Karlsruhe Institute of Technology

Connecting Symbolic Task Planning with Motion Control on ARMAR-III Using Object-Action Complexes

Student Research Thesis
of

Nils Adermann

At the faculty of Computer Science
Institute for Anthropomatics

Advisor: Prof. Dr.-Ing. Rüdiger Dillmann
Supervisor: Dr.-Ing. Tamim Asfour

Research Period: 1 June 2010 – 31 August 2010

I declare that I have developed and written the enclosed Student Research Thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 31. August 2010

Abstract
This thesis presents an architecture enabling humanoid robots to plan and execute sequences of hierarchical Object-Action Complexes (OACs). OACs define expected partial world state transformations resulting from OAC execution. A perception processing system populates short-term memory with abstractions of low-level sensory information. A Central Executive Agent (CEA) communicates short-term memory contents to the symbolic planning system PKS (Planning with Knowledge and Sensing) and receives instructions in the form of OACs. The CEA instantiates OACs from long-term memory with object information from short-term memory in order to execute them. Plan execution is continuously monitored and plans can be adjusted or regenerated autonomously.

Zusammenfassung
Diese Studienarbeit stellt eine Architektur für humanoide Roboter vor, die Objekt-Aktion Komplexe (OACs) verwendet um auf einem humanoiden Roboter Handlungsabläufe zu planen und auszuführen. OACs beschreiben die aus ihrer Ausführung resultierenden partiellen Transformationen des Weltzustands. Ein System zur Verarbeitung von Perzeption füllt das Kurzzeitgedächtnis des Roboters mit abstrahierten Repräsentationen der Sensorinformationen. Ein Zentraler Exekutiver Agent (CEA) sendet die Inhalte des Kurzzeitgedächtnisses an das symbolischen Planungssystem PKS (Planung mit Wissen und Wahrnehmung) und erhält von ihm in Form von OACs Anweisungen zur Ausführung. Der CEA instanziiert die erhaltenen OACs mit Wissen aus dem Langzeitgedächtnis und Objektinformationen aus dem Kurzzeitgedächtnis um sie anschließend auszuführen. Die Ausführung des Plans wird kontinuierlich überwacht, sodass der Plan gegebenenfalls autonom angepasst oder neu generiert werden kann.

Contents

1 Motivation

The majority of robots in use today solve only one or few specific problems in easily predictable constraint environments. Rigid requirements on predictability may suit industrial robots working in assembly lines but they pose a problem for general purpose service robots. A service robot collaborating with humans in environments made for humans like a household will encounter a much wider variety of objects and activities. A service robot operating in these environments will without doubt have to cope with unexpected situations.

Artificial intelligence has been a research subject for more than 50 years and many results have been found using simulation or simple robots. But only recent advances in hardware and mechanics have made it feasible to build general purpose robots like humanoid robots. The open environments these operate in and inaccuracies of their sensors pose additional problems for artificial intelligence. Decisions need to be made relying on uncertain data and unexpected situations need to be analyzed and understood to react appropriately.

Planning is one of the fields of artificial intelligence. It deals with generating sequences of actions – plans – to solve problems given an initial state and a set of constraints.

ARMAR-III [1] is a humanoid service robot which serves as an example of a robot with the outlined problems. This thesis describes the robust integration of planning software on ARMAR-III. It takes into account that ARMAR-III has only limited knowledge of the world's state, cannot always move accurately and receives inaccurate sensory information.

1.1 Current State of ARMAR in the Kitchen

At present ARMAR-III is able to perform numerous tasks in its kitchen environment. Using its stereo camera vision system it can recognise objects of various shapes, colours and textures. Combined with motion control of its 7-DOF arms ARMAR-III is capable of grasping these objects with its pneumatically actuated hands. A laser scanner allows the robot to locate itself inside the kitchen. ARMAR-III can recognise and grasp door handles allowing it to open the doors of the kitchen's dishwasher, fridge and cupboards [2].

ARMAR-III's software distinguishes scenarios, tasks and skills. At the lowest level skills combine perception and motion control to perform actions such as grasping of tableware based on Visual Servoing. Tasks sequentially apply multiple skills to achieve a more complex goal; e.g. bringing an object from the fridge to the robot's operator by opening the fridge, grasping the object and handing it over to the robot's operator. At the highest level of abstraction scenarios can receive speech commands and use tasks to perform the requested operations. The demonstration of new tasks and skills often

results in the creation of a new scenario which combines existing tasks with new ones to present them in a meaningful context. Creating and maintaining scenarios requires a significant amount of work – a lot of which could be avoided because each scenario needs to solve the same problems again.

1.2 Context and Goal of this Thesis

In this thesis I present a system for simplifying the creation and maintenance of scenarios using symbolic task planning software which communicates with an Central Executive Agent controlling the robot's perception and motion. The planning system dynamically generates instructions leading the robot to reach a goal supplied by an operator through speech. The combination of executive and planning system replace hardcoded scenarios for the purpose of demonstrations. The robot provides the planning system with symbolic abstractions of its sensory input. And it translates the planning system's instructions into motion primitives to be executed in the physical world. The concept of Object-Action Complexes from the PACO+ project serves as the basis for communicating knowledge, experience and instructions between the system's components. The system allows ARMAR-III's existing tasks and skills to be used for evaluation purposes.

2 State of the Art

2.1 Planning in Artificial Intelligence

A planning problem consists of an initial state, a set of actions an agent can peform and a set of goal conditions to be satisfied. Plans are sequences of actions transforming the initial state into a state which satisfies the set of goal conditions. In this context state often refers to something absolute but can also be understood as the agent's knowledge of the world. The descriptions of states, actions and goal conditions require a representation of objects and properties that exist in the world. Automated planners can then generate a plan when provided with a planning problem.

STRIPS [3] is an automated planner developed by Fikes and Nilsson in 1971. It considers a closed world in which the world state is represented as a set of facts which are always either true or false. A world state database contains all true facts or ground atoms. Actions can only be executed in states where their preconditions – a set of facts – evaluate to true. The effect of applying an action is expressed as additions to the world state database and deletions from it. Goal conditions can be expressed as a set of facts to be satisfied. When provided with a planning problem in this format STRIPS searches a sequence of actions that leads to a state satisfying the goal conditions. The STRIPS planning problem definition became the basis of the widely used Planning Domain Definition Language (PDDL) [4] for describing planning problems and many planners require input in a derivative form of STRIPS.

The closed world assumption of STRIPS is not realistic for a robot like ARMAR-III because it does not know the entire world's state but can only ever perceive a small subset thereof. Furthermore exploratory actions may have nondeterministic effects which cannot be expressed using STRIPS. There is no differentiation between knowledge and world state in STRIPS. So discrepancies between the robot's believe of the world's state based on its perception and the world's actual state cannot be modelled using STRIPS. To solve these shortcomings open world planners have been created which can be separated into three categories:

- Conformant planners deal with incomplete world states but can only use deterministic actions.
- Contingent planners understand incomplete world states and nondeterministic sensing or exploratory actions.
- Planning under uncertainty usually deals with probabilistic representations of the world state, modeling uncertainty in perception and execution.

2.1.1 PKS: Planning with Knowledge and Sensing

PKS developed by Petrick and Bacchus ([5], [6], [7]) falls into the second category of contingent planners. It uses a restricted modal logic of knowledge to model actions as changes to the planner's knowledge state, rather than the world state. While PKS models incomplete knowledge it assumes that all planner knowledge is correct and does not model the probability of its correctness. PKS uses a set of databases to store different kinds of knowledge which can be translated to formulas of modal logic representing the planner's knowledge state. Actions indirectly modify the knowledge state through direct modifications of the databases. These are the databases used in PKS:

- K_f contains positive and negative facts and is similar to the state representation in STRIPS. Negative facts need to be specified explicitly because an open world allows facts to be unknown, meaning neither false nor true.
- K_w can be used to express facts which will be known at execution time but are not known at plan time. K_w allows modelling sensing actions like $touch(x)$ which result in either $wet(x)$ or $dry(x)$ at execution time but are only known to result in either one of them at plan time.
- K_v is a specialised version of K_w for storing information about functions returning constants rather than facts resulting in truth values. So it allows the creation of sensing actions for sensors that return values rather than truth values.
- K_x contains information about facts that exclude each other. Every fact in a set of facts in K_x can only be true if all other facts in the same set are false. So if the planner comes to know one of the facts it automatically knows that all the other facts in the same set are false.
- LCW stores local closed world information meaning it allows the robot to assume a fact to be false when it is not contained in K_f. For example if the robot knows all the objects in the fridge then $inFridge(x) \in LCW$, so if $inFridge(orangeJuice)$ is unknown, it must in fact be false.

PKS actions have a set of parameters and just like STRIPS actions they consist of preconditions and effects. The preconditions are a conjunction of queries against the planner's knowledge state which must evaluate to true. Effects add or delete items from any of the PKS databases. Effects can be defined conditionally so that they only take place if a query against the planner's knowledge evaluates to true. Domain specific update rules which are defined just like effects can also modify knowledge. These rules can be used to specify state invariant knowledge as well as state independent conclusions using conditional effects. The operator K denotes that something is known, so an example for a domain specific update rule expressing that objects which break when they are dropped are fragile could be written as $K(dropped(x)) \wedge K(broken(x)) \Rightarrow add(K_f, fragile(x))$.

PKS itself does not deal with unexpected situations after the execution of actions. A plan execution monitor wraps the PKS library for generating plans and compares expectations to actual results continuously. If necessary the plan execution monitor will instruct PKS to generate a new plan starting with the current state as the initial state for the new plan.

2.2 State of the Art in Robotics

2.2.1 Task Planning on the Humanoid Robot HRP2

In [8] Okada et al. describe how they have constructed a three layered architecture integrating a task level planner with scenario recognition (further described by Tokutsu et al. in [9]) and a low-level control layer combined with the robot's vision system. In their work the scenario recognition system is responsible for using visual cues to detect a goal for the planner. Their task planner uses an action and state description similar to that of STRIPS and assumes a closed world. The actual plans are generated using partial order planning [10] which is a planning mechanism avoiding the generation of redundant plans which are permutations of themselves. The vision system generates symbolic representations of world state for the planning system prior to plan execution. The action selected by the task planner is sent to the visual behaviour control system. There the action primitive is disassembled and the vision system performs object recognition as necessary. The behaviour control system then generates a movement to perform the action. After completing the action the vision system is used to verify the correctness of the action's execution. The authors claim their system is very robust and almost never fails however there is no mention of error recovery after failed actions. The closed world planner requires all actions to work exactly as expected.

In [11] Haneda et al. further explain how they use a symbolic task planning system in combination with motion planning to rearrange objects in an interactive dynamics simulator. For this work they used the FF Planning System [12], a forward planning closed world planner. Just like before the motion planning system is used to plan the details of individual movements while the task planner is used to assemble a set of actions which need to be further refined.

2.2.2 Hierarchical Planning with TREX on the Mobile Manipulation Platform PR2

Another approach for integrating high-level planning with low-level motion control has been presented by McGann et al. in [13]. They propose an architecture in which a high-level planning system commands an executive which delegates its work to appropriate subsystems for supplied tasks. Their executive provides coherent access to subsystems and is responsible for recovering in the event of failure. It ensures safety constraints and manages access to shared resources. The TREX system providing this executive

uses the EUROPA[1] temporal planning library to generate its plans. TREX understands hierarchical action definitions allowing more granular details to be planned as well. Parts of the action domain can be modelled explicitly as state machines if that is beneficial over actions automatically generated through planning. TREX represents world state as state variables (complex compound values, not just primitives) which change over the course of a timeline. The planning system fills up future values in the timelines which serve as instructions for execution once the respective point in time is reached. Reactors allow a plan to be synchronised with recent perception so the planner can adjust its plan.

2.3 ICE in Robotics Projects

The Internet Communications Engine (ICE) [14], [15] is a middleware layer which has been used to build component systems for robotics projects including JAST [16] and Orca [17]. Brooks et al. describe their reasons for choosing ICE over alternative middleware solutions like CORBA in [18]. CORBA in particular is considered large, complicated and difficult to use, while ICE is described as more consistent and easier to use, yet providing all necessary features. Other reasons for choosing ICE include the wide variety of supported hardware platforms, its open source license and its suitability for all layers of their system. ICE interfaces are defined using its own interface definition language called Slice[2]. Slice can be mapped into many languages including C++, Java, Python, PHP and Ruby enabling interoperability of components programmed in any of these languages. ICE also provides the means to easily manage and maintain a distributed system where components are spread over multiple machines.

[1] http://opensource.arc.nasa.gov/page/nosa-software-agreement
[2] http://zeroc.com/doc/Ice-3.4.1-IceTouch/manual/Slice.html

3 Integration Concept with Object-Action Complexes

3.1 OACs in PACO+

Resulting from the PACO+ project Krüger et al. proposed a definition of Object-Action complexes [19]. Their summary states:

> PACO-PLUS project views Object-Action complexes as a dynamic (learnable, refinable) and grounded representation that binds objects, actions and attributes associated with an agent in a strong, causal way. They can carry low-level (sensorimotor) as well as high-level (symbolic) information and can thereby be used to join the perception-action space of an agent with its planning-reasoning space. In addtion, they enforce the storage of relevant information for further bootstrapping in episodic memory.

Their work presents a formal definition as well as a number of examples demonstrating the application of Object-Action complexes. One of the examples uses the PKS planning system which is also the basis for the work described in this thesis. According to [19] an Object-Action complex (OAC) is a triplet $(id; T; M)$ where id represents a unique OAC identifier, T is a prediction function of the expected change to the world through the OAC and M is a statistical measure of the OAC's sucess in the past.

The prediction funcion T encodes the system's belief of how the world will change. This definition takes into account that any learned prediction is naturally the subjective understanding of the observer. It is explicitly not an absolute truth which might be available in simulation. Even though T is defined as $T : \mathcal{S} \rightarrow \mathcal{S}$ where $\mathcal{S}$ is the global attribute space, in most cases only a small subspace is relevant to a particular OAC. To simplify the situation one defines an initial attribute space $\mathcal{A}$ and a predicted attribute space $\hat{\mathcal{A}}$ and restricts T to these.

The paper goes on to define system levels for the exeuction of OACs in hierarchical OAC systems. System levels restrict OACs to execution within only one layer of the hierarchy. So even though the system described in this thesis deals with hierarchical OACs it only uses a single system level and emulates hierarchy within that level to allow the reuse of OACs on all hierarchy levels.

3.2 Applying the OAC Concept in Software

One of the goals of OACs is to unify the representation of objects, actions and their combined instantiations across all layers of a cognitive system. As such it is necessary to build a system in which OACs can be executed on the lowest level as well as reasoned about on the highest. The SPOAC (Symbolic Planning Integration through Object-Action Complexes) library described in this thesis implements such a system.

The high-level reasoning system used in this thesis is a symbolic task planner. The planner requires access to perception to reason about world state. The representation for perception used in the context of OACs is that of objects with associated features or properties. Since the planning system works on a symbolic level some form of abstraction needs to happen between raw sensory input and the planner. Hence a processing system transforming raw sensory data into objects with symbolic state representation needs to be part of any system using OACs for both reasoning as well as low-level execution. Here the collection of known objects is considered the short-term memory of the robot because it contains recently perceived information on objects relevant to the task it is currently executing.

For the execution of OACs selected by the planner, a system which instantiates and configures motion control primitives is required. The system must also be able to handle feedback loops from perception to motion control so it has to have access to both the perception as well as the execution layer. This system is called an executive and is the central authority on the robot, which manages the other parts described. It receives high-level instructions from the planner. Sometimes the executive is called a central executive agent (CEA). The executive can ignore the planning system and make its own decisions which is useful for implementing intuitive behaviours similar to instincts.

Information about known OACs must be stored for access by the executive and the planning system. A learning system for OACs may create new OACs based on experience of the executive. The storage container for OACs is called long-term memory in this thesis because it provides the executive and planner with learned kownledge of its own abilities.

3.2.1 OACs in PKS

For PKS an Object-Action complex consists of a name, a list of parameters, preconditions and expected effects. Unfortunately PKS typically uses actions and instantiated actions to refer to OACs which is the more common terminology in planning systems. So attention needs to be paid to terminology in the context of PKS and OACs. Differing from the definition of OACs described in subsection 3.1 PKS models change to a robot's knowledge rather than change to the actual world state. An OAC is instantiated with parameters which are used in the abstract descriptions of precondition and effects. The parameters are constants and typically represent an object. Properties of objects are accessible as predicates and functions of arbitrary arity defined over the object identifiers.

3.2.2 Action and Object Representations

Perception starts at raw sensory information which needs to be further processed in order to enable reasoning on a more abstract level. The processing step consists of the extraction of features on one side and the grouping of features and analyzing their

relationships on the other side. Groups of extracted features and abstracted knowledge are the basis for objects. In combination with actions their actual object behaviours emerge. The actions define what information structure they require of an object. If an object is understood this way, it makes sense for an object to consist of any number of features and relationships collected in an arbitrary structure rather than a set of features in a fixed predefined structure.

If an object has no strictly defined set of properties and no strict structure it becomes possible to understand the processing of signals as continuous transformation of singals into objects through a chain of processors. The SPOAC library calls these processors perception handlers. Exposing objects to higher level processes becomes a simple task in this architecture because an export processor can be attached to the chain. The export processor can even filter or transform the information contained in the object before exposing it to another system if necessary.

In the SPOAC library, an action is essentially any piece of code with a set of parameters which transforms the real world when executed. An action is parameterised with objects from short-term memory and arbitrary further configuration data retrieved from long-term memory. Configuration in long-term memory can either apply to all object parameterisations or to any chosen subset of object parameterisations. For example a trajectory for looking around a workspace might be fine tuned by a developer, but will probably not differ based on the workspace. On the other hand an action like grasping might require configuration values very specific to each object like orientation and offsets for successful grasps. These values are not directly perceived but typically either inferred from pervious experience or when no learning mechanisms are available hardcoded by a developer.

Now an OAC can be understood as a reference to an action and a sequence of objects used to parameterise the action. Long-term memory provides the means to look up the action and its necessary configuration for a given OAC name and a corresponding sequence of objects. As such the OAC comes from long-term memory and needs to be instantiated with information from short-term memory in the form of objects to yield an executable action. At the same time reasoning about OACs needs to be possible in the planning system. Consequently, the OAC stored in long-term memory must additionally contain a definition of the requirements it imposes on parameter objects and the expected effects of executing the resulting action. Separating this information from the information required to instantiate an OAC has multiple advantages. The separation makes it possible to execute OACs that the system has not yet understood on an abstract level and cannot meaningfully reason about. This is particularly important for learning systems like the rule learning system described in [20] and [21]. which can learn preconditions and effects of OACs from gathered experience while executing OACs. Without exploration, meaning the execution of OACs without preconditions and known effects, it would be impossible to draw such conclusions. The separation of information necessary for execution from information required for reasoning also makes it possible to reason about OACs that might not yet be executable for technical reasons or to simply

test extensions to a high-level system using OACs for which there are no corresponding features in the execution code yet.

According to Krüger et al. [19] OACs contain a statistical measure to store information on the reliability of the OAC's execution. SPOAC does not yet include such a measure in its OACs. One major problem with implementing a statistical measure of reliability is that success needs to be measurable. The lack of sensors and sufficient image processing on ARMAR-III often makes it impossible to tell if the expected state was reached. Apart from that it is questionnable whether this measure should really be contained within the OAC itself. It would make learning processes that consider multiple OACs more complex if there is only statistical information about individual OACs. Instead, executed OACs and their results should be recorded in what could be called episodic memory. This addition to long-term memory would enable learning mechanisms to draw conclusions about reliability of individual OACs, sequences of OACs and interaction between OACs. Currently no learning mechanism is integrated in SPOAC but experiments with a combination of PKS and RULESYS have been performed on ARMAR-III.

3.2.3 An Executive for Object-Action Complexes

An executive for OACs does not need to differ significantly from other executives. Its implementation however is greatly simplified if the same concept — OACs — is used consitently across all involved components. OACs provide a consitent method for accessing short-term memory information in actions. OACs are a concept that cuts through all layers, applying to the most abstract high-level components as well as to low-level components dealing directly with sensors and actuators.

The executive implemented for this thesis makes use of three principal subsystems:

- the perception & processing subsystem dealing with short-term memory,
- the long-term memory subsystem providing knowledge about OACs,
- the activity control subsystem which is responsible for providing the executive with proposals on what it should do.

The perception & processing subsystem is responsible for populating short-term memory with objects and keeping their data synchronised with the robot's perception. Here an *object* simply refers to a group of related sensory information and extracted features as well as relationships to other objects. The system generating the data consists of a set of chained processors called *perception handlers*. Some of them extract information directly from the robot's sensors or low-level systems providing an abstraction of them while other *perception handlers* further process the data made available by the handlers of the former kind. An example would be a *perception handler* which uses the robot's vision library. It would create objects in short-term memory for every object the library recognises. Another *perception handler* would then analyse the relative positions of

all objects in short-term memory. The analysis would result in relationship properties like `onTopOf` and `under` to be stored in the same objects in short-term memory. All other systems can now reason about the objects using the extended properties instead of dealing with the details of extracting this abstract information from raw sensory data.

The long-term memory subsystem can be understood as a database for OACs. It has to be able to provide the planning system with a complete definition of the robot's capabilities and the executive with information on how to instantiate and execute an OAC. OAC learning mechanisms need to have access to the long-term memory to add new OACs and update existing ones.

The activity control subsystem provides the executive with instructions. While the executive decides what will really be exeucted on the robot, it needs to receive instructions that it can choose from. For this purpose the executive has a queue of OACs to be executed in order. This queue is accessible to activity controllers which can enqueue new OACs, remove OACs or inject OACs in the beginning for immediate execution. Activity controllers are notified about changes to execution state, for example the beginning or end of an OAC's execution. When the executive's OAC queue is empty it sends a message to all activity controllers requesting new instructions.

The planning system is integrated with the executive through an activity controller. The activity controller exposes the messages it receives through a network interface the planner is connected to. After successful execution of an OAC, the planner sends the next OAC from its plan to the activity controller. The activity controller then enqueues this OAC. At the same time a special perception handler provides the planner with updates on the state of short-term memory so it can react to unexpected situations and verify the success of its plan. If an unexpected situation is encountered and the plan execution monitor decides the plan needs to be changed, the planner can instruct the activity controller to stop the current OAC and enqueue a different one.

Activity controllers can also be used to implement instinctive behaviour. Such a controller would inject a new OAC in response to a perception handler perceiving a particular piece of information. Other possible uses for activity controllers include optimisation of OAC sequences based on past experience or spatial reasoning which the planner did not perform.

To execute an OAC, it is instantiated with objects from short-term memory and data from long-term memory. The resulting instance then runs inside the ARMAR control system, embedded in an ARMAR scenario. It is able to make use of preexisting skills and tasks allowing a simplified transition from the previous system to SPOAC. Because skills and tasks also handle perception short-term memory updates through perception handlers can be temporarily disabled or slowed down to improve the main ARMAR-III control loop speed.

4 Implementation of the OAC-based Integration Concept

4.1 Software Architecture

4.1.1 Conceptual View

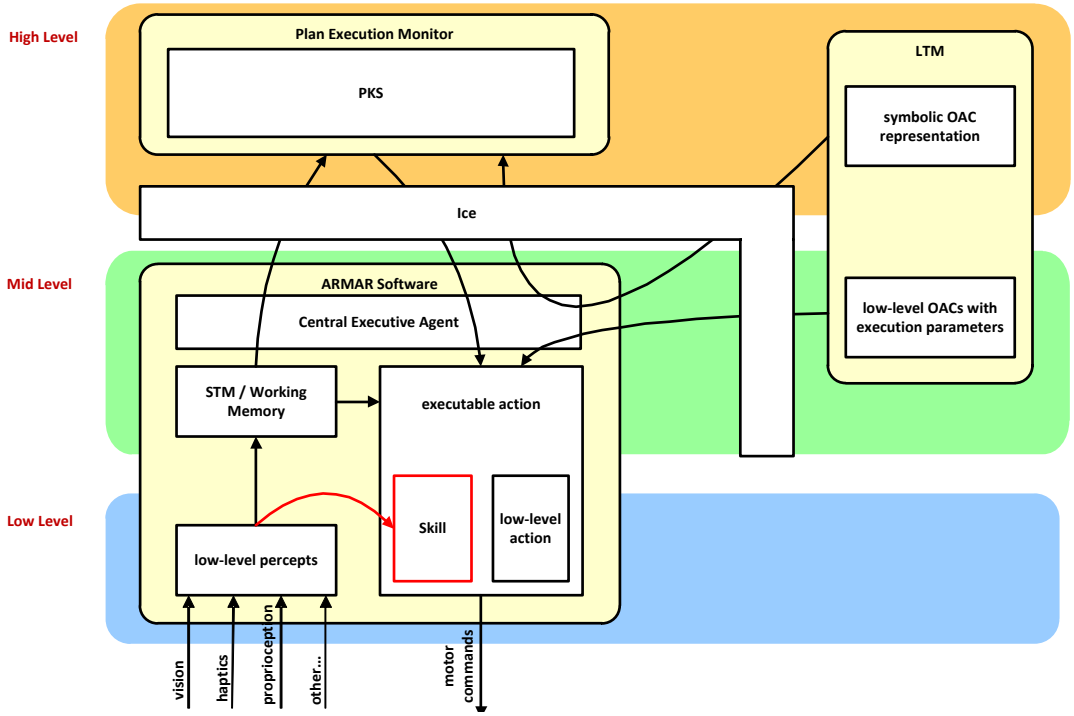

Figure 1: Component overview within the cognitive central architecture developed in PACO-PLUS

The system architecture depicted in Figure 1 follows directly from the solution concepts described in the previous section. Embedded in the cognitive architecture developed in PACO-PLUS consisting of three layers, the components are:

- the high-level symbolic planning system PKS,
- mid-level and high-level long-term memory storing OACs containing symbolic and low-level information,
- mid-level short-term memory storing object representations,
- the mid-level Central Executive Agent responsible for coordination,
- low-level perception processing,
- low-level skills and actions.

These components are connected using the ICE middleware. Short-term memory and the Central Exectutive Agent run within a single process to provide legacy skills and tasks with direct access to perception (marked in red in Figure 1).

4.1.2 Technology Used in the Implementation

The SPOAC library was developed in C++ using object oriented programming and test driven development. The employed build system CMake [22] comes with a test runner called CTest which is used to run test programmes. The tests themselves make use of the Boost Unit Test Framework [23]. API documentation is generated from source code comments using Doxygen [24].

An object oriented design pattern that has been applied to all components of the system is dependency injection [25]. Dependency Injection is meant to decouple dependent components of a system. The problem of coupled components is easily illustrated with an example. Assume that class A delegates some of its work to another class B. The naive solution is to create an instance of B in A's constructor. But if you now wish to replace B with another class C – for example a dummy class to test the behaviour of A in isolation from B – an adjustment of the instantiation code in A's constructor is required. To avoid this modification you need to decouple A from B. This can be done by separating the dependency resolution and instantiation from the implementation of A's behaviour. Instead of creating a new instance, A should receive an instance of B in its constructor or through a setter method. The instance it receives needs to be created outside of A. Now A can also receive an instance of C if it conforms to the same interface. To facilitate the use of dependency injection a lightweight dependency injection container called `DependencyManager` is a part of the SPOAC library. It takes care of resolving the dependencies of classes and creating the objects which are required to instantiate a class.

SPOAC's prefered format for encoding various objects for storage in files or serialisation is the JavaScript Object Notation (JSON) originating from JavaScript/ECMAScript [26]. JSON is a lightweight data-interchange format claimed to be easy to read and write for humans. This makes it suitable for OACs which are hand-crafted rather than generated through learning mechanisms and for the manual creation of scenario definitions for the demonstration of a particular set of OACs. JSON's minimalistic syntax makes it easy for machines to parse and generate JSON documents. Apart from the primitive types string, number, bool and null JSON knows the complex datatypes array and object. An array is an ordered list of values while an object is an unordered mapping of keys to values [27].

4.2 PKS: Planning with Knowledge and Sensing

PKS consits of two main parts: the library for planning itself and an application called Plan Execution Monitor which provides different ways of interfacing with the planning subsystem. The Plan Execution Monitor provides both a Command Line Interface (CLI) as well as network access through ICE. Apart from controlling the planner based on input received through the network or command line interfaces it also takes care of monitoring the consistency of the plan's expectations with percepted results. If a mismatch is found it can decide to generate a new plan leading to the goal from the current situation. There are several policies available for identifying matches of predicted and observed state:

- *equal*: The states only match if they are exactly identical.
- *subset-observed*: The states match if the observed state is a subset of the predicted state.
- *subset-predicted*: The states match if the predicated state is a subset of the observed state.
- *preconds-true*: The states match as long as the next planned action's preconditions are satisfied by the observed state.
- *any-equal*: Reports states as matching if any of the first four methods result in a match.
- *any-unequal*: Reports a state mismatch if any of the first four methods result in a match.
- *always-equal*: Always reports states as equal.
- *always-unequal*: Always reports states as unequal.

The default behaviour is *preconds-true* with which replanning only takes place once an action's preconditions are no longer satisfied.

4.3 Short-Term Memory

Short-term memory is understood as a model of the real world in SPOAC. Short-term memory contains a set of objects corresponding to objects in the real world. It is populated by PerceptionHandlers which add and modify objects. Figure 2 depicts the class model of short-term memory and the perception subsystem with three examples of PerceptionHandlers. Information decay is made possible through timestamping modifications to objects.

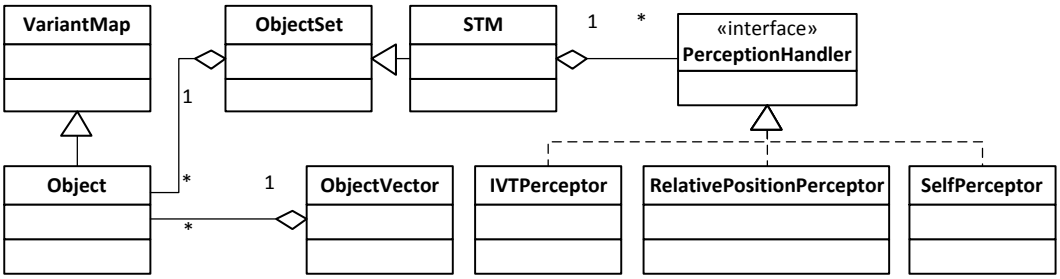

Figure 2: UML class diagram of short-term memory

An individual object stores key-value pairs. The values can be of a variety of types including references to other objects in short-term memory. This allows expressing complex situations like objects composed of other objects which can either be seen as a whole or as individual parts. An example of an object in short-term memory is shown in Listing 1.

Listing 1: JSON encoded object representation of a green cup

```
{
    "id": "cup23",
    "ivt.name": "cup",
    "ivt.color": "green",
    "ivt.world_point.x": 23.5,
    "ivt.world_point.y": 42.7,
    "ivt.world_point.z": 17.3,
    "isObject": true,
    "objLocation": {"sideboard": true},
    "onSurface": true
}
```

4.4 Long-Term Memory

The long-term memory service is essentially a database for OACs and scenario definitions. It can extract action definitions for the planner from the OACs as well as selecting an action configuration for the execution of an OAC when provided with short-term memory object information.

Both OACs and scenarios are encoded and stored using the JavaScript Object Notation (JSON). Presently JSON objects are stored in the file system, one file per OAC or scenario. If this ever became a performance bottleneck or if further processing of the OACs became necessary in the future the documents could be stored in a variety of databases available for storing arbitrary JSON documents.

Listing 2: JSON encoded OAC in Long-Term Memory

```
{
    "name": "grasp",
    "params": ["x", "l", "h"],
    "match":[{"x": {"ivt.name": "spruehflasche"},
              "action": "BoxGrasp"                },
             {"x": {"ivt.name": "burti"},
              "action": "BoxGrasp"                }],
    "action": "Grasp",
    "precondition": "
            K(isObject(?x)) &
            K(isLocation(?l)) &
            ...",
    "effect": "
            add(Kf, inHand(?x, ?h)),
            del(Kf, handEmpty(?h)),
            ..."
}
```

The example OAC in Listing 2 has a `match` section. This section allows to define a set of rules that the OAC's object parameters must match for the specified action or configuration values to apply. The main `action` section will be used if none of the rules apply. This feature is inspired by the pattern matching mechanisms common in functional programming languages. Functional programming languages use the mechanism to partially define functions for parameters matching the pattern [28] [29]. A set of these partial definitions makes up the entire function. The matching section can be used to define patterns for any combination of the parameter objects and any number of their properties. For more complicated patterns expressions are possible as well. The robot's state is available through the special object name `self`. A few examples can be found in Listing 3.

Listing 3: JSON encoded OAC in Long-Term Memory

```
Placing items into appliances, special code for dishwashers

    "params": ["what", "into"],
    "match":[{"into": {"appliance": "dishwasher"},
              "action": "PlaceIntoDishwasher"     }],
    "action": "PutInto",

Run, but walk slowly when the battery is nearly empty

    "params": ["destination"],
    "match":[{"self": {"battery": {"cmp": "<", "val": 10}},
              "action": "Walk"     }],
    "action": "Run",
```

The scenarios which are also stored as JSON documents in long-term memory contain a set of action controller names and a set of perception handlers to set up the Central Executive Agent for a scenario. Additionally scenarios contain a set of predicates and functions to be exposed to the planning system. This is necessary because there is no mechanism for learning the importance of different object properties integrated into the system yet. The planner would be overwhelmed if all raw data was given to it. The scenario allows limiting the number of OACs available for execution and can contain predefined goals. These goals are used when the language recognition system on ARMAR receives a new instruction from the robot's operator.

4.5 Implementation of the Central Executive Agent

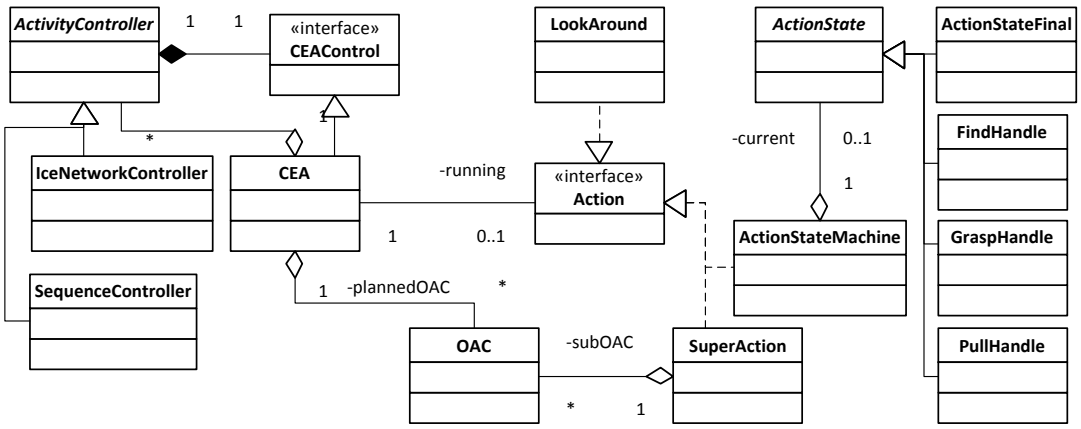

Figure 3: UML class diagram of the executive

The CEA is embedded in the robot's main control loop. So it is important that activities that might take a longer time are handled asynchronously. At present the majority of activity controllers and perception handlers operate synchronously because the current hardware available on ARMAR-III does not allow for truly parallel execution of high-level software. The activity controllers and perception handlers which provide the network interface using ICE however work asynchronously. So the CEA makes use of events, notifying activity controllers and perception handlers when necessary. These can then either block the process and perform their work or send off an asynchronous request. Either way the reply is sent as a separate message to the CEA because the events do not return any values.

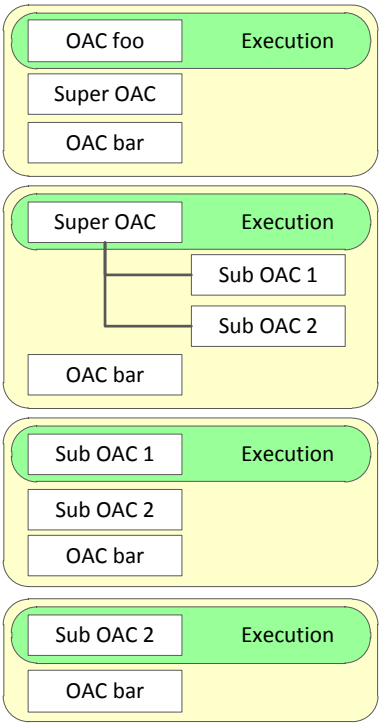

Figure 4: The Central Executive Agent's OAC queue

Another requirement for the CEA was the reuse of existing skills and tasks on executing OACs. This was easily achieved by allowing actions to run arbitrary code. They can then simply instantiate the existing skills or tasks and delegate the work to them. So in these cases an action configures a skill or task depending on which parameter objects it was passed and which configuration data it received from the OAC definition in long-term memory.

The OAC definitions from long-term memory do not provide any obvious mechanism for defining hierarchical OACs, OACs that delegate

their work to a sequence of other OACs. Instead OACs can reference the `SuperOAC` action and use the configuration data from long-term memory to select a sequence of sub-OACs to be executed in order. Any other action is also able to delegate work to an existing OAC. This is realised through a yield method of the CEA. An action can yield to an OAC which will then be injected in front of the OAC queue and started by the next run of the control loop. The previously running action and its corresponding OAC will continue their execution after the sub-OAC is finished. This means that the CEA really only deals with a linear queue of OACs but at the same time OACs can have a complex hierarchy.

Activity controllers are notified about OACs which are about to be started. This allows them to manipulate the contents of the queue prior to actual execution. Examples for such activity controllers would be ones replacing OACs based on past experience or ones that inject OACs to improve reliability based on spatial reasoning. These controllers can do exactly the same for OACs which were started as sub-OACs by a super OAC because the CEA does not itself differentiate between the two as explained in the previous paragraph.

4.6 Component Interfaces

Each component defines multiple interfaces for communication. The first interface allows other components to request information on the component's current state. The names of these interfaces are suffixed with *Info*. The second interface is used to receive information from other components. To allow listeners to intercept such communication and react to it, all communications of the second kind are broadcast rather than being directly transmitted to a particular component. These interfaces are suffixed with *Topic* indicating their use as a broadcast channel. The third and optional interface is suffixed with *InfoTopic* and is used to broadcast internal state transitions that might be of interest to other components.

The separation of these communication mechanisms allows other processes to query components for their current state on start up and then follow all further transmissions to react to them. A graphical user interface (GUI) can serve as an example of such a process. When the user launches the GUI the *Info* interfaces are used to retrieve the current state of the system to be displayed. The GUI can now listen to all broadcast messages to update the user interface when changes occur.

The interfaces are split into 3 different modules: `SymbolicExecution`, `Planning` and `LTM`. The idea is to make parts of the system independent, so one can use the SPOAC library without the need to use the `Planning` module for example.

In the following sections C++ notation is used to describe the details of each interface, the methods it defines and their paramters and return values.

4.6.1 SymbolicExecution::CEAControllerTopic

The Central Executive Agent listens for instructions that other components broadcast to this topic.

- `startAction` *Parameters:* Action a
  Appends the given OAC into the execution queue on the robot.
- `injectAction` *Parameters:* Action a
  Injects an action in front of the execution queue for immediate execution.
- `stopAction` *Parameters:* Action a
  Tells the Central Executive Agent to abort the given OAC's execution.
- `taskComplete`
  Informs the Central Executive Agent of the completion of a task so that it can notify the user.
- `pause`
  Instructs the Central Executive Agent to pause execution of the current OAC.
- `unpause`
  Allows the Central Executive Agent to continue executing the current OAC after it was paused using the previously mentioned request.
- `reset`
  Instructs the Central Executive Agent to abort the execution of the current OAC and clear its short term memory to restart the scenario. This is mostly useful for sending a reset instruction from a GUI during debugging.

4.6.2 SymbolicExecution::CEAInfoTopic

The Central Executive Agent uses this topic to broadcast messages when its state changes.

- `actionStarting` *Parameters:* Action a
  After the Central Executive Agent has selected the next OAC from its queue for execution this notification is sent out to give ActivityContollers a chance to prevent its execution in order to inject a different OAC.
- `actionStarted` *Parameters:* Action a
  If an OAC's execution was not cancelled after sending the `actionStarting` message, this notification is sent out to signal the beginning of its actual execution.
- `actionFinished` *Parameters:* Action a
  This notification is sent after an OAC has finished execution.

- `actionStopped` *Parameters:* Action a
  When an OAC has been stopped and cancelled during its execution this message is broadcast.
- `taskCompleted`
  After the completion of a task has been signalled through a message to the `CEA-ControllerTopic` this message is sent to notify all listeners on this channel.
- `paused`
  The message is sent when execution of the current OAC has been paused.
- `unpaused`
  Signals the continuation of OAC execution after a pause.
- `resetted`
  When the robot is reset this message is broadcast to allow all listeners to reset their state as well.

4.6.3 SymbolicExecution::CEAInfo

The Central Executive Agent's state is accessible through this interface.

- `getCurrentAction` *Returns:* Action
  This query returns the OAC currently being executed by the CEA.
- `isPaused` *Returns:* Boolean
  This accessor returns whether the CEA is currently paused.

4.6.4 Planning::PlanControllerTopic

This interface is used to configure the planner and update its knowledge of the world's state.

- `setSymbolDefinitions` *Parameters:* SymbolDefinition symbols
  Overwrites the current planner symbol definitions with the new given one. A symbol definition consists of predicate definitions, function definitions, type definitions and constant definitions. Constants represent object identities in short term memory.
- `setActionDefinitions` *Parameters:* ActionDefinitionList actions
  Overwrites the current planner action definitions.
- `setGoal` *Parameters:* Goal g
  Sets a goal for the planner in the form of a boolean PKS expression.
- `updateState` *Parameters:* StateUpdate update

After changes to information in short term memory this method is used to transmit the differences in knowledge so the planner can react to perception.

- `actionFinished` *Parameters:* Action a
  Informs the planner when an action has been finished.
- `startPlan`
  Instructs the planner to start planning toward the specified goal.
- `stopPlan`
  Resets the planner to start over when instructed to start planning.
- `resetState`
  When short term memory is reinitialised this is used to reset the planner's knowledge to an empty state as well.

4.6.5 Planning::PlannerInfo

The planner's current state is accessible through this interface.

- `getCurrentAction` *Returns:* Action
  Returns the action the planner believes should be executed next.
- `getActionDefinitions` *Returns:* ActionDefinitionList
  Returns the complete action definitions previously set through the `PlanControllerTopic`.
- `getSymbols` *Returns:* SymbolDefinition
  Returns the symbol defintions previously set through the `PlanControllerTopic`.
- `getGoal` *Returns:* Goal
  Retrieves the goal expression specified in the previous call to `setGoal` in the `PlanControllerTopic`.
- `getNextActions` *Parameters:* int max *Returns:* ActionList
  Returns at most *max* elements of the sequence of currently planned actions.

4.6.6 LTM::LTM

This interface is used to access the information stored in the long-term memory database. Since its only purpose is to provide information at this point it is not meant for broadcasting. Other components can connect to the long-term memory directly to request information.

- `getScenario` *Parameters:* String scenario *Returns:* Scenario
  Retrieves a scenario definition of the given name.

- `getAction` *Parameters:* String oac *Returns:* ActionDefinition
  Retrieves a planning action definition for the OAC of the given name.
- `getActionConfig` *Parameters:* OAC oacInstance *Returns:* ActionConfig
  Retrieves a detailed configuration for the execution of an OAC by the CEA. The `oacInstance` contains the actual short term memory object data.

5 Application of SPOAC to ARMAR-III

The goal of applying SPOAC to ARMAR-III is to run an experiment in which ARMAR-III is located in the kitchen of our lab and is given the task to serve a carton of apple juice and a cup. The apple juice is located in the fridge and the blue cup is standing on the stove. In order to achieve its goal ARMAR-III must open and close the fridge, pick up the apple juice and the cup as well as carrying them to the sideboard where they shall be placed for consumption.

Opening and closing the fridge poses several physical problems for ARMAR-III. The fridge's door opens toward the right so the right hand is used to grasp the handle and pull. However the fridge is located next to a wall so ARMAR-III cannot move to the right and its phyiscal properties prohibit it from fully opening the door with the right hand without moving to the side. So in order to finish opening the door ARMAR-III must use its left hand to push the door completely open. Consequently ARMAR-III cannot keep an object in one hand while opening the door with the other. However, ARMAR-III able to pass an object from one hand to the other. So the planner will have to instruct the robot to pass the object in its left hand to the right hand when opening a door with an object in its hand. Similar restrictions apply to removing an object from the fridge and closing it. Both actions can only be performed with the right arm. So if the fridge has to be closed after removing an object from it, the robot needs to pass the object to the other hand first.

Based on the results of the integration of PKS at the University of Southern Denmark in the PACO+ project [30] and the described limitations, the domain definition explained in this chapter was created. The follow sections describe the necessary properties and OACs, their meaning and implementation as well as constraints and translation to PKS symbols.

5.1 Properties

This section contains a list of all properties which are exposed to the planning system. This does not include properties of lower abstraction, such as coordinates for objects relative to the robot or exact values of orientation. Instead these properties either define the PKS type of an object which is used to limit the scope of PKS action definitions or abstract relative knowledge like object A is `onTopOf` object B.

	isObject
PerceptionHandler	`IVTPerceptor`
PKS Predicate	`isObject(?const)`

The ambiguously named property `isObject` actually describes if an object can be grasped. Consequenly it is not true for all objects in short-term memory. The property's value is boolean and thus represented as a predicate of arity 1 in PKS. Since only objects recognisable through the IVT vision library[3] were considered for grasping in the SPOAC experiments the only `PerceptionHandler` setting this flag is the `IVTPerceptor`. For example cups and boxes have `isObject` set, while it is false for locations like the sideboard or the stove.

	isHand
PerceptionHandler	`SelfPerceptor`
PKS Predicate	`isHand(?obj)`

Because PKS' new type system is not yet used this property is a boolean flag which is only set on the two short-term memory objects representing the robot's own hands. The flag is set on system startup by the `SelfPerceptor`.

	inHand
PerceptionHandler	None (assumption only)
PKS Predicate	`inHand(?obj1, ?hand)`

This property contains a link to another object, describing a relationship between two objects. The PKS representation is a predicate of arity 2. The property is set on a graspable object which has the `isObject` property set. The linked object must be a hand object with the `isHand` property. Because IVT is not currently able to recognise objects held in one of ARMAR-III's hands this property is automatically set in the `Grasp` action. ARMAR-III's hand has no sensors that would allow it to verify the assumption made after attempting to grasp an object. It is not sufficient to verify that the object has left its original position since it could have been dropped on the floor, moving it out of the robot's field of vision.

	handEmpty
PerceptionHandler	None (assumption only)
PKS Predicate	`handEmpty(?hand)`

The `handEmpty` property is set on a hand object and is true when there is no object with

[3] http://ivt.sourceforge.net/

an `inHand` property linking to this hand. Analogous to the `inHand` property there is no way for the robot to actually perceive whether this property is true or false so it is also an unverifiable assumption made after grasping or placing objects.

	isLocation
PerceptionHandler	`LocationPerceptor`
PKS Predicate	`isLocation(?obj)`

Similar to `isObject` and `isHand` this property is a replacement for the type system. Locations identify the different workspaces in between of which ARMAR-III can move around. The `LocationPerceptor` enters all known workspaces into short-term memory as necessary and sets this property on them.

	objLocation
PerceptionHandler	`IVTPerceptor`
PKS Predicate	`objLocation(?obj, ?l)`

This relationship property stores the location a graspable object was last seen at. The `IVTPerceptor` simply enters a reference to the current location of the robot, when the object is recognised. So rather than actually locating the object it stores which workspace the object could last be seen from and would thus likely be the best area to manipulate the object.

	onTopOf
PerceptionHandler	`RelativePositionPerceptor`
PKS Predicate	`onTopOf(?topObj, ?bottomObj)`

The `RelativePositionPerceptor` further processes information retrieved from IVT throught the `IVTPerceptor`. Based on object sizes and positions it estimates which objects stand on top of each other. This relationship is then stored in the property called `onTopOf`. This specifically excludes non-graspable objects, like the stove or the sideboard.

	onSurface
PerceptionHandler	`RelativePositionPerceptor`
PKS Predicate	`onSurface(?obj)`

The `RelativePositionPerceptor` sets the onSurface property for all graspable objects which are not `onTopOf` another graspable object. Meaning they stand on a surface such as the sideboard, the stove or on a board in the fridge.

	handedOver
PerceptionHandler	None (assumption only)
PKS Predicate	`handedOver(?obj)`

This property solely exists to make the action of handing over an object to a human being verifiable for PKS. Unfortunately the vision library is not actually able to detect whether the human being in front of ARMAR-III really grasped the object that was handed over. So this property is simply assumed to be true after the handing over action has been executed.

	objOpen
PerceptionHandler	None (assumption only)
PKS Predicate	`objOpen(?obj)`

The objOpen property currently only refers to the fridge, dishwasher and cupboard doors. There is no code for detecting the state of these doors yet so all of them are assumed to be closed in the beginning and opening and closing actions set the objOpen flag to true. This means that plans will fail when one of these actions is unsuccessful because the planner will not be made aware of the discrapancy between reality and its model in short-term memory.

	objPartialOpen
PerceptionHandler	None (assumption only)
PKS Predicate	`objPartialOpen(?obj)`

The fridge door cannot be opened with a single motion primitive by one hand as has been explained in the previous section. To allow the robot to pass an object from one

hand to the other before it continues opening the door with the other hand, this property is used to represent the partially open state of the fridge door in the meantime.

	robotLocation
PerceptionHandler	`SelfPerceptor`
PKS Function	`robotLocation()`

This property is defined on the robot object itself by the `SelfPerceptor`. It is updated when the robot moves to a different workspace. Unlike all previous properties this one is represented as a function of arity 0 in PKS.

5.2 OACs

	close
Parameters	l, h
Action	`CloseDoor`
PKS Precondition	

	`K(isLocation(?l)) &`
	`K(isHand(?h)) &`
	`(K(?l = Fridge) & K(?h = rightHand)) &`
	`K(robotLocation = ?l) &`
	`(K(objOpen(?l)) | K(objPartialOpen(?l))) &`
	`K(handEmpty(?h))`

PKS Effect	
	`del(Kf, objOpen(?l)),` `del(Kf, objPartialOpen(?l))`

This OAC uses the `CloseDoor` action which in turn uses the `CloseDoor` task already available on ARMAR-III to close the fridge door with ARMAR-III's right hand. The door can also be closed when it is only partially opened. This action requires no further configuration in the OAC because the majority of necessary data is still hardcoded into the `CloseDoor` task and the skills it uses, namely the `GraspHandle`, `MoveArm`, `ImpedenceVel`, `ImpedenceVisionVel` and `ExecuteTrajectory` skills.

	grasp
Parameters	x, l, h
Action	x.ivt.name ∈ {sprueflasche, burti} ⇒ `BoxGrasp` else `Grasp`

PKS Precondition	`K(isObject(?x)) &`
	`K(isLocation(?l)) &`
	`(K(?l = Sideboard) | K(?l = Stove)) &`
	`K(isHand(?h)) &`
	`K(robotLocation = ?l) &`
	`K(objLocation(?x, ?l)) &`
	`K(handEmpty(?h)) &`
	`K(onSurface(?x)) &`
	`(forallK(?y) !K(onTopOf(?y, ?x)))`

PKS Effect	`add(Kf, inHand(?x, ?h)),`
	`del(Kf, handEmpty(?h)),`
	`del(Kf, objLocation(?x, ?l)),`
	`del(Kf, onSurface(?x))`

The grasp OAC is responsible for grasping objects recognised by IVT. The `Grasp` action uses the `VisualGrasp` skill to grasp an object using visual servoing. No further configuration is used for this skill since it looks up the correct grasp orientation and approach vector itself. The two objects sprueflasche and burti however are not contained in the `VisualGrasp` object database and can thus only be grasped with the `BoxGrasp` action which uses the `BoxGrasping` skill that implements a mechanism for grasping unknown objects. To accomplish this separation pattern matching on the parameter object is used to differentiate based on the IVT database name as described in subsection 4.4.

	grasp-from
Parameters	x, l, h
Action	`Grasp`
PKS Precondition	

	`K(isObject(?x)) &`
	`K(isLocation(?l)) &`
	`K(isHand(?h)) &`
	`(K(?l = Sideboard) | K(?l = Stove)) &`
	`K(robotLocation = ?l) &`
	`(existsK(?y) K(onTopOf(?x, ?y))) &`
	`(forallK(?z) !K(onTopOf(?z, ?x))) &`
	`K(objLocation(?x, ?l)) &`
	`K(handEmpty(?h))`

PKS Effect	

	`add(Kf, inHand(?x, ?h)),`
	`del(Kf, handEmpty(?h)),`
	`(forallK(?y) (`
	`  K(onTopOf(?x, ?y)) =>`
	`  del(Kf, onTopOf(?x, ?y))))`

The `grasp-from` OAC uses the `Grasp` action just like the grasp OAC. However its PKS precondition and effect differ from grasp in so far as that this OAC is specifically meant for removing objects which stand on top of other graspable objects.

	hand-over
Parameters	x, l, h
Action	`HandOverObject`
PKS Precondition	

	`K(isObject(?x)) &`
	`K(isLocation(?l)) &`
	`K(isHand(?h)) &`
	`K(?l = DeliverNode) &`
	`K(inHand(?x, ?h)) &`
	`K(robotLocation = ?l)`

PKS Effect	

	`add(Kf, handedOver(?x)),`
	`del(Kf, inHand(?x, ?h)),`
	`add(Kf, handEmpty(?h))`

When ARMAR-III is instructed to bring an object to its operator it needs to hand that object over. It does so using the `HandOverObject` action which in turn uses the `HandOverSkill` which measures wrist forces to let go of the object when the operator pulls on it. This handing over procedure is always performed at the location called `DeliverNode` where the operator is typically located.

	look-around
Parameters	None
Action	`LookAround`

To see all objects reachable in one of ARMAR-III's workspaces its head has to be moved around a bit to increase the field of vision. The `look-around` OAC does not have a definition for PKS because PKS is not yet able to reason about finding objects that it is not aware of. Instead this OAC is used by the `SearchObjectsController` which injects it into the CEA's queue after the robot reaches a different workspace location. It is also included in the `open-complete` OAC so that ARMAR-III examines the contents of a cupboard after opening it.

	move
Parameters	l1, l2
Action	`GoTo`
PKS Precondition	

	`K(isLocation(?l1)) &`
	`K(isLocation(?l2)) &`
	`K(?l1 != ?l2) &`
	`K(robotLocation = ?l1)`

PKS Effect	
	`add(Kf, robotLocation = ?l2)`

The move OAC takes the robot from one workspace location to another using the `GoTo` action which does not make use of any preexisting ARMAR-III skills.

	open-partial
Parameters	l, h
Action	`OpenDoor`
Configuration	`partial`
PKS Precondition	

	`K(isLocation(?l)) &`
	`K(?l = Fridge) &`
	`K(isHand(?h)) &`
	`K(?h = rightHand) &`
	`K(robotLocation = ?l) &`
	`!K(objOpen(?l)) &`
	`!K(objPartialOpen(?l)) &`
	`K(handEmpty(?h))`

PKS Effect	
	`add(Kf, objPartialOpen(?l))`

The `open-partial` OAC applies only to the fridge door which needs to be opened partially with the right hand before being opened completely with the left hand.

	open-complete-not-looking
Parameters	l, h
Action	`OpenDoor`
Configuration	`complete`

This OAC is responsible for opening fridge, cupboard and dishwasher doors using the `OpenDoor` action. The fridge can only be completely opened using this OAC if it has already partially opened using `open-partial`. The `OpenDoor` action makes use of the existing `OpenDoor` and `OpenDishwasher` tasks which contain hardcoded configuration. The name of this OAC was chosen to distinguish it from the `open-complete` super OAC used by the planner which is described in the next section.

	open-complete
Parameters	l, h
Action	`SuperOAC`
Configuration	`[open-complete-not-looking(l, h), look-around()]`
PKS Precondition	

	`K(isLocation(?l)) &`
	`K(?l = Fridge) &`
	`K(isHand(?h)) &`
	`K(?h = leftHand) &`
	`K(robotLocation = ?l) &`
	`!K(objOpen(?l)) &`
	`K(objPartialOpen(?l)) &`
	`K(handEmpty(?h))`

PKS Effect	
	`add(Kf, objOpen(?l)),` `del(Kf, objPartialOpen(?l))`

The `open-complete` OAC is both responsible for opening the fridge completely after it has been partially opened and opening the dishwasher and cupboards. This OAC is a super OAC, meaning it delegates its work to a sequence of other OACs. After opening a door it yields to the `look-around` OAC so that ARMAR-III inspects the opened container for objects it might not have previously known about.

	pass-object
Parameters	x, h1, h2
Action	`PassObject`
PKS Precondition	

	`K(isObject(?x)) &`
	`K(isHand(?h1)) &`
	`K(isHand(?h2)) &`
	`K(?h1 != ?h2) &`
	`K(inHand(?x, ?h1)) &`
	`K(handEmpty(?h2))`

PKS Effect	

	`add(Kf, handEmpty(?h1)),`
	`add(Kf, inHand(?x, ?h2)),`
	`del(Kf, handEmpty(?h2)),`
	`del(Kf, inHand(?x, ?h1))`

The `pass-object` OAC allows ARMAR-III to pass an object currently held in one hand to the other. This is particularly useful when ARMAR-III needs to execute an OAC that only works with the hand the object is currently located in. For example if ARMAR-III wants to place an object in the fridge it can open the fridge partially with its right hand while holding the the object in its left hand. Then the object is passed to the right hand to free the left hand for opening the door completely.

	put-down
Parameters	x, l, h
Action	`Place`
PKS Precondition	

	`K(isObject(?x)) &`
	`K(isLocation(?l)) &`
	`(K(?l = Sideboard) | K(?l = Stove)) &`
	`K(isHand(?h)) &`
	`K(robotLocation = ?l) &`
	`K(inHand(?x, ?h))`

PKS Effect	

	`add(Kf, handEmpty(?h)),`
	`add(Kf, objLocation(?x, ?l)),`
	`del(Kf, inHand(?x, ?h)),`
	`add(Kf, onSurface(?x))`

The `put-down` OAC uses the Place action to place an object on a surface that ARMAR-III held in its hand. The Place action uses the preexisting `PlaceSkill` which contains hard-coded configuration making further configuration in the OAC unecessary.

	remove-from
Parameters	x, l, h
Action	`Grasp`
PKS Precondition	

	`K(isObject(?x)) &`
	`K(isLocation(?l)) &`
	`K(isHand(?h)) &`
	`((K(?l = Fridge) & K(?h = rightHand))) &`
	`K(robotLocation = ?l) &`
	`K(objOpen(?l)) &`
	`K(objLocation(?x, ?l)) &`
	`K(handEmpty(?h))`

PKS Effect	

	`add(Kf, inHand(?x, ?h)),`
	`del(Kf, handEmpty(?h)),`
	`del(Kf, objLocation(?x, ?l))`

The `remove-from` OAC is used to grasp an object located in the fridge. Just like the grasp and `grasp-from` OACs it uses the `Grasp` action to execute its task.

5.3 Scenario

The scenario used for the ARMAR-III experiment is called `make-drink`. It enables the `IceNetwork` and `PlanNetwork` `ActivityControllers` which take care of communicating with PKS. The `SearchObjectsController` makes sure that ARMAR-III explores the kitchen when it has not yet recognised any objects so that it can properly plan a solution aftwards. The `PlanNetworkPerceptionHandler` makes sure that short-term memory updates are passed on to the plan execution monitor. The `SelfPerceptor` and the `LocationPerceptor` ensure that ARMAR-III's state and current location are always up to date in short-term memory. For object recognition the `IVTPerceptor` is enabled and the `RelativePositionPerceptor` further processes the information provided by the vision system. The PKS goal condition is encoded as:

```
K(objLocation($CUP, Sideboard)) &
K(objLocation($JUICE, Sideboard)) &
!K(objOpen(Fridge)) &
!K(objPartialOpen(Fridge))
```

The placeholders $\$CUP$ and $\$JUICE$ are automatically replaced by the speech recognition system with the proper object identifiers at runtime.

5.4 Resulting plan

Putting all these pieces together the plan generated by PKS to solve the problem for *cup*1 and *applejuice* is the following:

```
move(Sideboard,Fridge)
open-partial(Fridge,rightHand)
open-complete(Fridge,leftHand)
remove-from(applejuice,Fridge,rightHand)
pass-object(applejuice,rightHand,leftHand)
close(Fridge,rightHand)
move(Fridge,Stove)
grasp(cup1,Stove,rightHand)
move(Stove,Sideboard)
put-down(applejuice,Sideboard,leftHand)
put-down(cup1,Sideboard,rightHand)
```

If one starts the plan with the robot already holding *cup*1 in its hand, the resulting shorter plan is:

```
put-down(cup1,Sideboard,rightHand)
move(Sideboard,Fridge)
open-partial(Fridge,rightHand)
open-complete(Fridge,leftHand)
remove-from(applejuice,Fridge,rightHand)
pass-object(applejuice,rightHand,leftHand)
close(Fridge,rightHand)
move(Fridge,Sideboard)
put-down(applejuice,Sideboard,leftHand)
```

If instead the robot is asked to retrieve two stacked objects *cup*1 and *cup*2 from the stove and bring them to the sideboard the plan looks as follows:

```
move(Sideboard,Stove)
grasp-from(cup2,Stove,leftHand)
grasp(cup1,Stove,rightHand)
move(Stove,Sideboard)
put-down(cup2,Sideboard,lefthand)
put-down(cup1,Sideboard,righthand)
```

Combining the two tasks to one in which the robot has to retrieve the stacked cups as well as the apple juice, the resulting plan deals with the applejuice separately from the two stacked objects. The robot cannot hold more than two objects at a time so it would be less efficient to go from the fridge to the stove, put down the applejuice in order to unstack and grasp the two cups only two have to move from the stove to the sideboard twice, than to drop the applejuice off at the sideboard first.

```
move(Sideboard,Fridge)
open-partial(Fridge,rightHand)
open-complete(Fridge,leftHand)
remove-from(applejuice,Fridge,rightHand)
pass-object(applejuice,rightHand,leftHand)
close(Fridge,rightHand)
move(Fridge,Sideboard)
put-down(applejuice,Sideboard,leftHand)
move(Sideboard,Stove)
grasp(cup1,Stove,rightHand)
move(Stove,Sideboard)
grasp-from(cup2,Stove,leftHand)
grasp(cup1,Stove,rightHand)
move(Stove,Sideboard)
put-down(cup2,Sideboard,lefthand)
put-down(cup1,Sideboard,righthand)
```

6 Conclusion and Outlook

In this thesis a working architecture for the execution of Object-Action Complexes on a humanoid robot is demonstrated. The presented system is capable of generating a variety of plans for complex tasks using a dynamically assembled planning domain rather than a completely hand crafted domain. Thus the system is scalable through the addition of new Object-Action Complexes alone and does not require a redesign of the entire domain when teaching the robot new capabilities.

The concept of Object-Action Complexes provides a practical method for exchanging information between different components of a cognitive system.

For a more complete evaluation of the proposed architecture, a more extensive library of OACs is required. Since ARMAR-III has been used for many other activities it continues to be a suitable platform for experiments. With a larger library of OACs it will become more feasible to evaluate significantly longer plans and plans involving more objects.

Episodic memory could be introduced into the architecture for the recording of OAC sequences and their success rates. The retrieved information could prove useful for optimising the order of OACs and potentially inserting OACs between OACs with problematic transitions based on past experience.

References

[1] T. Asfour, K. Regenstein, P. Azad, J. Schröder, N. Vahrenkamp, and R. Dillmann, "ARMAR-III: An Integrated Humanoid Platform for Sensory-Motor Control," in *IEEE/RAS International Conference on Humanoid Robots (Humanoids)*, 2006, pp. 169–175.

[2] T. Asfour, P. Azad, N. Vahrenkamp, K. Regenstein, A. Bierbaum, K. Welke, J. Schröder, and R. Dillmann, "Toward humanoid manipulation in human-centred environments," *Robotics and Autonomous Systems*, vol. 56, no. 1, 2008.

[3] R. E. Fikes and N. J. Nilsson, "STRIPS: a new approach to the application of theorem proving to problem solving," in *Proceedings of the 2nd international joint conference on Artificial intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1971, pp. 608–620.

[4] D. McDermott, "PDDL – The Planning Domain Definition Language," Yale Center for Computational Vision and Control, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.

[5] R. Petrick and F. Bacchus, "A Knowledge-Based Approach to Planning with Incomplete Information and Sensing," in *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, M. Ghallab, J. Hertzberg, and P. Traverso, Eds. Menlo Park, CA: AAAI Press, Apr. 2002, pp. 212–221.

[6] ——, "Extending the Knowledge-Based Approach to Planning with Incomplete Information and Sensing," in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, S. Zilberstein, J. Koehler, and S. Koenig, Eds. Menlo Park, CA: AAAI Press, Jun. 2004, pp. 2–11.

[7] R. Petrick, "A Knowledge-level approach for effective acting, sensing, and planning," Ph.D. dissertation, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 2006.

[8] K. Okada, S. Tokutsu, T. Ogura, M. Kojima, Y. Mori, T. Maki, and M. Inaba, "Scenario controller for daily assistive humanoid using visual verification, task planning and situation reasoning," in *Intelligent Autonomous Systems, 2010. IAS-10. 10th International Conference on*, Okada, pp. 398–405.

[9] S. Tokutsu, K. Okada, and M. Inaba, "Environment situation reasoning integrating human recognition and life sound recognition using DBN," in *Robot and Human Interactive Communication, 2009. RO-MAN 2009. The 18th IEEE International Symposium on*, Toyama, pp. 744–750.

[10] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[11] A. Haneda, K. Okada, and M. Inaba, "Realtime manipulation planning system integrating symbolic and geometric planning under interactive dynamics siumlator," in *Mechatronics and Automation, 2008. ICMA 2008. IEEE International Conference on*, Takamatsu, pp. 988–993.

[12] J. Hoffmann and B. Nebel, "The FF planning system: fast plan generation through heuristic search," *J. Artif. Int. Res.*, vol. 14, pp. 253–302, May 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=1622394.1622404

[13] C. McGann, E. Berger, J. Bohren, S. Chitta, B. Gerkey, S. Glaser, B. Marthi, W. Meeussen, T. Pratkanis, E. Marder-Eppstein, and M. Wise, "Model-based, Hierarchical Control of a Mobile Manipulation Platform," in *ICAPS Workshop on Planning and Plan Execution for Real-World Systems*, Thessaloniki, Greece, 2009.

[14] ZeroC Inc., "Internet Communications Engine (Ice)." [Online]. Available: http://www.zeroc.com/

[15] M. Henning, "A New Approach to Object-Oriented Middleware," *IEEE Internet Computing*, vol. 8, no. 1, pp. 66–75, 2004.

[16] M. E. Foster, T. By, M. Rickert, and A. Knoll, "Human-Robot dialogue for joint construction tasks," in *ICMI '06: Proceedings of the 8th international conference on Multimodal interfaces*. New York, NY, USA: ACM, 2006, pp. 68–71.

[17] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Orca: A Component Model and Repository," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, D. Brugali, Ed. Springer Berlin / Heidelberg, 2007, vol. 30, pp. 231–251, 10.1007/978-3-540-68951-5_13 http://dx.doi.org/10.1007/978-3-540-68951-5_13.

[18] M. Henning and S. Vinoski, *Advanced CORBA programming with C++*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] N. Krüger, J. Piater, F. Wörgötter, C. Geib, R. Petrick, M. Steedman, A. Ude, T. Asfour, D. Kraft, D. Omrcen, B. Hommel, A. Agostino, D. Kragic, J. Eklundh, V. Krüger, and R. Dillmann, "A Formal Definition of Object Action Complexes and Examples at Different Levels of the Process Hierarchy," EU project PACO-PLUS, Tech. Rep., 2009.

[20] A. Agostini, E. Celaya, C. Torras, and F. Wörgötter, "Learning Rules from Cause-Effects Explanations." Institut de Robòtica i Informàtica Industrial, CSIC-UPC, Tech. Rep. IRI-TR 04/2008, 2008.

[21] A. Agostini, F. Wörgötter, E. Celaya, and C. Torras, "On-Line Learning of Macro Planning Operators using Probabilistic Estimations of Cause-Effects," Institut de Robòtica i Informàtica Industrial, CSIC-UPC, Tech. Rep. IRI-TR 05/2008, 2008.

[22] Kitware Inc., "CMake." [Online]. Available: http://www.cmake.org/

[23] G. Rozental, "Boost Test Library: The Unit Test Framework." [Online]. Available: http://www.boost.org/doc/libs/1_44_0/libs/test/doc/html/utf.html

[24] D. van Heesch, "Doxygen." [Online]. Available: http://www.stack.nl/~dimitri/doxygen/

[25] M. Fowler, "Inversion of Control Containers and the Dependency Injection Pattern." [Online]. Available: http://martinfowler.com/articles/injection.html

[26] J. Fulman and E. L. Wilmer, "ECMAScript Language Specification." *Ann. Appl. Probab*, vol. 9, pp. 1–13, 1999. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[27] D. Crockford, "RFC 4627 - The application/json Media Type for JavaScript Object Notation (JSON)," IETF RFC, Tech. Rep. [Online]. Available: http://tools.ietf.org/html/rfc4627

[28] S. P. Jones, "The Haskell 98 Report." [Online]. Available: http://haskell.org/onlinereport/exps.html#pattern-matching

[29] Ericsson AB, "Erlang Reference Manual User's Guide." [Online]. Available: http://erlang.org/doc/reference_manual/expressions.html#pattern

[30] D. Kraft, E. Başeski, M. Popović, A. Batog, A. Kjær-Nielsen, N. Krüger, R. Petrick, C. Geib, N. Pugeault, M. Steedman, T. Asfour, R. Dillmann, S. Kalkan, F. Wörgötter, B. Hommel, R. Detry, and J. Piater, "Exploration and planning in a three-level cognitive architecture," in *Proceedings of the International Conference on Cognitive Systems (CogSys 2008)*, 2008.