Object-Action Complexes: Grounded Abstractions of Sensorimotor Processes

Norbert Krüger[a], Christopher Geib[b], Justus Piater[c], Ronald Petrick[b], Mark Steedman[b], Florentin Wörgötter[d], Aleš Ude[e], Tamim Asfour[f], Dirk Kraft[a], Damir Omrčen[e], Alejandro Agostini[g], Rüdiger Dillmann[f]

[a]*Mærsk McKinney Møller Institute, University of Southern Denmark, Odense, Denmark*
[b]*School of Informatics, University of Edinburgh, Edinburgh, Scotland, UK*
[c]*Institute of Computer Science, University of Innsbruck, Austria*
[d]*Bernstein Center for Computational Neuroscience (BCCN), Göttingen, Germany*
[e]*Jožef Stefan Institute, Department of Automatics, Biocybernetics, and Robotics, Ljubljana, Slovenia*
[f]*Institute for Anthropomatics (IFA), Humanoids and Intelligence Systems Laboratories (HIS), Karlsruhe Institute of Technology, Karlsruhe, Germany*
[g]*Institut de Robotica i Informatica Industrial (CSIC-UPC), Barcelona, Spain*

Abstract

This paper formalises Object-Action Complexes (OACs) as a basis for symbolic representations of sensorimotor experience and behaviours. OACs are designed to capture the interaction between objects and associated actions in artificial cognitive systems. This paper gives a formal definition of OACs, provides examples of their use for autonomous cognitive robots, and enumerates a number of critical learning problems in terms of OACs.

Keywords: Robotics, grounding, reasoning about action and change, execution monitoring, machine learning

1. Introduction

Autonomous cognitive robots must be able to interact with the world and reason about the results of those interactions, a problem that presents a number of representational challenges. On the one hand, physical interactions are inherently continuous, noisy, and require feedback (e.g., consider the problem of moving forward by 42.8 centimetres or until a sensor indicates an obstacle). On the other hand, the knowledge needed for reasoning about high-level objectives and plans is more conveniently expressed in a symbolic

form, as predictions about discrete state changes (e.g., going into the kitchen enables retrieving the coffee pot). Bridging the gap between low-level control knowledge and high-level abstract reasoning has been a fundamental concern of autonomous robotics [1, 2, 3, 4]. However, the task of providing autonomous robots with the ability to build symbolic representations of continuous sensorimotor experience *de novo* has received much less attention, even though this capability is crucial if robots are ever to perform at levels comparable to humans.

To address this need, this paper proposes a formal entity called an *Object-Action Complex* (OAC, pronounced "oak") as the basis for symbolic representations of sensorimotor experience. The OAC formalism is designed to achieve two ends. First, OACs provide a computational account that brings together several existing concepts from developmental psychology, behavioural and cognitive robotics, and artificial intelligence. Second, by formalising these ideas together in a shared computational model, OACs allow us to enumerate and clarify a number of learning problems faced by embodied agents. Some of these learning problems are known and have been well studied in the literature, while others have received little or no attention.

OACs are designed to formalise adaptive and predictive behaviours at all levels of a cognitive processing hierarchy. In particular, the formalism ensures that OACs are grounded in real-world experiences: all learning and refinement of OACs will be based on statistics gained from an agent's ongoing interaction with the world. To relate OACs operating at different processing levels, we will also allow OACs to be defined as combinations of other OACs in a hierarchy, in order to produce more complex behaviours. As a result, this formalism enables consistent, repeatable hierarchies of behaviour to be learnt, based on statistics gained during real-world interaction, that can also be used for probabilistic reasoning and planning. It also provides a framework that allows the OAC designer to focus on those design ideas that are essential for developing cognitive agents.

The goal of the OAC formalism is to provide a unifying framework for representing diverse interactions, from low-level reactive behaviour to high-level deliberative planning. To this end, we will build our computational models on existing assumptions and ideas that have been shown to be productive in previous research, in a number of different fields. In particular, we note six design ideas (DI) that have helped motivate our formalism:

DI-1 **Attributes:** Actions, objects, and interactions must be formalised over

an appropriate attribute space, defined as a collection of properties with sets of associated values. An agent's expectations and predictions (see [DI-2]) as to how the world will change if an action is performed must also be defined over such an attribute space. Different representations may require different attribute spaces, plus a method of mapping between them if they are to be used together.

DI-2 **Prediction:** A cognitive agent performing an action to achieve some effect must be able to predict how the world will change as a result of this action. That is, it must know which attributes of the world must hold for an action to be possible (which will typically involve reasoning about the presence of objects), which attributes will change when the action is performed, and how those attributes will change.

DI-3 **Execution:** Many previous efforts to produce fully autonomous robotic agents have been limited by simplifying assumptions about sensor, action, and effector models. We instead take the approach that complete robotic systems must be built with the ability to actually execute actions in the world and evaluate their success. This requires agents to be embodied within physical systems that can interact with the physical world.

DI-4 **Verification:** In order to improve its performance in a nondeterministic physical world, an agent must be able to evaluate the effectiveness of its actions, by recognising the difference between the states it predicted would arise from its actions, and those states that actually resulted from action execution.

DI-5 **Learning:** State and action representations are dynamic entities that can be extended by learning in a number of ways: continuous parameters can be optimised, attribute spaces can be refined or extended, new control programs can be added, and prediction functions can be improved. Embodied physical experiences characterised in terms of actions, predictions, and outcomes provide data for learning at all levels of a system.

DI-6 **Reliability:** It is not sufficient for an agent to merely have a model of the changing world. It must also learn the reliability of this model. Thus, our representations must measure and track the accuracy of their predictions over past executions.

These design ideas are widely accepted in the literature, where they have been discussed by various authors (see, e.g., [5, 6]). For example, a STRIPS-style planning operator [7] can be seen as a prediction function [DI-2] built from action preconditions and effects defined over an attribute space [DI-1]. Significant work has also been done on learning such prediction functions given an appropriate attribute space [8, 9]. The importance of embodiment [DI-3] in real-world cognitive systems has been pointed out by Brooks [1, 2]. Richard Sutton [10] has discussed the necessity of verifying the expected effects of actions [DI-4] to arrive at meaningful knowledge in AI systems. The interplay between execution [DI-3] and verification [DI-5] is associated with the *grounding problem* [11]. For example, Stoytchev [5] defines grounding as "successful verification", and discusses the importance of evaluating the success of actions [DI-6] and maintaining "probabilistic estimates of repeatability". We will discuss the relation of our work to prior work further in Section 3.

1.1. Paper Structure

In the remainder of the paper we will develop the OAC concept using the above design ideas. In particular, this paper will:

- formally define OACs for use by autonomous cognitive agents,
- identify problems associated with learning OACs, and
- provide examples of OACs and their interaction within embodied systems.

The rest of the paper is organised as follows. Section 2 further motivates this work and provides some basic terminology. Section 3 discusses the relation to prior research. Section 4 provides a formal definition of OACs, based on the above design ideas. Section 5 characterises a number of learning problems in terms of OACs. Section 6 describes how OACs are executed within a physical robot system. Section 7 provides detailed examples of OACs. Finally, Section 8 demonstrates a set of OACs interacting with each other to realise cognitive behaviour, including object grounding and associated grasping affordances, as well as planning with partly grounded entities.

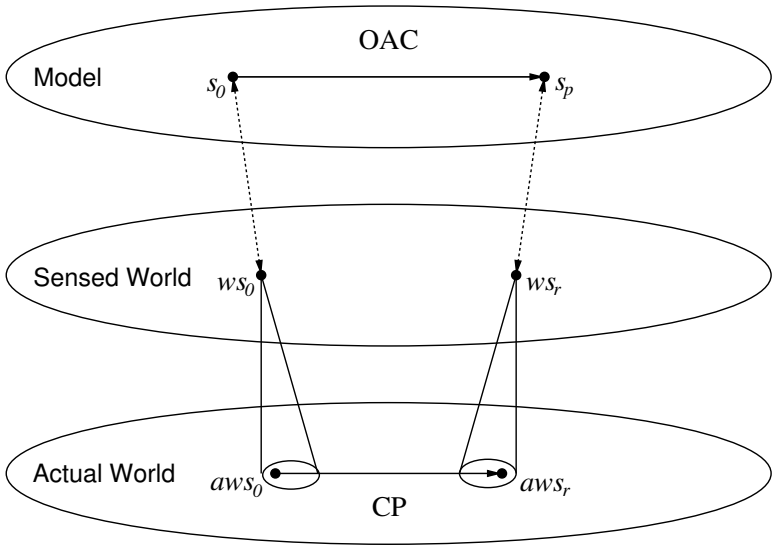

Figure 1: Graphical representation of an OAC and its relationship to a control program.

2. Prerequisites for Modelling OACs

To achieve its goals in the real world, an embodied agent must develop predictive models that capture the dynamics of the world and describe how its actions affect the world. Building such models, by interacting with the world, requires facing a number of representational challenges resulting from

- the continuous nature of the world,
- the limitations of the agent's sensors, and
- the stochasticity of real-world environments.

These problems make the task of efficiently predicting the results of real-world interactions challenging, and often require highly-specialised models of the interaction. As a result, any framework for representing such interactions must be able to support multiple models of the world, based on different attribute spaces. For example, differential equations can be straightforwardly used to predict the trajectory of straight line motions. However, this kind of representation will not be effective for symbolic planning. We will call each model of an interaction with the world an OAC, and stipulate that each OAC be defined over an attribute space.

Given the continuous nature of the world, all of an agent's interactions with the external world must be mediated by low-level continuous control routines. Such routines are necessary for the agent to sense and to act in a noisy, continuous, and uncertain real world. For this exposition, we will assume that the agent has a low-level control program (CP) that it uses to interact with the world.[1] Our objective then is for OACs to capture the interactions with the world meditated by the CPs. In order words, we will describe an OAC as *modelling* a CP.

Our first three design ideas suggest that an OAC must contain a prediction function defined on an attribute space that captures the regularities and results of its specific CP. Figure 1 illustrates this idea graphically with an OAC that predicts the behaviour of a specific CP functioning in the real world to move an agent's end effector. Here, the control program causes changes in the actual world that transform the actual initial state of the world, denoted by aws_0 (and sensed by the agent as ws_0), to the resulting actual world state, aws_r (sensed by the agent as ws_r).

An OAC that models this CP must also be able to map states of the sensed world to states represented in terms of its own attribute space, and to make predictions about the transitions that are caused by the CP. In Figure 1 this is captured by a correspondence between ws_0 in the sensed world and the initial state s_0 in the OAC's attribute space, and the OAC's predicted state s_p and the resulting sensed state ws_r.

In practice, we can simplify this diagram slightly. Since the agent's perception of the world is completely mediated by its sensors and effectors, any change in the world can only be observed by the agent through its (possibly faulty) sensors. Further, because the available sensor set of a given agent is fixed, we can treat the actual world and the sensed world as a single level, as shown in Figure 2. While we recognise the presence of errors in the sensors and the inherently un-sensed variation of the world, since we are not modelling learning over evolutionary time scales, we also assume that all embodied agents must learn based on the noisy and incomplete sensors provided to them. We will make this assumption for the remainder of the paper.

[1] We will discuss how new CPs can be learnt later in this document, but for the purpose of introducing this idea we will simply assume a CP is given.

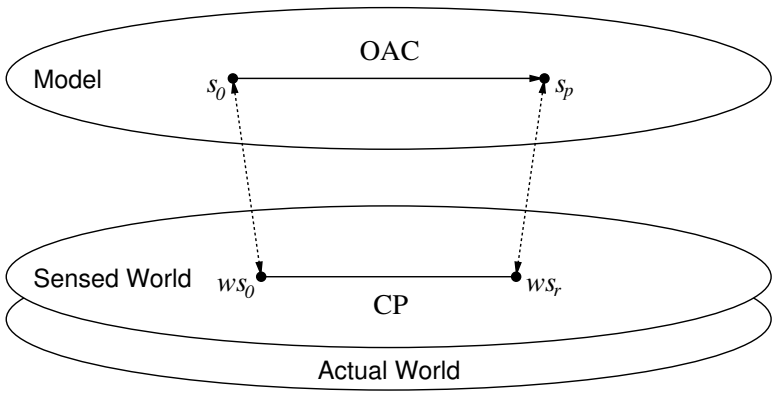

Figure 2: Graphical representation of an OAC and its relationship to the sensed world and a control program.

2.1. Representational Congruency and Grounding

For an OAC to model a CP and be effective for high-level reasoning tasks, it must consistently capture the underlying regularities present in the execution of the CP. One way to do this is to ensure that the states modelled by an OAC are inferable from sensed features of the world, and that relevant changes in the sensed world resulting from the execution of the CP are reflected in the states predicted by the OAC. We will call this property *representational congruency*, and will refer to OACs with this property as *congruent models* or *congruent representations* of the CP.

Representational congruency imposes strong conditions on an OAC's prediction function and attribute space, with respect to the CP it models, as illustrated in Figure 2. In particular, if s_0 is an initial state in the OAC's attribute space, corresponding to the sensed state ws_0, and the sensed state ws_r results from the execution of a CP in ws_0, then the state s_p predicted by the OAC (and represented in the OAC's attribute space) must map to the sensed state ws_r. In practical terms, such guarantees are necessary for ensuring the correctness of high-level reasoning tasks that have consequences at the sensed world level. (For instance, building high-level plans that are interpreted in terms of low-level effector commands.) We will provide a formal definition of representational congruency, and a further discussion, in Section 6.

We note that nothing about representational congruency requires OACs to share attribute spaces with the sensed world. For instance, a single state in the OAC's attribute space might denote a set of states at the sensed world

level. In general, a representationally congruent OAC is free to abstract its attribute space from the sensed world in any way that is effective for its reasoning task. This allows each OAC to develop and work with representations that are specific to their own reasoning tasks.

We also note that learning such congruent models requires a mapping from the sensed world state to the OAC's attribute state that is *consistent*. In other words, a given sensed state of the world must always map to the same state in the OAC's attribute space. Without such consistencies, regularities in the execution of the CP cannot be recognised, let alone learnt and modelled by an OAC. Given consistent mappings, we envision the congruency of an OAC increasing as experience extends the OAC's attribute space.

Following DI-3, DI-4, and DI-5, we will also require all OAC learning and refinement to be based on statistics gained through an agent's interaction with the world, in order to ensure that the resulting OACs are grounded in real-world execution and sensor feedback. Thus, we envision cognitive systems using OACs to solve a problem at a high level of abstraction while grounding their real-world interactions with low-level control programs and sensed world states. Further, while this section has discussed OACs as being grounded by executing a single control program, in Section 6 we will discuss how OACs can be defined as a combination of lower-level OACs. This will enable consistent, probabilistic reasoning and planning based on statistics gained during the execution of OACs in the world.

With these intuitions in hand, we will now discuss the relationship of OACs to prior work.

3. Relation to other Approaches

The OAC concept provides a *framework* for formalising actions and their effects in artificial cognitive systems, while ensuring that relevant components and prerequisites of action acquisition, refinement, chaining and execution are defined (e.g., the attribute space, a prediction of the change of the attribute space associated with an action together with an estimate of the reliability of this prediction, an execution function, and a means of verifying the outcome of an action). In particular, this framework ensures the grounding of an OAC in sensory experience by means of incremental verification and refinement ("ongoing learning"). It also specifies which components of an OAC are subject to learning as outlined in Section 5. Our OAC definition, however, does not specify the actual learning algorithms (e.g., whether this

learning takes place in a neural network, by means of reinforcement learning, or based on Bayesian statistics); this is up to the designer of the concrete OAC. As such, OACs ensure certain properties of action representation are fulfilled, leaving the designer free to specify the remaining content. The OAC framework thus provides a basis for the design of elementary cognitive units and their interaction. Naturally, it is based on a significant amount of prior work on action representations, as we will outline below.

A closely related concept from psychology is the (sensorimotor) schema as defined by Piaget and others [12, 13]. A sensorimotor schema is a dynamic entity that gathers together the perceptions and associated actions involved in the performance of behaviours. The schema represents knowledge generalised from all the experiences which have been involved in the executions of that behaviour. It also includes knowledge about the context in which the behaviour was performed as well as the agent's expectations about the action effects. Cognitive development takes place by refining and combining these schemas. OACs can be seen as a formalisation of such schemas to be used in artificial cognitive systems.

Together, the different components of OACs formalise concepts which have been derived over the last decades in cognitive science, artificial intelligence and robotics. We discuss related work in terms of four subjects that are addressed by the OAC concept: (1) the definition and learning of suitable attribute spaces and the predictions taking place in these spaces, (2) the concept of affordances, (3) the grounding of symbolic entities by the agent's interaction with the world, (4) the modularisation of actions allowing for their flexible combination, (5) hybrid control schemes, and (6) learning and memorisation.

Attributes and the prediction of expected change: The representation of world states in terms of discrete attribute spaces, and the representation of actions as expected changes to the values of these attributes, can be directly linked to STRIPS [7] and other classical formalisms [14, 15, 16]. Predictability of cause and effect (or the lack of it) is important for cognitive agents and has been treated in a large body of work [17, 18, 19, 20, 21, 22]. However, unlike classical formalisms, the prediction function associated with an OAC constitutes a dynamic and grounded entity, changing under the influence of ongoing learning processes in the cognitive system.

More specifically, OACs go beyond such classical representations by permitting both continuous and discrete attribute spaces, making it possible to

use OACs at different levels of a processing hierarchy, from low-level sensorimotor processes for robot perception and control, to high-level symbolic units for planning and language. As a consequence, OACs can be viewed as containers enabling subsymbolic as well as symbolic representations, and models of both symbolic and subsymbolic cognition can be formalised using OACs (see [23]).

Structures like POMDPs (see, e.g., [24]) are related to OACs in that they are also defined in terms of states, actions, and state transitions. However, unlike OACs, they employ specific probabilistic representations tailored to optimal action selection with respect to reward signals. POMDPs are not concerned with the issue of grounding their abstract representations in physical experience (see below). OACs provide more generic formalisations of actions in a cognitive system, also allowing for non-probabilistic representations in which action selection may not be the primary goal.

OACs also facilitate the learning of their associated prediction functions, an idea which is closely related to statistical structure learning [25, 26, 27, 28, 9, 8, 19], and learn how successful their executions are over particular time windows. In particular, in early development, when actions are likely to be unsuccessful, it is important to ensure that such execution uncertainties can be reasoned about. The storage of statistical data concerning execution reliability also has important applications to probabilistic planning [19], where an OAC's probability of success can be utilised to compute optimal plans. Consistently successful plans can then be memorised for future reference.

Affordances: OACs combine the representational and computational efficiency of STRIPS rules [7] and the object- and situation-oriented concept of affordance [29, 30]. Affordance is the relation between a situation, usually specified to include an object of a defined type, and the actions that it allows. While affordances have mostly been analysed in their purely perceptual aspect, the OAC concept defines them more generally as state-transition functions suited to prediction. Such functions can be used for efficiently learning the multiple representations needed by an embodied agent for symbolic planning, execution, and sensorimotor control.

Grounding and Situatedness: OACs reflect a growing consensus concerning the importance of grounding behaviour in sensorimotor experience, which has been stressed in the context of embodied cognition research (see, e.g., [11, 31, 32, 33]). To build a truly cognitive system, it is necessary to have the system's representations grounded by interacting with the physical

world in a closed perception-action loop [32]. OACs are necessarily grounded by their execution functions (Section 6), and are learnt from the sensorimotor experiences of the robot (Section 5). Thus, OACs realise grounding by "successful verification" [5] in an ongoing learning process.

The ability of OACs to formalise sensorimotor processes on different levels of the cognitive hierarchy allows high-level abstract actions to be formally grounded in sensory motor experience by means of lower-level actions. We have exemplified this by a "Birth of the Object" process [34, 35] described in Section 8.1. By this process, rich object descriptions and representations of grasping affordances (i.e., the association of potential grasping options to an object and their associated success likelihoods) emerge through interactions with the world. As we outline in Section 8.1, this process can be understood as the concatenation of several low-level perception-action interactions that are formulated in terms of OACs, leading to processes in which symbolic entities emerge (i.e., the notion of a specific object) and can be used on the planning level. Note that this is very much in line with prior work by others [6, 36] where representations and actions are likewise grounded through interaction. Differences in the specificities of our visual and motor representations compared to [6, 36] are discussed in detail in [35].

Modularity: The principle of modularity is widespread in cognitive process modelling (e.g., vision [37, 38] and motor control [39, 40, 41]), allowing the agent to make use of acquired perception and action competences in a flexible and efficient way. As we will demonstrate in Section 7, this concept is also inherent in the structure of OACs: OACs often operate at increasing levels of abstraction, each with a particular representation of situations and contexts. For instance, we will outline three examples of OACs for grasping objects. On the lowest level, continuous end-effector poses are associated to visual feature relations for grasping completely unknown objects. This OAC can be used to model reactive or affordance-based behaviours (see [42, 30]) as outlined in Section 7.2.2. At an intermediate level in another grasp-related OAC (described in Section 7.3), grasp densities are used to hypothesise possible grasps when the agent has some object knowledge [44]. Finally, at the highest level, plans effectively use grasps to manipulate objects on an abstracted symbolic scene representation (see Section 7.4).

Hybrid control schemes: OACs can be seen as a unifying representation for modelling control schemes in hybrid (i.e., discrete-continuous) dynamical systems (see, e.g., [45]). In this way, they are related to the idea of hybrid

control in systems which combine discrete events with continuous dynamics. In most practical cases, hybrid control architectures formalize discrete abstractions of inherently continuous control problems. For example, the task of manipulating an object can be decomposed into four subtasks: (1) reaching the grasp position, (2) grasping the object, (3) moving the object, and (4) placing the object at the goal position. From a hybrid control point of view, the subtasks associated with object manipulation can be described as discrete events, e.g., represented as finite state machines with continuous dynamics for each state. Each of the states might represent low-level, continuous controller operating on motor torques, sensor readings, etc., with discrete state transitions triggered by specific conditions of the lower-level controllers, or by external environmental stimuli.

Such hybrid control schemes can be implemented with OACs representing states, and their control programs implementing the low-level controllers. However, OACs can provide more than state models in hybrid dynamical systems. OACs can model open-loop, one-shot actions with stochastic outcomes, and be stacked into hierarchical architectures containing different layers of abstraction. At higher, symbolic levels, OACs can also be composed from other OACs, to be used for new tasks in different contexts.

Learning, Evaluation, and Memorisation: Cognitive agents must learn from past experience in order to improve their own development, a task that typically requires a form of memory as a means of tracking prior interactions (see, e.g., [46]). While memory itself is not often a problem, such processes must ensure efficient representation, with properties like associative completion and content addressability [47, 48, 49, 50], to enable machine learning from stored instances presented over a period of time.

Learning is also modularised through the OAC concept. In our example OACs, the lowest-level OAC learns the difference between successful and unsuccessful grasps. Using this as a base, another OAC learns alternative object-specific ways of posing the hand. Again, building on this OAC, another OAC learns the abstract preconditions and effects of grasping. Careful maintenance of the attribute spaces of the different OACs allows systems to benefit from the modularity of the information learnt for each OAC. As outlined in Section 8, the OAC formalism ensures that relevant data for learning is stored (in terms of "experiments"), and that learning is taking place at all times at all levels (even when learning is not the explicit goal of the agent).

4. Defining OACs

Our OAC definition is split into two parts, (1) a *symbolic description* consisting of a prediction function [DI-2] defined over an attribute space [DI-1], together with a measure of the reliability of the OAC [DI-6], and (2) an *execution specification* [DI-3] defining how the OAC is executed by the embodied system and how learning is realised [DI-5] by verification [DI-4].

This separation is intended to capture the difference between the knowledge needed for cause and effect reasoning (represented in the symbolic description), and the procedural knowledge required for execution (encapsulated in the execution specification). Since we do not constrain the form of the attribute space, OACs are not limited to continuous or discrete representations of actions. Instead, as we will see in Section 7, our definitions are flexible enough to accommodate both kinds of representations.

In the remainder of this section we will provide a formal definition of an OAC's symbolic description.

Definition 4.1. *We call the properties of the world captured by an OAC* ***attributes****. Each attribute has an associated range of possible values that can be assigned to that attribute.*

Intuitively, attributes can represent any sensed or derived property that we want our OACs to capture. In particular, Definition 4.1 does not make any commitments about attributes being continuous, discrete, or Boolean. This provides the OAC formalism with the flexibility to reason about very different problem spaces.

Definition 4.2. *An* ***attribute space*** $\mathcal{S}$ *is the set of all possible assignments of values to a set of attributes. A* ***state*** $s \in \mathcal{S}$ *denotes a (possibly partial) assignment of values to the attributes in the space.*

Since we have not limited the form of the attributes we permit, an attribute space can be very expressive, and an individual state description can abstract over a possibly large number of real-world states. Even a complete individual state in the OAC's attribute space can capture a possibly infinite number of real-world states. For example, a complete state specification that includes the assignment of the value "full" to the attribute "statusGripper" represents all the world states where the gripper is full, provided the other attributes of the world state are consistent with those of the OAC's state.

We also allow state descriptions to be partial, where values are only specified for a subset of the attributes in the space. For example, if the value "full" is assigned to the attribute "statusGripper", and no values are specified for any of the other attributes in the state space, then the resulting partial state denotes the set of all states where the gripper is full, regardless of the other attribute values. As a result, this state representation provides a powerful method for OACs to abstract over large state spaces.

We now turn our attention to formally defining OACs.

Definition 4.3. *An **Object-Action Complex (OAC)** is a triple*

$$(E, T, M) \tag{1}$$

where:

- *E is an identifier for an execution specification,*
- *$T : \mathcal{S} \rightarrow \mathcal{S}$ is a prediction function defined on an attribute space $\mathcal{S}$ encoding a model of how the world (and the agent) will change if the execution specification is executed, and*
- *M is a statistical measure representing the success of the OAC in a window over the past.*

Definition 1 characterises OACs using three main components. In the examples we will discuss here, the execution specification E identifies a single CP whose execution is modelled by the OAC. This means that multiple OACs can share the same underlying CP.[2]

In general, much of the actual world state will be irrelevant for most OACs. Therefore, we stipulate that the attribute space $\mathcal{S}$ captures all and only those attributes of the world that are needed for T to make its predictions. Thus, for a given OAC, $\mathcal{S}$ will often omit sizable portions of the sensed world, but may include specialised attributes derived from multiple sensors. Since observations are costly in real world systems, we can use the reduced space of $\mathcal{S}$ to constrain observations and allocate system resources more efficiently, resulting in a reduced sensor load for verifying OAC execution.

M codes an evaluation of the OAC's performance over a time window in the past. Given the diversity of attribute spaces we can define for OACs, M

[2]We will discuss more complex execution specifications in Section 6.

must be flexible enough to capture the reliability of many types of prediction functions. As a result, we allow each OAC to define M as a statistical measure appropriate for its needs. Thus, different OACs in a single system might define M in very different ways. For example:

- In a simple domain where an OAC is used until it fails and then is never used again, we might define M as a Boolean flag that indicates whether the OAC has failed.
- In a more complex domain where M tracks the accuracy of an OAC's prediction function over a certain time window in the past, we might define M as a pair made up of the expected value of the OAC's performance and the sample size used to compute the expected value.
- In even more complex domains it might be convenient to store statistical data beyond the expected value. For example, lower-level OACs might maintain statistical information about the differences between observed and expected changes in a number of specific attributes.

Note that the size of the temporal window over which M is collected is OAC dependent. In general, during learning (where large changes can significantly affect the success likelihood) smaller windows might be appropriate to judge whether learning is making good progress, whereas in the case of a mature OAC a larger window (and hence a more stable estimate of the success likelihood) might be appropriate.

To provide some intuition, we can imagine an agent with the following example OACs, defined on very different attribute spaces. These examples are described more formally in Section 7.

Ex-1 *An OAC that encodes how to push an object on a table based on the agent's end-effector pose space and the location of the object.* In this case, the OAC might predict the position of an object after a pushing action by the end-effector, depending on the velocity and force vector as well as the shape of the object. For M, the OAC might maintain the average deviation (over a certain time window) of the prediction of the position and the measured position after pushing the object.

Ex-2 *An OAC that encodes how to grasp an unknown "something" in a scene.* In this case, the OAC might predict the success or non–success (e.g., when the "something" is out of reach) of the grasping attempt. For M,

this OAC might store the likelihood of a successful grasp over a time window in the past.

Ex-3 *An OAC that encodes how to grasp a specific object in a specific scene suggesting an optimal gripper pose.* In this case, the OAC might also predict the success or non–success (e.g., in case the object is in a non-graspable pose) of the grasping attempt. For M, this OAC might store the likelihood of successfully grasping the object over a time window in the past.

Ex-4 *An OAC that encodes how to grasp an object for the purpose of planning (e.g., to systematically clean a table).* In contrast to Example Ex-3, at the planning level the precise control information required to grasp an object is not relevant. Rather, higher-level attributes such as the object affordances that become executable after a successful grasp need to be coded (e.g., the objects that are now movable to a shelf). For M, this OAC might store the likelihood that the grasp is successful over a time window in the past.

We will provide more detailed definitions of these example OACs and their reliability measures in Section 7. First, however, we will motivate the discussion of how OAC-based learning is formalised with the following definition.

Definition 4.4. *Let* `execute` *be a function with side effects that maps an OAC, defined on an attribute space $\mathcal{S}$, to a triple of states called an* ***experiment****, i.e.,*

$$\texttt{execute} : (E, T, M) \rightarrow (s_0, s_p, s_r), \tag{2}$$

where:

- $s_0 \in \mathcal{S}$ *is the state of the world before performing the OAC's execution specification,*
- $s_p \in \mathcal{S}$ *is the state of the world that T predicts will result from performing the OAC's execution specification in s_0, i.e., $s_p = T(s_0)$, and*
- $s_r \in \mathcal{S}$ *is the observed state resulting from actually performing E in state s_0.*

The side effect of this function is that the execution specification of the OAC is actually performed in the real world by the agent.

input : an OAC (E, T, M)
output: an experiment (`so`, `sp`, `sr`)
begin
    `so` = `stateCapture`(T);
    `sp` = T(`so`) ;
    `agentExec`(E);
    `sr` = `stateCapture`(T);
end

Algorithm 1: An implementation of `execute`.

Calling `execute` with an OAC causes the OAC's execution specification to be performed in the real world, producing an experiment as a result. This experiment is an *empirical event* dynamically created from the sensed and predicted states: its first element is derived by sensing the state of the world before execution, the middle term captures the OAC's prediction about the state that should have resulted, while the last element encodes the actual state of the world after execution. For example, an experiment for Example Ex-1 might include:

- the initial state of the end effector and the object,
- the predicted state of the object, and
- the actual state of the object after the execution.

We can imagine implementing `execute` with the pseudo-code in Algorithm 1. Here, `agentExec` is a function that causes the agent to perform the specified execution specification, and `stateCapture` is a function that captures the current state of the world, expressed in the attribute space of the given prediction function. For instance, in Example Ex-1, executing the "pushing OAC" launches a process that (1) captures the initial state, (2) invokes the prediction function on the initial state to predict the end state of the object after a pushing movement, (3) invokes the associated control program, (4) waits for it to terminate, (5) captures the resulting state of the object, and (6) reports all three states in the form of an experiment. We note that all OAC-specific processing takes place within the execution specification. For the rest of this paper, we will refer to the process of calling `execute` with a specific OAC as *executing an OAC*.

In our discussion up to this point, we have only considered a single OAC modelling a single control program, which simplifies the definition of the

execution specification: all we need to provide is the identifier of the control program that is to be executed. Given this mapping, `execute` has all the information it needs to invoke the specified control program, allow it to run until termination, and report the results as an experiment. In Section 7 we will see more detailed examples, and provide a discussion of how such one-to-one mappings can be built up in Section 8. However, we can also imagine much more complex specifications than the execution of a single control program. In particular, OACs might be defined in terms of other OACs, or sets of OACs. We will discuss this in more detail in Section 6.

As empirically grounded events, the experiments returned by `execute` can be used to update OACs in cycles of execution and learning (see Section 7) based on evaluations of their success [DI-4]. For instance, each of our example OACs might update their respective Ms on the basis of an experiment.[3] In the next section we explore particular learning problems in terms of OACs.

5. Learning OACs

The definition of an OAC as an entity that captures both symbolic and control knowledge for actions gives rise to a number of learning problems that must be considered for OACs to be effective. We note that each of these learning problems can be addressed by recognising that differences can exist between predicted states and actual sensed states. In practice, these problems may require different learning algorithms (e.g., Bayesian, neural network-like, parametric, non-parametric, etc.), and it is left to the OAC designer to choose an appropriate learning mechanism in each case.

As such, the following characterisations are intended to specify those aspects of the OAC that can be modified through learning, rather than a specific learning method. We consider four main learning problems, each of which is labelled in Figure 3, and illustrate these problems using the examples introduced in Section 4.

1. **Translation:** *(Learning the mapping of real-world states to OAC states)* This learning task produces the mapping from sensed world states to states in the OAC's attribute space. It also involves identifying and

[3] We leave open the possibility that an experiment might not be used immediately for learning, but could be stored in some type of short term memory (see, e.g., [46]) until resources for learning are available.

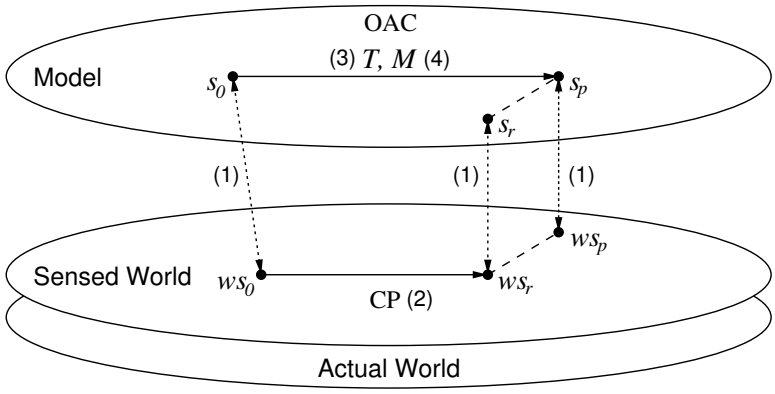

Figure 3: Graphical representation of the OAC learning problems: (1) Translation, (2) Control, (3) Prediction, and (4) Reliability.

adding to the OAC's attribute space those attributes of the world model that are required for effectively predicting interactions with the world. For instance, in Example Ex-1, this process would be responsible for adding new attributes (beyond object shape) such as the mass distribution of the object on the basis of more low-level sensory (visual and haptic) information, or the audio information caused by the object scraping along the surface.

2. **Control:** *(Learning control programs)* This learning task modifies an OAC's control program to minimise the distance between the world state ws_p predicted by the OAC and the actual sensed state ws_r. For instance, in Examples Ex-2 and Ex-3, when a grasp is not successful even though the OAC's T function predicts success, the control program can be modified to produce a successful grasp.
3. **Prediction:** *(Learning the prediction function)* This learning task modifies the prediction function to minimise the distance between a predicted model state s_p, and the actual resulting model state s_r. In Example Ex-1, this can be done by optimising the prediction function to produce a better estimate of the final state of the object after a push.
4. **Reliability:** *(Learning the prediction function's long term statistics)* This learning task updates the OAC's reliability measure M to reflect the long-term success of the OAC. In Examples Ex-2, Ex-3, and Ex-4, this process might record the last 100 attempts, and evaluate how many had been successful.

We reiterate that all of these learning problems can be addressed by recognising the differences between predicted states and actual sensed states as captured by experiments (i.e., through ongoing verification). However, the details and specifications of how each of these learning tasks might be performed at a given level of abstraction may vary wildly depending on the details of the attribute space. One of the critical contributions of this work is in enumerating and formalising these problems within the OAC framework.

In the following sections, we will use a set of common function names to denote each of these learning problems. Although these functions would have to be appropriately tailored to a particular OAC if we were to actually implement them, we will simply refer to them as: `updateModel`, `updateCP`, `updateT`, and `updateM`, respectively, and assume that each function takes an experiment as an argument.

6. Representational Congruency and Hierarchical Execution

Before we introduce hierarchical executions of OACs in Section 6.2 and Section 6.3, we begin by discussing a fundamental problem connected to OAC modelling, and a structural property for OACs that was earlier referred to as *representational congruency*.

6.1. Representational Congruency

When an OAC is executed, all the states returned to the OAC by an experiment are defined within the OAC's attribute space. This means that even in *mature* OACs (i.e., OACs that are well developed and are undergoing very little additional modification), it is possible for there to be states of the actual world that may not be predicted or (adequately) captured in the OAC's attribute space. In such OACs, there is no guarantee that its performance could be enhanced even by introducing additional attributes (e.g., by means of `updateModel`, where we actually extend an OAC's attribute space). This is even more true for less mature OACs that do not have fully developed attribute spaces: OACs that are "missing" attributes may fail to make accurate predictions. As a result, OACs are only as effective at predicting the outcomes of interactions as their learnt models allow them to be.

Representational congruency is a property that aligns an OAC's attribute space with that of a control program (or another OAC, as we'll see in Section 6.2), ensuring the completeness of the OAC's prediction function is improved. To formalise this idea, we provide the following definition.

Definition 6.1. *Let $\boldsymbol{A} = (E, T, M)$ be an OAC defined on an attribute space $\mathcal{S}_A$, and let $\mathcal{S}_{sense}$ be the agent's "foundational attribute space" defined by the agent's set of sensors and the complete set of their possible values. $\boldsymbol{A}$ is said to be* ***representationally congruent*** *to the control program captured by E iff $\forall ws_0, ws_r \in \mathcal{S}_{sense}$ and $\forall s_0, s_p \in \mathcal{S}_A$, such that s_0 and s_p are the respective projections of ws_0 and ws_r into $\mathcal{S}_A$, and the execution of E by $\boldsymbol{A}$ in a sensed world state ws_0 gives rise to a sensed world state ws_r, then it follows that T maps s_0 to s_p.*

Note that representational congruency is not a necessary property of an OAC. Since our OAC definition doesn't say anything substantive about the prediction function, any function is permitted. However, prediction functions that consistently fail to produce sound and complete mappings (with respect to the actual sensed world) won't be useful for reasoning, even if they are permitted by the OAC definition. As such, representational congruency provides the logical underpinning for an OAC's attribute space and prediction function to accurately model real-world interactions.

As a result, representational congruency as described in Definition 6.1 is not a property that we assume OACs begin with. Instead it is a "target" property that OACs converge towards as they improve their underlying models. In this view, Definition 6.1 captures a types of completeness property that may not be fully achieveable in practice. However, the intuition behind this definition, that representationally congruent OACs correctly predict the states that result from the execution of a control program, is a property that is essential if OACs are to be effective at certain reasoning tasks.

In the next section we will discuss more complex configurations of OACs and what executing such OACs means to representational congruency and our notion of an experiment.

6.2. Towers of OACs

It is worth recognising that, beyond being attached to external sensors, there is no significant difference between the attribute space of an OAC and the sensed world within which a CP operates. A CP moves the agent from one state of the sensed world to another, while the execution of an OAC moves the agent from one state of its attribute space to another. Building on this correspondence, we can consider OACs that use the attribute space of another, more basic OAC, as their "sensed world" and define their execution specification in terms of these more basic OACs.

Generalising this idea results in "towers" of OACs where each OAC stands in one-to-one relation with an OAC (or a control program in the base case) that is beneath it in the tower. In such cases, the execution specification of each OAC is just the recursive invocation of the OAC beneath it in the tower. Calling `execute` for the highest-level OAC results in a stack of calls to `execute`, one for each level of the tower, where each OAC invokes the OAC at the next level down until the process terminates with the execution of a single control program. The experiment that results from this execution must then be returned back up the tower, and appropriately translated into the attribute space of each OAC, as the result of each `execute` call.

For instance, consider the planning-level grasping OAC in Example Ex-4, operating in a discrete state space with an abstract description of objects and their graspability. This OAC's execution specification could invoke the OAC in Example Ex-3 which operates in the lower, continuous space of concrete gripper poses. A call to the high-level OAC in Ex-4 would then result in a call to the lower-level OAC in Ex-3 which computes a concrete end effector pose and triggers the execution of the control program. At each level, the resulting experiments would be passed back to the respective OACs.

We can modify our definition of representational congruency to permit towers of OACs. Recall that Definition 6.1 required an attribute space derived from the agent's sensor set. To extend representational congruency, we alter this definition to refer to the attribute space of the execution function in general. This results in the following revised version of Definition 6.1:

Definition 6.2. *Let $\boldsymbol{A} = (E, T, M)$ be an OAC defined on an attribute space $\mathcal{S}_A$, and let $\mathcal{S}_E$ be the attribute space of the OAC or CP specified by E. $\boldsymbol{A}$ is said to be **representationally congruent** to the execution specification captured by E iff $\forall s'_0, s'_p \in \mathcal{S}_E$ and $\forall s_0, s_p \in \mathcal{S}_A$, such that s_0 and s_p are the respective projections of s'_0 and s'_p into $\mathcal{S}_A$, and the execution of E by $\boldsymbol{A}$ in a state s'_0 results in a state s'_p, then it follows that T maps s_0 to s_p.*

We note that while Definition 6.2 is sufficient for describing representational congruency in towers of OACs, it will need further extension if we are to capture OACs with even more complex execution specifications, since the attribute spaces for such constructs could be substantially more complex.

We have also discussed how an experiment resulting from executing OACs in a tower must be passed back to each constituent OAC, and translated into the attribute space of that OAC. This means that the attributes and values

of a higher-level OAC's attribute space must be definable in terms of the attributes and values of the lower-level OAC. To ensure this property holds, we impose the following restriction on the attribute spaces of towers of OACs.

Definition 6.3. *Let* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *be OACs and let* $\mathcal{S}_A$ *and* $\mathcal{S}_B$ *be the attribute spaces of* $\boldsymbol{A}$ *and* $\boldsymbol{B}$*, respectively. If* $\boldsymbol{A}$ *has an execution specification defined in terms of* $\boldsymbol{B}$*, then all the attributes of* $\mathcal{S}_A$ *must be derivable from the attributes of* $\mathcal{S}_B$*. In such cases we will say that* $\boldsymbol{A}$ *and* $\boldsymbol{B}$ *are* ***hierarchically defined****.*

We will see examples of towers of OACs in Section 7.4 and Section 8.2.

6.3. One-to-Many Execution

One-to-one mappings are not the only kind of relationship we can envision for OACs. We can also imagine more complex scenarios, where an OAC is mapped to a sequence of OACs or control programs, or has an execution specification that involves iteration, conditional invocation, or parallel execution. For example, an OAC for opening a door might be comprised of a sequence of lower-level OACs that include actions to approach the door, grasp the doorknob, twist the doorknob, pull on the doorknob, etc. In order to execute such a higher-level OAC, each of these lower-level OACs must be successfully executed in the correct sequence.

A formal definition that permits one-to-many execution specification requires ordering constraints and success criteria for each of the sub-OACs. Furthermore, a correct understanding of the execution specification for such OACs must, like the one-to-one case, rest on recursively calling the `execute` function and continually monitoring the execution of the underlying OACs. We will not provide a detailed definition of such complex execution behaviour in this paper. Instead, we leave the specification and learning of such behaviours as an area for future work.

7. Examples of OACs

In this section, we give formal descriptions for a number of OACs. Some of these OACs have already been discussed informally as part of our running examples (Ex-1—Ex-4), while others are new. For each OAC, we provide a definition of its attribute space ($\mathcal{S}$), prediction function (T), success measure (M), and execution specification (E). We also discuss learning in these OACs, and show how they can be embedded within procedural structures

Section	Name	Attribute space/T	M	Learning
7.1 (Ex-1)	**AgnoPush**	End effector's pose space, object location and shape	Average deviation of prediction from actual final position	T, M
7.2 (Ex-2)	**AgnoGrasp**	Space of coplanar contour pairs, gripper status	Long term probability of successful grasp	CP, M
7.3 (Ex-3)	**ObjGrasp**	Object model, gripper status	Long term probability of successful grasp	CP, M
7.4 (Ex-4)	**PlanGrasp**	Logic-based rules	Long term probability of correct result prediction	T, M
7.4 (Ex-4)	**PlanPush**	Logic-based rules	Long term probability of correct result prediction	T, M

Table 1: Summary overview of example OACs.

to produce more complex behaviour. In Section 8, we will present examples of these OACs interacting with each other, to demonstrate grounding and planning. Table 1 provides an overview of the example OACs for comparison.

*7.1. Example Ex-1: Object Pushing (**AgnoPush**)*

In this example we define an OAC **AgnoPush** which models a pushing action that moves objects in a desired direction on a planar surface without grasping. Pushing as a nonprehensile action cannot be realised with sufficient accuracy to ensure a given object can be moved to a desired target in one step, i.e., by applying one pushing movement. If a higher-level planner specifies that object o should be pushed to a certain target, **AgnoPush** needs to be applied iteratively in a feedback loop until the target location is eventually reached. To achieve this, the system needs to know how objects move when short pushing actions are applied to them. In particular, the motion of the pushed object depends on various properties including shape, mass distribution, and friction. Here we will focus on shape. (A more detailed

description can be found in [51].)

7.1.1. Definition of ***AgnoPush***

Defining $\mathcal{S}$: Some prior knowledge needs to be available before **AgnoPush** can be learnt. In particular, we assume that the robot knows how to move the pusher (e.g., the robot hand or a tool held in its hand) along a straight line in Cartesian space. We also assume that the robot knows how to localise the observed objects by vision. The central issue for **AgnoPush** is to learn to predict the object's movement in response to the pusher's movement. To this end, the robot needs information about the object's shape, its current location on the planar surface, the duration of the pushing movement, and its direction relative to the point of contact on the object's boundary. We represent the object's shape by a 2D binarized image, such as those shown in Figure 4. Such images are sufficient as shape models (as opposed to full 3D shape models) because **AgnoPush** only encodes the response to an applied pushing action for objects that do not roll on planar surfaces.

More formally, $T_{\mathbf{AgnoPush}}$ is defined on the attribute space

$$\mathcal{S} = \{\text{bin}(o), \text{loc}(o), \tau, a\},$$

where $\text{bin}(o)$ is the shape model in the form of a binary image of the object to be pushed, $\text{loc}(o)$ denotes the initial location of the object o, τ is the duration of the push, and a denotes the parameters describing the pushing movement, i.e., the contact of push on the object's boundary and the direction of the movement of the pusher.

Defining T: Based on the information in this attribute space, we can predict the object's new location using the transformation

$$T(\text{bin}(o), \text{loc}(o), \tau, a) = V(\text{bin}(o), \text{loc}(o), a)\tau + \text{loc}(o), \tag{3}$$

where V is the function predicting the outcome of the push in terms of the object's linear and angular velocity.

T returns the expected position and orientation of the object after it has been pushed at a given point of contact and angle with constant velocity for a certain amount of time. The angle of push is defined with respect to the boundary tangent. These parameters are fully determined by the object's binary image and the pusher's Cartesian motion. Thus,

$$T : \mathcal{S} \longrightarrow \{loc(o)\}$$

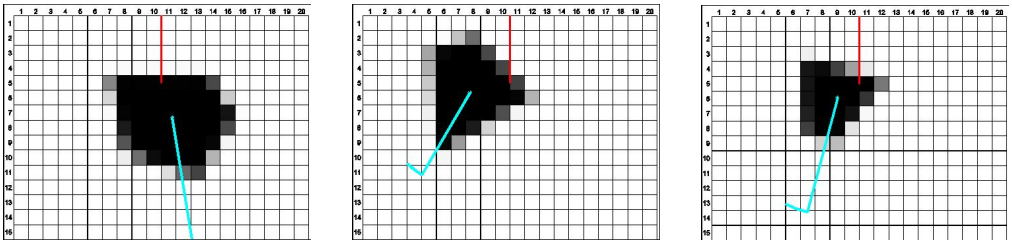

Figure 4: Samples of low resolution object images used as input to the neural network.

maps an initial state $(\text{bin}(o), \text{loc}_o(o), \tau, a)$ containing a concrete shape $\text{bin}(o)$, a location $\text{loc}_o(o)$ before the action and a specific poking action parameterized by (τ, a) to a predicted location $\{\text{loc}_p(o)\}$ after the action.[4]

Defining M: The statistical evaluation M measures how close the predicted object movement is to the real object movement over a certain time window. We define a metric $\text{d}(\text{loc}_p(o), \text{loc}_r(o))$ to measure the difference between the expected and actual object movement on the planar surface. The expectation of **AgnoPush**'s performance after N experiments is thus given by

$$M = \frac{1}{N} \sum_{i=1}^{N} \text{d}(\text{loc}_p(o)_i, \text{loc}_r(o)_i),$$

where i denotes different pushing trials (see Figure 5, right).

Defining E: An impulse to push an object in a certain direction must be provided by a higher-level cognitive process. The appropriate parameters to the pushing control program can be determined based on the available prediction function T. These issues will be discussed in Section 7.1.2. However, the control program modelled by **AgnoPush** is neither object nor target dependent. This means that the execution specification of this OAC simply calls the pushing control program with parameters a and τ computed by an external process.

Calling `execute` results in an experiment of the form

$$(\{\text{bin}(o), \text{loc}_o(o), \tau, a\}, T(\text{bin}(o), \text{loc}_o(o), \tau, a), \text{loc}_r(o))$$

[4]For brevity in Section 7 and Section 8 , we will often provide partial state descriptions when discussing T and the experiments resulting from `execute`, highlighting the significant parts of the state, rather than simply reporting complete states.

that is created by performing four major functions:

1. Capturing the initial state: For **AgnoPush**, this requires both extracting the binary image of the object $\text{bin}(o)$ and its location $\text{loc}_o(o)$, and acquiring the pushing movement parameters a.
2. Capturing the predicted resulting state: This is done by calculating $T(\text{bin}(o), \text{loc}_o(o), \tau, a)$.
3. Executing the execution specification: The pushing movement is performed by calling the pushing control program with parameters a and τ.
4. Capturing the actual resulting state: This is done by localising the object after the push, i.e., by observing $\text{loc}_r(o)$.

When the task is to push an object towards a given target location, the robot can solve this by successively applying `execute` in a feedback loop until the goal has been reached. Note that in this example the control program that realises straight-line motion of the pusher in Cartesian space is fixed and does not need to change while learning **AgnoPush**.

*7.1.2. Learning in **AgnoPush***

Learning in **AgnoPush** affects both its prediction function and its long-term statistics. A process for learning the prediction function (denoted by the function `updateT`) is realised using a feedforward neural network with backpropagation. The trained network encodes a forward model for object movements that have been recorded with each pushing action. To ensure that **AgnoPush** can be applied to different objects, the shape is specified in the form of a low resolution binary image, which is used as input to the neural network. Function T is updated incrementally based on the observed movements of the pushed objects. Statistical evaluation is also done incrementally as experiments are performed (a process denoted by the function `updateM`). Note, however, that since the prediction function T changes during learning, the statistical evaluation only converges to the true accuracy of the behaviour once T becomes stable (see Figure 5).

There are two modes of operation in which we consider **AgnoPush**:

A. Initial learning of the prediction function T, where the pushing movements encoded by the parameter a are randomly selected, and
B. Pushing the object towards a given target, where the current pusher movement a is determined based on the previously learnt prediction function and the given target location.

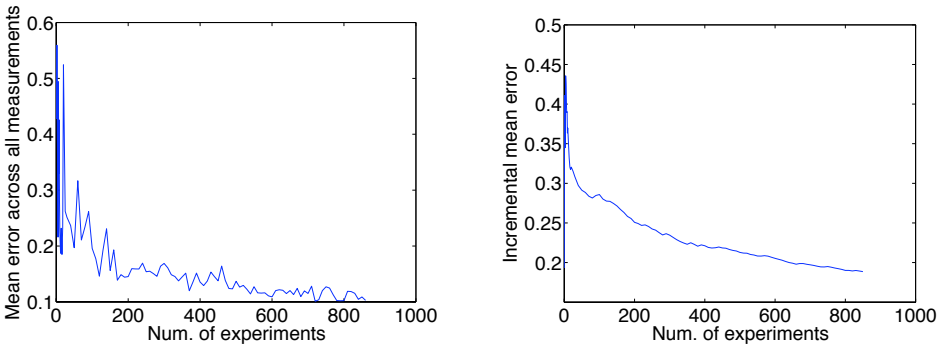

Figure 5: Mean error of robot pushing. The left figure shows the mean error of the predictor on the available data, i.e., after each update of the predictor we evaluate its performance on all previous experiments. The right figure shows the incremental statistical evaluation as realised by `updateM`. Four different objects were used in the experiment.

Figure 6: Pushing behaviour realised by **AgnoPush** after learning prediction function T.

As described above, the prediction function T is given in Equation (3), where velocity V is encoded by a neural network with the binary image of an object, the point of contact and the direction of the pusher movement used as input values, and the predicted final position and orientation of the pushed object as output. In mode B, we calculate the optimal pusher movement a (i.e., the point of contact and the direction of the push) by first extracting the object's binary image and determining the desired Cartesian movement of o from its current location towards the target location. The neural network is then inverted using nonlinear optimisation (see [51] for details). The resulting behaviour is shown in Figure 6.

The learning process has been implemented using explorative behaviour as shown in Algorithm 2. In this context, `updateT` estimates the weights of the neural network (for details on the learning algorithm, see [51]). To ensure that the data used for training is not used to estimate the performance of

while *true* **do**
  $a = \texttt{SelectRandomMotion};\ \mathrm{bin}(o);\ \mathrm{loc}_o(o);$
  $\texttt{expr} = \texttt{execute}(\textbf{AgnoPush});$
  if $\mathrm{d}(\mathrm{loc}_o(o), \mathrm{loc}_r(o)) > \epsilon$ **then**
    `updateM(expr);`
    `updateT(expr);`
  end
end

Algorithm 2: Explorative behaviour to learn **AgnoPush**. The constant $\epsilon > 0$ is used to determine whether the object has moved or not.

the prediction function, `updateM` is always applied to the data before it has been used to refine the prediction function. This loop also demonstrates how OACs can be embedded in procedural structures. We will see more examples of such procedures in the following sections.

*7.2. Example Ex-2: Object Independent Grasping (**AgnoGrasp**)*

Next, we consider an OAC **AgnoGrasp** that predicts the success of attempts to grasp unknown objects, based on an associated grasping hypothesis (specified in terms of the 6D pose of the gripper) for a co-planar contour pair (see Figure 7(a),(b) and [52]). Such grasp hypotheses are "agnostic" to the object being grasped, hence the name of the OAC. As a result, **AgnoGrasp** represents a visual feature/grasp association that enables an unknown "something" to be grasped (see Figure 7(d)).

*7.2.1. Definition of **AgnoGrasp***

Defining $\mathcal{S}$: We note that two co-planar contours define a plane which determines the orientation normal of the pose (i.e., two orientation parameters) for any possible grasp. The position and main orientation of a contour in 3D space determines the position of the 6D pose and the one remaining orientation parameter of the grasping hypothesis. This allows us to associate a grasp hypothesis $G^{\mathrm{H}}(C_i, C_j)$ with any pair of co-planar contours (C_i, C_j) (see Figure 7(b)). Such grasp hypotheses can then be executed by the system.

Formally, **AgnoGrasp** is defined on the attribute space:

$$\mathcal{S} = \{\texttt{statusGripper}, \Omega, \texttt{statusGrasp}\}\,.$$

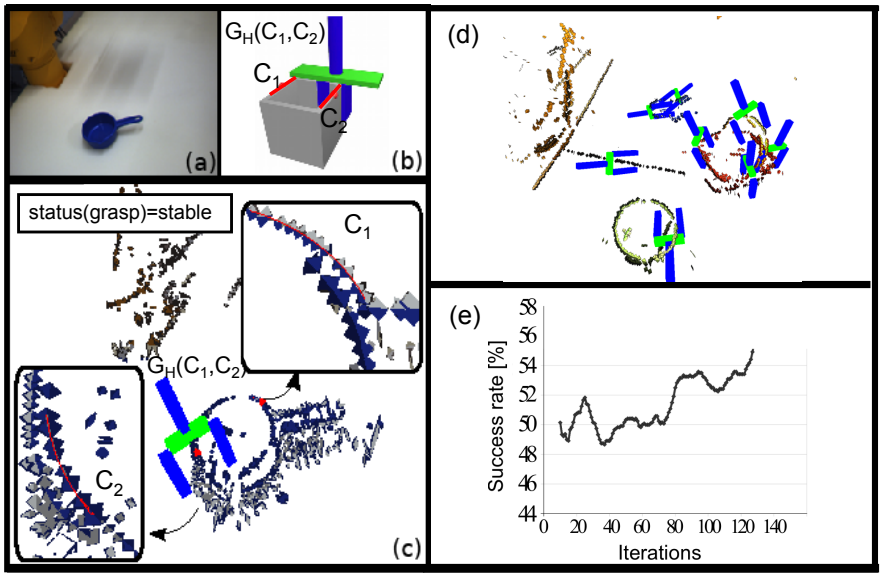

Figure 7: (a) The image of the scene captured by the left camera. (b) A possible grasping action type defined by using the two coplanar contours C_1 and C_2 shown in red. (c) A successful grasping hypothesis. The 3D contours from which the grasp was calculated are shown. Note that the information displayed is the core of an experiment. (d) A selected set of grasping hypotheses generated for a similar scene. (e) Change of performance as measured by M as a result of the learning process.

This attribute space contains the set Ω containing the co-planar contours in the scene and the status of the gripper `statusGripper` which either can take the value 'full' or 'empty'. In particular, it requires that (1) there are co-planar contours $C_i, C_j \in \mathcal{C}$ in the scene (i.e., the set of co-planar contours Ω is not empty), and (2) the gripper is empty.

Defining T: AgnoGrasp's prediction function determines the value of the attribute `statusGrasp`:[5]

$$\texttt{statusGrasp} \in \left\{ \begin{array}{l} \texttt{undefined}, \texttt{noplan}, \texttt{collision}, \\ \texttt{void}, \texttt{unstable}, \texttt{stable} \end{array} \right\}.$$

[5]Note that the current implementation of our learning algorithm only uses two classes, *success* which is equivalent to `stable`, and *failure* which corresponds to all other states except `noplan`, where the generated experiments are ignored for learning. More advanced learning algorithms might also use extended information on the stability of a grasp.

The possible values of `statusGrasp` each capture an outcome of the execution of **AgnoGrasp**. Before execution, `statusGrasp` is set to `undefined`. After selecting a specific grasping hypothesis, a motion planner tries to find a collision-free path that allows the arm to reach the pregrasping pose associated with the grasping hypothesis, which may result in a number of possible outcomes. If the planner fails to find a suitable trajectory or decides there is none, execution stops, and the result is `noplan`. If the hand unexpectedly enters into a collision, execution stops at that point, and the result is `collision`. If the closed gripper is determined to be empty, the result is `void`. If the gripper closes further while lifting the object, the result is `unstable`. Otherwise, the grasp is deemed successful, and the result is `stable`. In our case, $T_{\mathbf{AgnoGrasp}}$ simply maps to a state where $\texttt{statusGrasp} = \texttt{stable}$ holds.[6]

Defining M: In **AgnoGrasp**, the reliability measure $M_{\mathbf{AgnoGrasp}}$ is simply defined as the percentage of successful grasps in a time window of 100 grasping attempts.

Defining E: Like **AgnoPush**, **AgnoGrasp**'s execution specification is based on executing a low-level control program. In the case of **AgnoGrasp**, the CP requires as input a pair of co-planar contours from the scene (where a grasp hypothesis can be computed) that is chosen from the set of contours Ω. Thus, prior to execution, many grasping hypotheses from co-planar contour pairs are computed and a single pair is chosen for execution.[7]

This means that when performing `execute`, the initial state is given by:

$$\{\Omega, \texttt{statusGripper}\},$$

where Ω is the set of contours. The predicted state is simply an assertion that $\texttt{statusGrasp} = \texttt{stable}$ holds. After the chosen grasp hypothesis is performed, the grasp status $\texttt{statusGrasp}_{t+1}$ is sensed. This results in an experiment of the form:

$$(s_0, \texttt{statusGrasp}_{t+1} = \texttt{stable}, \texttt{statusGrasp}_{t+1}).$$

[6]The use of a constant mapping here not only represents the most likely outcome but, for certain reasoning tasks, the most wanted outcome. Space prohibits a comprehensive discussion of the motivation behind such mappings.

[7]In practice, the pair is chosen according to a ranking criterion. See [52] for details.

while *true* **do**
    compute contours pairs and associated grasping hypotheses
    `expr` = `execute`(**AgnoGrasp**);
    `updateCP`(`expr`);
    `updateM`(`expr`);
    drop object
end

Algorithm 3: A simple learning cycle for **AgnoGrasp**.

(See Figure 7(c) for the main components of an experiment.) Each experiment can either be used directly for on-line learning, as in the learning cycle in algorithm 3, or stored in an episodic memory for off-line learning at a later stage (see [53] for details).

*7.2.2. Learning in **AgnoGrasp***

In **AgnoGrasp**, learning affects the execution of the control program (through the `updateCP` function), and the updating of long-term statistics (via `updateM`; see Figure 7(e)). We do not consider other learning problems and, in particular, the OAC's prediction function always remains constant. Learning modifies the selection of the most promising grasping hypothesis and, thus, the control program underlying the execution function. In practice, the optimal choice of grasps depends on certain parameters, such as contour distance and the object position in working space (see Figure 7(d)). Based on an RBF network (see [53] for details), a function estimates the success likelihood that a certain grasp has been learnt in a cycle of experimentation and learning.[8] Algorithm 3 formalises this exploratory behaviour, which realises a simple learning cycle for **AgnoGrasp**.

*7.3. Example Ex-3: Object Specific Grasping (**ObjGrasp**)*

In this example, we consider an OAC **ObjGrasp** that models the grasping options for a specific object, and their associated success likelihoods, by means of *grasp densities* (see Figure 8 and [44]).

[8]In practice, such learning has provided an increase in the success rate from 42% to 51% (see [53] for details). Note that since **AgnoGrasp** uses very little prior knowledge, a high performance cannot be expected except in trivial scenarios.

*7.3.1. Definition of **ObjGrasp***

Defining $\mathcal{S}$: Object models o_i are stored in an object memory $\mathcal{M}^{\mathrm{O}}$.[9] An object model includes a learnt, structural object model that represents geometric relations between 3D visual patches (i.e., *early cognitive vision* (ECV) features [54]) as Markov networks [55]. In addition, it contains a continuous representation of object-relative gripper poses that lead to successful grasps by means of grasp densities [44]. Object detection, pose estimation, and the determination of useful gripper poses for grasping the object are all done simultaneously using probabilistic inference within the Markov network, given a scene reconstruction in terms of ECV features.

The attribute space for **ObjGrasp** is defined by

$$\mathcal{S} = \{\texttt{statusGripper}, \texttt{targetObj} = o, \texttt{statusGrasp}\}.$$

Here, the state description includes an attribute `targetObj` that specifies an object model o that is provided by the `execute` function as an input to the control program this OAC models. As notation, we will add a subscript to the OAC's name to identify this object model (e.g., $\textbf{ObjGrasp}_{\mathrm{Basket}}$). Like the two previous OACs, this model is chosen by processes external to **ObjGrasp**. The state description also includes `statusGripper` and `statusGrasp` as in Section 7.2, however, `statusGrasp` is only relevant to the predicted state.

Defining T: As with **AgnoGrasp**, the prediction function T always returns an assertion that `statusGrasp` = `stable` is true.

Defining M: The reliability measure M for **ObjGrasp** is defined as the cumulative outcome of statistics from executing this OAC (which is updated as part of a learning cycle; see Figure 9).

Defining E: Like **AgnoGrasp**, the execution of **ObjGrasp** requires its input parameters to be passed to a control program for execution. In the case of **ObjGrasp**, this parameter is the object to be grasped. When the `execute` function is performed, the process of capturing the initial state must:

1. access or reconstruct the current scene in terms of ECV features, and

[9] See Section 8.1 for more information about learning such models.

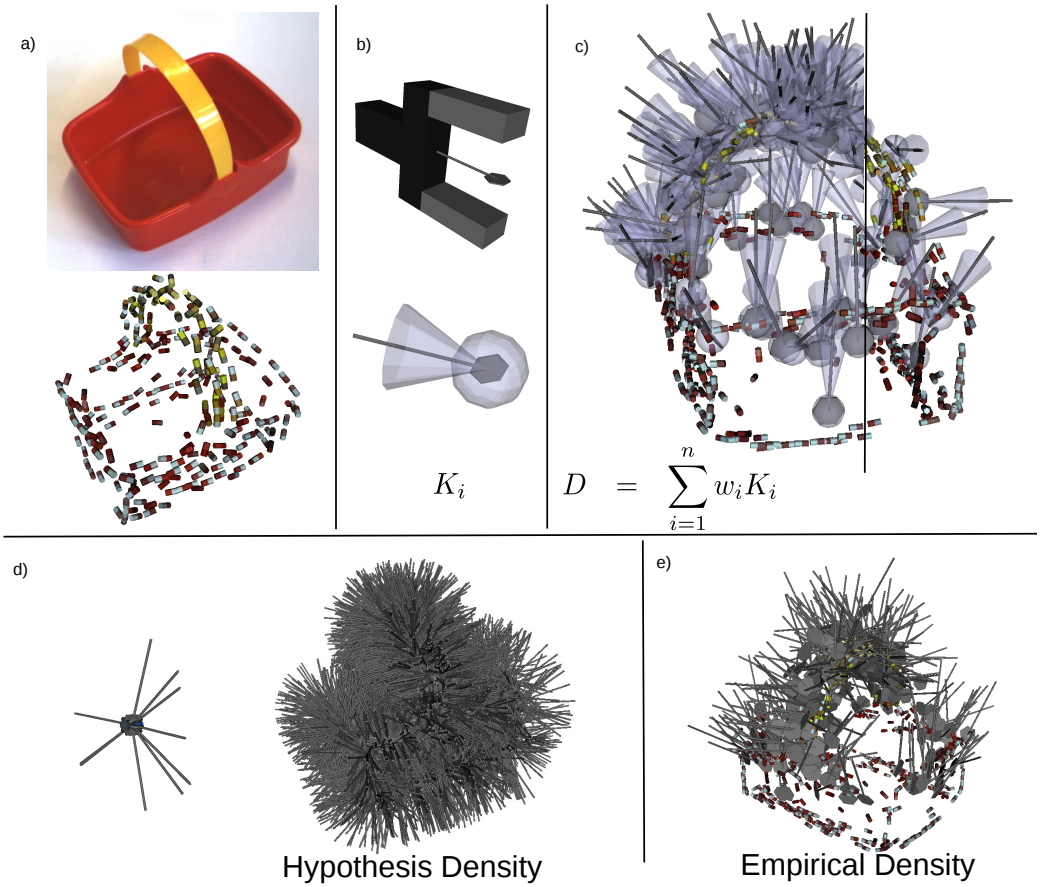

Figure 8: Mechanisms used by **ObjGrasp**. a) Objects (top) are represented as Markov networks [55] in terms of 3D ECV features [54] (bottom). b) In the following subfigures, gripper poses (grasps) are visualised as "paddles" (top). Grasp densities are obtained from individual grasps by kernel density estimation using $SE(3)$ kernels, as illustrated by unit-variance isosurfaces for 2 rotational and 3 positional degrees of freedom (bottom). c) A grasp density D associated with the basket (a). The right-hand side shows sparser samples for better visibility. d) Grasp hypothesis densities for specific objects such as the basket (right) are generated at uniform orientations around 3D ECV features (left). e) Empirical grasp density learnt by testing grasps drawn from a hypothesis density [44].

while *true* **do**
    compute ECV features
    `expr` = `execute`(**ObjGrasp**);
    `updateCP`(`expr`);
    `updateM`(`expr`);
    drop object
end

Algorithm 4: Exploration learning procedure for **ObjGrasp**.

2. retrieve the object model o from $\mathcal{M}^{\mathrm{O}}$, use it to locate the object, and determine a gripper position from the associated grasp density (see Figure 8).

Like **AgnoGrasp**, the OAC's prediction function returns `statusGrasp` = `stable`. The actual execution specification of the OAC encompasses a small, two-step control program:

1. First, a path planner generates a plan for manoeuvring the gripper to the intended position.
2. If such a plan is found, the CP executes the computed trajectory, and closes the gripper to grasp the object.

This yields a new state characterised by an attribute `statusGrasp` that can take on any of the values in the attribute space of the OAC **AgnoGrasp**, or the value `nopose`, which represents the case that no object instance can be reliably located. As a result of E, an experiment of the form

$$(\{\texttt{statusGripper}, o, \texttt{statusGrasp}\}, \texttt{statusGrasp}_{t+1} = \texttt{stable}, \texttt{statusGrasp}_{t+1}).$$

is returned.

We note that objects are always located within the currently-sensed part of a scene. Thus, it is up to other parts of the system to make sure that the scene reconstruction available to `execute` contains one and only one instance of the object o, e.g., by directing sensors accordingly.

*7.3.2. Learning in **ObjGrasp***

Algorithm 4 outlines how a higher-level process might acquire and refine grasping skills on a variety of objects. In this scenario, the scene contains up

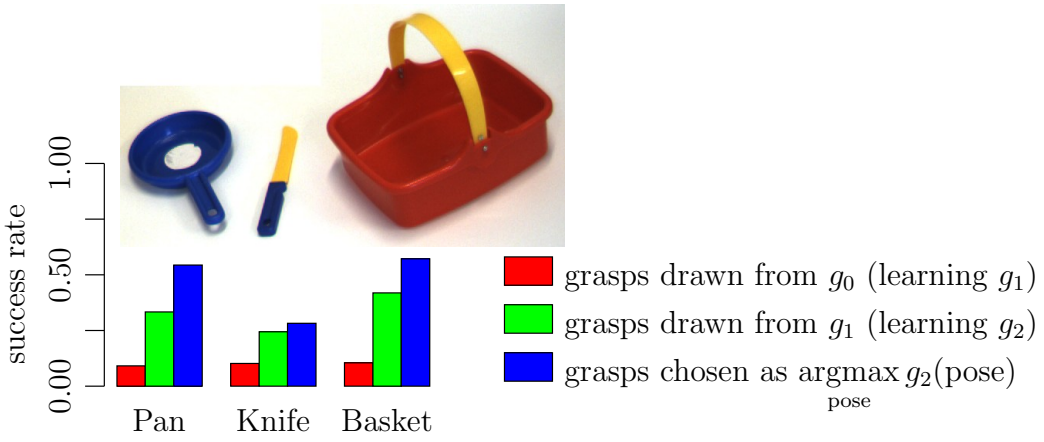

Figure 9: Evolving statistics M of `statusGrasp` = `stable` for the OACs $\mathbf{ObjGrasp}_{\text{Pan}}$, $\mathbf{ObjGrasp}_{\text{Knife}}$, and $\mathbf{ObjGrasp}_{\text{Basket}}$ over successive rounds of grasping trials [44]. In the first round, grasps are drawn from a hypothesis density g_0 generated from ECV features (Figure 8) for each object; the red bars show the empirical success rates, and the grasp density computed over the successful grasps is denoted g_1. In the second round (green bars), grasps are drawn from g_1, resulting in g_2. In the third round (blue), grasps are chosen as the maximum of g_2 for each object.

to one instance of each object of interest. The robot "plays" with the object by repeatedly grasping and dropping the object. This leads to a learning cycle similar to Algorithm 3, in which the system generates knowledge about the grasp affordances associated to the object.

*7.4. Example Ex-4: OACs for Planning (**PlanGrasp**, **PlanPush**)*

As a final example, we consider two high-level OACs suitable for planning: **PlanGrasp**, an OAC for grasping an object from a table, and **PlanPush**, an OAC for pushing an object into the reachable space so that it can be grasped [56]. Both of these OACs operate on discrete, attribute spaces defined in terms of a set of logical predicate and function symbols that denote properties and objects in the world. Such representations are standard in AI planning systems and we will structure our OACs in such a way that we can use prior planning work for building and executing plans.

*7.4.1. Definition of **PlanGrasp** and **PlanPush***

Table 2 shows a complete set of attributes that formalise a simple problem domain for picking up objects from a table and putting them onto a shelf. These attributes should be thought of as logical symbols (e.g., `clear`)

Attribute	Description
`clear(X)`	A predicate indicating that no object is stacked on `X`.
`focusOfAttn(X)`	A predicate indicating that object `X` is the focus of attention.
`gripperEmpty`	A predicate indicating that the robot's gripper is empty.
`inGripper(X)`	A predicate indicating that object `X` is in the gripper.
`onShelf(X)`	A predicate indicating that object `X` is on the shelf.
`onTable(X)`	A predicate indicating that object `X` is on the table.
`pushable(X)`	A predicate indicating that object `X` is pushable by the robot.
`reachable(X)`	A predicate indicating that object `X` is reachable for grasping by the gripper.

Table 2: A set of logical attributes for a simple planning domain.

with arguments represented by variables (e.g., `X`). Each ground term (e.g., $\texttt{clear(Obj}_0\texttt{)}$) has an associated truth value that is interpreted relative to the current world state.

Defining $\mathcal{S}$: To define the attribute spaces for **PlanGrasp** and **PlanPush** we restrict the set of attributes shown in Table 2. We define the attribute space for **PlanGrasp** in terms of the set of logical attributes:

$$\mathcal{S} = \left\{ \begin{array}{l} \texttt{focusOfAttn(X)}, \texttt{inGripper(X)}, \texttt{reachable(X)}, \\ \texttt{clear(X)}, \texttt{gripperEmpty}, \texttt{onTable(X)} \end{array} \right\}.$$

We also define the attribute space for **PlanPush** in terms of the attributes:

$$\mathcal{S} = \left\{ \begin{array}{l} \texttt{focusOfAttn(X)}, \texttt{reachable(X)}, \texttt{pushable(X)}, \\ \texttt{clear(X)}, \texttt{gripperEmpty}, \texttt{onTable(X)} \end{array} \right\}.$$

We note the only significant difference between these two attribute spaces is the inclusion of `inGripper(X)` for **PlanGrasp**, but not for **PlanPush**.

We also note that the representations used here could be more expressive (e.g., the arguments could be restricted to only allow objects of a particular type, or a multivalued logic could be used). In the interest of clarity we have used a simple representation: identifying the attribute space of an OAC is a challenging task no matter the level of abstraction, and learning the set of logical attributes in this OAC is a difficult process. (We recognise this as an instance of the "translation" learning problem.) A complete treatment of how this attribute space could be learnt is outside the scope of this paper.

Defining T: Given these attribute spaces, we can define T for each OAC as a pairing of initial conditions with predicted state descriptions (see Table 3). To

Name	Initial Conditions	Prediction
PlanGrasp	`focusOfAttn(X)`	`inGripper(X)`
	`reachable(X)`	`not(gripperEmpty)`
	`clear(X)`	`not(onTable(X))`
	`gripperEmpty`	
	`onTable(X)`	
PlanPush	`focusOfAttn(X)`	`reachable(X)`
	`not(reachable(X))`	
	`pushable(X)`	
	`clear(X)`	
	`gripperEmpty`	
	`onTable(X)`	

Table 3: Prediction functions T for planning-level grasping and pushing OACs.

specify the prediction function, both the initial conditions and the predictions are assumed to be conjunctions of specific attributes, i.e., all of the initial conditions must be true in the world for the prediction function to be defined, and all of the predictions are expected to be true in any state that results from the execution of the OAC. In terms of **PlanGrasp**, this means that if an object is the focus of attention, on the table, clear, reachable, and the agent's gripper is empty, then after executing this OAC we predict the object will be in the gripper, not on the table, and the gripper will no longer be empty. Likewise, for **PlanPush**, if an object is the focus of attention, unreachable, pushable, clear, on the table, and the agent's gripper is empty, then after executing this OAC we predict the object will be reachable.

Note that, like the other OACs we have discussed, these prediction functions do not make predictions in all states. Their predictive ability is restricted to those states where their initial conditions are met. In any world where these conditions do not hold the prediction function is undefined.[10]

Defining M: Taking the simplest possible approach, we define M for each OAC as the long-term probability that the OAC's T function correctly predicts the resulting state, assuming the OAC's execution began from a state for which the OAC's prediction function was defined.

[10]We can also imagine OACs whose prediction functions are best defined by disjunctions of separate prediction rules. In such cases, if one of the rules' initial conditions is true, that rule is used to predict the outcome of the action. If no rule matches then the prediction function is undefined.

We note that in classical AI planning systems, the reliability measure for all OACs would be fixed as $M = 1$. Such planners assume a deterministic and totally observable world, thereby removing all uncertainty from their prediction functions. More recent work in AI planning has moved beyond these assumptions (see, e.g., [19, 57]). For instance, there are now a number of planning algorithms that use probabilistic statements about an action's long-term success to build plans with probabilistic bounds on the likelihood that they will achieve their goals. Our definition of M makes our OACs suitable for use by such planners.

Defining E: The execution specifications of these two OAC are straightforward but differ significantly from our previous examples. While each of the previous example OACs indicated a specific control program to execute, our planning OACs define their execution in terms of executing other OACs. For example, the execution specification of **PlanGrasp** is defined in terms of executing **ObjGrasp**:

$$E_{\textbf{PlanGrasp}} = \texttt{execute}(\textbf{ObjGrasp}).$$

This means that invoking `execute`(**PlanGrasp**) calls `execute`(**ObjGrasp**).

Similarly the execution specification for **PlanPush** is defined in terms of our previously defined pushing OAC, **AgnoPush**:

$$E_{\textbf{PlanPush}} = \texttt{execute}(\textbf{AgnoPush}).$$

In other words, `execute`(**PlanPush**) calls `execute`(**AgnoPush**). In Section 8 we will see examples of the execution of these planning-level OACs.

7.4.2. Learning the Prediction Functions of Planning-Level OACs

The problem of learning prediction functions of the form we use in our planning OACs has been the focus of much recent research (see, e.g., [8]). One way this can be done is to use a training set of example actions in the world, and corresponding observations of the world before and after each action. For each example, a reduced world state consisting of a subset of the propositional attributes that make up the entire world model is computed and considered by the learning model. The attribute state is provided as input to the learning model in the form of a vector where each bit corresponds to the value of a single attribute. The learning problem is then treated as a set of binary classification problems, with one classifier for each attribute,

and the model learns the changes to each attribute in the reduced state. The actual learning can be performed, e.g., by using a kernelised voted perceptron classifier [58, 59], which is computationally efficient and can handle noise and partial observability. We refer the reader to [8] for a detailed account of how T and M can be learnt for this kind of OAC.

8. Interacting OACs

In this section, we describe two examples of OACs interacting in a single architecture. In Section 8.1, we illustrate the grounding of objects and object-related grasp affordances. In Section 8.2, we describe how such grounded representations can be used to execute plans.

8.1. Grounding Grasping OACs

In this first example of OAC interaction, we demonstrate the grounding of objects and object-related grasping affordances based on two learning cycles involving the OACs **AgnoGrasp** and **ObjGrasp** (see Figure 10). This process is shown in OAC notation in Algorithm 5 (and was previously described in [35]). The first cycle (Figure 10, top) learns a visual object model of an unknown object by grasping the object using **AgnoGrasp**. Once physical control is achieved, the model can be learnt by integrating the information gained from different views. The second cycle (Figure 10, bottom) learns how to grasp the object. The newly acquired visual model is used to identify and locate the object in the scene. **ObjGrasp** is then used to grasp the object. Through repeated applications of this procedure the performance of **ObjGrasp** is improved.

In this process, object knowledge and grasp knowledge is built up and stored in an internal representation (i.e., the object and grasp memory). Certain characteristics of our OACs play an important role in this process:

- Although the purpose of the first learning cycle is not to learn the OAC **AgnoGrasp** (the aim is to attain physical control over an object), learning is nevertheless taking place by calls to the `updateCP` and `updateM` functions, as a process parallel to those processes steered by, e.g., intentions or automated behaviours. This demonstrates the principle of "ongoing learning" mentioned in Section 3.

- OACs can be combined to produce complex behaviour. The interaction of multiple OACs, as demonstrated in the two learning cycles, can

First learning cycle
while `statusGrasp` $\neq$ `stable` **do**
    open gripper
    `expr` = `execute`(**AgnoGrasp**);
    `updateCP`(`expr`);
    `updateM`(`expr`);
end
Accumulate object representation o_i
if *accumulation successful* **then**
    transfer o_i into object memory $\mathcal{M}^{\mathrm{O}}$
    initialise $\mathbf{ObjGrasp}_{o_i}$ in $\mathcal{M}^{\mathrm{OAC}}$
    Second learning cycle
    while *instance of object* o_i *in scene* **do**
        state.`targetObj` = o_i
        `expr` = `execute`($\mathbf{ObjGrasp}_{o_i}$);
        `updateCP`(`expr`);
        `updateM`(`expr`);
        open gripper
    end
end

Algorithm 5: Grounding of object shape knowledge and object-specific grasp knowledge by cooperative application of the OACs **AgnoGrasp** and **ObjGrasp**.

result in the grounding of symbolic entities usable for planning (see Section 8.2).

8.2. Performing Plans

We now demonstrate how higher-level OACs can be executed by calling lower-level OACs, in the context of performing a plan. To do this, we consider an agent that is given the high-level goal of achieving `inGripper(`o`)` in a world described by the high-level state:

$$\{\texttt{focusOfAttn}(o), \texttt{gripperEmpty}, \neg\texttt{reachable}(o), \\ \texttt{pushable}(o), \texttt{onTable}(o), \texttt{clear}(o)\}.$$

Since `reachable(`o`)` does not initially hold in the world, a high-level planner must build a plan that makes this property true. Using the OACs from Section 7.4, one possible plan is an action sequence that first pushes o into a

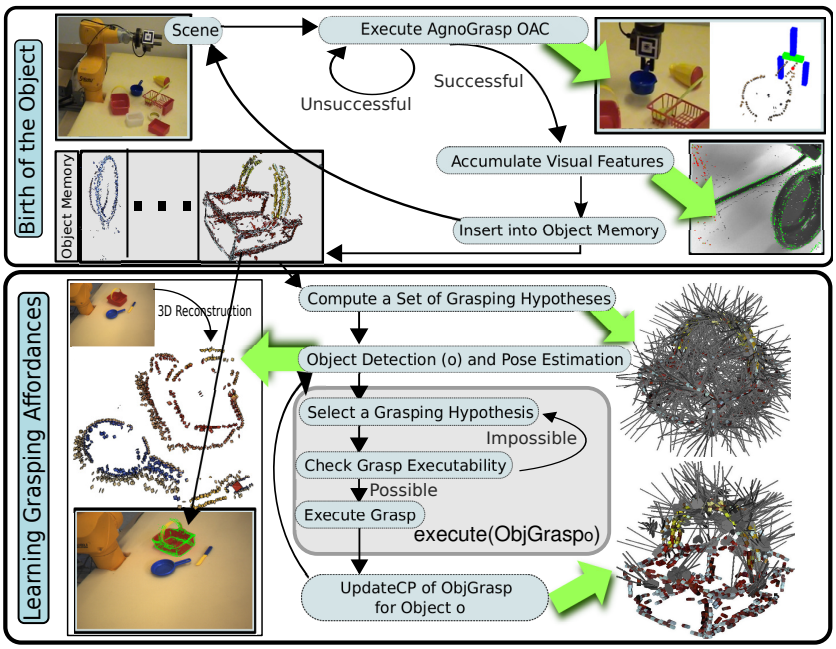

Figure 10: Grounding the OAC **ObjGrasp** in two learning cycles. In the first learning cycle, physical control over a potential object is obtained by the use of **AgnoGrasp**. Once control over the object is achieved and the visual structure changes according to the movement of the robot arm, a 3D object representation is extracted and stored in memory. In the second learning cycle, **ObjGrasp** is established and refined. First, the object representation extracted in the first learning cycle is used to determine the pose of the object in case it is present in the scene. Random samples of these are then tested individually. Successful grasps are turned into a probability density function that represents the grasp affordances associated to the object, in the form of success likelihoods of the grasp parameters.

graspable position, followed by an action that picks up o (see Figure 11).[11] This results in the following plan consisting of the two high-level OACs:

PlanPush, **PlanGrasp**.

Recall from Section 6.2 that successful planning using OACs relies on representational congruency and the hierarchical relationship between high-level OACs and lower-level OACs. Further, recall from Section 7.4 that the execution specifications of our high-level OACs are defined in terms of lower-

[11] We refer the reader to [56, 57] for more details on how such planning can be done.

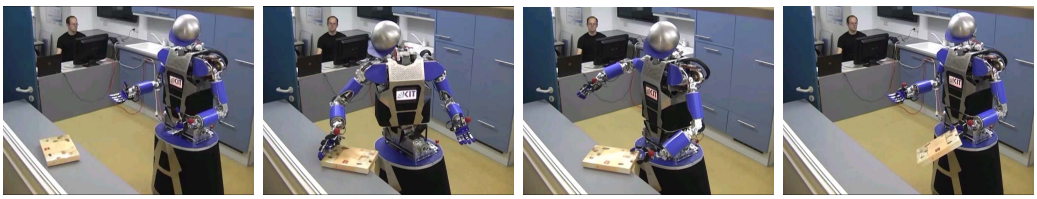

Figure 11: Execution of the plan involving the OACs **PlanPush**, **AgnoPush**, **PlanGrasp**, and **ObjGrasp**. From left to right: (1) the object is not graspable, (2) pushing moves the object into a graspable pose, (3) the object is grasped, and (4) the object can finally be picked up by the agent.

level OACs, so that the execution of a high-level OAC effectively calls a lower-level OAC as a subroutine, i.e.,

$$\begin{aligned} E_{\textbf{PlanPush}} &= \texttt{execute}(\textbf{AgnoPush}), \\ E_{\textbf{PlanGrasp}} &= \texttt{execute}(\textbf{ObjGrasp}). \end{aligned}$$

To understand the execution of the above plan, we must consider the ordering of the respective execution calls—and the experiments returned by those calls—in each of the component OACs in the plan. In this discussion, we assume that the world and the agent act as predicted and planned, without plan or execution failures. For reasons of space, we will also ignore all calls to the associated learning functions. However, even under such simplifying assumptions, the execution of the above plan requires a number of steps. We note that these steps should not be seen as a code fragment but rather as a trace of an executing system:

1. The execution of **PlanPush** is defined in terms of the execution of **AgnoPush**. By our definition of representational congruency, we are guaranteed that information can be translated from **PlanPush**'s high-level representation into **AgnoPush**'s model. For $\texttt{focusOfAttn}(o)$, a process must first be invoked to acquire $\text{bin}(o)$ and extract $\text{loc}(o)$ from the environment. Second, a process must identify τ and a for the desired push operation.
2. As we described in Section 7.1, executing **AgnoPush** invokes a low-level control program that performs the task of actually pushing o, by making use of the agent's end effector.
3. Executing **AgnoPush** returns the experiment:

$$(\{(\text{loc}(o), \text{bin}(o))\},\ \{T(\text{bin}(o), \text{loc}(o), a, \tau)\},\ \{\text{loc}(o)'\})$$

(which can, as all other experiments, either been stored in short term memory or used directly for learning). Representational congruency allows us to use $\text{loc}(o)'$ to determine the truth value of the high-level predicate $\texttt{reachable}(o)$ which is used in the experiment returned by **PlanPush**.[12]

4. Executing **PlanPush** therefore returns the experiment:

$$(\{\neg\texttt{reachable}(o), \texttt{pushable}(o), \texttt{clear}(o), \texttt{gripperEmpty}, \texttt{onTable}(o)\},\\ \{\texttt{reachable}(o)\},\\ \{\texttt{reachable}(o)\})$$

   indicating that $\texttt{reachable}(o)$ is now true in the actual world, and the agent can update its model with this information. This completes the execution of the first action in the plan.
5. The execution of **PlanGrasp** is defined in terms of the execution of $\textbf{ObjGrasp}_o$. As with **PlanPush**, information must be translated from the high-level representation into **AgnoPush**'s model. Since $\texttt{focusOfAttn}(o)$ is true in the world, the translation process, based on representational congruency, ensures that $\texttt{targetObj} = o$.
6. As we described in Section 7.3, executing $\textbf{ObjGrasp}_o$ invokes a low-level control program that performs the task of actually grasping o, by making use of the agent's end effector.
7. Executing $\textbf{ObjGrasp}_o$ returns the experiment:

$$(\{\texttt{statusGripper} = \texttt{empty}, \texttt{targetObj} = o\},\\ \{\texttt{statusGrasp} = \texttt{stable}\},\\ \{\texttt{statusGrasp} = \texttt{stable}\}).$$

   Again, representational congruency ensures $\texttt{statusGrasp} = \texttt{stable}$ can be translated into the attribute space of the higher-level OAC, to determine the truth value of the predicate $\texttt{inGripper}(o)$. This predicate can then be included in the experiment returned by **PlanGrasp**.

[12] We can imagine more complex execution specifications that would monitor the execution of the lower-level pushing OAC and call this OAC repeatedly until $\texttt{reachable}(o)$ is true. In this case, we have assumed that a single push is all that is necessary, and we leave the definition of such complex execution specifications as an area for future work.

8. Executing **PlanGrasp** returns the experiment:

$$(\{\texttt{reachable}(o), \texttt{clear}(o), \texttt{gripperEmpty}, \texttt{onTable}(o)\},$$
$$\{\texttt{inGripper}(o), \neg\texttt{gripperEmpty}, \neg\texttt{onTable}(o)\},$$
$$\{\texttt{inGripper}(o), \neg\texttt{gripperEmpty}, \neg\texttt{onTable}(o)\})$$

indicating that $\texttt{inGripper}(o)$ is now true in the world. As before, the agent can update its high-level model to reflect this fact. This ends the execution of the second action of the plan, and the plan as a whole.

As illustrated in the above example, the successful execution of a plan may typically require OACs to be invoked at multiple levels of abstraction, translating the calls between different models, and monitoring the results to confirm the success of the actions involved. We note that while our definitions support this simple example, more work is needed to extend our formal OAC definitions to more complex control structures for grounding, specifically in the areas of OAC execution, representation congruency, and OAC hierarchies. For instance, if planning is to be effective in real-world domains, *execution monitoring* is essential for detecting divergences of planned states from actual sensed states, and replanning accordingly (see, e.g., [60]). Such processes rely on the structural guarantees that properties like representational congruency provide. We leave the task of generalising such control structures as an area for future work.

9. Conclusion

This paper introduced Object-Action Complexes (OACs) as a framework for modelling actions and their effects in artificial cognitive systems. We provided a formal definition of OACs and a set of concrete examples, showing how OACs operate and interact with other OACs, and also how certain aspects of an OAC can be learnt.

The importance of OACs lies in their ability to combine the properties of multiple action formalisms, from a diverse range of research fields, to provide a dynamic, learnable, refinable, and grounded representation that binds objects, actions, and attributes in a causal model. OACs have the ability to represent and reason about low-level (sensorimotor) processes as well as high-level (symbolic) information and can therefore be used to join the perception-action space of an agent with its planning-reasoning space. In addition, OACs can be combined to produce more complex behaviours,

and sequenced as part of a plan generation process. As a consequence, the OAC concept can be used to bridge the gap between low-level sensorimotor representations, required for robot perception and control, and high-level representations supporting abstract reasoning and planning.

10. Acknowledgements

The research leading to these results received funding from the European Union through the Sixth Framework PACO-PLUS project (IST-FP6-IP-027657) and the Seventh Framework XPERIENCE project (FP7/2007-2013, Grant No. 270273). We thank Frank Guerin for fruitful discussions and his input to Piaget's understanding of sensory-motor schemas.

References

[1] R. A. Brooks, A robust layered control system for a mobile robot, IEEE Journal of Robotics and Automation 2 (1986) 14–23.

[2] R. A. Brooks, C. Breazeal, M. Marjanovic, B. Scassellati, M. M. Williamson, The Cog project: Building a humanoid robot, Lecture Notes in Computer Science 1562 (1999) 52–87.

[3] V. Braitenberg, Vehicles: Experiments in Synthetic Psychology, The MIT Press, 1986.

[4] M. Huber, A hybrid architecture for adaptive robot control, Ph.D. thesis, University of Massachusetts Amherst (2000).

[5] A. Stoytchev, Some basic principles of developmental robotics, IEEE Transactions on Autonomous Mental Development 1 (2) (2009) 1–9.

[6] J. Modayil, B. Kuipers, Bootstrap learning for object discovery, in: Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004, pp. 742–747.

[7] R. E. Fikes, N. J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3-4) (1971) 189–208.

[8] K. Mourão, R. Petrick, M. Steedman, Using kernel perceptrons to learn action effects for planning, in: Proceedings of the International Conference on Cognitive Systems, 2008, pp. 45–50.

[9] E. Amir, A. Chang, Learning partially observable deterministic action models, Journal of Artificial Intelligence Research 33 (2008) 349–402.

[10] R. Sutton, Verification, the key to AI, [Online]. Available from: http://www.cs.ualberta.ca/~sutton/IncIdeas/KeytoAI.htm (2001).

[11] S. Harnad, The symbol grounding problem, Physica D (42) (1990) 335–346.

[12] J. Piaget, The Origins of Intelligence in Children, London: Routledge & Kegan Paul, 1936, (French version published in 1936, translation by Margaret Cook published 1952).

[13] F. J. Corbacho, M. A. Arbib, Schema-based learning: Towards a theory of organization for biologically-inspired autonomous agents, in: Proceedings of the First International Conference on Autonomous Agents, 1997, pp. 520–521.

[14] A. Newell, H. Simon, GPS, a program that simulates human thought, in: E. A. Feigenbaum, J. Feldman (Eds.), Computers and Thought, McGraw-Hill, NY, 1963, pp. 279–293.

[15] C. Green, Application of theorem proving to problem solving, in: Proceedings of the First International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1969, pp. 741–747.

[16] E. D. Sacerdoti, The nonlinear nature of plans, in: Proceedings of the Fourth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1975, pp. 206–214.

[17] A. Samuel, Some studies in machine learning using the game of checkers, IBM Journal of Research and Development 3 (3) (1959) 210–229.

[18] N. J. Nilsson, Learning Machines, McGraw-Hill, 1965.

[19] L. P. Kaelbling, Learning functions in k-DNF from reinforcement, in: Proceedings of the Seventh International Workshop on Machine Learning, Morgan Kaufmann, 1990, pp. 162–169.

[20] T. Mitchel, Machine Learning, WCB McGraw Hill, 1997.

[21] H. Pasula, L. Zettlemoyer, L. P. Kaelbling, Learning symbolic models of stochastic domains, Journal of Artificial Intelligence 29 (2007) 309–352.

[22] F. Wörgötter, A. Agostini, N. Krüger, N. Shyloa, B. Porr, Cognitive agents — a procedural perspective relying on the predictability of object-action-complexes (OACs), Robotics and Autonomous Systems 57 (4) (2009) 420–432.

[23] D. Vernon, G. G. Metta, G. Sandini, A survey of artificial cognitive systems: implications for the autonomous development of mental capabilities in computational agents, IEEE Transactions on Evolutionary Computation 11 (2007) 151–180.

[24] L. P. Kaelbling, M. L. Littman, A. R. Cassandra, Planning and acting in partially observable stochastic domains, Artif. Intell. 101 (1998) 99–134. doi:10.1016/S0004-3702(98)00023-X.
URL http://portal.acm.org/citation.cfm?id=1643275.1643301

[25] J. Pearl, Probabilistic Reaasoning in Intelligent Systems: Networks of Plausible Inference, Morgan Kaufmann, 1988.

[26] A. P. Dempster, N. M. Laird, D. B. Rubin, Maximum likehood from incomplete data via the EM algorithm, Journal of the Royal Statistical Society (Series B) 39 (1) (1977) 1–38.

[27] D. Spiegelhalter, P. Dawid, S. Lauritzen, R. Cowell, Bayesian analysis in expert systems, Statistical Science 8 (1993) 219–283.

[28] S. Kok, P. Domingos, Learning the structure of Markov logic networks, in: Proceedings of the Twenty-Second International Conference on Machine Learning, ACM Press, 2005, pp. 441–448.

[29] J. J. Gibson, The Perception of the Visual World, Houghton Mifflin, Boston, 1950.

[30] E. Sahin, M. Çakmak, M. R. Doğar, E. Uğur, G. Ücoluk, To afford or not to afford: A new formalization of affordances toward affordance-based robot control, Adaptive Behavior 15 (4) (2007) 447–472.

[31] R. Brooks, Intelligence without reason, in: Proceedings of the International Joint Conference on Artificial Intelligence, 1991, pp. 569–595.

[32] R. Brooks, Elephants don't play chess, Robotics and Autonomous Systems 6 (1&2) (1990) 3–15.

[33] R. Pfeifer, M. Lungarella, F. Iida, Self-organization, embodiment, and biologically inspired robotics, Science 318 (2007) 1088–1093.

[34] D. Kraft, N. Pugeault, E. Başeski, M. Popović, D. Kragic, S. Kalkan, F. Wörgötter, N. Krüger, Birth of the Object: Detection of Objectness and Extraction of Object Shape through Object Action Complexes, Special Issue on "Cognitive Humanoid Robots" of the International Journal of Humanoid Robotics 5 (2009) 247–265.

[35] D. Kraft, R. Detry, N. Pugeault, E. Başeski, , F. Guerin, J. Piater, N. Krüger, Development of object and grasping knowledge by robot exploration, IEEE Transactions on Autonomous Mental Development 2 (4) (2010) 368–383.

[36] A. Stoytchev, Behavior-Grounded Representation of Tool Affordances, in: IEEE International Conference on Robotics and Automation, 2005, pp. 3060–3065.

[37] R. Jacobs, M. Jordan, A. Barto, Task decomposition through competition in a modular connectionist architecture: The what and where vision tasks, Cognitive Science 15 (2) (1991) 219–250.

[38] C. M. Vigorito, A. G. Barto, Intrinsically motivated hierarchical skill learning in structured environments, IEEE Transactions on Autonomous Mental Development (TAMD) 2 (2) (2010) 132–143.

[39] K. Narendra, J. Balakrishnan, Adaptive control using multiple models, IEEE Transaction on Automatic Control 42 (2) (1997) 171–187.

[40] M. Haruno, D. Wolpert, M. Kawato, MOSAIC model for sensorimotor learning and control, Neural Computation 13 (2001) 2201–2220.

[41] C. Miall, Modular motor learning, Trends in Cognitive Sciences 6 (1) (2002) 1–3.

[42] J. Gibson, The Ecological Approach to Visual Perception, Boston, MA: Houghton Mifflin, 1979.

[43] R. Detry, E. Başeski, N. Krüger, M. Popović, Y. Touati, O. Kroemer, J. Peters, J. Piater, Learning object-specific grasp affordance densities, in: Proceedings of the IEEE Eighth International Conference on Development and Learning, 2009, pp. 1–7.

[44] R. Detry, D. Kraft, A. G. Buch, N. Krüger, J. Piater, Refining grasp affordance models by experience, in: Proceedings of the IEEE International Conference on Robotics and Automation, 2010, pp. 2287–2293.

[45] T. A. Henzinger, The theory of hybrid automata, in: M. Inan, R. Kurshan (Eds.), Verification of Digital and Hybrid Systems, Vol. 170 of NATO ASI Series F: Computer and Systems Sciences, Springer, 2000, pp. 265–292.

[46] A. D. Baddeley, Essentials of Human Memory, Psychology Press, Taylor and Francis, 1999.

[47] D. Willshaw, P. Buneman, C. Longuet-Higgins, Non-Holographic Associative Memory, Nature 222 (1969) 960–962.

[48] D. Willshaw, Holography, Association and Induction, in: G. Hinton, J. Anderson (Eds.), Parallel Models of Associative Memory, Erlbaum, Hillsdale, NJ, 1981, pp. 83–104.

[49] F. T. Sommer, G. Palm, Bidirectional Retrieval from Associative Memory, Advances in Neural Information Processing Systems 10 (1998) 675–681.

[50] T. Plate, Holographic Reduced Representations: Convolution Algebra for Compositional Distributed Representations, in: Proceedings of the Twelfth International Joint Conference on Artificial Intelligence, Morgan Kaufmann, San Francisco, CA, 1991, pp. 30–35.

[51] D. Omrčen, A. Ude, A. Kos, Learning primitive actions through object exploration, in: Proceedings of the International Conference on Humanoid Robots, Daejeon, Korea, 2008, pp. 306–311.

[52] M. Popović, D. Kraft, L. Bodenhagen, E. Başeski, N. Pugeault, D. Kragic, T. Asfour, N. Krüger, A strategy for grasping unknown objects based on co-planarity and colour information, Robotics and Autonomous Systems 58 (5) (2010) 551–565.

[53] L. Bodenhagen, D. Kraft, M. Popović, E. Başeski, P. E. Hotz, N. Krüger, Learning to grasp unknown objects based on 3D edge information, in: Proceedings of the IEEE International Conference on Computational Intelligence in Robotics and Automation, 2009, pp. 421–428.

[54] N. Pugeault, F. Wörgötter, N. Krüger, Visual primitives: Local, condensed, semantically rich visual descriptors and their applications in robotics, Special Issue on "Cognitive Humanoid Vision" of the International Journal of Humanoid Robotics 7 (3) (2011) 379–405.

[55] R. Detry, N. Pugeault, J. Piater, A probabilistic framework for 3D visual object representation, IEEE Transactions on Pattern Analysis and Machine Intelligence 31 (10) (2009) 1790–1803.

[56] R. Petrick, D. Kraft, K. Mourão, C. Geib, N. Pugeault, N. Krüger, M. Steedman, Representation and integration: Combining robot control, high-level planning, and action learning, in: Proceedings of the Sixth International Cognitive Robotics Workshop, Patras, Greece, 2008, pp. 32–41.

[57] R. Petrick, F. Bacchus, A knowledge-based approach to planning with incomplete information and sensing, in: Proceedings of the International Conference on Artificial Intelligence Planning and Scheduling, 2002, pp. 212–221.

[58] Y. Freund, R. Schapire, Large margin classification using the perceptron algorithm, Machine Learning 37 (1999) 277–296.

[59] R. Khardon, G. M. Wachman, Noise tolerant variants of the perceptron algorithm, Journal of Machine Learning Research 8 (2007) 227–248.

[60] R. Petrick, D. Kraft, N. Krüger, M. Steedman, Combining cognitive vision, knowledge-level planning with sensing, and execution monitoring for effective robot control, in: Proceedings of the Fourth Workshop on Planning and Plan Execution for Real-World Systems, Thessaloniki, Greece, 2009, pp. 58–65.

Norbert Krüger is a Professor at the Mærsk McKinney Møller Institute, University of Southern Denmark. He holds an M.Sc. from the Ruhr-Universität Bochum, Germany and his Ph.D. from the University of Bielefeld. Norbert Krüger leads the Cognitive Vision Lab which focuses on computer vision and cognitive systems, in particular the learning of object representations in the context of grasping. He has also been working in the areas of computational neuroscience and machine learning.

Christopher Geib is a Research Fellow at the University of Edinburgh School of Informatics. He holds an M.S. and Ph.D. from the University of Pennsylvania. His research focuses broadly on decision making and reasoning about actions under conditions of uncertainty, including planning, scheduling, constraint-based reasoning, human-computer interaction, human-robot interaction, and probabilistic reasoning. His recent research has focused on probabilistic intent recognition through weighted model counting and planning based on grammatical formalisms.

Justus Piater is a professor of computer science at the University of Innsbruck, Austria. He earned his Ph.D. degree at the University of Massachusetts Amherst, USA, where he held a Fulbright graduate student fellowship. After a European Marie-Curie Individual Fellowship at INRIA Rhône-Alpes, France, he was a professor at the University of Liège, Belgium, and a Visiting Research Scientist at the Max Planck Institute for Biological Cybernetics in Tübingen, Germany. His research in computer vision and machine learning is motivated by intelligent and interactive systems, where he focuses on visual learning, closed-loop interaction of sensorimotor systems, and video analysis.

Ronald Petrick is a Research Fellow in the School of Informatics at the University of Edinburgh. He received an M.Math. degree in computer science from the University of Waterloo and a Ph.D. in computer science from the University of Toronto. His research interests include planning with incomplete information and sensing, cognitive robotics, knowledge representation and reasoning, generalised planning, and natural language dialogue. He is currently the Scientific Coordinator of the EU JAMES project.

Mark Steedman is a Professor of Cognitive Science in Informatics at the University of Edinburgh, working in computational linguistics, artificial intelligence, the communicative use of prosody, tense and aspect, and wide-coverage parsing using Combinatory Categorial Grammar (CCG). Prof. Steedman is a Fellow of the Association for the Advancement of Artificial Intelligence (AAAI), the Royal Society of Edinburgh (FRSE), and the British Academy (FBA). He is a member of the Academy of Europe and a former President of the Association for Computational Linguistics (ACL).

Florentin Wörgötter has studied Biology and Mathematics in Düsseldorf. He received his Ph.D. in 1988 in Essen working experimentally on the visual cortex before he turned to computational issues at the Caltech, USA (1988-1990). After 1990 he was researcher at the University of Bochum concerned with experimental and computational neuroscience of the visual system. Between 2000 and 2005 he had been Professor for Computational Neuroscience at the Psychology Department of the University of Stirling, Scotland where his interests strongly turned towards "Learning in Neurons". Since July 2005 he leads the Department for Computational Neuroscience of the Bernstein Center at the University of Göttingen. His main research interest is information processing in closed-loop perception-action systems, which includes aspects of sensory processing, motor control and learning/plasticity. These approaches are tested in walking as well as driving robotic implementations. His group has developed the RunBot a fast and adaptive biped walking robot.

Aleš Ude studied applied mathematics at the University of Ljubljana, Slovenia, and received his doctoral degree from the Faculty of Informatics, University of Karlsruhe, Germany. He was awarded the STA fellowship for postdoctoral studies in ERATO Kawato Dynamic Brain Project, Japan. He has been a visiting researcher at ATR Computational Neuroscience Laboratories, Kyoto, Japan, for a number of years and is still associated with this group. Currently he is a senior researcher at the Department of Automatics, Biocybernetics, and Robotics, Jožef Stefan Institute, Ljubljana, Slovenia. His research focuses on imitation and action learning, perception of human activity, humanoid robot vision, and humanoid cognition.

Tamim Asfour received his diploma degree in electrical engineering and his Ph.D. degree in computer science from the University of Karlsruhe, Germany in 1994 and 2003, respectively. He is leader of the humanoid robotics research group at the institute for Anthropomatics at the Karlsruhe Institute of Technology (KIT). His research interests include humanoid robotics, grasping and manipulation, imitation learning, system integration and mechatronics.

Dirk Kraft obtained a diploma degree in computer science from the University of Karlsruhe (TH), Germany in 2006 and a Ph.D. degree from the University of Southern Denmark in 2009. He is currently employed as an assistant professor at the Mærsk McKinney Møller Institute, University of Southern Denmark. His research interests lie within cognitive systems, robotics and computer vision.

Damir Omrčen received his PhD in robotics from the University of Ljubljana, Slovenia, in 2005. He is employed as a research assistant at the Department of Automation, Biocybernetics and Robotics at the "Jozef Stefan" Institute in Ljubljana. His fields of interest include vision and robot control where he combines classical model-based approaches and more advanced approaches based on exploration and learning.

Alejandro Agostini received the B.S. degrees in Bioengineering with honours from the National University of Entre Ríos, Argentina, and in Electronic Engineering from the National University of Catalonia (UPC), Spain. He is currently a senior Ph.D. student in Artificial Intelligence at the UPC, and is working at the Institut de Robòtica Industrial (IRI) under a work contract drawn up by the Spanish Research Council (CSIC). His research interests include machine learning, robotics, decision making, and cognitive systems. He performed several research stays at the Bernstein Centre for Computational Neuroscience, Göttingen, Germany, and at the University of Karlsruhe, Germany.

Rüdiger Dillmann is Professor at the Computer Science Faculty, Karlsruhe Institute of Technology (KIT), Germany. He is director of the Research Center for Information Science (FZI), Karlsruhe. He is scientific leader of German collaborative research centre Humanoid Robots (SFB 588). His research interests include humanoid robotics, technical cognitive systems, machine learning, computer vision, robot programming by demonstration, and medical applications of informatics.