic mechanisms provided by the existing language like conditions and looping constructs, for example, can be used within the new language. The new language has the expressiveness of the existing language on its own. Thus, the development time of the new language is reduced and the focus can be laid on the development of the new concepts instead of implementing basic programming language mechanisms. Furthermore, the exact set of basic mechanisms is hard to define. Language extensions ensure the flexibility that behavior developers are not restricted to a predefined (maybe too much restricted) subset of mechanisms. Otherwise, the behavior developers are not allowed to use the whole set of mechanisms the existing language provides. For example, using synchronisation constructs or multitasking mechanisms leverages the mechanisms provided by the new language. This requires a documentation giving the hints, what kind of mechanisms are allowed in which situation.

In general, standalone languages have the advantages, that they are typically easier to analyse and verify [22]. But the expressiveness is limited. Taking, for example, a closer look into the standalone language *RAP*, shows that the so-called *primitives* are implemented in *Lisp*. The *RAP* language is not expressive enough to encode all aspects needed to describe the *primitive* behaviors. For the implementation of the *primitives* only a subset of *Lisp* is allowed for the same reasons as described above for the language extensions.

Thus, the focus of this work is to define the mechanisms and constructs that are necessary to describe the behavior for service robots. For a “first” reference implementation, that can be used for implementing real-world scenarios the decision comes to a language extension. After gaining enough experience to be able to define a standalone language that addresses all aspects necessary to implement real-world scenarios this can be the way to go. Independent of that decision such a new language can nowadays be implemented as a domain specific language (DSL) [19] within the *Eclipse Framework*, for example. The *Eclipse Modeling Framework (EMF)* [1] provides the tools to define and verify *DSL*'s. But again, for a first step the additional work to define a new *DSL* based on *EMF* is omitted due to time constraints.

Therefore, the most important contribution of this work is the mechanisms and concepts. They are independent of the implementation technology. However, for the reference implementation *Lisp* is used. The major advantage of *Lisp* is the reduced development time. This is because *Lisp* compilers are designed to be incremental and interactive. Running programs can easily be modified without stopping and restarting the program. The previous state of the program is preserved. This holds not just true for the development of the new language, but also for the development of the behaviors using the new language. Furthermore *Lisp* provides powerful abstraction facilities that allow to write complex algorithms in a few lines of code. Especially the constructs for unification and dynamic binding of variables can comfortably be implemented in *Lisp*. [16]

Details how *Lisp* is interfaced to SMARTSOFT/C++ can be found in [33].

5.4 Task Coordination Block (TCB)

The *TCB* definition is described using the following syntax:

```
(define-tcb (tcb-name ?x ?y => ?z)
  (precondition (<lisp condition>))
  (priority <number>)
  (rules (rule-1 rule-2))
  (action (<lisp code>))
  (abort-action (<lisp code>))
  (plan ( [parallel|one-of] <list of tcbs> )))
```

Using *Lisp Makros* the *TCB* definition is stored in the *KB*.

The following example illustrates a *TCB* definition used to drive to a specific location which is given as input variable.

```
(define-tcb (approach-location ?location)
  (rules (wrong-map unknown-location maps-not-connected))
  (plan (
    (get-pos-from-location ?location => ?x ?y)
    (drive-to-pos ?x ?y 500))))
```

The *TCB* has no *action/abort-action* clauses. It defines three rules to handle different contingencies. The *TCB* contains a plan with two sequential steps. The first one binds the variables `?x` and `?y` to the coordinates of the given `?location`. The second one then is responsible to approaches the location with a proximity of 500mm.

5.5 Task-Net and Interpreter Structure

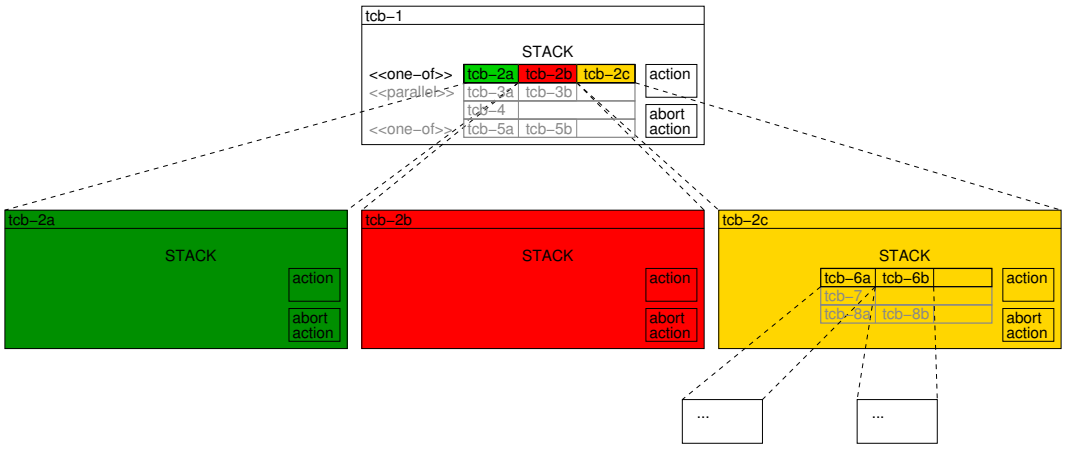

Figure 5.3: Task-Net structure: Child *TCBs* are managed within a stack in the parent *TCB*. *TCBs* can be grouped as *parallel* or *one-of* groups in one stack entry.

The *TCBs* have a parent/child relationship with constraints describing whether they run in sequence or in parallel (*parallel, one-of*). In the approach applied in this work, these constraints are directly mapped onto the structure the task-net is stored. It is stored as a task-tree reflecting the parent/child relationship. A *TCB* manages its children within a stack (fig. 5.3). That reflects the sequential

ordering. Parallel execution is addressed as a stack entry can contain multiple *TCBs*. Entries containing multiple *TCBs* are assigned with the properties whether they follow the *parallel* or *one-of* paradigm. Furthermore, the stack can easily be modified at runtime. If a *TCB* execution fails, the *rule* handling that contingency could, for example, push another or several *TCBs* on the stack. After successfully executing these *TCBs* the actual *TCB* execution continuous.

5.6 Conditional Task Execution

The final selection of a *TCB* is done at runtime. The *TCB* selection is performed in several steps according to the ascending priority. At first, the number of input and output variables as well as the binding of the input variables are matched against the *TCBs* in the *KB*. *TCBs* can define already bound input variables. The second step is to evaluate the *precondition* clause if one is defined in the *TCB* description. Otherwise, the result is evaluated to true. In the *precondition*, *Lisp* conditions as well as *TCL* specific function can be used. This allows, for example, to consider the current configuration of the components, which are stored in the *KB*. As the selection is performed in the ascending priority, the first match is taken for execution.

5.7 Binding of Variables

Figure 5.4 shows the binding of variables in two different situations. On the one hand, the variable bindings are passed from the parent *TCB* to its children and between the children. Whether a child *TCB* defines an already existing variable as output the variable value is overwritten. Thus, the parent *TCB* defines a scope for the variables used by the children.

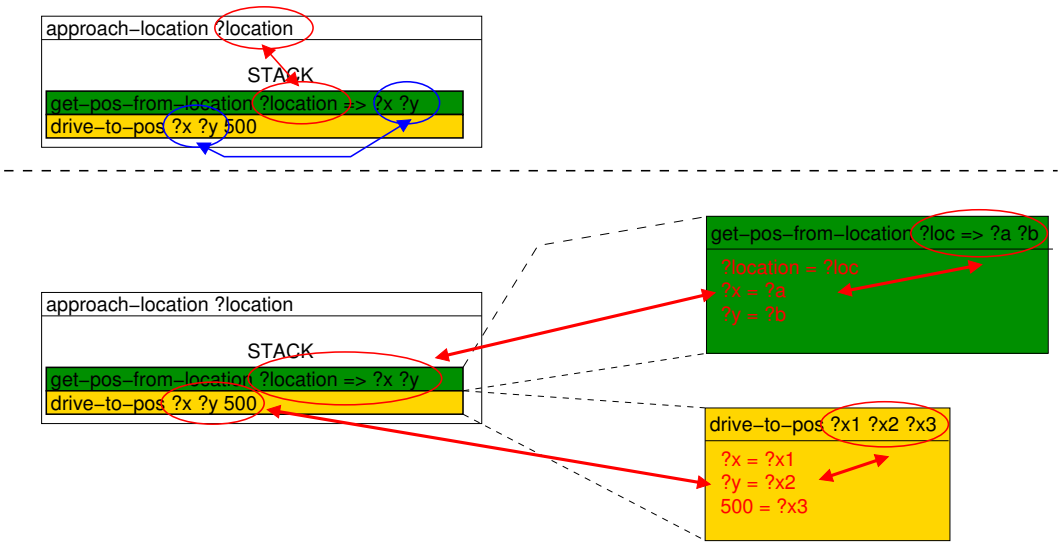

Figure 5.4: *top:* Binding of variables between parent and children. *bottom:* Mapping of external and internal names of the variables.

On the other hand, the names of the variables can differ between the external representation and the internal representation of a *TCB*. These different representations are mapped and the mapping is stored in the *TCB*. Each *TCB* manages its “own” mapping of the external and its internal names of the

variables. That mapping is especially important for the ouput variables. The output variable (external name) has to be assigned with the value instead adding the internal representation with the assigned value to the scope of the parent.

5.8 The Action Clause

Actions are defined within the *TCBs* (*action, abort-action)*, rules and event-handler. They can contain SMARTTCL specific functions and *Lisp* code, with certain restrictions. *Actions* should not execute computations or invoke blocking calls that take a long time relative to the reactivity which is expected from the *sequencer*. This is important since SMARTTCL does not support true concurrency.

SmartTCL specific functions are:

tcl-param (server slot value) to send parameters to a component specified by *server*. The parameter is send by a name/value pair where *slot* specifies the name and *value* specifies the value. Additionally the modified parameter is updated in the component representation stored in the *KB*. The component representation in the *KB* specifies the slots taken into account for that update. This is because it is not reasonable to store every parameter in the component representation. That, for example, holds true for commands like `DELETEGOAL`, which is send to the planner to delete the active goal regions.

tcl-read-param (server slot) to read the parameter value specified by *slot* from the component representation specified by *server* in the *KB*. These values are, for example, used in the *precondition* clause of the *TCBs* to support situation dependent decisions depending on the components parameters. The parameter is not retrieved from the component as that is currently not supported by SMARTSOFT.

tcl-state (server state) to set the component specified by *server* into the state specified by *state*. As with the parameters, the state is updated in the component representation in the *KB*.

tcl-read-state (server) to read the state of the component specified by *server* from the representation in the *KB*.

tcl-activate-event (name handler server service mode param) to activate an event with the name *name* corresponding to the service specified by *server* and *service*. The *event-handler* is assigned to the event. Event-handler are stored in the *KB*. Furthermore, the *mode* specifies whether the event activation is *single* or *continuous*. The *parameter* defines the parametrization of the event activation (for more information on *mode* and *param* see SMARTSOFT documentation).

tcl-event-message returns the event message for further processing. Can only be used inside of the *event-handler*, as the invoking event is known there. The event message can be any *Lisp* structure.

tcl-delete-event deletes the event which invoked the *event-handler*. Can only be used inside of the *event-handler*, as the invoking event is known there. By default events with the assigned mode `continuous` remain active until they are deletetd.

tcl-query (server service request) to send a query specified by *request* to the component specified by *server* and *service*.

tcl-send (server service param) to send parameters and commands to the component specified by *server* and *service*.

tcl-abort to abort a *TCB*. Can, for example be called from a *rule* or *event-handler*. The *abort* is forwarded to all children of the *TCB* and all activated events are deleted.

tcl-bind-var (name value) to bind a variable specified by *name* to a *value*. Is, for example, used to bind the output variables of a *TCB* inside the *action* clause.

tcl-push (tcb) to push a single *TCB* on the *plan* stack.

tcl-push-plan (plan) to push a whole plan on the *plan* stack.

tcl-delete-plan to delete the *plan* stack.

tcl-kb-update (key value) to update a frame in the *KB*. *Key* specifies the key slots of the frame.

tcl-kb-query (key value) to query a frame from the *KB*. In case several frames match the query (*key, value*) only the first one is returned.

tcl-kb-query-all (key value) to query a list of frames from the *KB*. All frames matching the query (*key, value*) are contained in the list.

5.9 Rules

Rules are specific to the exact signature of the *TCB*. That is important especially in case the *TCB* defines output variables that could not be bound during the *TCB* execution because of the failure. These variables have then to be bound by the rule.

The following rule, for example,

```
(define-rule (unknown-location)
  (tcb (get-pos-from-location ?location => ?x ?y))
  (return-value (ERROR (UNKNOWN LOCATION)))
  (action (
    (tcl-delete-plan)
    (tcl-push-plan :plan '(
      (say "I don not know this location. Please
        show me. I will follow you.")
      (parallel (
        (follow-person) (memorize-locations))))))))
```

is associated to the `get-pos-from-location ?location => ?x ?y` *TCB* which is responsible to resolve the position from a by name given location. The *rule* will be executed if it is activated in the parent *TCB* and the *return-value* is `(ERROR (UNKNOWN LOCATION))`. In the *action* clause, at first the current plan of the *TCB* is deleted. Afterwards a new plan is inserted, which contains the steps: ① announce text (*say*) ② run *follow-person* and *memorize-locations* in parallel. Several *rules* for the same *TCB* and *return-value* can exist, but only the desired one should be activated in the parent *TCB*.

5.10 Events and Event-Handler

The events are directly mapped onto the SMARTSOFT event pattern.

The example event activation, called from a *TCB action* clause illustrates the activation of the *goalevent* of the *cdl* component.

```
(tcl-activate-event
  :name 'evt-cdlgoal
  :handler 'handler-cdl
  :server 'cdl
  :service 'goalevent
  :mode 'continuous)
```

The event has the name *evt-cdlgoal* and an assigned event-handler with the name *handler-cdl*. The event activation mode is *continuous*.

An event can be deleted by calling the `tcb-delete-event` function. Furthermore, if the *abort-action* of a *TCB* is called, all activated events if the *TCB* are automatically deleted.

The event handler which is, for example, assigned to the *evt-cdlgoal* is shown below:

```
(define-event-handler (handler-cdl)
  (action (
    (tcl-state :server 'cdl :state "neutral")
    (tcl-state :server 'mapper :state "neutral")
    (tcl-state :server 'planner :state "neutral")
    (tcl-abort))))
```

In the *action* clause of the *event-handler* the components *cdl, mapper* and *planner* are deactivated (state = *neutral*) and the *TCB* calling the handler is aborted. That is the activated events are deactivated and all children are aborted. That causes the *TCB* to having finished execution.

5.11 Online Modification of Plans

The plans stored inside the *TCB* instances can easily be modified at runtime. SMARTTCL functions are provided to delete the whole plan, delete single steps of a plan, push a new step onto the plan or push a whole plan onto the current plan. These plan modifications are typically invoked by executing the *actions* of an event-handler or a *rule* to react on events and contingencies.

The example demonstrates a code snippet where the current plan is deleted and a new plan is pushed onto the stack. The new plan contains two *TCBs* which run in parallel.

```
(tcl-delete-plan)
(tcl-push-plan :plan '((parallel (
  (tcb-follow-person)
  (tcb-memorize-location)))))
```

Chapter 6

Experiments and Results

6.1 Example 1: Guided Tour

This scenario addresses several of the use cases. It is composed out of the behaviors “follow person”, “memorize location” and “approach location”.

In the scenario the robot is guided through its environment by following a person (fig. 6.1). While following, the person shows the robot different locations, like the sofa, the kitchen or the dinner table. The person following, as well as the memorizing of the locations is commanded by speech interaction. Furthermore, the robot can be commanded to approach the locations. In case the location was not shown to the robot and is thus not approachable, the robot will announce that situation and ask the person to show it the location. In the whole scenario there is no fixed ordering in that the behaviors of the robot are applied. The robot can easily be command to follow the person again, after approaching a location. A already known location can be overwritten by guiding the robot to the new position of the location and asking it to memorize it. This is, for example, necessary whether the sofa is moved to a different position. As the different locations can be located in different current map representations the robot switches between the maps (fig. 6.2) as described in chapter 2.

For guiding the robot around, the behaviors “follow person” and “memorize location” are executed in parallel. Thus, the robot can be asked to follow a person or to stop following while telling it a new location. The robot guiding *TCB* looks as follows:

```
(define-tcb (tcb-guiding)
  (action (
           (tcl-send :server 'tts :service 'say-wait
                     :param "Welcome, my name is Kate. I am new to this
                       environment. Can someone please show me around.")
           [...] ;; some initialization
           (tcl-activate-event :name 'evt-approach  :handler 'handler-approach
                               :server 'stt  :service 'sttevent
                               :mode 'continuous
                               :param '(approachLocation 0.3))
           (tcl-activate-event :name 'evt-exit  :handler 'handler-exit
                               :server 'stt  :service 'sttevent
                               :mode 'continuous
                               :param '(command 0.3))))
  (plan ( (parallel (
                     (tcb-follow-person)
                     (tcb-memorize-location))))))
```

Figure 6.1: Robot Guiding Tour: ① User commands the robot to follow him. ② Robot is following. ③ Memorizing the dinner table. ④ Memorizing the sofa. ⑤ Memorizing the kitchen. ⑥ Dinner table approached.

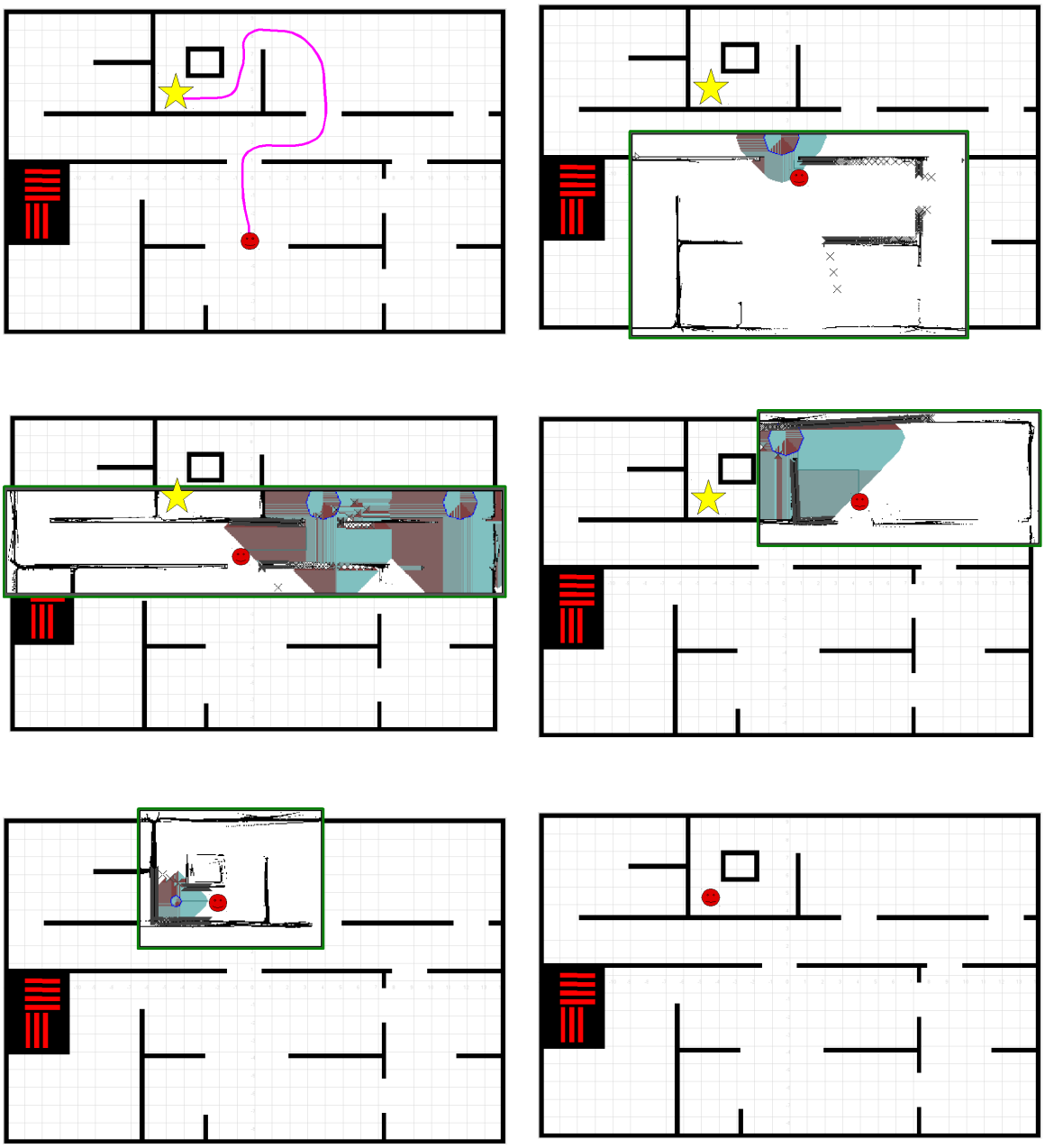

Figure 6.2: Approaching a location including switching between maps performed in the *STAGE* simulator. The goal is marked by a star, the robot by the smiley. Intermediate goals are illustrated as blue circles. The surrounding colored area marks the wavefront performed by the path planner. The currently selected current map is marked by a green rectangle. On the top left image the path the robot will drive to the goal location is illustrated exemplary.

At the beginning of the *action* clause execution a text is send to the speech synthesis component. Afterwards, some initialization is performed. Finally, two events are activated on the speech recognition component. The first one is fired whether the person asks the robot to approach a specific location. The second one is fired whether the person asks the robot to exit the scenario[1]. The *plan* clause contains the two parallel *TCBs* "follow person" and "memorize location".

In case the approach-event is fired the assigned `handler-approach` handler is executed. The handler looks as follows:

```
(define-event-handler (handler-approach)
  (action (
           (tcl-send :server 'tts :service 'say
                     :param (format nil "I will approach the ~s" (second (first
                        (tcl-event-message)))))
            (let* ( (robot-pose (tcl-query :server 'base :service 'pose))
                    (robot-room (get-room-from-pose (first robot-pose)
                                                    (second robot-pose))))

             (tcl-kb-update :key '(is-a)
                            :value `((is-a robot)(current-room ,(get-value
                               robot-room 'name))))
             (tcl-delete-plan)
             (tcl-push-plan :plan `(
                                    (tcb-approach-location
                                       ,(second (first (tcl-event-message))))
                                    (parallel (
                                               (tcb-follow-person)
                                               (tcb-memorize-location)))))))))
```

At first, the robot announces that it will approach the given location. Then, the `robot-pose` is queried from the base component. Based on this pose the current room the robot is located in is determined by calling the helper function `get-room-from-pose`. The current room is updated in the *KB*. Afterwards, the current plan is deleted. Finally, the new plan is inserted. The new plan comprises the *TCB* to approach the given location and afterwards executing the "person follow" and "memorize location" *TCBs* in parallel. These two parallel *TCBs* activate the events that the robot can be commanded to follow a person or to memorize a location. Furthermore, the approach event is still activated and therefore the robot can also be commanded to approach a different location. Approaching a location is performed by the following *TCB*:

```
(define-tcb (tcb-approach-location ?location)
 (rules (rule-wrong-room rule-unknown-location rule-rooms-not-connected))
 (abort-action (
                (tcl-state :server 'cdl :state "neutral")
                (tcl-state :server 'mapper :state "neutral")
                (tcl-state :server 'planner :state "neutral")))
 (plan (
        (tcb-get-location-pose ?location => ?x ?y)
        (tcb-drive-to-pos ?x ?y 500))))
```

The *TCB* contains an *abort-action*, which will deactivate the components for local motion control (CDL), mapping and path planning. The plan itself is a sequence of two *TCBs*. The first one resolves the position of the given location and returns the coordinates. The latter is responsible to drive to the given coordinates. Whether the location can not be resolved in the *KB* by

[1] Explicitly telling the robot to exit the scenario is part of the Robocup@Home rulebook, for example. In this scenario it is used to add further situations the robot has to master.

the `tcb-get-location-pose` *TCB*, it will return with the result value `(ERROR (UNKNOWN LOCATION))`. This contingency will then be handled by the *rule* `rule-unknown-location` which is assigned to the *TCB* definition as shown in the example. Executing the *rule*, the robot will announce that it is not possible to approach the location as it does not know where the location is. The robot is furthermore configured that it is able to be guided to the location. This is done by pushing the parallel *TCBs* "follow person" and "memorize location" on the current plan.

Another contingency occurs, for example, whether the given goal location is not in the same current map the robot is located in. In that situation the *rule* `rule-rooms-not-connected` will be executed. This *rule* is also assigned to the *tcb-approach-location TCB*. In the *action* block of this rule the knowledge about the maps and the doors between the maps is exported to a *PDDL* model. Afterwards, the symbolic task planner *LAMA* is called and provides feedback, in form of the sequence in that the maps have to be passed, to the rule. Finally, this generated plan is imported. This is done by pushing the plan steps onto the plan clause in the *TCB*. Therefore the plan steps are transformed into the corresponding *SmartTCL* representation for a *TCB*. In the following the plan generated by *LAMA* of the example depicted in figure 6.2 is illustrated:

```
((result ok)
 (ops 3)
 (plan ((drive-to-room room-5 room-7)
        (drive-to-room room-7 room-3)
        (drive-to-room room-3 room-2))
```

The plan contains the result, that the plan could be found (`result ok`), the number of operators (`ops 3`) and the plan itself. This plan is, for example, transformed into the SMARTTCL plan steps, which are pushed onto the plan after deleting the whole plan:

```
(tcb-drive-to-room 'room-7)
(tcb-drive-to-room 'room-3)
(tcb-drive-to-room 'room-2)
```

The scenario is completely implemented as part of this work and performed in the *STAGE* simulator, as well as in the real world with the robot "Kate". Further information on the execution are depicted in figure 6.1 and 6.2. The example demonstrates the successful coverage of several of the requirements, such as hierarchical tasks, composition of tasks, parallel execution, managing events, handling contingencies, plan modification at runtime, integration of a symbolic task planner and task selection at runtime.

6.2 Example 2: Cleanup Table Scenario

In this scenario the robot has to cleanup a table. The scenario is an extension of the scenario published on youtube[2]. The basic idea is already described in the use cases in chapter 2. The robot approaches the dinner table in case it is asked to cleanup the table. It recognizes the objects on the table and throws them either into the kitchen sink or into the trash bin. Three different objects exist: crisp cans, beverage cans and cups. The cups can be stacked into each other and have to be thrown into the kitchen sink. The beverage cans can be stacked into the crisp can, but have to be thrown into the trash bin. As the huge amount of different situations of objects on the table is unknown beforehand a symbolic planner generates the sequence how to stack the objects and throw them away.

One challenge to build this scenario is how to export the PDDL model to the symbolic task planner and how to import the found solution. The task planner has to provide capabilities to express

[2] `http://www.youtube.com/roboticsathsulm#p/u/5/40d4Dlk5LCQ`

constraints. The amount of objects that can be stacked into each other is constrained. For example, only three cups can be stacked into each other. Furthermore always the single cup has to be stacked into a stack of cups. Two beverage cans can be stacked into one crisp can. These constraints have to be taken into account by the symbolic planner. Therefore, the *Metric-FF* planner is chosen. For that decision the standard *FF*, *Metric-FF* and *LAMA* planners were taken into account. *Metric-FF* is the only one of them which is able to handle the constraints.

Several of the basic navigation mechanisms are reused in this scenario. But this scenario is more complex and comprises more contingencies which have to be handled by the robot. The crucial part of the scenario is implemented as part of this work. It is not completely finished and the final stress testing has to be done. However, as the critical parts are solved and tested separately the composition to form the whole scenario seems to be a feasible task.

6.3 Overview and Discussion of the Results

In this section the requirements depicted in section 4.1 are discussed how they are fulfilled in this work. An overview can be found in table 6.1. The most relevant requirements are fulfilled. In the following each of the requirements is discussed on its own.

No.	Requirement	Relevance	Status
I.	Hierarchical Tasks	very high	fulfilled
II.	Reuse of Tasks	very high	fulfilled
III.	Parallel Execution	very high	fulfilled
IV.	Managing Events	very high	fulfilled
V.	Handling Contingencies	high	fulfilled
VI.	Plan Modification at Runtime	medium	fulfilled
VII.	Integration of Deliberative Tools	medium	fulfilled
VIII.	Integration of a *Knowledge Base*	high	fulfilled
IX.	Information Exchange between Tasks	high	fulfilled
X.	Task Selection at Runtime	very high	fulfilled
XI.	Debugging the Task Expansion	low	not fulfilled

Table 6.1: Requirement fulfillment.

I. Hierarchical Tasks

Hierarchical tasks are completely supported. Tasks are organized in a parent/child relationship. The children are managed within the *plan* clause of the parent *TCB*. In the reference implementation the *children* are managed within a stack. Hierarchical tasks are widely used in the experiments. Mapping the hierarchical constraints between the tasks directly to the way the tasks are stored, seems to be a practical solution, instead of storing the tasks in a kind of task pool with annotated constraints as it is done, for example, in *RAP*.

II. Reuse of Tasks

The *TCBs* are reused in several different scenarios. Two of them are described in the experiments section. These experiments also gain from the reuse of *TCBs* which where developed for other scenarios. The *TCBs* can almost be reused as a kind of black box. Currently, constraints indicating which resources are used by a *TCB* are not expressed in the *TCB* definition and are not evaluated during execution. Integrating constraints will further improve reusability and robustness of the *TCBs*.

III. Parallel Execution

Parallel execution of *TCBs* is supported in a quasi-parallel way. Two different modes *parallel* and *one-of* are available. This completely covers the requirements placed on this thesis. Quasi-parallel execution is seen as superior to true concurrency and will be a sustainable solution also for the future.

IV. Managing Events

Events are the basic mechanism how feedback is sent from the components to the *sequencer*. The events can be activated and deleted by providing SMARTTCL specific functions. The life cycle and the expansion of the task-net are based on events. The way events are integrated in SMARTTCL is seen as reasonable solution.

V. Handling Contingencies

Contingency handling is supported by providing the concept of *rules*. The *rules* are used to react on contingencies occurring at runtime. Similar mechanisms have already demonstrated their usefulness in other approaches. In the experiments a large amount of different contingencies is handled successfully. The concept of *rules* provides a powerful mechanism to react on the dynamic and unpredictably occurring changes in real world.

VI. Plan Modification at Runtime

Plan modification at runtime is supported by specific SMARTTCL functions. These functions are typically used to react on events or to recover from a contingency by modifying the current plan. In the experiments this mechanism is widely used. For example, plans generated by a symbolic planner are integrated into the current plan by deleting the current plan and adding the generated sequence of steps.

VII. Integration of Deliberative Tools

Three different symbolic task planners are integrated and thus accessible by SMARTTCL as examples of integrating deliberative tools. As symbolic planners typically only provide an output of the found solution on the terminal, a feature to save the found solution into a specified file has been added[3]. This is necessary to be able to import the generated plan into SMARTTCL and modify the current plan. In the experiments two examples are given how symbolic planners are used at runtime. The first one demonstrates how to generate the sequence in which the maps have to be passed to reach the goal location. The latter one demonstrates how to generate the sequence which objects have to be stacked into each other to cleanup the table while minimizing driving around with the robot. The current state

[3] As the sources of all three symbolic planners are available this feature could easily be integrated.

shows that different deliberative tool can be integrated and can be used to support the further task expansion and decision making at runtime. The mechanism how to further integrate several other tools is illustrated.

VIII. Integration of a *Knowledge Base*

The storage of the *TCBs*, *rules* and *event-handlers* is done in a *KB*. In the reference implementation the *SimpleKB* is used. The experiments show that the *KB* is used, for example to store the knowledge about the different rooms and the doors connecting them. Furthermore, the locations the robot memorizes at runtime and can be asked to approach are stored in the *KB*. The *SimpleKB* implementation is quite easy, but provides enough features to be useful for a huge amount of different scenarios. The *SimpleKB* can easily be exchanged with *KB* systems which are more complex as, for example, *PowerLoom* [30].

IX. Information Exchange between Tasks

Information exchange between tasks is supported in two different ways. The *TCBs* provide input and output variables. These variables can be used to pass information from one *TCB* to another one. This mechanism is used to pass local information. Furthermore, information can be stored in and retrieved from the *KB*. Thus, two different options covering different facilities are provided.

X. Task Selection at Runtime

Task selection at runtime is supported. The *TCBs* are stored in the *KB* and at runtime the signature defined in the *plan* clause is matched against the signature of the *TCBs* stored in the *KB*. For the task selection the name of the *TCB*, the precondition, the matching of variables and the priority are taken into account. The current state of the task selection already provides promising results. However, for each *TCB* the evaluation of the precondition results either in true or false – no weighting of the applicability is given. Thus the decision making is somehow still simple and can be extended to use, for example, objective functions to rate how useful a *TCB* is in the current situations.

XI. Debugging the Task Expansion

Debugging the task execution is only supported in a very simple way. The steps which *TCB* is selected, which *rule* and which *event-handler* is executed are only printed in the *Lisp* interpreter in a almost unreadable way. The debugging functionality is strongly limited and is only useful for experienced developers. In the current reference implementation debugging the task expansion is not satisfying and has to be improved. This requirement is not satisfyingly fulfilled, but is also rated with minor relevance for this work. The basic idea is available and can further be implemented to completely fulfill the requirement.

Chapter 7

Summary and Future Work

7.1 Summary

As soon as service robots have to operate in the same environment as humans or even along with them, they have to be able to handle the high dynamics of everyday environments. In reference to the best available knowledge they have to perform the desired tasks in a robust way. Robustness is achieved by integrating the use of symbolic and subsymbolic mechanisms of information processing.

Not only that the components and algorithms of such a system are of considerable complexity, the dynamic management and online reconfiguration of the components further increase the complexity. Various skills are composed at runtime to form behaviors adjusted to the current task, context and perceived situation. The situation dependent composition and selection of skills is the only way to execute tasks in dynamic environments. It is very unlikely that a single and mostly static approach can handle all situations and contingencies experienced in real world operation. Thus, situation and task dependent composition of behaviors is a powerful approach to robustly perform everyday tasks in dynamic environments.

A utilization of the *Three Layer Architecture* to bridge the gap between symbolic and subsymbolic mechanisms is developed and implemented. The major aspect of the utilization of the *Three Layer Architecture* is the issue, that the *sequencer* is the driving part managing the overall system. The *sequencer* is the place to store task dependent procedural knowledge on how to configure skills to behaviors, when to use a symbolic task planner and what kind of action plot are suitable to achieve certain goals. SMARTTCL is tailored to these requirements. Situation dependent task execution is supported by task selection at runtime and online modification of plans. Contingencies are handled by so-called *rules* which contain the steps how to recover from the failure.

The overall architecture and especially SMARTTCL has been successfully used to develop different scenarios. A reference implementation of SMARTTCL based on *Lisp* is implemented to gain more experience and as proof of concept. Several experiments were performed. The basic idea how the *Three Layer Architecture* is utilized in this work and details on the concepts behind SMARTTCL are accepted for publication as paper with the title "SMARTTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots." [49] for the *International Workshop on DYnamic languages for RObotic and Sensors systems (DYROS)*, which is affiliated with the *2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*.

This work provides an important contribution to sequencing in *Three Layer Architectures*. Similar languages which were developed several years ago were not able to demonstrate the power of those

concepts, although they already addressed aspects that are up-to-date nowadays. The progress in robotics allows to proceed the development of those ideas. First experiments and scenarios performed in this work already demonstrate the power and flexibility of that approach.

7.2 Future Work

This thesis provides the first steps towards a novel instantiation of the *Three layer Architecture* and the definition of a task coordination language. It proved to be crucial to store procedural knowledge on how to perform a task in a declarative way within the task nets. The basic mechanisms and concepts provided by SMARTTCL build a promising starting point for further research in this area.

Further investigation on the task coordination language should focus on improving the task selection at runtime, composability of the *TCBs*, verification and validation of the SMARTTCL programs, facilities for debugging the task expansion and decision making and finally the integration of several other tools in the deliberative layer.

Task selection at runtime is currently done in a quit simple way. The precondition clause is only designated to support binary decisions. The overall decision making can be improved. For example, to use a utility function to rate the different *TCBs* according to their expected utility in the current situation.

Currently only *preconditions* are supported by SMARTTCL. The precondition is evaluated before the execution of the task and not further be evaluated during execution automatically. To ensure that a specific condition remains during execution, the components in the skill level have to provide events which can be monitored by the *sequencer*. Further capabilities for monitoring the system and integrating that into the overall architecture is another interessting topic to work on.

In the current reference implementation new behaviors can be composed out of existing ones. However, knowledge about constraints and some other aspects is still necessary to be known by the behavior developer. For example, two *TCBs* operating with the robots manipulator should never run at the same time. To be aware of the constraints these have to be annotated to the *TCBs* and checked at runtime. Another problem occurred, for example, whether two *TCBs* are executed in parallel where both change the currently activated grammar specification of the speech recognition component. The desired behavior would be, that both specifications are merged. But such a behavior is not supported and the specification send as last is active and overwrites the other one.

In the current state no verification of the SMARTTCL programs is supported. For example, the *rules* and *event-handler* are assigned to the *TCBs* and *events* by their string identifier. Typing errors can lead to runtime failures that could be avoided by checking whether the specified *rules* and *event-handlers* are also specified as blocks in the *KB*. Verification and analysis can, for example, be performed by defining SMARTTCL in a textual modeling level. Having the behavior programs in such a representation, verification tools can be used to ensure, for example, that the mapping of the string identifiers are correct. Further analysis can be performed based on the model representation. One possible solution to implement such a representation can be to use *xText* [55], which is part of the *Eclipse Modeling Project*.

The proposed concept of SMARTTCL allows to easily integrate state chart, for example, generated by *Visual State*. In that situation a *TCL* node is represented by a state, which can contain hierarchies and parallel regions. State charts can be used to model static assured behaviors for situations where no variability in the task execution is wanted. In safety critical situations the overall execution control can be given to the state chart by preventing parallel *TCBs* from execution. This integration in supported thanks to the very similar underlying mechanisms how events are handled. The integration of this two

mechanisms has further to be investigated on. Especially, how the events are mapped between the different representations.

Debugging the task decomposition and decision making at runtime is currently only supported in a very rudimentary way. A graphical representation, for example, based on *Graphviz* [18] will be very useful. Further execution traces illustrating why which decision was taken will be mandatory to efficiently develop complex scenarios. Especially for behavior developers which are not familiar with the details of SMARTTCL the currently available status messages do not provide enough support.

In this work three different symbolic task planners are integrated as example of tools in the deliberative layer. Further tools to support the decision making at runtime have to be added. This includes, for example, simulators and analysis tools. As an example of a simulator, *Gazebo* [17] can be used to obtain the maximum allowed velocities taking the current payload of the robot into account. Furthermore, analysis tools can be used to check at runtime whether the desired configuration of the components in the skill layer is feasible, before setting the configuration.

Appendix A

Sources Guiding Tour

This appendix completely illustrates the source files of the guiding tour scenario. The *TCBs*, *rules* and *event-handlers* are defined in the `scenario.lisp` file. The file `symbolicPlannerImport-Export.lisp` contains the *Lisp* function to export the *PDDL* model from the *KB* specific to the *map switching task*. It also contains the import function to add the generated sequence into the current *plan* clause. Depending on the environment the scenario can be performed in different *KB* setups. The first setup (`memory-kate.lisp`) is used on the real robot "Kate". The setup is performed in one room (robot-lab C26). The second setup (`memory-stage.lisp`) is performed in the stage simulator. In that setup seven different rooms exist which are represented in the *KB*.

```
;;;; S E T U P   --    S T R A T E G Y
(defun get-room-from-pos (x y safety)
  (let ( (result nil))
    (dolist (room (tcl-kb-query-all :key '(is-a) :value '((is-a room))))
      (let (
            (size-x (first (get-value room 'size)))
            (size-y (second (get-value room 'size)))
            (offset-x (first (get-value room 'offset)))
            (offset-y (second (get-value room 'offset))))
        (cond
          ((and
            (>= (- x safety) offset-x)
            (<= (+ x safety) (+ offset-x size-x))
            (>= (- y safety) offset-y)
            (<= (+ y safety) (+ offset-y size-y)))
           (format t "---------------->>> room ~s -- ~s ~s ~s ~s ~%" (get-value room 'name) size-x size-y offset-x offset-y)
           (setf result room);(get-value room 'name))
           (return)))))
    result))



;; tcb-drive-to-room -- 1
(define-tcb (tcb-drive-to-room ?room)
  (precondition (equal '?room (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'current-room)))
  (priority 5)
  (action (
           (format t "tcb-drive-to-room: ~s ~%" '?room)
           (format t "robot is in correct room ~%")
           '(SUCCESS ()))))


;; tcb-drive-to-room -- 2
(define-tcb (tcb-drive-to-room ?room)
  (action (
           (format t "tcb-drive-to-room: ~s ~%" '?room)
           (format t "robot has to switch to another room ~%" '?room)
           (let* (
                  (robot-pos (tcl-query :server 'base :service 'pose))
                  (robot-current-room-name (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'current-room))
                  (robot-room (tcl-kb-query :key '(is-a name) :value `((is-a room)(name ,robot-current-room-name)))))

             (let ((room-pair (sort (list (get-value robot-room 'name) '?room) #'string<)))
               (format t "roomList: ~s~%" room-pair)
               (let ((door-list (query-kb-all *MEMORY* '(is-a from to) `((is-a door)(from ,(first room-pair))(to ,(second room-pair))))))
                 (format t "doorList length: ~d ~%" (length door-list))
                 (cond
                   ((> (length door-list) 0)
                    (tcl-activate-event :name 'evt-cdlgoal :handler 'handler-cdl :server 'cdl :service 'goalevent :mode 'continuous)
                    (tcl-activate-event :name 'evt-planner :handler 'handler-planner :server 'planner :service 'plannerevent :mode 'continuous)
                    (tcl-kb-update :key '(is-a) :value `((is-a robot)(goalid ,(+ 1 (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid)))))
                    (tcl-state :server 'planner :state "neutral")
                    (tcl-state :server 'mapper :state "neutral")
                    ;; cdl
                    (tcl-param :server 'cdl :slot 'ID :value (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))
                    ;; mapper
                    (tcl-param :server 'mapper :slot 'CURPARAMETER :value (append (get-value robot-room 'size) (get-value robot-room 'offset)
                                                                                  (list (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))))
                    ;; planner
                    (tcl-param :server 'planner :slot 'ID :value (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))
                    (tcl-param :server 'planner :slot 'DELETEGOAL)

                    (dolist (door door-list)
                      (let ((pos (get-value door '?room)))
                        (tcl-param :server 'planner :slot 'SETDESTINATIONCIRCLE :value `(,(first pos) ,(second pos) 500))))
                    (tcl-state :server 'planner :state "pathplanning")
                    (tcl-state :server 'mapper :state "buildbothmaps")
                    (setf *new-room* '?room)
                    '(SUCCESS))
                   (T
                    (format t "Switching rooms requires asking the symbolic planner for ordering of rooms~%")
                    '(ERROR (ROOMS NOT DIRECTLY CONNECTED)))))))))))
```

```
;; tcb-drive-to-pos -- 1
(define-tcb (tcb-drive-to-pos ?x ?y ?dist)
  (precondition (equal
                 (get-room-from-pos '?x '?y 0)
                 (tcl-kb-query :key '(is-a name) :value `((is-a room)(name ,(get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'current-room))))))
  (priority 5)
  (action (
           (let* (
                  (robot-pos (tcl-query :server 'base :service 'pose))
                  (robot-current-room-name (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'current-room))
                  (robot-room (tcl-kb-query :key '(is-a name) :value `((is-a room)(name ,robot-current-room-name)))))
             (format t "tcb-drive-to-pos: already correct room ~s ~%" (get-value robot-room 'name))
             (tcl-activate-event :name 'evt-cdlgoal :handler 'handler-cdl :server 'cdl :service 'goalevent :mode 'continuous)
             (tcl-activate-event :name 'evt-planner :handler 'handler-planner :server 'planner :service 'plannerevent :mode 'continuous)
             ;;(setq *goalId* (+ 1 *goalId*))
             (tcl-kb-update :key '(is-a) :value `((is-a robot)(goalid ,(+ 1 (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid)))))
             (format t "before planner state -> neutral ~%")
             (tcl-state :server 'planner :state "neutral")
             (format t "after planner state -> neutral~%")
             (format t "before mapper state -> neutral ~%")
             (tcl-state :server 'mapper :state "neutral")
             (format t "after mapper state -> neutral ~%")
             ;; cdl
             (tcl-param :server 'cdl :slot 'ID :value (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))
             ;; mapper
             (tcl-param :server 'mapper :slot 'CURPARAMETER :value (append (get-value robot-room 'size) (get-value robot-room 'offset)
                                                                           (list (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))))
             (format t "param: ~s ~%" (append (get-value robot-room 'size) (get-value robot-room 'offset)
                                              (list (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))))
             ;; planner
             (tcl-param :server 'planner :slot 'ID :value (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'goalid))
             (tcl-param :server 'planner :slot 'DELETEGOAL)
             (tcl-param :server 'planner :slot 'SETDESTINATIONCIRCLE :value '(?x ?y ?dist))

             (tcl-state :server 'planner :state "pathplanning")
             (tcl-state :server 'mapper :state "buildbothmaps")
             ;;(tcl-state :server 'cdl :state "moverobot")
             '(SUCCESS ())))))


;; tcb-drive-to-pos -- 2
(define-tcb (tcb-drive-to-pos ?x ?y ?dist)
  (action (
           (format t "tcb-drive-to-pos: switching room necessary ~%")
           '(ERROR (WRONG ROOM)))))



;; tcb-follow-person
(define-tcb (tcb-follow-person)
  (action (
           (format t "=========================>>> tcb-follow-person ~%")
           (tcl-activate-event :name 'evt-follow :handler 'handler-follow-person :server 'stt :service 'sttevent :mode 'continuous :param '(robotControl 0.3))))
  (abort-action (
                 (format t "=========================>>> ABORT TCB-FOLLOW-PERSON ~%~%~%")
                 (tcl-state :server 'cdl :state "neutral")
                 (tcl-state :server 'laserpersontracker :state "neutral"))))


;; tcb-memorize-location
(define-tcb (tcb-memorize-location)
  (action (
           (format t "=========================>>> tcb-memorize-location~%")
           (tcl-activate-event :name 'evt-memorize :handler 'handler-memorize-location :server 'stt :service 'sttevent :mode 'continuous :param '(memorizeLocation 0.3)))))



;; tcb-get-location-pose
(define-tcb (tcb-get-location-pose ?location => ?x ?y)
  (action (
           (format t "=========================>>> tcb-get-location-pose: ~s ~%" '?location)
           (let ((location-pose (get-value (tcl-kb-query :key '(is-a name) :value '( (is-a location) (name ?location))) 'pose)))
```

```
            (cond
              ((null location-pose)
                '(ERROR (UNKNOWN LOCATION)))
              (T
                (tcl-bind-var :name '?x :value (first location-pose))
                (tcl-bind-var :name '?y :value (second location-pose)))))))



;; tcb-approach-location
(define-tcb (tcb-approach-location ?location)
  (rules (rule-wrong-room rule-unknown-location rule-rooms-not-connected))
  (abort-action (
                (tcl-state :server 'cdl :state "neutral")
                (tcl-state :server 'mapper :state "neutral")
                (tcl-state :server 'planner :state "neutral")))
  (plan (
        ;;(tcb-setup-approach-halt)
        (tcb-get-location-pose ?location => ?x ?y)
        (tcb-drive-to-pos ?x ?y 500))))





;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; H A N D L E R ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; handler-goal-reached
(define-event-handler (handler-cdl)
  (action (
           (format t "========================>>> GOAL REACHED HANDLER~%~%")
           (cond
             ((equal (tcl-event-message) '(reached))
               (cond
                 ((equal (tcl-current-tcb-name) 'tcb-drive-to-pos)
                   (format t "========================>>> GOAL REACHED !!! ~%")
                   (tcl-state :server 'cdl :state "neutral")
                   (tcl-state :server 'mapper :state "neutral")
                   (tcl-state :server 'planner :state "neutral")
                   (tcl-abort))
                 ((equal (tcl-current-tcb-name) 'tcb-drive-to-room)
                   ;;; TODO
                   (tcl-kb-update :key '(is-a) :value '((is-a robot)(current-room ?ROOM)))
                   (tcl-abort))))))))


;; handler-planner
(define-event-handler (handler-planner)
  (action (
           (format t "========================>>> PLANNER HANDLER~%~%")
           (cond

             ((equal (tcl-event-message) '(start occupied by goal))
               (format t "========================>>> start occupied by goal !!! ~%")
               (tcl-param :server 'mapper :slot 'CURLOADLTM))

             ((equal (tcl-event-message) '(start occupied by obstacle))
               (format t "========================>>> start occupied by obstacle !!! ~%")
               (tcl-param :server 'mapper :slot 'CURLOADLTM))

             ((equal (tcl-event-message) '(wrong map id))
               (format t "========================>>> wrong map id !!! ~%"))

             ((equal (tcl-event-message) '(no path))
               (format t "========================>>> no path !!! ~%")
               (tcl-param :server 'mapper :slot 'CURLOADLTM))

             ((equal (tcl-event-message) '(ok))
               (format t "========================>>> ok !!! ~%")
               (sleep 1) ;; TODO
               (cond
                 ((equal (tcl-current-tcb-name) 'tcb-drive-to-pos)
```

```
                    (tcl-param :server 'cdl :slot 'GOALMODE     :value 'PLANNER)
                    (tcl-param :server 'cdl :slot 'STRATEGY     :value 'APPROACH_HALT))
                  ((equal (tcl-current-tcb-name) 'tcb-drive-to-room)
                    (tcl-param :server 'cdl :slot 'GOALMODE     :value 'PLANNER)
                    (tcl-param :server 'cdl :slot 'STRATEGY     :value 'APPROACH)))

                (tcl-state :server 'cdl :state "moverobot"))

               (T
                 (format t "=========================>>> unsupported event ~s !!! ~%" *MSG* ))))))



;; handler-follow-person
(define-event-handler (handler-follow-person)
  (action (
           (format t "=========================>>> FOLLOW PERSON HANDLER~%~%")
           (cond
             ((equal (second (first (tcl-event-message))) 'followPerson)
               (cond
                 ((not (equal (get-value (tcl-kb-query :key '(is-a) :value '((is-a robot))) 'strategy) 'FOLLOW))
                   (tcl-send :server 'tts :service 'say :param "I will follow you. If you want me to stop please tell me.")
                   ;; CDL
                   (tcl-param :server 'cdl :slot 'STRATEGY     :value 'FOLLOW)
                   (tcl-param :server 'cdl :slot 'GOALMODE     :value 'PERSON)
                   (tcl-param :server 'cdl :slot 'FREEBEHAVIOR :value 'ACTIVATE)
                   (tcl-param :server 'cdl :slot 'LOOKUPTABLE  :value 'DEFAULT)
                   (tcl-param :server 'cdl :slot 'APPROACHDIST :value 500)
                   ;; LASER PERSON TRACKER
                   (tcl-param :server 'laserpersontracker :slot 'SETMAXCOV :value '(640 0))
                   (tcl-param :server 'laserpersontracker :slot 'RESET :value '(800 0))))
               (tcl-state :server 'laserpersontracker :state "follow")
               (tcl-state :server 'cdl :state "moverobot"))

             ((equal (second (first (tcl-event-message))) 'stop)
               (tcl-state :server 'laserpersontracker :state "neutral")
               (tcl-state :server 'cdl :state "neutral")
               (tcl-send :server 'tts :service 'say :param "I have stopped following you."))

             (T
               nil)))))


;; handler-memorize-location
(define-event-handler (handler-memorize-location)
  (action (
           (format t "=========================>>> MEMORIZE HANDLER: ~s ~%~%" (tcl-event-message))
           (tcl-send :server 'tts :service 'say :param (format nil "I have memorized the ~s"(second (first (tcl-event-message)))))
           (tcl-kb-update :key '(is-a name) :value '(
                                                     (is-a location)
                                                     (name ,(second (first (tcl-event-message))))
                                                     (pose ,(tcl-query :server 'base :service 'pose)))))))

;; handler-exit
(define-event-handler (handler-exit)
  (action (
           (format t "=========================>>> EXIT HANDLER: ~s ~%~%" (tcl-event-message))
           (tcl-send :server 'tts :service 'say-wait :param "Bye bye")
           (tcl-abort))))


;; handler-approach
(define-event-handler (handler-approach)
  (action (
           (format t "=========================>>> APPROACH HANDLER: ~s ~%~%" (tcl-event-message))
           (tcl-send :server 'tts :service 'say :param (format nil "I will approach the ~s"(second (first (tcl-event-message)))))
           (let* ( (robot-pos (tcl-query :server 'base :service 'pose))
                   (robot-room (get-room-from-pos (first robot-pos) (second robot-pos) 2000)))
             (cond
               ((null robot-room)
                 (setf robot-room (get-room-from-pos (first robot-pos) (second robot-pos) 0))))
             (format t "robot-pos: ~s  --  robot-room-name: ~s ~%" robot-pos (get-value robot-room 'name))
             (tcl-kb-update :key '(is-a) :value '((is-a robot)(current-room ,(get-value robot-room 'name))))
           (tcl-delete-plan)
```

```
          (tcl-push-plan :plan `(
                                    (tcb-approach-location ,(second (first (tcl-event-message))))
                                    (parallel (
                                                (tcb-follow-person)
                                                (tcb-memorize-location))))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; R U L E S  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; tcb-drive-to-pos -- (ERROR (WRONG ROOM))
(define-rule (rule-wrong-room)
  (tcb (tcb-drive-to-pos ?x ?y ?dist))
  (return-value (ERROR (WRONG ROOM)))
  (action (
          (format t "RULE: (ERROR (WRONG ROOM)) ~%")
          (let ((room (get-room-from-pos '?x '?y 0)))
            (tcl-delete-plan)
            (tcl-push-plan :plan `((tcb-drive-to-room ,(get-value room 'name)) (tcb-drive-to-pos ?x ?y ?dist)))
            ;;(tcl-push :tcb '(tcb-drive-to-pos ?x ?y ?dist))
            ;;(tcl-push :tcb `(tcb-drive-to-room ,(get-value room 'name)))
          '(SUCCESS ()))))


;; tcb-drive-to-pos -- (ERROR (UNKNOWN LOCATION))
(define-rule (rule-unknown-location)
  (tcb (tcb-get-location-pose ?location => ?x ?y))
  (return-value (ERROR (UNKNOWN LOCATION)))
  (action (
          (format t "RULE: (ERROR (UNKNOWN LOCATION)) ~%")
          (tcl-send :server 'tts :service 'say-wait :param (format nil "I do no know where the ~s is." '?location))
          (tcl-delete-plan)
          (tcl-push-plan :plan '((parallel (
                                               (tcb-follow-person)
                                               (tcb-memorize-location)))))
          '(SUCCESS ()))))


;; tcb-drive-to-room -- (ERROR (ROOMS NOT CONNECTED))
(define-rule (rule-rooms-not-connected)
  (tcb (tcb-drive-to-room ?room))
  (return-value (ERROR (ROOMS NOT DIRECTLY CONNECTED)))
  (action (
          (format t "RULE: (ERROR (ROOMS NOT DIRECTLY CONNECTED)) ~%")
          (export-pddl-rooms '(at ?room))
          (cond
            ((tcl-query :server 'symbolicplanner :service 'query :request (list
                                                                            '(operator "~/SOFTWARE/smartsoft/src/master/smartSymbolicPlanner/PDDL/roomSwitch-3-domain.pddl")
                                                                            '(fact     "~/SOFTWARE/smartsoft/src/master/smartSymbolicPlanner/PDDL/roomSwitch-3-fact.pddl")
                                                                            '(result   "~/SOFTWARE/smartsoft/src/master/smartSymbolicPlanner/PDDL/roomSwitch-3-result.pddl")))
              ; everything ok
              (cond
                ((import-plan-pddl-room-switch)
                  ; everything ok
                  (format t "generated new plan")
                  '(success))
                (T
                  ; could not import plan
                  (format t "no plan available")
                  '(ERROR (NO PLAN AVAILABLE)))))

              (T
                ; something went wrong with query in general
                '(ERROR (UNKNOWN)))))))



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; S C E N A R I O  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;; tcb-simple-shopping-mall-1
(define-tcb (tcb-simple-shopping-mall-1)
  (action (
           (format t "=========================>>> tcb-simple-shopping-mall-1~%")
           ;; setup scenario
           (tcl-send :server 'tts :service 'say-wait :param "Welcome, my name is Kate. I am new to this environment. Can someone please show me around.")
           (tcl-state :server 'stt :state "active")
           (tcl-param :server 'cdl :slot 'TRANSVEL :value '(0 500))
           (tcl-param :server 'cdl :slot 'ROTVEL :value '(-50 50))
           (tcl-param :server 'mapper :slot 'CURLTM         :value '(DISABLE 10))
           (tcl-param :server 'mapper :slot 'CUREMPTY       :value 'ACCUMULATE)
           ;(tcl-param :server 'mapper :slot 'LTMLOAD :value 1)
           (tcl-param :server 'mapper :slot 'LTMINITIALIZE  :value 0)           ;; TODO

           (tcl-state :server 'mapper :state "buildbothmaps")
           (tcl-state :server 'mapper :state "neutral")

           (tcl-param :server 'cdl :slot 'FREEBEHAVIOR :value 'ACTIVATE)        ;; TODO
           (tcl-param :server 'cdl :slot 'LOOKUPTABLE  :value 'DEFAULT)         ;; TODO
           (tcl-param :server 'cdl :slot 'APPROACHDIST :value 100)              ;; TODO

           ;(tcl-kb-update :key '(is-a) :value '((is-a robot)(current-room room-5)))
           (tcl-kb-update :key '(is-a) :value '((is-a robot)(goalid 0)))
           (tcl-kb-update :key '(is-a) :value '((is-a robot)(cur-map-size (30000 20000))))
           (tcl-kb-update :key '(is-a) :value '((is-a robot)(cur-map-offset (-15000 -10000))))
           (tcl-activate-event :name 'evt-approach :handler 'handler-approach :server 'stt :service 'sttevent :mode 'continuous :param '(approachLocation 0.3))
           (tcl-activate-event :name 'evt-exit :handler 'handler-exit :server 'stt :service 'sttevent :mode 'continuous :param '(command 0.3))))
  (plan (
         (parallel (
                    (tcb-follow-person)
                    (tcb-memorize-location))))))
```

```
;; E X P O R T
(defun export-pddl-rooms (goal-description)
  (setf path (make-pathname :directory "~/SOFTWARE/smartsoft/src/master/smartSymbolicPlanner/PDDL/" : name "roomSwitch-3-fact.pddl"))
  (setf str (open path :direction :output :if-exists :supersede))
  ;;
  (format str "(define (problem roomSwitch-1)~%")
  (format str "  (:domain switch-to-room)~%")
  ;; objects -- city
  (format str "  (:objects~%")
  (let ((objects (query-kb-all *MEMORY* '(is-a) '((is-a room)))))
    (dolist (obj objects)
      (let ((name (get-value obj 'name)))
        (format str "    ~s - room~%" name))))
  (format str "  )~%")

  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
  ;; init
  (format str "  (:init~%")
  ;; current-room
  (let ((objects (query-kb-all *MEMORY* '(is-a) '((is-a robot)))))
    (dolist (obj objects)
      (let ((current-room (get-value obj 'current-room)))
        (format str "    (at ~s)~%" current-room))))
  ;; doors
  (let ((objects (query-kb-all *MEMORY* '(is-a) '((is-a door)))))
  (dolist (obj objects)
    (let ((from (get-value obj 'from))
          (to (get-value obj 'to))
          (open (get-value obj 'open)))
      (cond ((equal open 'true)
             (format str "    (door ~s ~s)~%" from to))))))

  (format str "  )~%")

  ;; goal
  (format str "  (:goal~%")
  (format str "    ~s~%" goal-description)
  (format str "))")

  (close str))


;; I M P O R T
(defun import-plan-pddl-room-switch ()
  (setf path (make-pathname :directory "~/SOFTWARE/smartsoft/src/master/smartSymbolicPlanner/PDDL/" : name "roomSwitch-3-result.pddl"))
  (setf str (open path :direction :input))
  (let* ((result (read str))
         (status (second (assoc 'result result)))
         (plan   (second (assoc 'plan result))))
        (close str)
        (cond
          ((equal status 'error)
                ;no plan generated
                (format t "no plan generated~%")
                nil)
          ((equal status 'ok)
            ;correct plan available
            (format t "correct plan available~%")
            (let ((new-plan nil))
              (dolist (step (reverse plan))
                (push '(tcb-drive-to-room ,(third step)) new-plan))
              (tcl-push-plan :plan new-plan))
            T)
          (T
            ;unknown format
            (format t "unknown format~%")
            nil))))
```

```
(defun show-locations ()
  (let ((obj-list     (query-kb-all *MEMORY* '(is-a) '((is-a location)))))
    (dolist (obj obj-list)
      (format t "~%--------------------------------~%")
      (format t "name      :~s ~%" (get-value obj 'name))
      (format t " pose     :~s ~%" (get-value obj 'pose)))))


(defun show-robot ()
  (let ((robot (query-kb *MEMORY* '(is-a) '((is-a robot)))))
    (format t "~%--------------------------------~%")
    (format t " goalid    :~s ~%" (get-value robot 'goalid))
    (format t " current-room :~s ~%" (get-value robot 'current-room))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; M E M O R Y    - - -    K N O W L E D G E B A S E

;; R O O M S
(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-1)
     (speech (room one))
     (size (40000 40000))
     (offset (-20000 -20000))))
```

```lisp
(defun show-locations ()
  (let ((obj-list     (query-kb-all *MEMORY* '(is-a) '((is-a location)))))
    (dolist (obj obj-list)
      (format t "~%--------------------------------~%")
      (format t "name      : ~s ~%" (get-value obj 'name))
      (format t " pose     : ~s ~%" (get-value obj 'pose)))))


(defun show-robot ()
  (let ((robot (query-kb *MEMORY* '(is-a) '((is-a robot)))))
    (format t "~%--------------------------------~%")
    (format t " goalid     : ~s ~%" (get-value robot 'goalid))
    (format t " current-room  : ~s ~%" (get-value robot 'current-room))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; M E M O R Y     - - -     K N O W L E D G E B A S E

;; R O O M S
(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-1)
     (speech (room one))
     (size (10500 7500))
     (offset (-14500 2000))))


(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-2)
     (speech (room two))
     (size (10000 7500))
     (offset (-6500 2000))))


(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-3)
     (speech (room two))
     (size (16000 7500))
     (offset (-1500 2000))))


(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-4)
     (speech (room three))
     (size (10000 11500))
     (offset (-14500 -9500))))


(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-5)
     (speech (room four))
     (size (19000 11500))
     (offset (-9000 -9500))))


(tcl-kb-update :key '(is-a name) :value
  '(
     (is-a room)
     (name room-6)
     (speech (room five))
     (size (8500 11500))
     (offset (6000 -9500))))


(tcl-kb-update :key '(is-a name) :value
```

```
  '(
     (is-a room)
     (name room-7)
     (speech (hallway))
     (size (29000 6000))
     (offset (-14500 -1000))))


;; D O O R S
(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-1)
     (from room-5)
     (to room-6)
     (room-5 (6500 -1500))
     (room-6 (8000 -1500))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-2)
     (from room-5)
     (to room-6)
     (room-5 (6500 -6500))
     (room-6 (8000 -6500))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-3)
     (from room-4)
     (to room-5)
     (room-4 (-7500 -500))
     (room-5 (-5500 -500))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-4)
     (from room-5)
     (to room-7)
     (room-5 ( -500 -500))
     (room-7 ( -500 1750))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-5)
     (from room-6)
     (to room-7)
     (room-6 (9000 -500))
     (room-7 (9000 1750))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-6)
     (from room-3)
     (to room-7)
     (room-3 (12000 4250))
     (room-7 (12000 2500))
     (open true)))
```

```
(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-7)
     (from room-3)
     (to room-7)
     (room-3 (4000 4250))
     (room-7 (4000 2500))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-8)
     (from room-2)
     (to room-3)
     (room-2 (   0 8000))
     (room-3 (1500 8000))
     (open true)))


(tcl-kb-update :key '(is-a name from to) :value
  '(
     (is-a door)
     (name door-9)
     (from room-1)
     (to room-7)
     (room-1 (-12500 4500))
     (room-7 (-12500 2500))
     (open true)))
```

List of Figures

List of Tables

Bibliography

[1] Eclipse Modeling Framework (EMF). http://www.eclipse.org/modeling/emf/, visited on October 31th 2010.

[2] Technical Comittee on Mobile Manipulation. http://mobilemanipulation.org, visited on October 31th 2010.

[3] ROS actionlib. http://www.ros.org/wiki/actionlib, visited on October 31th 2010.

[4] Aibo. Robot Dog, http://support.sony-europe.com/aibo/, visited on October 31th 2010.

[5] F. Baader, D. Calvanese, D. L. McGuinness, Nardi D., and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[6] J.-C. Baillie. URBI: Towards a Universal Robotic Low-Level Programming Language. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 820 – 825, 2005.

[7] R.P. Bonasso, R.J. Firby, E. Gat, D. Kortenkamp, D.P. Miller, and M.G. Slack. Experiences with an architecture for intelligent, reactive agents. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2):237–256, 1997.

[8] Cheddar. a free real time scheduling analyzer, http://beru.univ-brest.fr/~singhoff/cheddar/, visited on October 31th 2010.

[9] M. Egerstedt. Behavior Based Robotics Using Hybrid Automata. In *HSCC '00: Proceedings of the Third International Workshop on Hybrid Systems: Computation and Control*, pages 103–116, London, UK, 2000. Springer-Verlag.

[10] R. J. Firby. *Adaptive Execution in Complex Dynamic Worlds*. PhD thesis, Yale University, 1989.

[11] R. J. Firby. An Architecture for A Synthetic Vacuum Cleaner. In *In Working Notes: AAAI Fall Symposium on Instantiating Real-World Agents*, pages 55 – 63, 1993.

[12] E. Gat. Integrating Planning and Reacting in a Heterogeneous Asynchronous Architecture for Controlling Real-World Mobile Robots. *Architecture*, 1992.

[13] E. Gat. ESL: a language for supporting robust plan execution in embedded autonomous agents. In *IEEE Aerospace Conference Proceedings. 1997.*, volume 1, pages 319–324, 1997.

[14] E. Gat. The ESL user's guide. Unpublished, August 1997. Jet Propulsion Lab.

[15] E. Gat. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*, chapter On Three-Layer Architectures, pages 195–210. AAAI Press/The MIT Press, 1st edition, March 1998.

[16] E. Gat. Lisp as an Alternative to Java. *Intelligence*, 11, 2000. http://www.flownet.com/gat/papers/lisp-java.pdf.

[17] Gazebo. 3D multiple robot simulator with dynamics, `http://playerstage.sourceforge.net/gazebo/gazebo.html`, visited on October 31th 2010.

[18] Graphviz. Graph Visualization Software, `http://www.graphviz.org/`, visited on October 31th 2010.

[19] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[20] J. Hoffmann. The metric-FF planning system: translating "Ignoring delete lists" to numeric state variables. *J. Artif. Int. Res.*, 20(1):291–341, 2003.

[21] D. Kortenkamp, R. P. Bonasso, and R. Murphy, editors. *Artificial Intelligence and Mobile Robots – Case Studies of Successful Robot Systems*. AAAI Press/The MIT Press, 1st edition, March 1998.

[22] D. Kortenkamp and R. Simmons. *Springer Handbook of Robotics*, chapter Robotic Systems Architectures and Programming, pages 187–206. Springer-Verlag, 2008.

[23] Lua. Scripting Language, `http://www.lua.org/`, visited on October 31th 2010.

[24] D. McDermott. A Reactive Plan Language. Technical report, Yale University, 1993.

[25] H. Moser, T. Reichelt, N. Oswald, and S. Förster. Context-sensitive Plan Execution Language for Adaptive Robot Behaviour. In Max Bramer, Richard Ellis, and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXVI*, pages 233 – 246. Springer London, 2010.

[26] NAO. Humanoid Robot Platform, `http://www.aldebaran-robotics.com/en`, visited on October 31th 2010.

[27] D. Nau, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[28] T. Niemueller, A. Ferrein, and G. Lakemeyer. A Lua-based Behavior Engine for Controlling the Humanoid Robot Nao. In *RoboCup Symposium 2009*, Graz, Austria, 2009.

[29] F. R. Noreilst. Integrating Error Recovery in a Mobile Robot Control System*. *Supervision*, pages 396–401, 1990.

[30] PowerLoom. Knowledge Representation & Reasoning System, `http://www.isi.edu/isd/LOOM/PowerLoom/`, visited on October 31th 2010.

[31] S. Richter and M. Westphal. The LAMA planner. Using landmark counting in heuristic search. In *Proceedings of IPC-6, International Planning Competition*, 2008.

[32] ROS. Robot Operating System, `http://www.ros.org`, visited on October 31th 2010.

[33] C. Schlegel. Schnittstelle SmartSoft/Lisp. FAW Ulm, SFB 527, Unpublished.

[34] C. Schlegel. SimpleAgenda. Unpublished source files.

[35] C. Schlegel. SimpleKB. Unpublished source files.

[36] C. Schlegel. Fast local obstacle avoidance under kinematic and dynamic constraints for a mobile robot. In *International Conference on Intelligent Robots and Systems*, pages 594 – 599, 1998.

[37] C. Schlegel. *Navigation and Execution for Mobile Robots in Dynamic Environments: An Integrated Approach*. PhD thesis, University Ulm, 2004.

[38] C. Schlegel. *Software Engineering for Experimental Robotics*, volume 30 of *STAR series*, chapter Communication Patterns as Key Towards Component Interoperability, pages 183–210. Springer, 2007.

[39] C. Schlegel, T. Haßler, A. Lotz, and A. Steck. Robotic software systems: From code-driven to model-driven designs. In *International Conference on Advanced Robotics (ICAR)*, June 2009.

[40] C. Schlegel, J. Illmann, H. Jaberg, M. Schuster, and R Wörz. Integrating Vision-Based Behaviors with an Autonomous Robot. In *VIDERE - Journal of Computer Vision Research*, pages 32–60, 2000.

[41] C. Schlegel, A. Steck, D. Brugali, and A. Knoll. Design Abstraction and Processes in Robotics: From Code-Diven to Model-Driven Engineering. In *2nd International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Darmstadt, Germany, 2010.

[42] C. Schlegel and R. Wörz. Interfacing different layers of a multilayer architecture for sensorimotor systems using the object-oriented framework SMARTSOFT. In *Third European Workshop on Advanced Mobile Robots (Eurobot)*, pages 195–202, 1999.

[43] R. G. Simmons. Concurrent Planning and Execution for Autonomous Robots. *Control Systems Magazine, IEEE*, 12(1):46–50, February 1992.

[44] R. G. Simmons. Structured control for autonomous robots. *Robotics and Automation, IEEE Transactions on*, 10(1):34–43, February 1994.

[45] R. G. Simmons and D. Apfelbaum. A Task Description Language for Robot Control. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, 1998.

[46] R. G. Simmons, R. Goodwin, C. Fedor, and J. Basista. *Task Control Architecture - Programmer's Guide to Version 8.0*. Carnegie Mellon University, August 1995.

[47] SMACH. `http://www.ros.org/wiki/smach`, visited on October 31th 2010.

[48] IAR Visual State. `http://www.visualstate.com`, visited on October 31th 2010.

[49] A. Steck and C. Schlegel. SmartTCL: An Execution Language for Conditional Reactive Task Execution in a Three Layer Architecture for Service Robots. In *International Workshop on Dynamic languages for RObotic and Sensors systems (DYROS)*, Darmstadt, Germany, 2010.

[50] A. Steck and C. Schlegel. Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development. In *1st International Workshop on Domain-Specific Languages and models for ROBotic systems (IROS - DSLRob)*, Taipei, Taiwan, 2010.

[51] V. Verma, A. Jónsson, R. Simmons, T. Estlin, and R. Levinson. Survey of Command Execution Systems for NASA Spacecraft and Robots. In *"Plan Execution: A Reality Check" Workshop at The International Conference on Automated Planning and Scheduling (ICAPS)*, 2005.

[52] M. Weser, D. Off, and J. Zhang. HTN Robot Planning in Partially Obserable Dynamic Environments. In *Proceedings of the 2010 IEEE International Conference on Robots and Automation (ICRA)*, pages 1505–1510, Anchorage, Alaska, USA, 2010.

[53] M. Weser and J. Zhang. Autonomous Planning for Mobile Manipulation Services Based on Multi-Level Robot Skills. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1999–2004, St. Louis, MO, USA, 2009.

[54] M. Wopfner, J. Brich, S. Hochdorfer, and C. Schlegel. Mobile Manipulation in Service Robotics: Scene and Object Recognition with Manipulator-Mounted Laser Ranger. In *Proceedings for the joint conference of ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, Munich, Germany, 2010.

[55] Xtext. Language Development Framework, `http://www.eclipse.org/Xtext/`, visited on October 31th 2010.