Project no.:	**IST-FP6- IP-027657**
Project full title:	**Perception, Action & Cognition through Learning of Object-Action Complexes**
Project Acronym:	**PACO-PLUS**
Deliverable no.:	**D 1.2.1**
Title of the deliverable:	**Technical report describing the adaptive algorithms and the designed software**

Contractual Date of Delivery to the CEC:	January 31st, 2008
Actual Date of Delivery to the CEC:	January 31st, 2008
Organisation name of lead contractor for this deliverable:	BCCN
Author(s):	R. Petrick, Ch. Geib, M. Steedman, P. Haazebroek, B. Hommel, K. Welke, P. Azad, T. Asfour, F. Wörgötter and R. Dillmann
Participant(s):	UEDIN, UL, UniKarl
Work package contributing to the deliverable:	WP1
Nature:	R/D
Version:	1.0
Total number of pages:	27
Start date of project:	1st Feb. 2006 **Duration:** 48 months

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level		
PU	Public	
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	X

Abstract:

In this report, we briefly review the general architecture guiding this development, the relevant processing levels, and recent advances in interfacing these levels. We also describe a relatively simple scenario used to test and demonstrate implemented functionalities of the system, and explain how the processes on the three levels interact in order to produce intelligent behavior suited to meet the challenges the scenario poses. Furthermore, we present the ongoing work towards the implementation of the proposed architecture on the humanoid robot ARMAR-III.

Keyword list: Three level architecture, high-level planning representation, API for ARMAR-III.

Table of Contents

1 Introduction

A Scenario for Integrating Low-Level Robot/Vision, Mid-Level Affordance Memory, and Grounded High-Level Conditional Planning:

The PACO-PLUS project aims at developing and implementing a biologically plausible, intelligent cognitive architecture for a humanoid robot, with an emphasis on integrating perception and action into Object-Action Complexes (OACs). This paper will summarize the current state concerning the cognitive architecture for PACO-PLUS and how planning and acting can be performed in a simple scenario based on this architecture.

1.1 The architecture

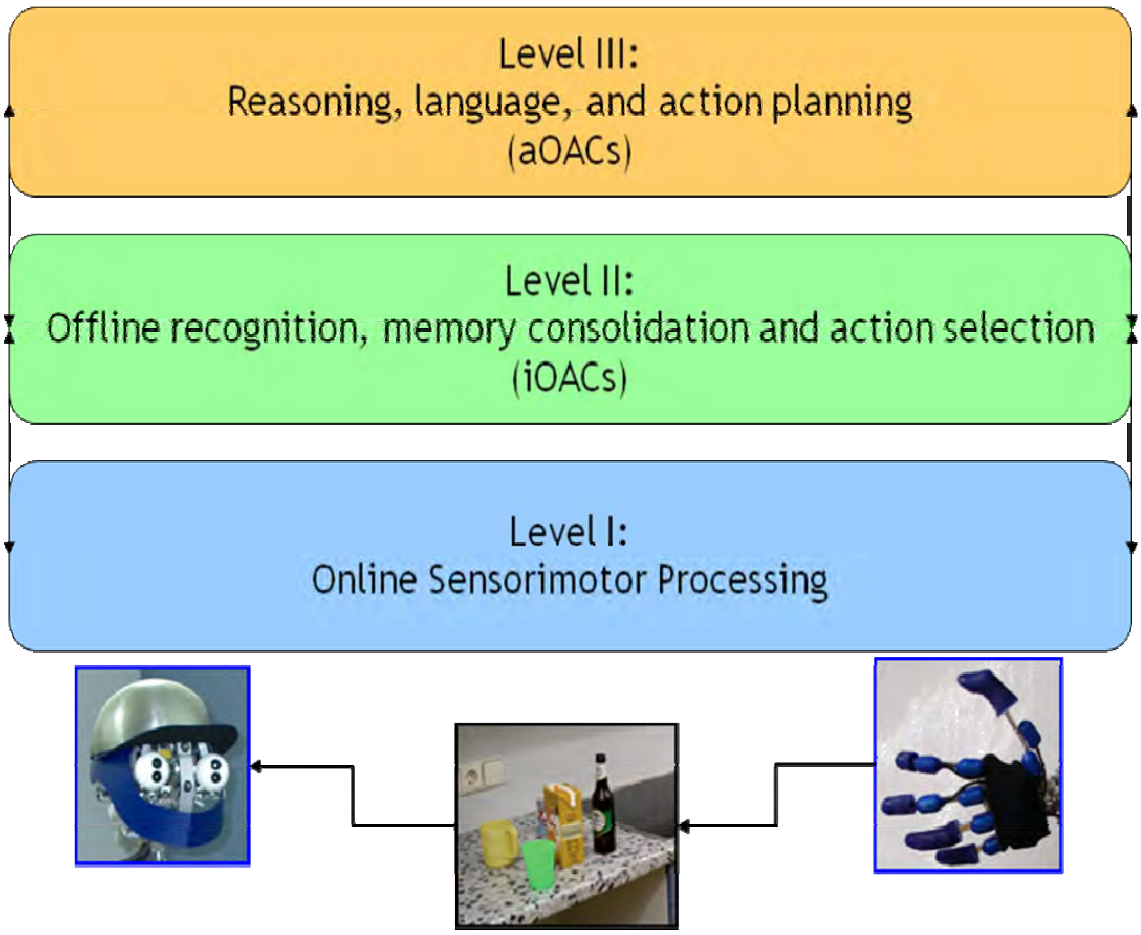
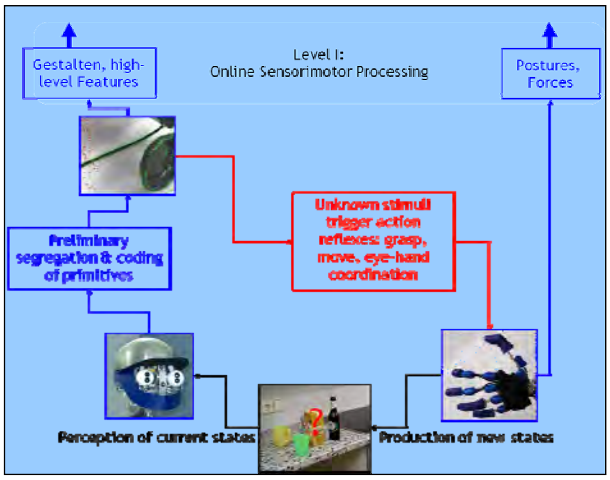

Level III:
Reasoning, language, and action planning
(aOACs)
Level II:
Offline recognition, memory consolidation and action selection
(iOACs)
Level I:
Online Sensorimotor Processing

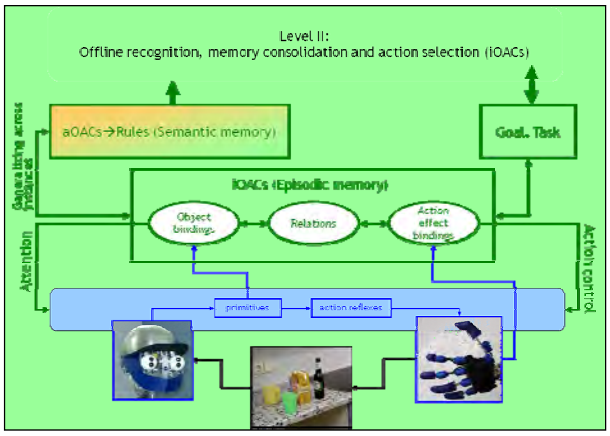
Figure 1 Overview of the proposed cognitive control architecture

The basic architecture of the aimed-at cognitive system consists of three communicating processing levels. As shown in Figure 1, the lowest processing level is responsible for sensorimotor processing, that is, for the registration and preprocessing of sensory (mainly visual, auditory, and tactile) information and for driving the available motor systems (head, eyes, hands, fingers, and body movement) —see Figure 2A. Perception and action interact in a cyclic fashion, in the sense that sensory processing informs and partially controls motor actions, which again provide further sensory information about the environment and the agent-environment relationship.

The low level sends information to the mid level for further processing, integration into episodic event representations, encoding into memory, and eventual action selection if necessary (see Figure 2B). The mid level provides top-down information for the low level to facilitate object recognition and specifies abstract action plans developed by the high level.


Gestalten, high-level Features
Level I:
Online Sensorimotor Processing
Postures, Forces
Unknown stimuli trigger action reflexes: grasp, move, eye-hand coordination
Preliminary segregation & coding of primitives
Perception of current states
Production of new states

The mid level provides top-down information for the low level to facilitate object recognition and is responsible for refining abstract action plans developed by the high level. The high level constructs sequential action plans or more complex conditional plans with branches. It interacts with the mid level to access updated information about particular world properties, and passes plans to the mid level for further specification.



The mid level provides top-down information for the low level to facilitate object recognition and is responsible for refining abstract action plans developed by the high level. The high level constructs sequential action plans or more complex conditional plans with branches. It interacts with the mid level to access updated information about particular world properties, and passes plans to the mid level for further specification.

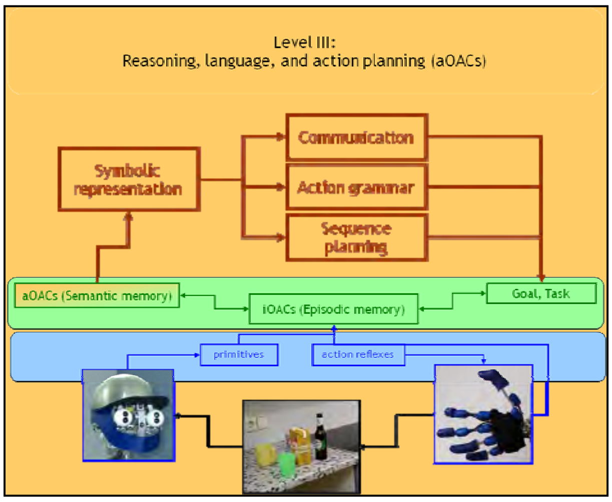


Figure 2 Detailed schemata of the three levels

The object stacking scenario

Here we provide an overview of a proposed scenario for integrating SDU's robot/vision system, Leiden's mid-level memory/reasoning component, and Edinburgh's high-level planner and plan execution monitor[1]. We describe both the proposed functional architecture and the progress made towards realizing it. The domain we have chosen for our integration task is a simple object manipulation scenario that builds on the integration scenario involving SDU and Edinburgh (see the Edinburgh document *A Scenario for Integrating Low-Level Robot/Vision and High-Level Planning* for details). We assume a *table* with a number of *objects* that are graspable by the robot. We consider situations with no more than 5 objects and, initially, only 1-2 objects. For simplicity we assume that objects are generally cylindrical in shape but not necessarily identical. In particular, each object can have a different *radius* which determines its size. Objects may or may not be *open* containers, which determines whether or not we can *stack* objects inside other objects, provided the object sizes permit such stacking. The objects can vary in colour and the system is prepared to process this feature dimension in principle (e.g., to guide top-down identification or visual search), but at this point colour is ignored.

In principle, our system is equipped to represent and control multiple categories of actions (such as grasping, pointing, moving, etc.) and several types of actions defined within these categories. At this point, however, we have agreed to restrict the possibilities to four types of grasping actions, as illustrated in Figure 3:

Grasp Type A

This action can only be used to grasp objects at the top of a stack, or an empty object on the table. Objects must also satisfy a minimum and maximum radius restriction. Note that successful performance of this action on a given object implies that this object is “pokeable” and, hence, open.

Grasp Type B

This action can only be used to grasp objects on the table that are not part of a stack. Objects must also satisfy a minimum radius restriction. Note that successful performance of this action on a given object implies that this object is “pokeable” and, hence, open.

Grasp Type C

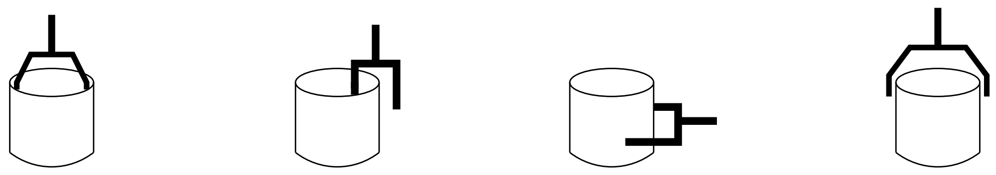
This action can only be used to grasp objects that aren't contained in other objects, i.e., the “outermost” object which must be on the table. Objects must also satisfy a maximum radius restriction. Note that successful performance of this action on a given object does not imply that this object is “pokeable” or open.

Grasp Type D

This action can only be used to grasp objects that aren't contained in other objects, i.e., objects that are on the table. Objects must also satisfy a maximum radius restriction. For simplicity, we will assume that objects stacked within the object being grasped will not affect the grasp. Note that successful performance of this action on a given object does not imply that this object is “pokeable” or open.

[1] For the purposes of distinguishing between the three levels in this document we will use the terms “robot” and “low level”, “memory” and “mid level”, and “planner” and “high level” to denote the components developed by SDU, Leiden, and Edinburgh respectively.

The goal of the scenario is to clear all open objects from the table, by removing them to some designated location (e.g., a box, a shelf, a hole, a corner of the table, etc.). The location may furthermore be restricted in such a way as to force object stacking in order to successfully complete the task. For instance, there might only be room for 2 objects to sit side by side on a shelf, meaning all other objects would have to be appropriately stacked.



a) Grasp Type A **b) Grasp Type B** **c) Grasp Type C** **d) Grasp Type D**

Figure 3 Robot grasps available for the planner

The high-level planning system will typically have only incomplete information concerning the openness of objects and must therefore construct *conditional plans* that contain explicit *knowledge-producing* actions. Such actions are used by the planner to interact with the mid-level memory in order to obtain the information it requires about the state of the world. The mid-level, in response, can gather the requested state information by either (a) retrieving the information from existing episodic memory traces that the given object is “pokeable”, or (b) triggering the exploration of the object to create such traces ad hoc (affordance learning). Object openness plays two important roles in this scenario: (i) as a goal condition that determines which objects should be removed from the table, and (ii) as a prerequisite for stacking operations.

The planner will also construct plans that contain object-manipulation operations in the form of *grasp* actions, in order to stack objects or remove them from the table. At the high level, such actions are described in an abstract fashion, without reference to primitive grasp types, gripper orientation, object location, etc. It is the job of the mid level to *refine* such actions for execution at the robot level, for instance by choosing the appropriate primitive grasp type given an object’s location and the position of the gripper. (We will only focus on grasp type initially.)

This scenario is meant to provide a basis for integrating the robot/vision, mid-level memory, and high-level planning components of the system. The planner is responsible for constructing a plan that achieves the goal of clearing open objects from the table, by working with a high-level representation of the scenario. The job of the mid-level component is to extend such plans by providing information about object features and affordances, and by selecting the concrete action needed, before passing the augmented plan to the robot/vision system. Ultimately, the robot/vision system must be able to translate the plans generated and refined by the upper levels into concrete motor actions.

Given that we have described the robot/vision front end of the proposed architecture in some detail elsewhere [Kraft et al., 2008], in this document we will mainly focus on the interface between the low-level and the mid-level, and the top-down generation of goal-directed actions, including a description of how a high-level plan is passed to the mid-level memory system, and the message passing protocol that supports the exchange of messages between the levels.

2 Mid-level memory representation

In our first scenario, we directly linked the low level to the high level. This provides many advantages by being able to directly ground symbols in sensory data and translate them into concrete motor action. At the same time, however, it requires all predicates being predefined, which necessarily restricts the systems’ flexibility. It also means that the flow of information is rather unidirectional, so that sensory processing does not yet receive any top-down support. To overcome these restrictions, we introduced a biologically plausible mid-level representational system that is motivated by the Theory of Event coding [Hommel, Müsseler, Aschersleben & Prinz, 2001] and that intervenes between the low, sensorimotor level and the high-level planner. Four functionalities of the mid-level are of particular importance for input processing (see Figure 4) and one for output generation:

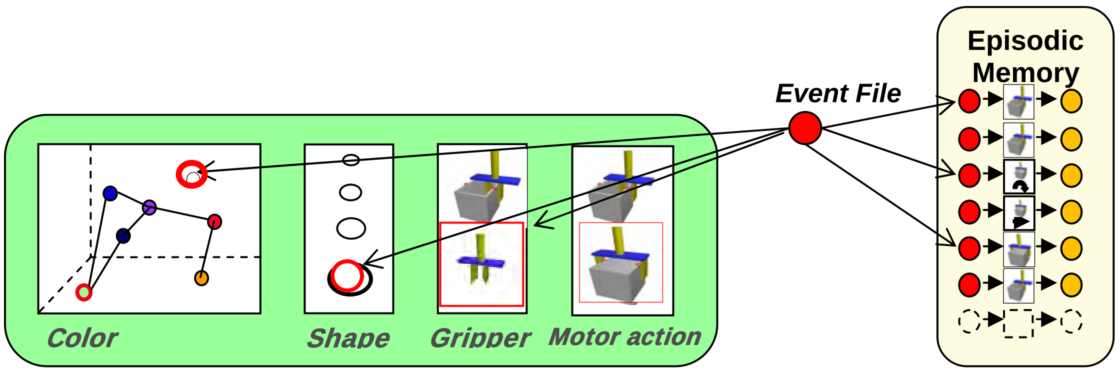


Figure 4 Representation of perceptual input and episodic memory

2.1 Feature coding

Preprocessed visual input is coded by multiple self organizing Kohonen-like maps of nodes. Nodes in these maps encode objects in a distributed fashion and the maps can grow new nodes when a novel perceptual feature (e.g., a novel colour) is presented. At this point, we have adapted the system to the characteristics of the SDU camera and gripper and simplified feature coding in the following ways:

As there is only one cup attended to at a time, we need to represent only one object in the feature maps. This allows us to effectively disable the 3D colour map (corresponding to the RGB value of the global cup colour) and restrict coding to

- a 1D shape map corresponding to the circle radius
- a 1D gripper map, corresponding to the distance of the gripper fingers
- the set of four grasp types introduced above projected onto a 1D motor action map

2.2 Working memory

The eventual system contains a working memory that provides pointers to all the sensory and action-related aspects of an event (*event files*). This structure is updated every time a change is registered, whether it is brought about by a voluntary action of the robot or by other, external forces. Event files can be activated both bottom up (by current perceptual input) and top down (by the episodic memory traces). This allows for interaction effects such as selecting motor actions (e.g., type of grasping) that seem possible with current input (e.g., a large red cup) and that have previously led to desired effects (e.g., having a gripper sensation of holding this cup). Event files will also help address the problem of object constancy/consistency: Due to changes (e.g., lighting conditions) in the environment and sensor inaccuracy, the continuous sensor input can fluctuate. Working memory representations maintain an (abstracted) active representation of the input, yielding a more consistent object representation. Even though this functionality is not yet implemented, working memory representations will also facilitate sensory preprocessing by providing top-down information.

At this point, the working memory contains only one unit or event file: pointers to the current object (cup) and the actions it affords (i.e., the actions that have been successfully performed on this object). In later versions, we intend to extend the capacity of the working memory to multiple events and to include context information.

2.3 Episodic memory (iOACs)

The episodic memory contains enduring traces of previous working memory units (instantiated Object-Action Complexes or iOACs). iOACs link object-feature information to types of motor actions in a bidirectional fashion. These bidirectional links are particularly important for informing the high-level planner. Importantly for the present scenario, iOACs will provide information whether a given cup has ever been successfully poked in previous exploration or planned action (i.e., whether it is linked to any instance of Grasp A or Grasp B), which implies that this cup must be open. Only if no relevant episodic information is available would it be necessary to engage in ad hoc exploration to test whether the current cup is pokeable (and thus “open”) or not (and thus “closed”). (Once this information has been obtained—by either means—the mid level can successfully respond to plan-level requests for information about this object’s openness.) At this point, the episodic memory is restricted to integrating the few feature dimensions that are perceptually coded and the four action types. Future versions will be extended to include more feature dimensions from multiple sensory modalities and multiple actions from several categories. This will introduce capacity problems in terms of search time and storage space, as the complexity and number of episodic traces this will grow rapidly. We will encounter this problem by introducing recency- and interference-based forgetting and thus eliminate episodic traces depending on their lifetime and the number of more recent traces. Given that episodic traces will be read out by semantic memory (see below) and transformed into rule-like representations, this will not lead to a loss of important information however.

2.4 Semantic memory (aOACs)

The introduction of an episodic memory makes it possible to collect multiple data sets related to the same object and object-action relationship. This introduces both variability and consistency: features that are not essential for a given object, affordance, or action will vary (such as the colour of cups in the context of grasping or the speed of the grasping movement) and features that are essential (such as openness that indicates pokeability, and vice versa, or the radius of a cup). In principle, it would seem more parsimonious to store only relevant object and action characteristics and immediately “forget” inessential aspects. But there is an interesting trade-off between parsimony and robustness of knowledge [cf., O’Reilly & Rudy, 2001]. Extracting only relevant information presupposes that it is known which features are relevant, which however is impossible without extensive testing. Building an episodic database provides an alternative, as simple regression techniques suffice to isolate essential object and action features from inessential characteristics: essential object features should reliably predict (i.e., be significantly correlated with) essential action features, and vice versa. Once isolated, these features can be taken to construct rule-based knowledge (abstracted Object Action Complexes or aOACs) that will strongly facilitate the interaction between the high-level planner and the mid-level memory. Particularly attractive is the fact that episodic memory traces outlive the availability of sensory information. Thus, if a rule turns out to be invalid, the system does not need to await another encounter with the respective object but can construct alternative rules by considering other features laid down in the episodic traces. At this point, the semantic memory is not yet implemented.

2.5 Action selection

In our earlier scenario we directly linked the low-level motor programs realizing the four considered types of grasps to their symbolic representations at the high level. Introducing the mid-level and an episodic memory that includes action-relevant information provides the opportunity to make the planning more abstract. Hence, the planner no longer needs to specify the particular grasp type needed to grasp a particular object but can leave this specification to the interaction between the low-level and the mid-level.

At the mid-level, this is realized as follows: if the high-level planner issues a command to grasp a particular object, this primes all iOACs that include a grasping action. Given that we currently restrict our modeling to the mentioned four types of grasps anyway, this means that all available iOACs are primed. (Hence, this step does not yet help in the action selection process but it will do so as more types of actions are considered.) Further priming is received from the currently activated event file(s) in working memory, so that both types of priming (top-down and bottom-up) will converge onto those iOACs that match the currently available object(s). At this point, this can only be a single cup, so that episodic traces related to this cup will be primed the strongest, and so will be the action-related information they include. In the present system, this means that priming will be strongest for those of the four actions that have been successfully carried out on this object and for the grasp-relevant shape information—sufficient information to select the appropriate grasp type and parameterize the end posture of the gripper or hand.

3 High-level planning representation

Given the above scenario, we define a set of high-level actions and properties that allows the planner to operate in this domain. We also provide some insights as to how such actions and properties relate to the memory and robot/vision levels.

3.1 Actions

Although the robot has four primitive grasp types available to it at the lower levels (grasp types A, B, C, and D), from the planner's perspective grasping is treated as a single abstract operation. Plans that include grasping actions therefore need to be refined by the mid level before execution, in order to determine the actual grasp type that will be used at the motor program level by the robot. Actions at the planning level are modelled as STRIPS-style actions. As a result, the effects of the single grasping operation are subdivided into two plan-level actions that account for different world contexts where the action could be applied (i.e., grasping an object from the table versus grasping an object from the top of a stack). Each grasping action takes a single argument, ?x, denoting the label of an object in the world. Such labels are designated by strings of the form objN, where N is a non-negative integer, e.g., obj42. Although we assume the existence of particular object labels for the purpose of our examples, in practice such information must be provided to the planner by the lower levels. Our representation includes the following two high-level grasp actions:

`grasp-fromTable(?x)`

Grasp object ?x from the table.

`grasp-fromTopOfStack(?x)`

Grasp object ?x from the top of a stack of objects.

We have also encoded four actions for moving and manipulating objects when successfully grasped:

`putInto-objectOnTable(?x,?y)`

Put object `?x` into object `?y`, which is on the table.

`putInto-stack(?x,?y)`

Put object `?x` into object `?y`, which is at the top of a stack on the table.

`putOnTable(?x)`

Put object `?x` onto the table.

`putAway(?x)`

Put object `?x` away.

Each manipulation action is *object centric* and modelled with a high degree of abstraction. For instance, we do not provide plan-level actions that specify 3D spatial coordinates, joint angles, or similar real-valued parameters. The `putAway` action is particularly generic and should be considered a placeholder for a more complex (possibly, predefined) operation that clears an object from the table to its final destination location. For the purpose of this document we will assume that objects are being put away onto a shelf. We also note that both `putInto-objectOnTable` and `putInto-stack` actions denote stacking operations which will have as a prerequisite the property that objects can only be stacked into open objects.

The high-level planning representation also includes a single *knowledge-producing* action:

`findout-open(?x)`

Determine whether object `?x` is open or not.

At the planning level, this action is modelled as a type of information gathering operation that provides the planner with additional information about an object's state. In practice, this operation must interact with the mid level: the planner has no direct access to the world state and, thus, must request information from the lower levels of the system. The mid level is therefore responsible for gathering the information required by the planner to satisfy such requests (e.g., by accessing existing episodic memory traces, or initiating exploratory actions), and must return this information to the upper level in a form that can be understood by the planner.

3.2 Properties

Our planning-level domain encoding makes use of a set of predicates and functions which we have agreed could reasonably be provided to the planner as a result of sensor information from the robot/vision level. (These properties are still subject to change as the domain model is refined through further discussions.) We have the following properties in our high-level domain representation:

`clear(?x)`

A predicate indicating that no object is stacked in `?x`.

`gripperempty`

A predicate describing whether the robot's gripper is empty or not.

`ingripper(?x)`

A predicate indicating that the robot is holding object `?x` in its gripper.

`instack(?x,?y)`

A predicate indicating that object `?x` is in a stack with object `?y` at its base.

`isin(?x,?y)`

A predicate indicating that object `?x` is stacked in object `?y`.

`onshelf(?x)`

A predicate indicating that object `?x` is on the shelf.

`ontable(?x)`

A predicate indicating that object `?x` is on the table.

`open(?x)`

A predicate indicating that object `?x` is open.

`radius(?x) = ?y`

A function indicating that the radius of object `?x` is `?y`.

`reachable(?x)`

A predicate indicating that object `?x` is reachable by the gripper.

`shelfspace = ?x`

A function indicating that there are `?x` empty shelf spaces.

`graspable(?x)`

A function indicating that object `?x` is capable of being grasped.

(This is a generic property that may be replaced by more specific properties at a later time.)

3.3 Domain encoding

Using the above properties we can write planning operators for the actions we require. Our target planner in this case is the PKS (Planning with Knowledge and Sensing) planning system [Petrick & Bacchus, 2002, 2004]. For simplicity, we have made the following restrictions in our domain encodings: (i) all objects are initially assumed to be on the table, and (ii) the `putOnTable` action will initially be omitted (since there are no initial stacks).

Our current domain encoding is given in Table 1. (These actions have been formalized for use with PKS, however, we have simplified the syntax here.) Although most of the details of the actual action encodings can be ignored, we mention two important points. First, each action operator is parameterized with a set of arguments that can denote any object in the world. Thus, all of our actions are object centric. Second, our encoding takes advantage of PKS's ability to work with functions and simple numerical expressions, which we include as part of the action preconditions and effects. (E.g., the radius of an object plays a role in determining whether or not it can be stacked inside another object.) Our domain encoding can be extended as needed to accommodate new actions or properties that may arise from future discussions.

PKS action description notation: The domain encoding in Table 1 is very much like a standard STRIPS encoding except that PKS, unlike STRIPS, uses multiple databases as the basis for its representation. Thus, references to K_f and K_w in the “effects” section of an action denote two of PKS's databases. (K_f is very much like a standard STRIPS database that stores the planner's knowledge of facts, and K_w is a specialized database for storing the plan-time effects of knowledge-producing actions.) References to expressions like $\neg K_w$`(open(?x))` denote knowledge preconditions that ensure the planner does not unnecessarily include a knowledge-producing action in a plan if it already knows the information that would be provided by such an action. (I.e., if the planner already knows whether an object is open or not then it shouldn’t include a `findout-open` action for that object.)

Table 1: PKS-style action descriptions

Action	Preconditions	Effects
`findout-open`	$\neg K_w$`(open(?x))` `ontable(?x)`	`add(`K_w`,open(?x))`

`grasp-fromTable`	`graspable(?x)`
	`reachable(?x)`
	`clear(?x)`
	`gripperempty`
	`ontable(?x)`
		`add(`K_f`,ingripper(?x))`
		`add(`K_f`,¬gripperempty)`
		`add(`K_f`,¬ontable(?x))`

`grasp-fromTopOfStack(?x)`	`graspable(?x)`
	`reachable(?x)`
	`clear(?x)`
	`gripperempty`
	`exists(?z).`
	    `instack(?x,?z)`
	    `ontable(?z)`
		`add(`K_f`,ingripper(?x))`
		`add(`K_f`,¬gripperempty)`
		`forall(?y).isin(?x,?y)→`
		    `del(`K_f`,isin(?x, ?y))`
		    `add(`K_f`,clear(?y))`
		`forall(?z).instack(?x,?z)→`
		    `del(`K_f`,instack(?x,?z))`

`putInto-objectOnTable(?x,?y)`	`?x ≠ ?y`
	`ingripper(?x)`
	`open(?y)`
	`clear(?y)`
	`ontable(?y)`
	`radius(?y) > radius(?x)`
		`add(`K_f`,gripperempty)`
		`add(`K_f`,isin(?x,?y))`
		`add(`K_f`,instack(?x,?y))`
		`del(`K_f`,clear(?y))`
		`del(`K_f`,ingripper(?x))`
		`forall(?w).instack(?w,?x)→`
		    `del(`K_f`,instack(?w,?x))`
		    `add(`K_f`,instack(?w,?y))`

`putInto-stack(?x,?y)`	`?x ≠ ?y`
	`ingripper(?x)`
	`open(?y)`
	`clear(?y)`
	`radius(?y) > radius(?x)`
	`exists(?z).`
	    `instack(?y,?z)`
	    `ontable(?z)`
		`add(`K_f`,gripperempty)`
		`add(`K_f`,isin(?x,?y))`
		`add(`K_f`,clear(?y))`
		`del(`K_f`,ingripper(?x))`
		`forall(?z).instack(?y,?z)→`
		  `add(`K_f`,instack(?x,?z))`
		  `forall(?w).instack(?w,?x)→`
		    `del(`K_f`,instack(?w,?x))`
		    `add(`K_f`,instack(?w,?z))`

`putAway(?x)`	`ingripper(?x)`
	`shelfspace > 0`
		`add(`K_f`,onshelf(?x))`
		`add(`K_f`,gripperempty)`
		`del(`K_f`,ingripper(?x))`
		`shelfspace = shelfspace - 1`

4 Example plans

In this section we give three examples of planning problems we can solve using PKS and the above action descriptions. In each example we consider a scenario with 2 objects initially on the table. Each object also has a size as indicated by its radius. (For simplicity we use integer values in our examples however we also permit real-valued quantities.) In each example we assume the following initial conditions:

- Objects: `obj1`, `obj2`
- `radius(obj1) = 1`, `radius(obj2) = 4`
- `shelfspace = 1`
- All objects are on the table (no initial stacks)

The goal in each example is to clear the open objects from the table by placing the objects on a shelf which has limited space. In Example 1, the planner initially knows that both objects are open and, thus, does not need to include knowledge-producing actions in the plan. In Examples 2 and 3, knowledge-producing actions are required: in the second example, the planner knows that one object is not open but does not know whether the second object is open or not; in the third example, the planner does not know whether either object is open or not.

When PKS constructs a plan that includes knowledge-producing actions, it can build into the plan a set of *conditional branches* for reasoning about the possible outcomes of such an operation. In particular, one branch is constructed for each possible value the operation might return. The resulting plans in this case are structured as trees rather than simple linear sequence of actions. In our examples, branch points are denoted by expressions like "`branch(open(objX))`," meaning "branch on the truth value of `open(objX)`." In this scenario, we will only consider branches on binary properties, i.e., properties that can be either true or false. A branch point is followed by two plan sections, labelled as "`K+`" and "`K-`," denoting two disjoint plan branches. The `K+` branch indicates the "knowledge positive" branch where `open(objX)` is assumed to be true. The `K-` branch indicates the "knowledge negative" branch where `open(objX)` is assumed to be false. Each branch can contain a sequence of actions and possibly other branch points. A `nil` tag along a branch indicates that no further operation takes place along that branch. At execution time, the information returned from a knowledge-producing action (i.e., information gathered by the mid level and passed to the high level in response to such an action) lets the high-level plan execution monitor decide which branch of the plan it should follow at a branch point. The planner ensures that when conditional plans are constructed, the goals are achieved along every branch of the plan.

4.1 Example 1

The planner initially knows that open(obj1) and open(obj2) are both true.

Plan:

```
grasp-fromTable(obj1)
putInto-objectOnTable(obj1,obj2)
grasp-fromTable(obj2)
putAway(obj2)
```

Since obj1 and obj2 are both initially known to be open the planner does not need to include any knowledge-producing actions in the plan. The two objects can simply be stacked and removed from the table.

4.2 Example 2

The planner initially knows that `open(obj1)` is true but does not know the state of `open(obj2)`.

Plan:

```
findout-open(obj2)
branch(open(obj2))
K+:
    grasp-fromTable(obj2)
    putAway(obj2)
K-:
    Nil
```

Since the planner does not initially know whether `obj2` is open or not it includes a knowledge-producing `findout-open` action in the plan. The plan then branches on the two possible outcomes of `open(obj2)`. If `open(obj2)` is true (the `K+` branch) then `obj2` is grasped and removed from the table; if `open(obj2)` is false (the `K-` branch) then no further action is taken. Since the planner initially knows that `obj1` is not open, this object does not need to be removed from the table.

4.3 Example 3

The planner does not initially know the state of `open(obj1)` and `open(obj2)`.

Plan:

```
findout-open(obj1)
findout-open(obj2)
branch(open(obj2))
K+:
    branch(open(obj1))
    K+:
        grasp-fromTable(obj1)
        putInto-objectOnTable(obj1,obj2)
        grasp-fromTable(obj2)
        putAway(obj2)
    K-:
        grasp-fromTable(obj2)
        putAway(obj2)
K-:
    branch(open(obj1))
    K+:
        grasp-fromTable(obj1)
        putAway(obj1)
    K-:
        nil
```

Since the planner does not initially know whether `obj1` or `obj2` is open, it includes two `findout-open` actions in the plan. It then considers each possible outcome of these actions by constructing a plan with four branches (an initial branch point, followed by a second branch point along each of the top-level branches):

(1) Along the `K+`/`K+` branch where `open(obj2)` and `open(obj1)` are true, both objects are grasped and put away as in Example 1.

(2) Along the `K+`/`K-` branch where `open(obj2)` and $\neg$ `open(obj1)` are true, object `obj2` is grasped and put away.

(3) Along the `K-`/`K+` branch where $\neg$ `open(obj2)` and `open(obj1)` are true, object `obj1` is grasped and put away.

(4) Along the `K-/K-` branch where ¬`open(obj2)` and ¬`open(obj1)` are true, no further action is taken.

We note that in the above examples we only describe the high-level structure of the plans. During plan execution, the mid level must refine certain actions like `grasp-fromTable` and `grasp-fromTopOfStack` (by choosing an appropriate grasp types), and satisfy `findout-open` requests for information, which can result in changes to the plan content.

5 Message passing protocol

In this section we describe a simple message passing protocol for exchanging information between the low-level robot/vision, mid-level memory, and high-level planning levels. We begin by defining the kinds of messages that can be passed between the system levels. We then describe a simple control architecture that is sufficient for our proposed integration task, and provide some details of a communication library (supplied by Edinburgh) that implements this protocol.

5.1 Message definitions

We define a set of 10 messages that capture the interactions between the three levels of the system. Each message is defined by its *type* and its *content*. A message's type is simply its name or label. Depending on the message type, a message may also contain specific content or data to be sent. The message passing protocol we have defined is currently based on a *point-to-point* model, where each message is sent by a particular system component to another component. Moreover, the message set is designed in such a way that messages are (generally) defined in send/receive pairs so that only certain messages can be initiated by a "sending" level, with an appropriate response being sent by the "receiving" level. The complete set of messages is given in Table 2 and the send/receive message pairs are given in Table 3. (This message set is subject to change and may be expanded or streamlined in the future.).

Table 2: Available message types in the message passing protocol

`MSG_STATE_UPDATE` — Provide updated state information

Sender/Destination: Robot to Memory, or Memory to Planner

Content: World state specification

`ACK_STATE_UPDATE` — Acknowledge state update message

Sender/Destination: Planner to Memory, or Memory to Robot

Content: NONE

`MSG_ACTION_REQUEST` — Request a new action

Sender/Destination: Robot to Memory, or Memory to Planner

Content: NONE

`ACK_ACTION_REQUEST` — Acknowledge new action request for execution

Sender/Destination: Planner to Memory, or Memory to Robot

Content: NONE

`MSG_ACTION_SUBMIT` — Submit a new action for execution

Sender/Destination: Planner to Memory, or Memory to Robot

Content: Action specification

`ACK_ACTION_SUBMIT` — Acknowledge receipt of new action and start of action execution

Sender/Destination: Robot to Memory, or Memory to Planner

Content: NONE

`MSG_ACTION_STOPPED` — Provide alert that execution of last submitted action has stopped

Sender/Destination: Robot to Memory, or Memory to Planner

Content: Action execution return value (1 = success or 0 = failure)

`ACK_ACTION_STOPPED` — Acknowledge termination of last submitted action

Sender/Destination: Planner to Memory, or Memory to Robot

Content: NONE

`MSG_PLAN_REQUEST` — Request entire plan from planner

Sender/Destination: Memory to Planner

Content: NONE

`MSG_PLAN_SUBMIT` — Submit a complete plan

Sender/Destination: Planner to Memory

Content: Plan specification

Table 3: Send/receive message pairs

Message type sent	**Expected response**
`MSG_STATE_UPDATE`	`ACK_STATE_UPDATE`
`MSG_ACTION_REQUEST`	`ACK_ACTION_REQUEST`
`MSG_ACTION_SUBMIT`	`ACK_ACTION_SUBMIT`
`MSG_ACTION_STOPPED`	`ACK_ACTION_STOPPED`
`MSG_PLAN_REQUEST`	`MSG_PLAN_SUBMIT`

5.2 Message passing control loop

The message passing protocol is initially driven by the robot/vision level of the system. Because of the paired send/receive nature of our message set, the upper system levels are forced to coordinate their operations in order to respond appropriately to lower-level messages. Currently, communication only takes place between two "adjacent" levels of the system, i.e., the robot and memory, or the memory and planner (see Figure 5). This means that all communication between the robot and planner must flow through the memory level, which typically acts as a forwarding service, but may also observe or refine the flow of messages (see below). Because the message passing protocol is mainly driven by the robot level, the memory and planning levels operate as message servers that respond to message queries. This protocol also permits certain message exchanges between the planner and memory levels, however, that can interrupt the standard robot-driven process. It is also worth noting that nothing in the implementation of the communication architecture prevents us from expanding this protocol in the future to permit direct point-to-point communication between any two components of the system.

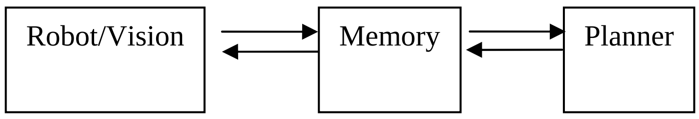


Figure 5 Flow of messages between the three system levels

5.2.1 Robot-level control loop

At the robot level, the message-processing control loop follows a relatively simple repeating pattern where the robot essentially drives the message-passing process and the upper levels of the system respond to queries. The robot-level control loop defines a very simple synchronous cycle wherein a message is sent and its acknowledgement is received before the next message can be sent. As a result, the robot only executes one action at a time and provides updates on the state of the world before the next action begins.

At an abstract level, we see the interaction between the robot and the higher levels follow the *RobotLevelControlLoop* pseudo code given in Figure 6(a). After an initial report on the world state, the main communication cycle consists of an action request by the robot, which is fulfilled by the upper levels (ultimately the planner), an indication from the robot when the action has finished executing, followed by an update on the new state of the world. Messages to and from the robot level all pass through the memory level. Thus, a request made by the robot for a planning-level service (e.g., requesting a new action) will ultimately reach the planner after being forwarded through the memory.

5.2.2 Memory-level control loop

Unlike the more tightly-regulated control loop of the robot level, communication at the memory level is more loosely structured using a client-server architecture. In particular, the memory is able to respond to requests from both the robot and the planner, as well as initiate certain messages of its own.

In most cases, the memory will initially act as a forwarding service that delivers messages from the robot to the planner, and from the planner to the robot. One notable exception is the receipt of `MSG_ACTION_SUBMIT` messages from the planner. In order to properly process such messages, the memory level must first examine the message contents. In the case of a `findout-open` action, the memory must decide how to best satisfy the request for information (by examining semantic memory or initiating object exploration), before reporting the results to the planner. Reporting is done in the high level's language, ensuring that it will be understood by the planner. If object information is readily available in the semantic memory, then this information can be communicated directly from the mid level to the high level. If the mid level has to initiate object exploration, then additional message exchanges between the robot and mid level may be required before the final results are returned to the high level. (This protocol also supports a future bottom-up role for the memory, where the middle level "abstracts" sub-symbolic robot-level information into a symbolic form understandable by the planner, using the transfer from episodic to semantic memory.) In the case of `grasp-fromTable` and `grasp-fromTopOfStack` actions, the memory must refine the message contents to specify a particular grasp type before forwarding the message to the robot level.

The memory is also able to directly request information about the structure of a plan from the planner. The planner will respond with a complete description of the current plan, which may be a conditional plan with branches. The memory can then use this information as needed, for instance to help in its decision making concerning refinement activities.

The pseudo code for the memory-level control algorithm is given in Figure 6(b).

5.2.3 Planning-level control loop

The planning level control loop also operates in a client-server fashion, responding to messages sent from the memory level (but typically originating from the robot level). The planning level is responsible for constructing high-level plans and feeding the actions, one at a time, to the robot level through the memory level. The planner also receives world state updates from the robot (again, through the memory) as well as status reports as to the success or failure of performed actions.

The memory level is also able to interact with the planner to request a complete description of the current plan. This part of the protocol provides the memory level with greater information about a plan's structure, which could be analysed in order to help direct future operations of the memory level, or refine future actions sent to the robot. Future versions of the communication protocol may also allow the planner to directly "push" such plan information to the memory, for instance as a result of re-planning operations. The general planning-level control algorithm is given in Figure 6(c).

The message passing architecture we have outlined has a number of advantages. First, the protocol clearly separates the operations of the three system levels and the interactions between the levels, with the memory level acting as a form of mediator or interpreter. For instance, this protocol allows for the possibility of different content formats for messages flowing between the lower and upper levels of the system (e.g., messages with sub-symbolic information between the robot and memory, and messages with symbolic information between the memory and planner). The only proviso is that the contents of all message exchanges be understood by the receiving level. Also, future changes to the communication protocol involving one pair of levels need not force changes to the interaction of another pair of levels. Finally, we have designed our message set to support much more complex and flexible control architectures, which might arise in the future. For our initial integration tasks, however, the existing process we have outlined is more than sufficient.

Direct link between low level and high level: In terms of the initial integration efforts between SDU and Edinburgh, the above protocol does not specify any major changes to existing work. Instead, the memory level can be viewed as a message-forwarding module that holds the place of the full mid-level memory component, in order to bring the previous SDU/Edinburgh architecture in line with the protocol described here. Although this module will simply pass messages to the other system levels, its addition should facilitate the inclusion of a more fully-featured module into current integration efforts at a later date when a memory system is made available. The necessary code for the message-forwarding module is provided as part of Edinburgh's supplied communication library.

```
Proc RobotLevelControlLoop
    Send: MSG_STATE_UPDATE; Receive: ACK_STATE_UPDATE;
    while !termination loop
        Send: MSG_ACTION_REQUEST; Receive: ACK_ACTION_REQUEST;
        Receive: MSG_ACTION_SUBMIT; Send: ACK_ACTION_SUBMIT;
        Send: MSG_ACTION_STOPPED; Receive: ACK_ACTION_STOPPED;
        Send: MSG_STATE_UPDATE; Receive: ACK_STATE_UPDATE;
    endLoop
endProc
```

a)

```
Proc MemoryLevelControlLoop
    while !termination loop
        choose
            Send: MSG_PLAN_REQUEST;
        or
            Wait for message receive;
            case MSG_ACTION_SUBMIT:
                Process action (action refinement/information gathering,  forwarding and response);
            case MSG_PLAN_SUBMIT:
                Update memory with received plan;
            case all other message types:
                Forward message;
        endChoose
    endLoop
endProc
```

b)

```
Proc PlannerLevelControlLoop
    while !termination loop
        Wait for message receive;
        case MSG_STATE_UPDATE:
            Update world model;
            Send: ACK_STATE_UPDATE;
        case MSG_ACTION_UPDATE:
            Send: ACK_ACTION_REQUEST;
            Construct plan/get next action in plan;
            Send: MSG_ACTION_SUBMIT; Receive: ACK_ACTION_SUBMIT;
        case MSG_ACTION_STOPPED:
            Process action success/failure;
            Send: MSG_ACTION_SUBMIT;
        case MSG_PLAN_REQUEST:
            Construct plan/get entire plan;
            Send: MSG_PLAN_SUBMIT;
    endLoop
endProc
```

c)

Figure 6: Message passing control algorithms

5.3 Socket communication library and sample code

For ease of implementation Edinburgh has defined a set of C++ classes for manipulating message types and message contents. These classes work in conjunction with a lightweight socket library (also written in C++) that has been developed for Linux, to facilitate communication between system components.

At the code level, message types are chosen from a list of predefined `enum` types, and message contents are simple C++ strings.[2][3] Currently, the content of the `MSG_STATE_UPDATE` message must be a list of instantiated high-level properties that form the world state. Similarly, the action specification content of the `MSG_ACTION_SUBMIT` message is a single instantiated high-level action. The content of the `MSG_PLAN_SUBMIT` message will be a plan similar to those in the example plans, but encoded as a Prolog-style list (see below for an example). A plan iterator class is provided for inspecting the structure of conditional plans in this format. (For more details, refer to the sample code provided with the socket library, available from Edinburgh.)

For initial testing purposes we terminate a plan by having the planner send a `MSG_ACTION_SUBMIT` message to the memory level in response to an action request, with the string `"EOP"` as its content. The memory level will then pass this message on to the robot. Both the memory and robot levels must then send a final `ACK_ACTION_SUBMIT` message to the level above, at which point all system levels are free to terminate communication. In the future, plan termination will force the suspension of the main control loop (i.e., the planner will not send an action) until a new goal is given to the planner and a new plan is constructed.

The communication library is distributed with a set of sample programs that implement the basic message passing protocol described in this document for the robot, memory, and planner components. These

[2] The current version of the communication library also defines messages for introducing new objects, new properties, and new actions into the planning-level domain description. We are still in the process of extending the message passing protocol to include these new message types and, thus, we have not included a discussion of these messages at this time.

programs focus solely on the communication interface, with little additional functionality. (For instance, the memory level program currently forwards messages without refining actions and always requests a complete plan after the first robot level request for an action.) It is hoped that these programs can serve as the basis for the development of more sophisticated modules that can simply be "plugged" into the communication architecture. A series of pregenerated plans are included with this software, however, to test the message exchanges between the three levels.

5.4 Message passing example

To better understand the flow of messages between the three system levels, we consider the scenario in Example 2, where the robot is tasked with the goal of clearing the open objects from a table. Figure 7 shows the messages sent by the three levels during the execution of the first action, `findout-open(obj2)`, in the conditional plan constructed for Example 2 (i.e., one complete cycle of the robot-level control loop).

We note that the first message sent by the robot, `MSG_STATE_UPDATE`, provides the planner with its initial description of the world. We assume that on initialization the robot/vision system will sent such an "unusually complete" world description, as a bootstrapping action. From the perspective of the planning system such a message is no more than a particularly large state update, and requires no extra machinery.

Given an initial state description, the planner can construct a plan to achieve a given high-level goal. The planner sends the actions in this plan to the robot/vision system one step at a time, through the memory, in response to action requests. After the execution of each action the robot/vision system reports an update of the world state back to the planner, again, through the memory. In Figure 7 these updates are described in terms of state changes, however, we have agreed that state updates will initially include a complete (or as near as possible to complete) description of the new world state.

For many of the messages sent in the system, the memory level acts as a forwarding service between the robot and the planner. (In the future the memory will take on a more active role as a mediator or translator between the robot and planner.) The first notable exception is the occurrence of the `MSG_ACTION_SUBMIT` message. Since the action specified in this message is a knowledge-producing action, `findout-open(obj2)`, the memory must retrieve particular state information about `obj2` for the planner. In this case, we assume that the mid level does not possess this information in its semantic memory and must direct the robot level to try "poking" the object using grasp type A. (We use `graspA-poke(obj2)` to denote the modified exploratory action sent to the robot.) The results of this action are ultimately sent to the planner as a state update, from the robot level through the memory level, returning the predicate `open(obj2)`to the planner. (In the example the robot generates the result `open(obj2)` which is sent to the mid level and then to the planner. In practice the mid level may have to infer this conclusion from low-level sensory/visual information.)

Figure 7 also illustrates the results of a `MSG_PLAN_REQUEST` message from the memory to the planner. In this case, the planner responds with a plan of the form:

```
[findout-open(obj2), branch(open(obj2),
     [grasp-fromTable(obj2), putAway(obj2)],[])]
```

This plan corresponds to the complete conditional plan given in Example 2, encoded in a Prolog-style list format for transmission using the communication library. (The communication library provides a helper class for processing plans in the compact list format.) Note that if we continued the execution of this plan, the planner would follow the branch that includes the actions `grasp-fromTable(obj2)` and `putAway(obj2)`, since `open(obj2)` was true. Furthermore, the mid level would have to refine the `grasp-fromTable(obj2)` action by choosing an appropriate grasp type, before forwarding this action to the robot.

	ROBOT	MEMORY	PLANNER
1.	**`MSG_STATE_UPDATE`**: `"ontable(obj1),...,!clear(obj1)"`		
2.		(Forward to planner) **`MSG_STATE_UPDATE`**: `"ontable(obj1),...,!clear(obj1)"`	
3.			**`ACK_STATE_UPDATE`**
4.		(Forward to robot) **`ACK_STATE_UPDATE`**	
5.	**`MSG_ACTION_REQUEST`**		
6.		(Forward to planner) **`MSG_ACTION_REQUEST`**	
7.			**`ACK_ACTION_REQUEST`**
8.		(Forward to robot) **`ACK_ACTION_REQUEST`**	
9.			**`MSG_ACTION_SUBMIT`**: `"findout-open(obj2)"`
10.		*Refine* `findout-open(obj2)` *to* `graspA-poke(obj2)` (Forward to robot) **`MSG_ACTION_SUBMIT`** **`"graspA-poke(obj2)"`**	
11.		(Send to planner) **`MSG_PLAN_REQUEST`**	
12.			**`MSG_PLAN_SUBMIT`**: `"[findout-open(obj2), branch(open(obj2)), [grasp-fromTable(obj2), putAway(obj2)], [])]"`
13.	**`ACK_ACTION_SUBMIT`**		
14.		(Forward to planner) **`ACK_ACTION_SUBMIT`**	
15.	**`MSG_ACTION_STOPPED`**: `"1"`		
16.		(Forward to planner) **`MSG_ACTION_STOPPED`**: `"1"`	
17.			**`ACK_ACTION_STOPPED`**
18.		(Forward to robot) **`ACK_ACTION_STOPPED`**	
19.	**`MSG_STATE_UPDATE`**: `"open(obj2)"`		
20.		(Forward to planner) **`MSG_STATE_UPDATE`**: `"open(obj2)"`	
21.			**`ACK_STATE_UPDATE`**
22.		(Forward to robot) **`ACK_STATE_UPDATE`**	
23.	...	...	...

Figure 7: Example of messages sent during the execution of `findout-open(obj2)` in Example 2.

We note that according to the message passing protocol, `MSG_PLAN_REQUEST` messages could be sent by the memory at other times during its control loop, or not at all, producing slightly different message orderings than those shown in Figure 7. (In the sample code the memory sends a `MSG_PLAN_REQUEST` after receipt of the first `MSG_ACTION_SUBMIT` message from the planner.) Similarly, different message orderings (and additional messages) could be produced if the mid level immediately returned a response to the planner's `findout-open` action without initiating an exploration operation. Alternate message orderings — including messages sent in parallel from different levels — could also arise since the robot, memory, and planner all run as independent processes. The implementation of our message passing protocol ensures that such ordering differences do not lead to problems like deadlock, however.

5.5 High-level plan execution and monitoring

Although the high level is able to construct plans for the proposed object manipulation scenario, and communicate those plans to the other system levels using the message passing protocol, we must still be concerned with how plan failure information should be exchanged between the planner and the other system levels.

In discussions with SDU we have identified the need for a high-level mechanism that would operate closely with the planner in order to monitor plan execution and control re-planning activities. A *plan execution monitor*, currently being built by Edinburgh, will be responsible for assessing both action failure and unexpected state information that result from feedback provided to the planner from the execution of planned actions at the robot level. The difference between predicted and actual state information will be used to decide between (i) continuing the execution of an existing plan, (ii) requesting a more detailed state report about particular properties of the world, and (iii) re-planning from new/unexpected states.

In terms of the integration scenario described in this document, the plan execution monitor will have the added task of managing the execution of plans with conditional branches (resulting from the inclusion of knowledge-producing actions like `findout-open`). When a knowledge-producing action is resolved by the lower levels, the results will be returned to the planner through the memory level, as part of the standard state update cycle. When faced with a conditional branch point in a plan, the plan execution monitor must then make a decision as to the correct plan branch it should execute, based on current state information. If such information is unavailable, for instance due to a failure at the robot/vision level, re-planning activities or requests for additional state information will be triggered as above. It is important to note that the robot/vision system will never be aware of the conditional nature of a plan, and will never receive a "branch" operation like those shown in the example plans above. From the point of view of the robot, it will only receive a sequential stream of actions. This will also be the case for the memory level, except when a complete plan is requested. In such situations a fully-specified conditional plan will be transmitted to the memory level.

Initially, we expect that most plans will fail early, and often, and that most monitoring operations will trigger re-planning activities. Our goal is to implement the basic framework for the plan monitor in the short term, in order to evaluate its effectiveness on plans being executed in the actual robot environment.

6 Towards an implementation on ARMAR-III

An ongoing task throughout the project is the integration of the control architecture as proposed in the previous sections on the humanoid platform ARMAR-III at UniKarl. Within this task, the different modules of the proposed architecture are adapted to match the requirements which follow from the low-level control mechanisms, the kinematic structure and the perceptual abilities of ARMAR-III. The software architecture of the ARMAR-III interface is adapted to provide mechanisms which allow integrating modules of the different partners in a flexible manner. Furthermore the perceptual and manipulative abilities of ARMAR-III are continuously extended in order to perform complex tasks on the humanoid platform.

6.1 Software and control architecture of ARMAR-III

The software and control architecture of the humanoid robot ARMAR-III consists of three layers. On the lowest level, digital signal processors (DSP) perform low-level sensorimotor control realized as cascaded velocity-position control loops. On the same level, hardware such as microphones, loudspeakers, and cameras are available. All these elements are connected to the PCs in the mid-level, either directly or via

CAN bus. The software in the mid-level is realized using the Modular Controller Architecture (MCA2, www.mca2.org). The PCs in the mid-level are responsible for higher-level control (forward and inverse kinematics), the holonomic platform, or speech processing.

The first two levels can be regarded as stable i.e. the implemented modules remain unchanged. The programming of the robot, in particular for the partners, takes place on the highest level only. Here, the so-called *robot interface* (see also Deliverable D1.2, month 6) allows convenient access to the robot's sensors and actors via C++ variables and method calls. As an example, the robot's head can be moved by different variants of the method `MoveHead`, e.g. using an inverse kinematics algorithm or setting all joint angles directly. At the same time, the sensor values for the current joint positions and velocities are available via variables. Access to the arms, the hip, and the platform are offered in the same manner.

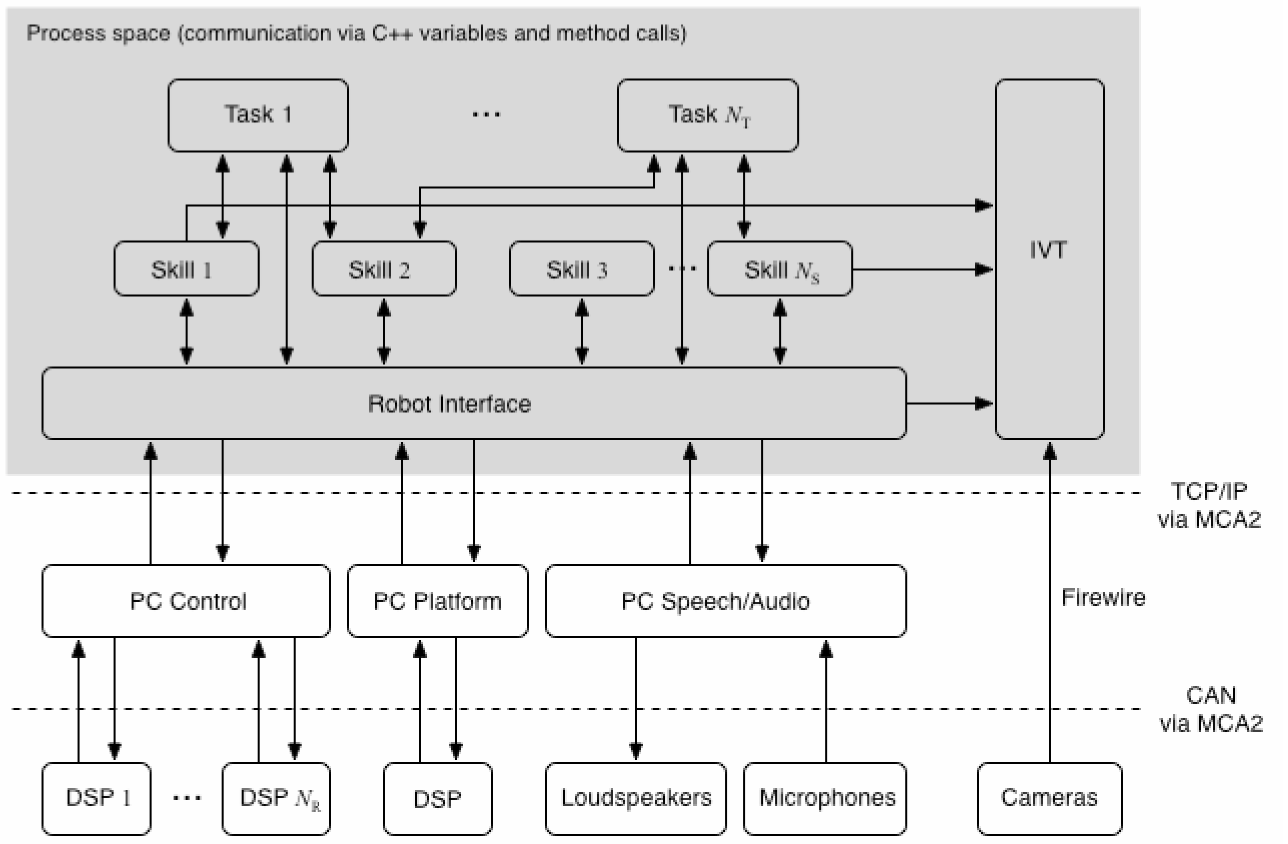


Figure 8 Software and control architecture of ARMAR-III

To allow effective and efficient programming of the robot, in addition to direct access to the robot's sensors and actors, two abstraction levels are defined: tasks and skills. While so far tasks have been implemented manually by hard coded combination of several skills for a specific purpose, the next step is to automatically generate tasks at run-time based on the output of the planning module. In contrast, skills are implemented capabilities of the robot that can be regarded as atomic. Currently, on ARMAR-III the following skills are available:

1. SearchForObjectSkill: Searching for known objects using the active head.

This skill scans the space in front of the robot by moving the head and performing a full scene analysis at each target position. The skill can be parameterized to either search for a specific object or include all object representations currently available in the database.

2. VisualGraspSkill: Grasping of objects.

This skill makes use of a visual servoing approach including one arm and the hip to grasp an object that has been previously recognized and localized. It controls the position closed-loop while continuously updating the position of the robot's hand and the target object. The hand is tracked by using a marker.

3. PlaceSkill: Placing objects on even surfaces

With this skill, it is possible to place a previously grasped object on an even surface, such as a table. Force information acquired from a 6D force sensor in the wrist is evaluated to determine the contact forces with the surface while placing the object.

4. HandOverSkill: Handing over objects to or from the robot

With this skill, objects can be either handed over to the robot or received from the robot. In both cases, information acquired from a 6D force sensor is evaluated to determine when to close respectively open the hand.

5. OpenDoorSkill: Opening various doors

This skill can be regarded as a more complex respectively higher-level skill compared to the previously introduced ones, since it makes use of other skills. It first uses a module that can recognize and localize handles of doors, which are then grasped making use of the VisualGraspSkill. Once the handle has been grasped, this skill opens the door using a force-control approach.

6. CloseDoorSkill: Closing various doors

Using this skill, open doors can be closed, given an initial target position to touch the door of interest. Again, a force-control approach is used to perform the action.

All skills have in common a continuously called run method, in which they perform their action and sensing. The return value of this method signalized its state, i.e. whether the skill is still *operating*, finished with *success*, or finished with *failure*. After success or failure, a skill switches to the state *waiting*. Each skill can be parameterized and has its own specific configuration data structure for this purpose. In the same way, the result of a skill is communicated.

6.2 Connecting High-Level Planning

The software and control architecture introduced in 6.1 provides the basic interfaces for connecting the high-level planning module to ARMAR-III. In order to plan and execute complex tasks, the planning module requires a set of atomic actions which are executable on the robot. Together with required action preconditions and effects a plan towards the desired target can be constructed using these atomic actions. The skills introduced with the ARMAR-III programming interface provide a flexible mechanism to implement such atomic actions on the robot. The skills build the basis for mapping plan-level actions to the robot. The perceptual abilities of the ARMAR-III vision system (IVT) are available on the skill level. With the vision methods provided in IVT, the necessary properties required in the planning level can be evaluated and deployed for plan execution monitoring.

The scenario for implementing high-level planning on ARMAR-III is an extension to the scenario described in Section 1. The extension focuses on action failure and recovery. Initially only two objects are considered for the stacking task. In order to accomplish object stacking on the humanoid platform in a robust manner, the set of actions and properties was adapted. Execution of actions with the five finger hand and the 7 DOF anthropomorphic arm of ARMAR-III, in comparison to the demonstrator at SDU, is more complex and as a consequence will fail more likely. The required properties and actions were extended by taking into account the failures that can occur during perception and action execution on ARMAR-III while performing the object stacking task. Failures will lead to an unexpected state which can be assessed in the plan execution monitor introduced in Section 5.5. The plan execution monitor can decide to re-plan from unexpected states.

The following sections give an overview of the properties and actions which extend the high-level planning representation introduced earlier in order to achieve robust execution on ARMAR-III.

6.2.1 Additional Properties

The following properties are introduced in order to take into account failures during action execution and perception.

`toppled(?x)`

A predicate indicating that object ?x is lying toppled on the table.

An object can be toppled when a manipulation task failed, e.g. object was dropped due to an inaccurate grasp, object was knocked over during arm movements.

`blocked(?x,?y)`

A predicate indicating that object ?x is physically blocked by object ?y.

This predicate indicates a configuration of objects, where the kinematics of the arm and the structure of the manipulator do not allow to grasp object ?x, because access is blocked by object ?y.

`occluded(?x)`

A predicate indicating that object ?x is occluded.

Visual occlusion can result from the initial positioning of objects or from execution failure.

6.2.2 Additional Actions

The following actions are introduced in order to resolve states resulting from action failures or disadvantageous initial setups.

`graspC-placeUpright(?x)`

Grasp a toppled object ?x from the table using Grasp Type C and place it upright.

This action resolves failures which result in a toppled object.

`moveAway(?x)`

Move object ?x to a new position on the table.

Position will be chosen in a way that blocking and occlusion states are resolved.

UniKarl and UEDIN are currently in the process of implementing these extensions as part of ongoing integration activities. The additional properties and actions described above are being incorporated into the existing planning-level domain model (see p.12), and sample plans are being generated for a series of failure scenarios.

References

Hommel, B., Müsseler, J., Aschersleben, G., & Prinz, W. (2001). The theory of event coding (TEC): A framework for perception and action planning. *Behavioral and Brain Sciences*, *24*, 849-878.

Kraft, D., Başeski, E., Popović, M., Batog, A. M., Kjær-Nielsen, A., Krüger, N., Petrick, R., Geib, C., Pugeault, N., Steedman, M., Asfour, T., Dillmann, R., Kalkan, S., Wörgötter, F., & Hommel, B. (2008). Exploration and planning in a three level cognitive architecture. Submitted.

O'Reilly, R.C., & Rudy, J.W. (2001), Conjunctive representations in learning and memory: Principles of cortical and hippocampal function. *Psychological Review*, *108*, 311-345.

Petrick, R., & Bacchus, F. (2002). A knowledge-based approach to planning with incomplete information and sensing. *Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2002)*, 212-221.

Petrick, R., & Bacchus, F. (2004). Extending the knowledge-based approach to planning with incomplete information and sensing. *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-04)*, 2-11.