Architecture and Representations

Nick Hawes[1], Jeremy L Wyatt[1], Aaron Sloman[1], Mohan Sridharan[1], Richard Dearden[1], Henrik Jacobsson[2], and Geert-Jan Kruijff[2]

[1] Intelligent Robotics Lab, School of Computer Science, University of Birmingham, Birmingham, UK, {nah,jlw,axs,rwd}@cs.bham.ac.uk
[2] DFKI GmbH, Saarbrücken, Germany, {henrikj,gj}@dfki.de

1 Introduction

The study of architectures to support intelligent behaviour is certainly the broadest, and arguably one of the most ill-defined enterprises in AI and Cognitive Science. The basic scientific question we seek to answer is: "What are the trade-offs between the different ways that intelligent systems might structured?" These trade-offs depend in large part on what kinds of tasks and environment a system operates under (*niche space*), and also what aspects of the *design space* we deem to be architectural. In CoSy we have tried to answer that question in several ways. First by thinking about the requirements on architectures that arise from our particular scenarios (parts of niche space). Second by building systems that follow well-defined architectural rules, and using these systems to carry out experiments on variations of those rules. Third by using the insights from system building to improve our understanding of the trade-offs between different architectural choices, i.e. between different partial designs. Our objective in CoSy has not been to come up with just another robot architecture, but instead to try to make some small steps forward in a new science of architectures.

When the project started we had several specific scientific goals for our work. First, we wanted *to identify a space of possible architectures that includes most of the good ones for our scenarios.* Our methodology is to define this space of possible architectures using an *architectural schema*, which is a simply a set of constraints and rules on what architectures we allow. We refer to specific points within it as *architectural instantiations.* Second *we aimed to develop a toolkit and an experimental methodology to explore this space, identifying the trade-offs as we move through it.* Finally, our third goal was to use our toolkit *to implement quickly and easily robotic systems that adhere to certain architectural principles.* Those systems are described in Chapters 9 and 10. In this chapter we will describe our schema; some specialisations of it; experiments showing the trade-offs between some of those specialisations; and the toolkit we devised. In addition we will describe how our work sits in between the two major camps of work on architectures for intelligence. The specific contributions are:

- a set of requirements for embodied cognition as a way of generating an architectural schema.
- a new architectural schema that draws on important work from both cognitive architecture and robotic architecture traditions.
- a toolkit for implementing architectures within this schema.
- a series of robotic implementations that utilise the schema and provide a proof of concept (covered in Chapters 9 and 10).
- a set of well defined problems in architectures, posed if the schema is accepted: binding; filtering; management; action fusion.
- experimental profiling of the effects of steps through schema space, specifically with respect to the filtering problem.

The contributions are sequenced in the chapter as follows. First we make some introductory remarks about the science of architectures, our methodology, and define our terminology. Second we describe run-time and design-time requirements that arise from the needs of the CoSy scenarios, and of which our architectural theory must take account. After this we describe our architectural schema (CAS) and show how it meets these requirements. Fourth, we describe how the schema poses at least four further problems that we refer to as the *binding problem*, the *filtering problem*, the *processing management problem*, and the *action fusion problem*. Solutions to these problems provide a second level of constraints and thus a more specific architectural schema. Fifth, we describe the relationship between CAS and other schemas. We follow these contributions with a brief discussion of future research directions.

2 Architectures and the science of cognitive systems

We start by clarifying the role of work on "architectures". AI is a science that attempts to understand intelligent systems partly through the process of synthesising them, and partly by analysis of the systems that result. In most areas of AI while the problems are technically difficult, the methodology is clear. When considering integrated systems, however, the methodology itself is a stumbling block. How can we objectively compare the architectures for two systems that might perform different tasks, in different environments, using different processing content? The short answer is that we can't. Yet most examples of work on architectures in robotics are precisely systems that combine architectural choices with task specific processing content, and that are evaluated in different environments from one another. From the point of view of a science of architectures for embodied intelligence this is no good. For a science of architectures we need to separate the processing content from the architectural bones on which it is hung. In other words we want to assess how the changes in the architecture alter the run time properties of that system, independent of changes in the information processing content.

This is tricky, since in both natural and artificial systems the shape of the architecture is intimately related to the shape of the pieces it pulls together. In nature, moreover, the information processing architecture is intimately related to the hardware implementation and the information processing functions implemented in that hardware. On the engineering side, complete cognitive systems are still so difficult to construct that the architecture is essentially almost always bespoke. Separation of architecture and content in a theory is made harder by the fact that the processing in an intelligent system can be described at a variety of levels of abstraction. We may speak of the neural or machine architecture, of information processing architecture, or of the cognitive architecture. In a complete theory of architectures for intelligence we need a clear account of how these different levels of description are related. At the current time we do not even have a clearly defined terminology to distinguish the different uses of the word.

There is an important distinction in the literature between architectures that are entirely specified in advance and those that are partially specified. In CoSy, following [26, 21], we use the notion of an *architectural schema* to refer to architectures of the latter type. In simple terms we can think of an *architectural schema* as a set of constraints that defines a space of possible architectures and ways of moving through that space. We can add constraints to a schema to create sub-schemas, i.e. sub-spaces. If we add enough constraints we can create a fully specified architecture, and these we refer to as *architectural instantiations*. Hereon, we frequently use the terms *schema* and *instantiation* as shorthand for these terms.

If we have a software toolkit that implements a particular schema, it can be used to implement specific working systems that follow the schema. These implemented systems require us to make both a complete set of architectural choices, and to have processing content. As described above the two are very hard to separate. Our approach must therefore be based on a judicious division of our design choices into architectural ones (i.e. ones we can easily vary in the schema), and those concerned with content (i.e. which vary outside the schema).

In CoSy the specific schema we have designed is called CAS (Cognitive systems Architecture Schema). Our toolkit implementing this schema is called CAST. It allows researchers to implement systems that fall within the space defined by CAS. CAST is particularly powerful because it allows us to implement a system, and then change some of the architectural choices independent of the processing content of that specific system. Thus CAST allows us to take steps through the architectural space, and to isolate the effects on system behaviour due to architectural changes. Thus CAST provides an important element necessary for a science of architectures as described above.

Overall our architectural work in CoSy is neither concerned with trying to model humans or any other specific type of animal, nor with trying to compete on performance on a specific set of tasks. Rather it is concerned with trying to understand the possibilities and trade-offs involved in different designs in relation to different sets of requirements. In our approach we have several stages through which we must go:

- We identify sets of scenarios (tasks and environments) that we want our systems to solve. We then identify the requirements on architectures arising from these scenarios.
- These requirements lead us to hypothesise an initial architectural schema, together with a toolkit implementing the schema.
- Using the scenarios and the schema toolkit we produce one or more systems based on one or more instantiations of the schema. In these designs there is, as far as possible, a clear separation between the architectural choices and the processing content.
- We analyse our architectural choices, both empirically, and by introspecting on the flaws in our designs. We use the toolkit to make architectural changes to these designs, while holding the processing content fixed.
- Using the insights gained we can refine our schema.

In the rest of this document we will step through this process, stopping along the way to reflect on the choices we made, and to compare our schema with previous work. We now turn to the two scenarios driving CoSy, and to the requirements arising from them.

3 Requirements for architectures for cognitive robots

As stated previously our starting point is to analyse scenarios in order to define requirements for architectures. This is an example of backward chaining research in which we consider goals beyond our immediate capabilities, and work backwards to define sub-problems that are more easily tackled — while also understanding a little of the way their solutions might be composed. In the CoSy project we picked scenarios based on robot systems that are able to interact with humans, either while mobile in an office or home setting (the Explorer); or while manipulating objects in a space shared with a human (the . PlayMate). In the Explorer scenario typical tasks might be human augmented mapping of an office environment, using natural language utterances to name and describe places, and with mixed initiative dialogue. For example in this scenario the robot can ask the human what type of room it is in, if it is unsure, and must understand questions, instructions to learn, and instructions to act. In the PlayMate scenario the tasks include the robot answering questions about the identity and properties of objects on the table and also about the spatial relationships between them. The robot may be given instructions to manipulate the objects to alter their spatial relationships, and to learn not just word labels for objects, but also the meanings of the property words used to describe an object (e.g. what visual features correspond to the word red).

What sorts of requirements do these scenarios and other typical robot scenarios place upon the robot designs, and in particular upon the manner in which the pieces of the intelligent system are put together? There are a number of well-known properties of the robot-environment interaction that any robot system operating in human environments typically has to deal with. Our run-time requirements therefore include the ability to deal with the following properties of this interaction:

- *Dynamism:* The world changes frequently, rapidly and independently of the robot.
- *Uncertainty:* Sensors are inaccurate, and the actions of the robot often fail to have the planned for consequences.
- *Multiple modalities:* Many robots — but especially the robots in our two scenarios — must use information from multiple sensory modalities in order to make decisions. In our case these include simple haptics, vision, proprioception and speech. All but the most trivial robot also has multiple modes of action (manipulation, looking, facial expression, utterances, locomotion) enabled by multiple motor systems and the actions of these must be coordinated.
- *Re-taskability:* In our scenarios our robots must be re-taskable, either by others, or autonomously. Behaviour should be goal directed, but not to the extent that it cannot be re-tailored to the context, perhaps on the fly, switch to a new task, or interleave new tasks with old ones.

These are quite general *run-time requirements* on the interaction between a robot and its environment, i.e. requirements on the system during performance. Out of these arise run-time and *design-time requirements* on the architectural schema itself. Architectures for artificial cognition do not just structure the way components work together during a system run, but structure the engineering efforts of the designers. A good software implementation of an architecture or architecture schema should therefore assist the design process. This means that the architecture should make it easier to design a cognitive system, and easier to evaluate a system, and understand the causes of its particular behaviour. A proper engineering science of cognition will also require the architecture to have a number of other properties:

- *Understandability:* Cognitive systems of the order of complexity we are describing must be understandable. This means that at least some of the tokens within the system need to be semantically transparent to the designers at some level of abstraction. If we are to understand why our robots succeed or fail in a task, and how they can be re-engineered, then we must have the ability to look at the tokens within the system, and to allocate them meaning as designers. An architecture schema for cognitive systems therefore needs to provide a variety of clear ways by which tokens can be related to one another.

- *Incrementality in design:* Large, bug-free, complex software systems need to be constructed and tested incrementally. This requires that a schema allow new sub-systems to be added without completely redesigning the existing sub-systems.
- *Multiple specialist representations:* the field of AI has fragmented, and the sub-disciplines have developed their own specialist representations for inference and decision- making. In designing robots with multiple forms of sensing and acting we need to bridge the gaps between them.
- *Parallel processing:* many of the algorithms employed in vision, language processing, planning, and learning are computationally demanding. A serial model of processing is thus unworkable. There is a need for perceptual components to run in parallel, so that the system may respond rapidly to change. Action components must also run in parallel so that the robot can do more than thing at a time, e.g. looking and reaching.
- *Asynchronous updating:* information arrives in different modalities at different rates. In addition processing in some modalities is slower than in others. This requires us to accept that updating of information will occur asynchronously across the system.

Given these constraints the management of information flow in the robot system becomes key. How should information from one sub-system be communicated to others? How should decisions to act be combined and sequenced? How should we determine whether separate pieces of information are related? These are questions to which we should provide architectural answers. Following from our requirements together with the specification of the Explorer and PlayMate scenarios are a number of useful properties of robot control systems that would satisfy those requirements (though they may not be the only way of satisfying them), and of an architectural schema should take account. In order to satisfy our requirements in our scenarios we will assume the following principles:

1. Our robots will have representations of entities in the world at a variety of levels of abstraction from the sensory information.
2. Some of these representations will have roots in multiple sensory modalities.
3. There will be many concurrently running sub-systems in our robots refining and using these representations.
4. Both the PlayMate and the Explorer must represent and reason about hypothetical future states, in order to be able to plan, answer questions, etc. They will therefore require representations that support this kind of reasoning.
5. A large part of what our architectural solution will be concerned with is the refinement, sharing and transmission of these representations by and to these different sub-systems.

6. The system will not be able to draw all possible inferences from the available sensory information, and thus will have to make judicious choices about which processing to perform.
7. There will multiple modes of action which have to be coordinated.

In the next section we describe the architectural schema CAS. CAS encompasses systems that satisfy our assumptions above, and also satisfies the requirements that precede them. In the following sections we also explain why.

4 A new architectural schema

The requirements and assumptions described in the previous section give rise to a space of possible architecture schema designs. In order to produce a single schema to constrain our research work we must design a schema that satisfies all of these requirements, whilst still being general enough to capture a selection of the space of possible designs. It is also important that any design reflects both our previous experiences as system designers (i.e. we have knowledge about what works and what doesn't work), and the experience of the wider research community (i.e. what concrete designs have proven successful in the past). Given all of these (interacting) constraints, it is not possible to claim that the following design is the best possible schema design for our scenarios. Instead we put it forward as an initial attempt at producing a schema to satisfy our requirements given our experience.

4.1 Key Features of CAS

To quickly convey the features of CAS we summarise them below. More detail is given in the following sections.

- Distributed shared memory: The schema contains sub-architectures each of which has a blackboard (working memory). These sub-architectures are loosely coupled to reduce complex inter-dependencies. Systems could contain sub-architectures for motor control, vision, action, planning, language etc. The structure is recursive: sub-architectures can contain other sub-architectures.
- Parallel refinement of shared representations: Each sub-architecture contains a number of processing components which run in parallel and that asynchronously read and update shared information via the sub-architecture specific working memory.
- Limited privileges: Each of these sub-architecture working memories is only writable by processes within its sub-architecture, and by a small number of privileged global processes (e.g. a global goal manager).
- Control of information and processing: Information flow is controlled by goals generated within the architecture at run time, allowing it to deal with

new problems and opportunities. This allows the schema to support different approaches to processing (e.g. incremental processing, forward chaining, backward chaining etc.). The schema distinguishes between two classes of goal: global goals (that require coordination across sub-architectures), and local goals (that are dealt with inside a single sub-architecture).

- Knowledge management by ontologies: The knowledge that can be used within a sub-architecture is defined by a set of ontologies for that sub-architecture. Relationships between the ontologies in different sub-architectures are defined by a set of general ontologies. These ontologies can also be used to define knowledge at an architecture-general level.

4.2 Sub-architecture design

Components

Our schema starts on the level of a collection of processing components. Every component is concurrently active, allowing them to process in parallel. This satisfies our requirement of supporting concurrent processing. We do not specify any constraints on the contents of components: they could have behave like a node in a connectionist network, an activity in a behaviour based system [2], or a unit of processing in a decomposition by information processing function. Components can take input either directly from sensors, or from the working memory. They can also directly control actuators in the manner of closed loop controllers, or initiate fixed action patterns. Components can have processing triggered by the appearance of certain information on the shared working memory, and can modify structures on that memory. Components may also have their own private memory. Finally components are of two types: managed and unmanaged. *Unmanaged components* are low-latency processes that run all the time. They are useful for several types of processing. They can be used for low-latency early processing of information coming directly from sensors at a high rate. In a visual system, for example, they could correspond to the earliest stages of pre-attentive processing, pulling high bandwidth data from cameras at frame rate and pumping the processed frames onto the working memory. Alternatively they could implement reflexes, providing rapid reaction to sensory information; or they could implement monitors on signals that raise alarms or actions elsewhere in the system, or modify global parameters. *Managed components* by contrast are typically computationally expensive processes, and the schema assumes that there are not the computational resources available to run them all. They therefore post requests to run to the *task manager* associated with the sub-architecture.

Shared Workspaces

Rather than exchange information directly, processing components are connected to a *shared working memory.* The content of the working memory is solely composed of the outputs of processing components. Working memories also connect to and exchange information with other working memories in other sub-architectures. In our implementation of CAS the communication method between the working memory and the components determines the efficiency of the model. But for now let us consider simply that the schema itself allows reading and writing to working memories, and transfer of information between them.

This use of shared working memories is particularly well suited to the *collaborative refinement of shared structures*. In this approach to information processing, a number of components use the data available to them to incrementally update an entry on shared working memory. In this manner the results of processing done by one component can restrict the processing options available to the others in an informed way. As all components are active in parallel, the collective total processing effort (i.e. the amount of work done by all the components in solving a problem) may be reduced by sharing information in this way. This feature turns out to be very powerful aspect of the schema.

Processing management

Our previously discussed requirements included the requirement that any design should support the explicit control of processing. Although control strategies could be implemented using communication via working memory entries, failing to support control requirements in the schema would mean that they would fall outside of the regions of design space we could explicitly manipulate with it. It would also mean that system designers would have to reinvent the control strategies they required with each new instantiation. The schema therefore supports control strategies by including a dedicated control component, the *task manager*, in each sub-architecture. In addition to the usual component connections to working memory, each task manager has a dedicated control connection to each component in its sub-architecture. Task managers are also connected across sub-architectures, allowing control strategies to be coordinated across entire instantiations. The task manager can operate in either a demand driven mode or in request mode. In the demand driven mode components request permission to perform a particular task and then have this request accepted or rejected by the task manager. In request mode, the task manager sends task requests to components which can then be accepted or rejected.

4.3 System wide design

While a system could be composed of a single sub-architecture we intend that there should typically be several sub-architectures in operation. Support for multiple sub-architectures is required in the schema for a number of reasons. It allows the designer of an instantiation to include separate modules in their

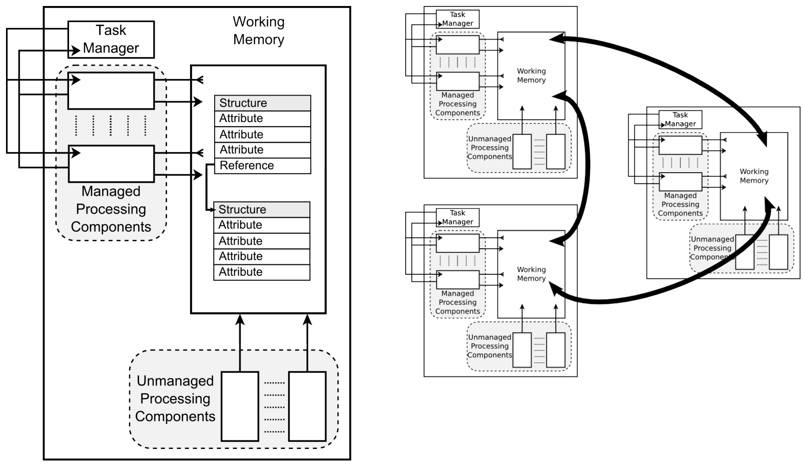


Fig. 1. Left panel: A single sub-architecture within CAS. There are components, which run in parallel, asynchronously updating shared structures on a common working memory. They can also take input from sensors or give output to actuators. The task manager determines whether managed components are able to run, while unmanaged components always run. **Right panel:** sub-architectures are coupled to make an overall system. Local changes are transmitted globally between working memories, not components directly.

system, where each sub-architecture plays a specialised role in processing. This contributes towards satisfying the requirement of supporting the multiple representations required to develop a robotic system from currently available technology. Multiple sub-architectures also, if implemented correctly, improve support for concurrency.

In the integrated systems we describe in Chapters 9 and 10 we have between three and nine sub-architectures. The schema, note, makes no assumptions about whether system decomposition should be predominantly according to behaviour or information processing function. What is important is that the decomposition groups components that commonly exchange information via shared structures. When events occur on a working memory computation linear in the number of components must be performed to trigger processing by the right components. Early on in our system building experiences we discovered that doing this for all components in the whole system meant the system was paralysed by information change. One of the main benefits of having distributed working memories is precisely that the working memory can act as a gate-keeper for its local components, only letting them become aware of events elsewhere in the system when necessary. We show why this arrangement is beneficial in our exploration of different sub-schemas later in the chapter. Having loosely coupled sub-architectures also allows us to explore architecture sub-schemas where there are global controllers that utilize very abstract representations. We explore some specific architectural instantiations in Chapter 10.

In the next subsection we describe in more the way that CAS is implemented in CAST. This includes discussion of the memory model, and the communication model. Following that we define four problems that are raised by CAS, and indeed by any architecture schema which satisfies our assumptions from Section 3. After describing these we describe sub-schemas for CAS that address each of the four problems.

4.4 CAST: A toolkit implementing CAS

In Section 4 we described how CAS works at an abstract, conceptual level. However, the details of the implementation of our schema in CAST determine a great deal about its efficiency, and how easy (or hard) it is for it to be specialised in one way or another — i.e. the kinds of moves the implementation supports through the space of architectures. In particular the communication, filtering and memory access models employed by CAST are key in how efficient particular architectural instantiations of the schema tend to be.

In our schema, a working memory is an associative container that maps between unique identifiers and *working memory entries*. Each entry is an instance of a *type*, which can be considered as analogous to a class description. A working memory entry can be any information that can be encapsulated into a single object class. A system that includes visual components may, for example, include entries that describe regions of interest and visually determined objects. A system that must navigate through a building may have entries representing maps of various floors, and objects that have been detected.

Components can add new entries to working memory, and overwrite or delete existing entries. Components can retrieve entries from working memory using three access modes: id access, type access and change access. For id access the component provides a unique id and then retrieves the entry associated with that id. For type access the component specifies a type and retrieves all the entries on working memory that are instances of this type. Whilst these two access modes provide the basic mechanisms for accessing the contents of working memory, they are relatively limited in their use for most processing tasks. Typically most component-level processing can be characterised by a model in which a component waits until a particular change has occurred to an entry on working memory, before processing the changed entry (or a related entry). To support this processing model, components can subscribe to *change events*. Change events are generated by the working memory to describe the operations being performed on the entries it contains. Different instantiations of the schema may support different content for change events, but a minimum set of useful information is the unique id and type of the changed entry, the name of component the made the change, and the operation performed to create the change (i.e. whether the entry was added, overwritten or deleted).

As stated previously an instantiation of the schema can be composed of one or more sub-architectures. The addition of multiple sub-architectures requires that the single sub-architecture schemes for working memory access are extended. We assume that a design would not include multiple sub-architectures unless necessary for imposing modularity on the processing, and so use sub-architecture boundaries to impose restrictions on cross sub-architecture communication. By default a change event is only broadcast within the sub-architecture in which it was generated. This restricts knowledge about the changes on working memory to within a sub-architecture. If a component should require information about changes on a working memory in a different sub-architecture, it can choose to subscribe to these changes. This action opens up a connection between the two working memories in question (the working memory where the change was generated, and the working memory local to the component requesting the information), down which all requested changes are sent as they are generated. The receiving working memory then includes any changes it receives from other sub-architectures in the list of changes it broadcasts within its sub-architecture. The schema allows restrictions to be placed on which working memories a component can access. The default is that a component can read entries from any working memory, but only write to the working memory in its own sub-architecture. Any variation of this scheme can be specified by an instantiation of the schema, allowing components to write to working memories as required.

As described in Section 4 support for multiple sub-architectures is required in the schema for a number of reasons. One of these is that in the given an efficient implementation it increases the support for concurrency in the system. In CAST although components are concurrently active, their parallel interactions via working memory have the potential to become a bottleneck in processing. By distributing processing across multiple working memories, CAST allows designers to avoid these potential delays caused by unrelated processing.

5 Four problems

From our first year of system building experience in the PlayMate, and our attempts to build systems using other architectures (e.g. OAA) we realised that there were a number of problems that must be addressed by all embodied intelligent systems that exchange representations between different sub-systems, and which must satisfy the requirements and assumptions we listed in Section 3. We refer to these as the *binding problem*, the *filtering problem*, the *processing management problem* and the *action fusion problem*. We define them as follows.

The *binding problem* arises as soon as we have a system with two pieces of information in different places that refer to the same entity. In some instances, to produce coherent decision-making and action execution we need to relate those pieces of information to one another. For example, if I have several blue objects in front of me, and someone refers to an entity as being "blue" how do I decide which entity that I can see is the object of the referent? In general, given many pieces of information residing in different sub-systems, how should the overall system efficiently decide which pieces are related to which other pieces and in what ways? In short *how do we match information from one component with information from another?* The binding problem occurs in many forms, and is a well studied phenomenon in the visual system [30], and in neural architectures [29].

The *filtering problem* arises as soon as a piece of information is created in one sub-system. How should the system decide where else that piece of information is needed? The key issue here, as stated above, is that we do not want to share all information with all sub-systems. The problem is that where information is needed depends both on the context and the problem the system is trying to solve [24]. We call this the filtering problem because we think of it precisely as the problem of deciding what information to send, and what information to filter away, i.e. to hold back from other sub-systems. Filtering mechanisms need to be cheap, context sensitive and to generate few false positives, and very few false negatives.

The *processing management problem* is the problem that arises because we do not have enough computational resources to draw all possible inferences from the sensory data. It is acknowledged that many animals utilise some form of attention to limit processing. On the one hand we can limit what information is processed, and on the other we can limit the processing that is done to the information selected. In other words we want to manage the processing according to the task. We want to investigate the different possible mechanisms and the trade-offs between them. In some solutions or models of human processing the solutions to the binding, filtering and processing management problems are intimately related.

The *action fusion* problem arises when different sub-systems recommend actions to motor systems that need to be fused or otherwise coordinated. Perhaps two behaviours are vying for control of a motor subsystem, or perhaps the activities of two separate motor systems need to be coordinated. In either case the architectural schema must have mechanisms for enabling this coordination. Behaviour based systems use arbitration or fusion mechanisms, which are limited to a very small number of tasks. Planners use action models to coordinate activity, and can produce controllers on the fly for many different tasks, but each of which has only limited feedback. The architectural question is again what the trade-offs are between different approaches. In CoSy we have not yet investigated this question, but sketch some possible choices at the end of the chapter.

We now go on to describe, for each of the first three problems, the solution spaces that we have investigated. We have captured the best of these solutions and incorporated them into CAS as sub-schemas, i.e. as more specific parts of our architectural schema.

5.1 Binding

Requirements on binding

Earlier, we stated that systems suitable for the PlayMate and Explorer scenarios often need to connect related information across sub-systems. We referred to this as the binding problem[3] [15, 13]. As a simple example take a system that can discuss and manipulate objects in a tabletop scene. Perhaps it receives the instruction ”put the blue things to the left of the red thing”. To carry this out it must be able to relate disparate pieces of information. The object identities and their properties from vision must be connected with corresponding information from the utterance. The robot also needs to connect not just information about physical entities, but also about relations between them — there is information in this example about current spatial relationships from both vision and language. Finally the goal state from language must be related to a possible spatial configuration, and the objects in this hypothetical state related to the objects in the current state. In fact the binding problem exists in a much larger range of designs than those necessary for the PlayMate or Explorer or indeed those covered by CAS. It exists in *any* system where information from multiple sub-systems must be *fused* in order to make decisions. The binding problem is related to theory tethering[4] and symbol grounding [8], in that some kind of binding must underlie either approach. After four years of investigating different approaches we have included a method for solving the binding problem in CAS. In this section we describe our solution as it currently stands.

There are two constraints from our scenarios that have influenced our approach to binding. The first is that one of the main features of both our scenarios is the need for an ability to perform deliberative reasoning, by which we mean processes that explicitly represent and reason about hypothetical world states. The second is that because the systems are also embodied, deliberation relies on representations derived from perceptual subsystems that are unreliable and update at unpredictable rates. Embodiment also requires that representations are interpretable by effector sub-systems.

These constraints lead us to impose the following three requirements on our solution to the binding problem. First, *binding must produce representations that are stable* for the duration of the deliberative processes they are involved in. For example, a representation of an object from vision should remain stable across multiple image frames if that object is to be involved in a planning process. Second, these *representations produced by binding must be at a level of abstraction appropriate for the processing they will be involved in.* For example, the positions of objects on a tabletop could be represented metrically or as predicate relations. The first is necessary for visual servoing,

[3] We realise that there are other binding problems in other fields, e.g. neuroscience, but they are somewhat different to the problem here.

[4] See http://www.eucognition.org/wiki/index.php?title=Symbol_Tethering

the second for understanding utterances about the scene. The two requirements are linked: the level of detail influences temporal stability, in that more abstract representations are typically more durable. Our third requirement arises because the symbols to be bound come from concurrently active, asynchronously updating sub-systems, binding cannot happen in a synchronous manner. To keep a representation of the state as current as possible, it is important that perceptual information is processed as soon as it is generated. Therefore it is important that any representation generated by binding can be incrementally and asynchronously extended as soon as new information is available. To summarise, the requirements on our approach to binding are:

- The representations it produces should be stable across the lifetime of the processing for which they are necessary.
- The representations should have the appropriate level of abstraction for the processing for which they are necessary.
- The process of binding must be proceed in an asynchronous and incremental manner.

The binding solution: overview

Implicit Binding

Our solution to the binding problem within CAS relies on the separation of representations into two levels of abstraction. At the low level we have sub-architecture specific representations (Figure 2). Within a sub-architecture the representations that reside on the working memory can be structured, they could for example be slot and filler structures. These structures allow the results of components to be bound together: two components can fill in different slots in the same structure. This could be because the different pieces of information in the two slots were derived from the same original data (e.g. the same Region of Interest in an image), or from two different data items that the system designers deem to be related (e.g. a ROI tracked across two frames). This kind of binding therefore relies entirely on the structure of the representations, and the relationship of the processing components as decided at design time. The binding is not explicitly decided by the system, and thus we call it *implicit binding*. Implicit binding has turned out to be a very powerful feature of CAS. In addition since structures on the working memories may contain links to other structures — possibly on other working memories — there is the ability to use implicit binding to bind information along a single *processing trail*. This kind of book-keeping turns out to be another powerful feature of CAS. The reason for this is that it allows us to abstract information that changes slowly from information that changes rapidly, e.g. the identity of an object from the visual description of that object. We can then store the slowly changing information in a new structure that has a reference back to the fast changing structure. So, in our example, the pose of the object could be recovered without re-abstraction by using the references in the processing trail. This is another kind of implicit binding.

Implicit binding, while powerful, is not suitable in all situations. If the decision to bind two pieces of information from two different sub-architectures must be made at run-time then we need *explicit binding*. In explicit binding the two pieces of information are translated into a general representation, and in this form are written to the working memory of a specialised binding sub-architecture. A third comparison component decides whether they should be bound into a set. This set is written to the binding working memory and is globally readable by the whole system.

The motivation for this centralised approach rather than any other is one of minimising the effort of translation. Suppose we have no general representation. This means that if every one of N sub-architectures needs information from all the others $N \cdot (N-1)$ pair-wise translation processes are required. Whereas if we have a general representation only at most $N \cdot 2$ translators are needed[5]. From a systems engineering viewpoint, translation to a single general representation makes the translation process more modular and less redundant (although this depends on implementation details). It is more modular in the sense that all translations from a collection of sub-architecture specific representations happen in one place in the system, rather than (potentially) in $N-1$ places. It reduces redundancy because the $N-1$ translations may feature many similar operations, whereas these can all be grouped into a single translation into the general representation.

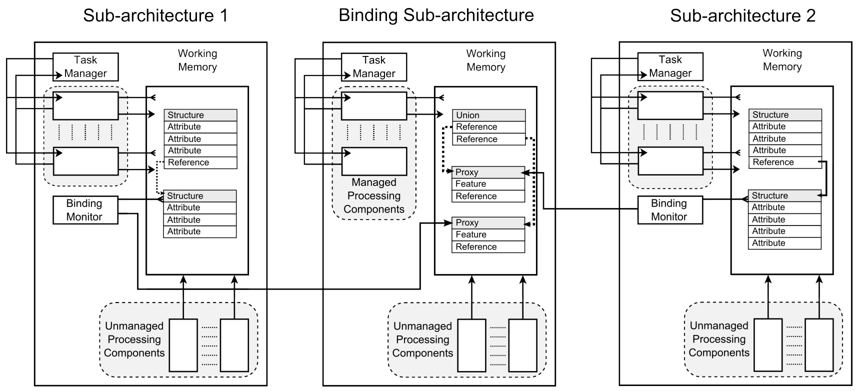


Fig. 2. The binding sub-architecture.

To implement this scheme (see Figure 2) each sub-architecture has a special component called a *binding monitor* that translates any structures on its working memory into *binding proxies* that are written to the binder's work-

[5] This assumes separate steps for translating into and out of the general representation, which may be reduced into a single step in practice.

ing memory. A proxy is simply a set of *binding features*. Features are in turn simply attribute-value pairs. For example, a visual sub-architecture may create proxies for visible objects that have feature attributes for colour, shape, ontological category etc. The role of a binding monitor is to keep its proxies linked to the source information in its sub-architecture, and to update them as that source changes. This link allows other components to operate on the proxies in place of the source data.

All the binding monitors in the system write their proxies to the working memory in the *binding sub-architecture*. There a collection of components which we will refer to as *the binder* groups these proxies into *binding unions* based on whether their features match. Unions also have minimal internal structure, but are instead composed of the union of the sets of features from its bound proxies. The set of unions on binding working memory represents the current best architecture-wide hypothesis of the current state of the things the sub-architectures can represent about the world (including its own internal processing). This is based on the assumption that the underlying proxies and features are also the best hypotheses from the corresponding sub-architectures. The comparison of features is performed by feature *comparators*. Each comparator compares features to determine whether they could refer to the same underlying item of information (e.g. whether the colour or spatial location of two objects is the same). The collected results of these comparisons is used to determine whether a proxy could be bound into a union with other proxies. We now give a formal description of binding.

The binding solution: details

The set of possible features is broad:

Definition 1. A feature space $\Phi^x \in \mathbf{\Phi}$ is any data format in the space of all possible data formats, $\mathbf{\Phi}$. $\phi_i^x \in \Phi^x$ denotes an instantiation of a particular representation where x should be interpreted as any feature space name.

For example, $\phi_{red}^{ColourLabel} \in \Phi^{Colour}$ denotes the colour "red" in the representation space of colours. In our CAST instantiation, $\mathbf{\Phi}$ corresponds to any representation that may inhabit a working memory.

Information from the sub-architectures (SAs) is shared as a collection of proxies:

Definition 2. A binding proxy is a structure $p = \langle F_p, u_p \rangle$ where F_p is a set of instantiated features of different types (i.e. $F_p = \{\phi_1^{x_1}, \phi_2^{x_2} \dots \phi_n^{x_n}\}$) and u_p refers to a binding union with which the proxy is bound (see below).

The unions should express information from proxies that, by all accounts (cf. Algorithm 3), refer to the same entity. Unions simply inherit the features of the bound proxies and are defined as:

Definition 3. A binding union is a structure $u = \langle F_u, \mathbf{P}_u \rangle$ where $\mathbf{P}_u$ refers to the subset of proxies unified by the union u and F_u is defined as the union of the features in all proxies in $\mathbf{P}_u$.

The problem for the binder is to assess whether two proxies are matching or not. By matching we mean that they should refer to the same thing. To do this, all new or updated proxies are compared to all unions on the basis of their respective features. The basis of this comparison is that each pair of feature types has an associated comparator function:

Definition 4. A feature comparator is a function $\Delta : \Phi^x \times \Phi^y \rightarrow \{true, false, indeterminate\}$ returning a value corresponding to whether two feature instances are equivalent (or similar enough) or not. The comparator can also choose to not return a definite answer if the answer is undefined, or the uncertainty is too big (i.e. *indeterminate*).

Obviously, *indeterminate* is the only answer most such comparators can return, e.g. the comparison of a Φ^{Colour} and a $\Phi^{Position}$ is likely undefined[6]. However, for many pairs of features there exist informative comparators. For example, features such as linguistic concepts can be compared to other concepts (with ontological reasoning) or physical positions can be compared to areas.

Definition 5. Two feature spaces (Φ^x, Φ^y) are *comparable* iff $\exists(\phi_i^x, \phi_j^y) \in (\Phi^x, \Phi^y)$ such that $\Delta(\phi_i^x, \phi_j^y) \neq indeterminate$.

The more pairs of features from different SAs that are comparable, the more likely it is that proxies from these SAs will be accurately matched.

To compare a proxy and a union, the corresponding feature sets are the basis for scoring:

Definition 6. The binding scorer is a function $S^+ : \mathcal{P} \times \mathcal{U} \rightarrow \mathbb{N}$ where $\mathcal{P}$ and $\mathcal{U}$ denote the set of all proxies and unions respectively and

$$S^+(p, u) = \sum_{\phi_i^x \in F_p} \sum_{\phi_j^y \in F_u} \begin{cases} 1 \text{ if } \Delta(\phi_i^x, \phi_j^y) = true \wedge \phi_i^x \neq \phi_j^y \\ 0 \text{ otherwise} \end{cases}$$

where F_p and F_u are the feature sets of p and u respectively.

Note that identical features are not counted. This to prevent a union getting a higher score just because it is compared to one of its member proxies (this would sometimes prevent a proxy switching to a better union). The number of feature mismatches is also counted (i.e. with $true$ replaced with $false$ in S^+). That function is here denoted $S^- : \mathcal{P} \times \mathcal{U} \rightarrow \mathbb{N}$.

S^+ and S^- are the basis for selecting the best among all unions for each new or updated proxy. This is conducted by the function `bestUnionsforProxy`

[6] Of course, in the implementation, such undefined comparators are never invoked. Mathematically, however, this is exactly what happens.

```
bestUnionsforProxy($p$,$\mathcal{U}$)
Input: A proxy, $p$, and the set of all unions, $\mathcal{U}$.
Output: Best union(s) with which a proxy should bind.
begin
    $best := \emptyset$;
    $max := 0$;
    for $\forall u \in \mathcal{U}$ do
        if $S^{-}(p,u) = 0 \wedge S^{+}(p,u) > max$ then
            $best := \{u\}$;
            $max := S^{+}(p,u)$;
        else if $S^{-}(p,u) = 0 \wedge S^{+}(p,u) = max$ then
            $best := best \cup \{u\}$;
        end
    end
    return $best$;
end
```

Fig. 3. The algorithm which computes the set of best candidate unions for being bound with a new or updated proxy (see definitions 1-6 for an explanation of the notations).

described in Algorithm 3. The result of $best =$ `bestUnionsforProxy` is a set of zero, one or more unions. If $|best| = 0$ then a new union will be created for the proxy p alone (i.e. with all the features of p). If $|best| = 1$, then the proxy is bound to that union.

When $|best| \geq 2$ we are faced with a *ambiguity.* To avoid deadlocks in such cases the binder can select a random union from *best* for binding. However, bindings are *sticky*, meaning that if an already bound proxy subsequently matches a union in a larger "best"-list, then it will not switch to any of those unions. This to avoid excess processing in, and signaling from, the binder. This also helps to satisfy our requirement for symbols to be stable as far as possible. Ambiguities cannot be solved by the binder itself, but it can request help from other SAs. This may result, in principle, in the communication SA initiating a clarification dialogue with a human tutor.

Relations and Groups

The proxies and unions described so far have been assumed to roughly correspond directly to physical objects. They may also correspond to more abstract entities as well. To support this, two special proxy types are implemented in a slightly different manner: proxies denoting groups of proxies, and proxies denoting relationships between proxies.

Since proxies contain features that are of any representable type, proxies can also have features attributable to groups and relations, e.g. cardinality and relative metric information respectively, and explicit references to relating proxies. Currently we handle groups in a fairly simple yet direct way: a special kind of "group proxy" is created exactly like an ordinary binding proxy with all the features that the members of the group have in common (e.g. "the blue balls to the left of the mug" creates a group with features $\phi^{Concept}_{ball}$ and $\phi^{ColourLabel}_{blue}$ and with a spatial relation $\phi^{SpatialRel}_{left\_of}$-proxy to the $\phi^{Concept}_{mug}$-proxy. A separate process in the binding SA (the "group manager") then spawns off individual proxies which inherit the features of the group proxy. Every time an individual is bound to something, a new proxy is spawned[7]. To all the other processes, the individuals appear as an endless supply of normal proxies.

Relation proxies are implemented in a similar way as standard proxies, but with additional features indicating the other proxies involved in the relation. Features of relation proxies are thus compared using the same mechanism that compares the features of standard proxies. For example, spatial metric features, e.g. $\phi^{\mathbb{R}^3}_{\mathbf{(x,y,z)}}$, could in principle be compared to a linguistic feature describing the same relation, e.g. $\phi^{SpatialRel}_{left\_of}$. It has turned out that features that link relations to normal proxies and vice versa make the scoring inefficient. Therefore, a separate scoring scheme similar to that in definition 6 is used to assess how well proxies match to unions w.r.t. their relational context.

Assume that union u_1 has a relation (union) $u_{1\rightarrow 2}$ to union u_2 If a now three additional proxies are added, arranged internally as two proxies and a relation proxy between them, p_a, p_b and $p_{a\rightarrow b}$ respectively, it is possible that they will be bound with the existing unions. First of all, if $S^+(p_a, u_1) = 0$ and $S^-(p_a, u_1) = 0$, then p_a may be unified despite no convincing score if $p_{a\rightarrow b}$ is unified with $u_{1\rightarrow 2}$. In other words, the relational context of an ordinary proxy, as defined by the relational proxies, can tip over the balance and favour that it binds with an existing union despite that there are no features that match. The relational proxy $p_{a\rightarrow b}$ may also be unified under similar conditions (i.e. where p_a is unified with u_1). If $S^-(p_a, u_1) > 0$ or $S^-(p_b, u_2) > 0$, however, the relation will not bind even if $S^+(p_{a\rightarrow b}, u_{1\rightarrow 2}) > 0$ and $S^-(p_{a\rightarrow b}, u_{1\rightarrow 2}) > 0$. The reason is that two relations that are between entities that cannot be the same, can also not be the same relation.

Visual & Spatial Reference Resolution

To illustrate how our binder supports a number of behaviours typically required of robots that interact with humans, the following sections present a number of examples taken from the Explorer and PlayMate systems. Perhaps the most common use of information fusion systems is to interpret linguistic references in terms of visual information. Our binder handles this task as an instance of a more general problem of information fusion. We will here consider the simple situation where we have a red object and two blue objects on the table. The objects are arranged in a straight line of alternating colours. The human then asks the robot to "put the blue objects to the left of the red objects".

[7] With some obvious limitations to allow finite groups and to prevent excess proxies being generated when members of different groups merge.

We will start our example in the visual sub-architecture, where change detection, tracking and segmentation components create representations of the objects in the scene. These objects have 3D poses and bounding boxes and a number of slots for visual properties such as colour, shape and size. These slots are filled by a recogniser that has been previously trained (see Chapter 7) using input from a human trainer [25]. For this example we assume the recogniser correctly extracts the colours of the objects as red and blue. When the scene becomes stable (determined by the change detector) the visual subarchitecture binding monitor creates a proxy for each of the currently visible objects. As the visual property recogniser processes the objects, the monitor updates the proxies with features reflecting these properties. This is an incremental process, so the visual proxies are updated asynchronously as the objects are processed. At this point only the visual proxies are present in the binding working memory, each one is bound to its own union.

The presence of objects in the visual working memory is also noticed by the components in the spatial subarchitecture. These abstract the objects as points on the tabletop, and the spatial binding monitor creates a proxy for each. These proxies are tagged with the ID of the visual proxy for the corresponding object so they are bound correctly[8]. Concurrently with the proxy creation, qualitative spatial relations between the spatial objects are added to working memory. These are generated by components using potential-field-based models of spatial relations [3]. In our example the two blue objects are to the left and to the right of the red object respectively. They are both also near the red object (but not near each other). As these relations are added, the spatial binding monitor reflects them on the binding working memory as relation proxies between the spatial proxies. The binder uses these as the basis of relations between the unions featuring the spatial proxies. This provides our basic model of the current state.

When the human speaks to the robot, a speech recognition module in the communication subarchitecture is triggered. The resulting speech string is written to the communication working memory. This triggers a cycle of deep sentence analysis and dialogue interpretation, yielding a structured logical description of the utterance's content. From this structure the communication binding monitor generates communication proxies for the discourse referents from the utterance and the relations between them. These proxies include features that can match against both those attached to visual proxies (colour, shape and size), and those attached to spatial proxies (relations based on spatial preposition). In the example two proxies are generated: one normal proxy for the red object, and one group proxy for the blue objects. The binder uses the features of these communication proxies to bind them into unions with the visual and spatial proxies. In the example the $\phi_{red}^{ColourLabel}$-proxy is bound together with the visual and spatial proxies relating the red object,

[8] A similar, but more general, functionality could be generated by matching location-derived features.

and the $\phi_{blue}^{ColourLabel}$-proxies spawned from the corresponding group proxy (see Section 5.1) are bound with the remaining proxies for the blue objects. This provides the system with an interpretation of the utterance in terms of the visual scene.

In this example, the process of reference resolution involves simply ensuring that the communication proxies referring to visual entities (i.e. those referring to objects in the tabletop scenario) are bound to unions that have a visual component. If the utterance contains spatial language, then relation proxies are generated by the communication binding monitor. This causes the binding process to bind proxies via the relations between proxies as well as the features of single proxies. Failure to bind proxies can trigger a number of different processes.

Binding summary

In this section we have described two main mechanisms to tackle the binding problem: implicit binding and explicit binding. Implicit binding is essentially a design time choice, while explicit binding is a run time decision by the system itself. We have described how implicit binding also allows us to implement stable abstract features that are linked to rapidly changing features via the creation of *processing trails*. Finally we have described how explicit binding occurs. In addition to the basic mechanism we have also explored the problem of early binding. This is when possible bindings can be used to cut down the number of possible interpretations in some sub-architecture specific process. In other words tentative bindings across sub-architectures can prune hypotheses within those sub-architectures. This kind of approach we term *incremental binding* and it is described in Chapter 8. The key issue with binding is whether or not a centralised approach to the problem is the right one. We have shown that it is possible, not for which kinds of niches it is the right choice.

5.2 Filtering

Previously we described the *filtering problem* as being how to decide efficiently where a piece of information that arises in one sub-system needs to be sent. In other words it is a problem of efficient information sharing. In CAS our atomic information generating sub-systems are components. There is a space of possible models for information sharing between components, ranging from point-to-point communication (i.e. that used by our OAA-based first PlayMate system) to a broadcast model. Between these two extremes exist a range of possible systems in which components share information with a subset of components. Which model is chosen can have a great impact on the final system behaviour. In this section we use the shared memory-based design of CAS to explore the effects of varying the information sharing patterns between components empirically. Specifically we achieve this by altering the ratio of components to sub-architectures in a subset of the PlayMate system.

We start with an n-m design where n components are divided between m sub-architectures, where $n > m > 1$. The design is part of the Play-Mate system described in Chapter 10, in which components are assigned to sub-architectures based on functionality (vision, binding or qualitative spatial reasoning), although for this experimental work arbitrary n-m assignments are also possible (and would explore a wider area of design space). We then reconfigure this system to generate architectures at two extremes of the design space for information sharing models. At one extreme we have an n-1 design in which all n components from the original system are in the same sub-architecture. At the other extreme of design space we have an n-n design in which every component is in a sub-architecture of its own. Each of these designs can be considered a schema specialisation of the CAS schema from which a full instantiation can be made.

These various designs are intended to approximate, within the constraints of CAS, various possible designs used by existing systems. The n-1 design represents systems with a single shared data store to which all components have the same access. The n-m design represents systems in which a designer has imposed some modularity which limits how data is shared between components. The n-n design represents a system in which a no data is shared, but is instead transmitted directly between components. In the first two designs a component has do to extra work to determine what information it requires from the available shared information. In the latter two designs a component must do extra work to obtain information that is not immediately available to it (i.e. information that is not in it's subarchitecture's working memory).

In order to isolate the effects of the architectural alterations from the other runtime behaviours of the resulting systems, it is important that these architectural differences are the *only* differences that exist between the final CAS instantiations. It is critical that the systems are compared on the same task using the same components. CAST was designed to support this kind of experimentation: it allows the structure of instantiations to be changed considerably, with few, if any, changes to component code. This has allowed us to take the original implementation described above and create the n-1, n-m, and n-n instantiations without changing component code. This means that we can satisfy our original aim of comparing near-identical systems on the same tasks, with the only variations between them being architectural ones.

To measure the effects of the architecture variations, we require metrics that can be used to highlight these effects. We previously presented a list of possible metrics that could be recorded in an implemented CAS system to demonstrate the trade-offs in design space [9]. Ultimately we are interested in measuring how changes to the way information is shared impacts on the external behaviour of the systems, e.g. how often it successfully completes a task. However, given the limited functionality of our experimental system, these kind of behaviour metrics are relatively uninformative. Instead we have

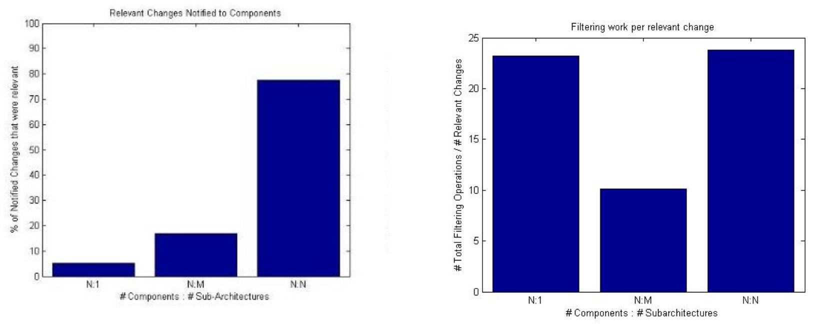


Fig. 4. Left panel: Average number of relevant change events received per component. Right Panel: Average filtering effort per relevant change event received.

chosen to focus on lower-level properties of the system. We have compared the systems on:

1. variations in the number of **filtering operations** needed to obtain the change events necessary to get information to components as required by the task.
2. variations in the number of **communication events** required to move information around the system.

As discussed previously communication and change events underlie the behaviour of almost all of the processing performed by a system. Therefore changes in these metrics demonstrate how moving through the space of information sharing models supported by CAS influences the information processing profile of implemented systems.

We studied the three different designs in two configurations: one with vision and binding sub-architectures, and the second with these plus the addition of the QSR subarchitecture. This resulted in six final instantiations which we tested on three different simulated scenes: scenes containing one object, two objects and three objects. Each instantiation was run twenty times on each scene to account for variations unrelated to the system's design and implementation.

The results for the filtering metric are based around the notion of a *relevant event*. A relevant event is a change event that a component is filtering for (i.e. an event that it has subscribed to). Figure 4 demonstrates the percentage of relevant events received per component in each instantiation. 100% means that a component only receives change events it is listening for. A lower percentage means that the connectivity of the system allows more than the relevant change events to get the component, which then has to filter out the relevant ones. This is perfectly natural in a shared memory system. The results demonstrate that a component in an n-1 instantiation receives the lowest percentage of relevant events. This is because within a subarchitecture, all changes are broadcast to all components, requiring each component to do a lot of filtering work. A component in an *n*-*n* instantiation receives the greatest percentage of relevant changes. This is because each component is shielded by a subarchitecture working memory that only allows change events that are relevant to the attached components to pass. In the *n*-*n* case because only a single component is in each subarchitecture this number is predictably high[9]. This figure demonstrates the benefits of a directly connected instantiation: components only receive the information they need.

However, this increase in the percentage of relevant changes received comes at a cost. If we factor in the filtering operations being performed at a subarchitecture level (which could be considered as “routing” operations), we can produce a figure demonstrating the total number of filtering operations (i.e. both those at a subarchitecture and a component level) per relevant change received. This is presented in Figure 4. This shows a striking similarity between the results for the *n*-1 and *n*-*n* instantiations, both of which require a larger number of filtering operations per relevant change than the *n*-*m* instantiations. In the *n*-*m* systems, the arrangement of components into functionally themed sub-architectures results in both smaller numbers of change events being broadcast within sub-architectures (because there are fewer components in each one), and a smaller number of change events being broadcast outside of sub-architectures (because the functional grouping means that some changes are only required within particular sub-architectures). These facts mean that an individual component in an *n*-*m* instantiation receives fewer irrelevant change events that must be rejected by its filter. Conversely a component in the other instantiations must filter relevant changes from a stream of changes containing *all of the change events in the system.* In the *n*-1 instantiations this is because all of these changes are broadcast within a subarchitecture. In the *n*-*n* instantiations this is because all of these changes are broadcast between sub-architectures. Figure 5 (left panel) shows that these results are robust against changes in the number of objects in a scene. Also, the nature of the results did not change between the systems with vision and binding components, and those with the additional QSR components.

Figure 5 (centre panel) demonstrates the average number of communication events per system run across the various scenes and configurations for the three different connectivity instantiations. This shows that an *n*-*n* instantiation requires approximately 4000 more communication events on average to perform the same task as the *n*-1 instantiation, which itself requires approximately 2000 more communication events than the *n*-*m* instantiation. Figure 5 (right panel) demonstrates that this result is robust in the face of changes to the number of objects in a scene. The nature of the results also did not change between the systems with vision and binding components, and those with the additional QSR components.

[9] The events required by the manager component in each subarchitecture mean the relevant percentage for the *n*-*n* instantiations is not 100%.

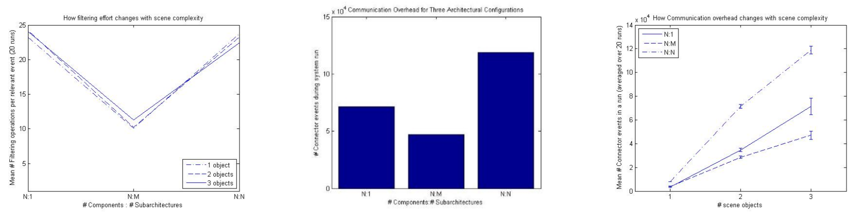


Fig. 5. Left panel: Average filtering effort per relevant event compared to scene complexity. Centre panel: Average total communication events per instantiation run. Right panel: Average total communication events per instantiation run compared to scene complexity.

This result is due to two properties of the systems. In the *n*-*n* system, every interaction between a component a working memory (whether it's an operation on information or the propagation of a change event) requires an additional communication event. This is because all components are separated by sub-architectures as well as working memories. In addition to this, the number of change events propagated through the systems greatly effect the amount of communication events that occur. In the *n*-*n* and *n*-1 instantiations, the fact that they effectively broadcast all change events throughout the system contributes significantly to the communication overhead of the system.

5.3 Filtering summary

From these results we can conclude that a functionally-decomposed *n*-*m* CAS instantiation occupies a "sweet spot" in architectural design space with reference to filtering and communication costs. This sweet spot occurs because having too much information shared between components in a system (the *n*-1 extreme) means that all components incur an overhead associated with filtering out relevant information from the irrelevant information. At the other extreme, when information is not shared by default (the *n*-*n* extreme) there are extra communication costs due to duplicated transmissions between pairs of components, and (in CAS-derived systems at least) the "routing" overhead of transmitting information to the correct components (i.e. the filtering performed by working memories rather than components).

In addition we have demonstrated here an empirical approach to comparing different points within design space, where we have held the content of the system constant, while making architectural changes. In this way we have also shown how to use CAST to help carry out the empirical part of the science of architectures we discussed at the beginning of the chapter. We now proceed to discuss the third of our problems, that of managing processing across the system.

5.4 Processing Management

In this section we discuss the problem of how a complex artificial cognitive system such as the Explorer or PlayMate systems we describe in Chapters 9 and 10 should manage their internal activity. In particular, when the processing possibilities exceed the processing resources, how should the robot choose what kind of processing to do? We refer to this as the *processing management problem*. We sketch several possible solutions to it, and then discuss the solution we have been exploring, which relies on technologies for planning under uncertainty.

To begin with consider a visual system that contains some of the many algorithms and representations described in Chapters 4 and 7. Each of these requires considerable computation to run, even in their classification (or non-learning) mode. In a robot with multiple competences we will need all of those vision algorithms and many more besides. For any natural visual scene running all such algorithms on all parts of the image is not feasible. This is not a problem in so far as we never need to perform all visual processing on a scene: the vision we need is determined by the task we are performing. To tackle this there has been much work on attention, and in particular the use of visual saliency models to identify which parts of the scene to process. There has been little or no work, however, on how to select which algorithms to run. It is this issue that we address here.

In our PlayMate domain, both a robot and human can converse about and manipulate objects on a table top (see [10]). As described previously, typical visual processing tasks in this domain require the ability to find the colour, shape identity or category of objects in the scene to support dialogues about their properties; to see where to grasp an object; to plan an obstacle free path to do so and then move it to a new location; to identify groups of objects and understand their spatial relations; and to recognize actions the human performs on the objects. Each of these vision problems is hard in itself, together they are extremely challenging. The challenge is to build a vision system able to tackle all these tasks. One early architectural approach to robot vision was to attempt a general purpose, complete scene reconstruction, and then query this model for each task. This is still not possible and in the opinion of many vision researchers will remain so. An idea with a growing body of evidence from both animals and robots is that some visual processing can be made more effective by tailoring it to the task/environment ([18, 12, 17].

Consider the scene in Figure 6, and consider the types of visual operations that the robot would need to perform to answer a variety of questions that a human might ask it about the scene: “is there a blue triangle in the scene?”, “what is the colour of the mug?”, “how many objects are there in the scene?”. In order to answer these questions, the robot has at its disposal a range of information processing functions and sensing actions. But, in any reasonably complex scenario (such as the one described above), it is not feasible (and definitely not efficient) for the robot to run all available information processing

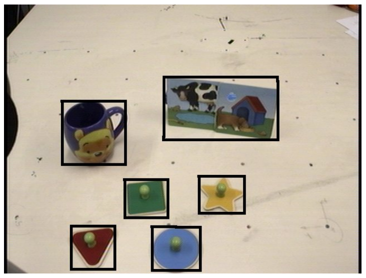


Fig. 6. A picture of the typical table top scenario—ROIs detected for processing are bounded by rectangular boxes.

functions and sensing actions, especially since the cognitive robot system needs to respond to human queries/commands in real-time.

There are many approaches that could be taken. The key choices concern how bottom up and top down processing are mixed. Within the constraints provided by CAS we have explored at least three approaches. In the first two approaches, processing opportunities can be identified as data arrives, and requests for processing resources be posted by the components to a local task manager. The task manager may simply have a policy which is unvarying, e.g. it permits all requests, or grants them up to some load threshold. This is what we have done for most our visual systems to date, because they have been very small. Alternatively the allocation policy could change according to the mode that the task manager is in. This is the approach we took in the communication sub-system for the Explorer and PlayMate systems. In this sub-architecture the task manager has modes, which it can switch between, and which are associated with different resource allocation policies. The third approach, and the one we detail here is to drive the processing in a largely top down way. In reality a mix of both top down and bottom up processing will be required. A top down approach essentially takes the current robot task, and uses this to determine which processing will be performed. There are several ways that we can conceive of the top determination of processing. In principle a (perhaps learned) task specific visual routine could be invoked from a library. The problem with this approach is that it is not at all obvious how to generalise from one task to another from a set of learned instances. A different approach would be to use planning to compose a completely new visual routine on the fly. The problem with this approach is that the planning process is itself expensive. However, we explore this approach, and show that it can work.

There already exists a body of impressive work on planning of image processing ([6, 28, 22, 19]). However, it is largely used for single images, requires specialist domain knowledge to perform re-planning or plan repair, has only been extended to robotic systems in the most limited ways, and poses the problem in an essentially deterministic planning framework or as a MDP ([19]). In our approach we push the field of planning visual processing in a new direction by posing the problem as an instance of probabilistic sequential decision making. We pose it as a Partially Observable Markov Decision Process (POMDP), thereby taking explicit, quantitative account of the unreliability of visual processing. Our main technical contribution is that we show how to contain one aspect of the intractability inherent in POMDPs for this domain by defining a new kind of hierarchical POMDP. We compare this approach with an earlier formulation based on the Continual Planning (CP) framework of [4]. Using a real robot domain, we show empirically that both planning methods are quicker than naive visual processing of the whole scene, even taking into account the planning time. The key benefit of the POMDP approach is that the plans, while taking slightly longer to execute than those produced by the CP approach, provide significantly more reliable visual processing than either naive processing or the CP approach. We give an overview of the POMDP approach here, describe the results, and relate it back to the CAS framework.

A POMDP formulation of visual processing

In robot applications, typically the true state of the world cannot be observed directly. The robot can only revise its belief about the possible current states by executing actions, for instance one of the visual operators.

We pose the problem as an instance of probabilistic sequential decision making, and more specifically as a Partially Observable Markov Decision Process (POMDP) where we explicitly model the unreliability of the visual operators/actions. This probabilistic formulation enables the robot to maintain a probability distribution (the *belief state*) over the true underlying state. To do this we need an observation model that captures the likelihood of the outcomes from each action. In this paper, we only consider actions that have purely informational effects. In other words, we do not consider actions such as poking the object to determine its properties, with the consequence that the underlying state does not change when the actions are applied. However, the POMDP formulation allows us to do this, which is necessary if we wish to model the effects of operators that split ROIs, move the camera, or move the objects to gain visual information about them.

Each action considers the true underlying state to be composed of the normal class labels (e.g. *red(R), green(G), blue(B)* for color; *circle(C), triangle(T), square(S)* for shape; *picture, mug, box* for sift), a label to denote the absence of any object/valid class—*empty* (E), and a label to denote the presence of *multiple* classes (M). The observation model for each action provides a probability distribution over the set composed of the normal class labels, the class label *empty* (E) that implies that the match probability corresponding to the normal class labels is very low, and *unknown* (U) that means that there is no single class label to be relied upon and that multiple classes may therefore be present. Note that U is an observation, whereas M is part of the underlying state: they are not the same, since they are not perfectly correlated.

Since visual operators only update belief states, we include “special actions” that answer the query by “saying” (not to be confused with language-based communication) which underlying state is most likely to be the true state. Such actions cause a transition to a terminal state where no further actions are applied. In the description below, for ease of explanation (and without loss of generality) we only consider two operators: *color* and *shape*, denoting them with the subscripts c, s respectively. States and observations are distinguished by the superscripts a, o respectively.

Consider a single ROI in the scene—it has a POMDP associated with it for the goal of answering a specific query. This POMDP is defined by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{Z}, \mathcal{O}, \mathcal{R} \rangle$:

- $\mathcal{S} : \mathcal{S}_c \times \mathcal{S}_s \cup \text{term}$, the set of states, is a cartesian product of the state spaces of the individual actions. It also includes a *terminal* state (term). $\mathcal{S}_c : \{E^a_c, R^a_c, G^a_c, B^a_c, M_c\}$, $\mathcal{S}_s : \{E^a_s, C^a_s, T^a_s, S^a_s, M_s\}$
- $\mathcal{A} : \{color, shape, sRed, sGreen, sBlue\}$ is the set of actions. The first two entries are the visual processing actions. The rest are special actions that represent responses to the query such as “say blue”, and lead to *term.* Here we only specify “say” actions for color labels, but others may be added trivially.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ represents the state transition function. For visual processing actions it is an identity matrix, since the underlying state of the world does not change when they are applied. For special actions it represents a transition to *term.*
- $\mathcal{Z} : \{E^o_c, R^o_c, G^o_c, B^o_c, U_c, E^o_s, C^o_s, T^o_s, S^o_s, U_s\}$ is the set of observations, a concatenation of the observations for each visual processing action.
- $\mathcal{O} : \mathcal{S} \times \mathcal{A} \times \mathcal{Z} \rightarrow [0, 1]$ is the observation function, a matrix of size $|\mathcal{S}| \times |\mathcal{Z}|$ for each action under consideration. It is learned by the robot for the visual actions (described in the next section), and it is a uniform distribution for the special actions.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \Re$, specifies the reward, mapping from the state-action space to real numbers. In our case:

$$\begin{aligned} \forall s \in \mathcal{S},\ \mathcal{R}(s, shape) &= -1.25 \cdot f(\text{ROI-size}) \\ \mathcal{R}(s, color) &= -2.5 \cdot f(\text{ROI-size}) \\ \mathcal{R}(s, \text{special actions}) &= \pm 100 \cdot \alpha \end{aligned}$$

For visual actions, the cost depends on the size of the ROI (polynomial function of ROI size) and the relative computational complexity (the *color* operator is twice as costly as *shape*). For special actions, a large +ve (-ve) reward is assigned for making a right (wrong) decision for a given query. For e.g. while determining the ROI’s color:
$\mathcal{R}(R^a_c T^a_s, \text{sRed}) = 100 \cdot \alpha, \mathcal{R}(B^a_c T^a_s, \text{sGreen}) = -100 \cdot \alpha$
but while computing the location of *red* objects:

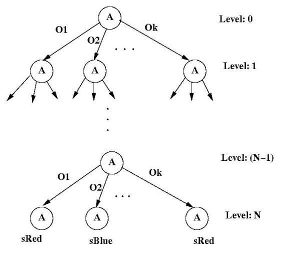


Fig. 7. Policy Tree of an ROI—each node represents a belief state and specifies the action to take.

$\mathcal{R}(B_c^a T_s^a, \text{sGreen}) = 100 \cdot \alpha$. The variable α enables the trade-off between the computational costs of visual processing and the reliability of the answer to the query.

Our visual planning task for a single ROI now involves solving this POMDP to find a policy that maximizes reward from the initial belief state. Plan execution corresponds to traversing a policy tree, repeatedly choosing the action with the highest value at the current belief state, and updating the belief state after executing that action and getting a particular observation. In order to ensure that the observations are independent (required for POMDP belief updating to hold), we take a new image of the scene if an action is repeated on the same ROI.

Actual scenes will have several objects and hence several ROIs. Attempting to solve a POMDP in the joint space of all ROIs soon becomes intractable due to an exponential state explosion, even for a small set of ROIs and actions. For a single ROI with m features (color, shape, etc.) each with n values, the POMDP has an underlying space of n^m; for k ROIs the overall space becomes: n^{mk}. Instead, we propose a *hierarchical decomposition*: we model each ROI with a lower-level (LL) POMDP as described above, and use a higher-level (HL) POMDP to choose, at each step, the ROI whose policy tree (generated by solving the corresponding LL-POMDP) is to be executed. This decomposes the overall problem into one POMDP with state space k, and k POMDPs with state space n^m. Space does not permit us to give the full details of the hierarchical POMDP formulation here, but these can be found in [27]. The key technical point is that in the HL-POMDP the observation function and the cost/reward specification for each action is based on the policy tree of a LL-POMDP that corresponds to that action. An example of the type of policy tree found for a LL-POMDP is given in Figure 7 where the LL-POMDP's policy tree has the root node representing the initial belief when the visual routine is called. At each node, the LL-POMDP's policy is used to determine the best action, and all possible observations are considered to determine the resultant beliefs and hence populate the next level of the tree.

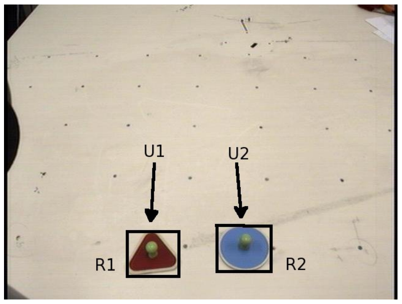
Once the observation functions and costs are computed, the HL-POMDP model can be built and solved to yield the HL policy. During execution, the HL-POMDP's policy is queried for the best action choice, which causes the execution of one of the LL-POMDP policies, resulting in a sequence of visual operators being applied on one of the image ROIs. The answer provided by the LL-POMDP's policy execution causes a belief update in the HL-POMDP, and the process continues until a terminal action is chosen at the HL, thereby answering the query posed. Here it provides the locations of all *blue* objects in the scene. For simpler occurrence queries (e.g. "Is there a blue object in the scene?") the execution can be terminated at the first occurrence of the object in a ROI. Both the LL and HL POMDPs are query dependent. Solving the POMDPs efficiently is hence crucial to overall performance.

A Continual Planning Formulation

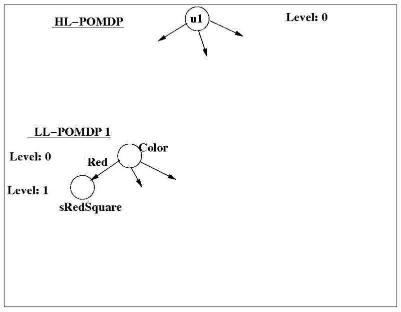
The Continual Planning (CP) approach of [4] interleaves planning, plan execution and plan monitoring. Unlike classical planning approaches that require prior knowledge of state, action outcomes, and all contingencies, an agent in CP postpones reasoning about unknowable or uncertain states until more information is available. It achieves this by allowing actions to assert that the preconditions for the action will be met when the agent reaches that point in the execution of the plan, and if those preconditions are not met during execution (or are met earlier), replanning is triggered. But there is *no* representation of the uncertainty/noise in the observation/actions. It uses the PDDL ([20]) syntax and is based on the FF planner of [11]. Consider the example of a *color* operator:

```
(:action colorDetector
:agent (?a - robot)
:parameters (?vr - visRegion ?colorP - colorProp )
:precondition (not (applied-colorDetector ?vr) )
:replan (containsColor ?vr ?colorP)
:effect (and
      (applied-colorDetector ?vr)
      (containsColor ?vr ?colorP) )   )
```

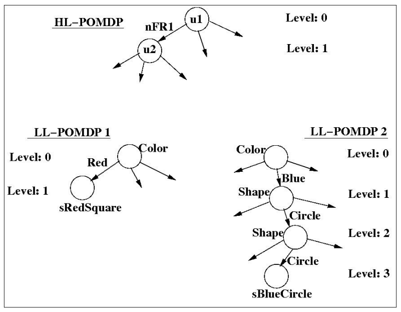
The parameters are a color-property (e.g. *blue*) being searched for in a particular ROI. It can be applied on any ROI that satisfies the precondition i.e. it has not already been analyzed. The expected result is that the desired color is found in the ROI. The "replan:" condition ensures that if the robot observes the ROI's color by another process, replanning is triggered to generate a plan that excludes this action. This new plan will use the *containsColor* fact from the new state instead. In addition, if the results of executing a plan step are not as expected, replanning (triggered by execution monitoring) ensures that other ROIs are considered. Other operators are defined similarly, and based on the goal state definition the planner chooses the sequence of operators whose effects provide parts of the goal state—the next section provides an example. The CP approach to the problem is more responsive to an unpredictable world than a non-continual classical planning approach would be, and it can therefore reduce planning time in the event of deviations from expectations.



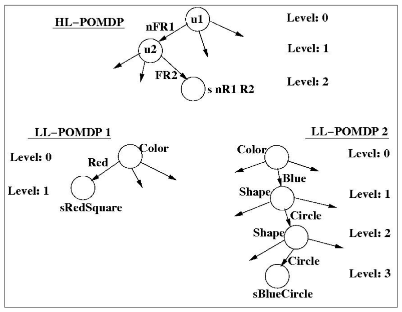
(a) Input image.



(b) Execution Step 1.



(c) Execution Step 2.



(d) Execution Step 3.

Fig. 8. Example query: "Where is the Blue Circle?" Dynamic reward specification in the LL-POMDP allows for early termination when negative evidence is found.

But, while actions still have non-deterministic effects, there is no means for accumulating belief over successive applications of operators. We show that the HiPPo formulation provides significantly better performance than CP in domains with uncertainty.

An example query

Figs 8(a)-8(d) show one execution example for an image with two ROIs.

The example query is to determine the presence and location of one or more *blue circles* in the scene (Fig 8(a)). Since both ROIs are equally likely target locations, the HL-POMDP first chooses to execute the policy tree of the first ROI (action u_1 in Fig 8(b)). The corresponding LL-POMDP runs the color operator on the ROI. The outcome of applying any individual operator is the observation with the maximum probability, which is used to update the subsequent belief state—in this case the answer is *red*. Even though it is more costly, the color operator is applied before shape because it has a higher likelihood of success, based on the learned observation functions. When the outcome of *red* increases the likelihood (belief) of the states that represent the

"Red" property as compared to the other states, the likelihood of finding a blue circle is reduced significantly. The dynamic reward specification ($\alpha = 0.2$) ensures that without further investigation (for instance with a shape operator), the *best* action chosen at the next level is a terminal action associated with the "Red" property—in this case it is *sRedSquare*. The HL-POMDP receives the input that a red square is found in R_1, leading to a belief update and subsequent action selection (action u_2 in Fig 8(c)). Then the policy tree of the LL-POMDP of R_2 is invoked, causing the color and shape operators to be applied in turn on the ROI. The higher noise in the shape operator is the reason why it has to be applied twice before the uncertainty is sufficiently reduced to cause the choice of a terminal action (*sBlueCircle*)—the increased reliability therefore comes at the cost of execution overhead. This results in the belief update and terminal action selection in the HL-POMDP—the final answer is ($s\neg R_1 \wedge R_2$), i.e. that a *blue circle* exists in R_2 and not R_1 (Fig 8(d)).

In our HiPPo representation, each HL-POMDP action chooses to execute the policy of one of the LL-POMDPs until termination, instead of performing just one action. The challenge here is the difficulty of translating from the LL belief to the HL belief in a way that can be planned with. The execution example above shows that our approach still *does the right thing*, i.e. it stops early if it finds negative evidence for the target object. Finding positive evidence can only increase the posterior of the ROI currently being explored, so if the HL-POMDP were to choose the next action, it would choose to explore the same ROI again.

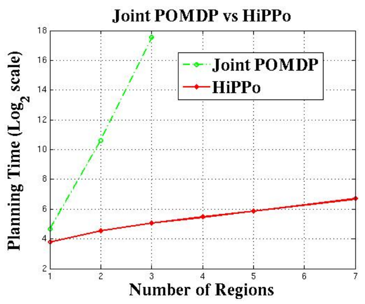
If the same problem were to be solved with the CP approach, the goal state would be defined as the PDDL string:

(and (exists ?vr - visRegion) (and (containsColor ?vr Blue) (containsShape ?vr Circle)))

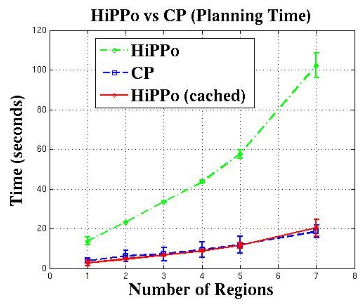
i.e. the goal is to determine the existence of a ROI which has the color *blue* and shape *circle*. The planner must then find a sequence of operators to satisfy the goal state. In this case it leads to the creation of the plan:

(colorDetector robot vr0 blue)
(shapeDetector robot vr0 circle)

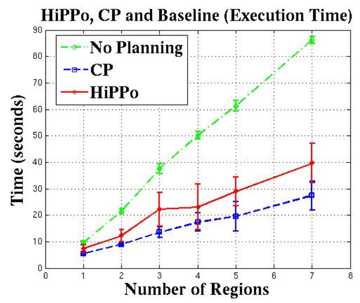
i.e. the robot is to apply the color operator, followed by the shape operator on the first ROI. There is a single execution of each operator on the ROI. Even if (due to image noise) an operator determines a wrong class label as the closest match with a low probability, there is no direct mechanism to incorporate the knowledge. Any thresholds will have to be carefully tuned to prevent mis-classifications. Assuming that the color operator works correctly in this example, it would classify the ROI as being *red*, which would be determined in the plan monitoring phase. Since the desired outcome (finding *blue* in the first ROI) was not achieved, replanning is triggered to create a new plan with the same steps, but to be applied on the second ROI. This new plan leads to



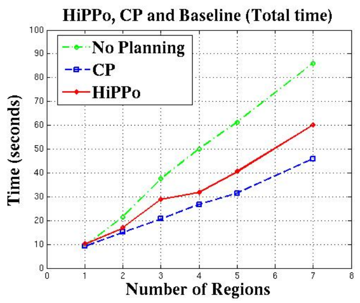
(a) HiPPo vs. joint POMDP. Joint POMDP soon becomes intractable.



(b) Planning times of HiPPo vs. CP. Policy-caching makes results comparable.



(c) Execution times of HiPPo, CP vs. No planning. Planning makes execution faster.



(d) Planning+execution times of HiPPo, CP vs. No planning. Planning approaches reduce processing time.

Fig. 9. Experimental Results—Comparing planning and execution times of the planners against no planning.

the desired conclusion of finding the *blue circle* in R_2 (assuming the operators work correctly).

An experimental study

In considering the trade-offs between the different types of solution to the processing management problem we consider several hypotheses that we can test. Specifically we hypothesise that:

- The hierarchical-POMDP planning (HiPPo) formulation is more efficient than the standard POMDP formulation.
- HiPPo and CP have comparable plan-time complexity.
- Planning is significantly more efficient than blindly applying all operators on the scene.
- HiPPo has higher execution time than CP but provides more reliable results.

In order to test these hypotheses we ran several experiments on the robot in the tabletop scenario. Objects were placed on the table and the robot had to analyze the scene to answer user-provided queries. Query categories include:

- Occurrence queries: Is there a red mug in the scene?
- Location queries: Where in the image is the blue circle?
- Property queries: What is the color of the box?
- Scene queries: How many green squares are there in the scene?

For each query category, we ran experiments over ~ 15 different queries with multiple trials for each such query, thereby representing a range of visual operator combinations in the planning approaches. We also repeated the queries for different numbers of ROIs in the image. In addition, we also implemented the naive approach of applying all available operators (color, shape and sift in our experiments) on each ROI, until a ROI with the desired properties is found and/or all ROIs have been analyzed.

Unlike the standard POMDP solution that considers the joint state space of several ROIs, the hierarchical representation does not provide the optimal solution (policy). Executing the hierarchical policy may be arbitrarily worse than the optimal policy. For instance, in the search for the *blue* region, the hierarchical representation is optimal *iff* every ROI is blue-colored. But as seen in Figure 9(a) that compares the planning complexity of HiPPo with the standard POMDP solution, the non-hierarchical approach soon becomes intractable. The hierarchical approach provides a significant reduction in the planning time and (as seen below) still increases reliability significantly.

Next, we compare the planning times of HiPPo and CP approaches as a function of the number of ROIs in the scene—Figure 9(b). The standard hierarchical approach takes more time than CP. But, the computationally intensive part of HiPPo is the computation of the policies for the ROIs. Since the policies computed for a specific query are essentially the same for all scene ROIs, they can be cached and not repeated for every ROI. This simple adjustment drastically reduces the planning time and makes it comparable to the CP approach.

Figure 9(c) compares the execution time of the planning approaches against applying all the operators on each ROI until the desired result is found. The HiPPo approach has a larger execution time than CP because it may apply the same operator multiple times to a single ROI (in different images of the same scene) in order to reduce the uncertainty in its belief state. In all our experiments the algorithms are being tested on-board a cognitive robot which has multiple modules to analyze input from different modalities (vision, tactile, speech) and has to bind the information from the different sources. Hence, though the individual actions are optimized and represent the state-of-the-art in visual processing, they take some time to execute on the robot.

A key goal of our approach is not only to reduce overall planning and execution time, but to improve the reliability of the visual processing. In these

Approach	**% Reliability**
Naive	76.67
CP	76.67
HiPPo	91.67

Table 1. Reliability of visual processing

terms the benefits are very clear, as can be seen in Table 1. The direct application of the actions on all the ROIs in the scene results in an average classification accuracy of 76.67%, i.e. the sensing actions misclassify around one-fourth of the objects. Using CP also results in the same accuracy of 76.67%, i.e. it *only reduces the execution time* since there is no direct mechanism in the non-probabilistic planner to exploit the outputs of the individual operators (a distribution over the possible outcomes). The HiPPo approach is designed to utilize these outputs to reduce the uncertainty in belief, and though it causes an increase in the execution time, it results in much higher classification accuracy: 91.67%. It is able to recover from a few noisy images where the operators are not able to provide the correct class label, and it fails only in cases where there is consistent noise. A similar performance is observed if additional noise is added during execution. As the non-hierarchical POMDP approach takes days to compute the plan for just two ROIs we did not compute the optimal plan for scenes with more than two ROIs, but for the cases where a plan was computed, there was no significant difference between the optimal approach and HiPPo in terms of the execution time and reliability, even though the policy generated by HiPPo can be arbitrarily worse than that generated by the non-hierarchical approach.

A significant benefit of the POMDP approach is that it provides a ready mechanism to include initial belief in decision-making. For instance, in the example considered above, if R_2 has a higher initial belief that it contains a *blue circle*, then the cost of executing that ROI's policy would be lower and it would automatically get chosen to be analyzed first leading to a quicker response to the query.

Figure 9(d) shows a comparison of the combined planning and execution times for HiPPo, CP, and the naive approach of applying all actions in all ROIs (no planning). As the figure shows, planning is worthwhile even on scenes with only two ROIs. In simple cases where there are only a couple of operators and/or only one operator for each feature (color, shape, object recognition etc) one may argue that rules may be written to decide on the sequence of operations. But as soon as the number of operators increase and/or there is more than one operator for each feature (e.g. two actions that can find color in a ROI, each with a different reliability), planning becomes an appealing option.

Summary of processing management work

In this sub-section we have explored the implications of the fact that within a complex cognitive system with many goals it will not be possible to perform all processing. We have studied this problem within the context of vision, specifically the kind of visual sub-architecture we use for the PlayMate scenario. Architecturally there are many possible solutions. Bottom up, data-driven processing is implemented naturally in CAS. In this section we have shown how to augment it with top down control, achieved using techniques for planning under uncertainty, and continual planning.

6 The relationship of CAS to previous work on architectures

6.1 Cognitive Architectures

There have been several attempts at unified theories of intelligence from within cognitive science. At least two of these emphasise the role of production systems. In SOAR Newell and Laird [16] proposed a production system model in which serial application of rules, written in a common form, modified representations held in a workspace shared by those rules. An important component of the theory was that there was a single unified representational language within which all data held in the shared workspace was expressed. Another key idea was that a set of meta-rules controlled the serial application of these productions. These three key ideas: a single shared workspace, serial application of processing elements, and a common representational language have been extremely influential. They are both simple to comprehend and allow the construction of effective systems. In ACT-R [1] John Andersen and colleagues have taken some of the elements of production systems and used them to produce models of aspects of human cognition that produce testable predictions. In ACT-R productions now represent the serial actions of processing in the thalamus and connect to information in buffers. Together these simulate the behaviour of multiple thalamic-cortical loops. ACT-R has been used to construct models that give impressively accurate predictions for human performance on a range of tasks, including reading and mental arithmetic. ACT-R models rely heavily on the provision of timing information about delays in each stage of processing. Both SOAR and ACT-R have in common the fact that they have widely available languages that allow researchers to implement computational models. Finally in Global Workspace Theory (GWT), Baars and Shanahan [24] have proposed the idea that conscious thought is explainable at an abstract level by the idea of a global workspace. The key idea in GWT is that local processes propose items to be posted onto a single global workspace, and that mechanisms exist that select one collection of items that are in turn re-transmitted to all the local processes. It can be seen that all three theories emphasise the idea of a single shared workspace for parts of cognition. There is evidence however, at least from robotics, that such a single workspace is an incomplete architectural account of intelligence. I now turn to describe ideas from robotics on architectures, in order to compare and contrast them with the ideas from work on cognitive architectures.

6.2 Robotic Architectures

The first significant attempt to implement what might be loosely called a cognitive robot was the Shakey project [23]. Detailed examination of their approach bears fruit. The architecture in Shakey was dominated by a central workspace within which all data about the contingent state of the world was written in a single representational language. In the case of Shakey this was a form of first order predicate logic minus quantification. Sensory processing was essentially a business of abstracting from the raw sensor data to this predicate description. Typing of entities was captured using predicates, and the representation also captured some metric information for the highly simplified world. Qualitative action effects were captured using STRIPS operators, which addressed some of the difficulties previously encountered by the situation calculus. This declarative knowledge about action effects was used in a planning process that had available all knowledge in the world model. In addition to this the robot had fixed routines that would update the world model when sufficient uncertainty had accumulated about its state. This was the way that gross error recovery occurred: through re-sensing and then recalculating the world model. Simpler errors were handled using Intermediate Level Actions (ILAs). These were essentially discrete closed loop controllers that relied upon the world to settle between each step and thus couldn't deal easily with ongoing rapid change. Overall, Shakey shares, at an architectural level, some of the assumptions of SOAR and ACT-R. It relies on serial application of planning operators to simulate trajectories through the state space and selects courses of action based on those. It uses a single representational language, although it does reason with that representation using two different kinds of reasoning. It has completely serial control of execution: only one ILA is in control of the robot at a time. It collects all contingent knowledge about the world in a single shared workspace. It handles error recovery in two ways: re-sensing leading to model updating and re-planning, and closed loop recovery from errors without planning. Finally it does not provide an architectural answer to the problem of sensory interpretation: perceptual routines existed in Shakey, but there are no architectural constraints or aids to how they operate, communicate, or share information. They serve only to provide information to a central model in a unified language. The classic story about Shakey given by behaviour based roboticists, is that it could never have worked outside of its carefully controlled environment, and that even within it performance was unreliable. Shakey was able to perform different tasks, but it relied upon an accurate world model. It was able to construct a sufficiently accurate representation under benign sensory conditions, but robot vision researchers unsuccessfully spent the decade following the Shakey project trying to extend this approach of scene reconstruction to more natural visual environments. Thirty-five years later our ability to perform scene or surface reconstruction is still poor, although it has improved. Of course to be fair to the designers of Shakey it is not clear that they took a principled stance on whether all aspects of a scene should be recovered, only that attempts to extend their approach often sought to do this, and have largely failed to date. A strong reaction to this paradigm that occurred in the mid 1980s was exemplified by the behaviour based approach to robotics [5]. The approach is characterised by a number of authors including proponents and sceptics. One key idea is that the system is almost entirely representation free. The meaning of this statement depends on what is meant by representation. Kirsch [14] describes three types of representation: data items the values of which co-vary with features in the world; declarative statements which release their information when queried; and predicate descriptions that allow generalisation by type, leading to the ability to reason about inheritance. Brooks' early robots were certainly representation free on the basis of either of the last two definitions. Furthermore, while modules may have representations of the first kind, they do not typically transmit these representations to other modules for further processing or consumption. In other words there is no sharing of information, only competition for control of the robot. Other important aspects of the approach are that many controllers run in parallel, that each is relatively simple, and that their action recommendations are fused through a single global mechanism. The obvious weakness of the behaviour based approach is the lack of evidence of its ability to scale to higher cognitive functions, despite nearly a quarter of a century of effort. Instead, roboticists typically use behaviours as the lowest level of control in a hierarchical system. Three tiered architectures such as 3T [7] employ behaviours at the lowest level, and link these to a symbolic planning level via a sequencing level in which transitions are made from behaviour to behaviour using a finite state machine like representation. Representations have made a re-appearance via advances in filtering and statistical approaches imported from machine learning. Behaviours, rather than truly behaviour based approaches have thus been merged into the tool-box of techniques employed by most roboticists within architectures, rather than being an architectural choice in their own right.

7 Summary of contributions and conclusions

In this chapter we have explored a small part of the space of designs for a particular part of niche space. Working from run and design time requirements imposed by our combination of task and environment (our two scenarios) we have suggested one architectural schema (CAS). We have argued that CAS includes a large number of architectural instantiations and sub-schemas that meet those requirements. From our experience of building real robot systems using the software implementation of the schema (CAST) we have identified four problems which are common to a very large range of systems, and which we argue warrant architectural solutions. These are the problems of *binding*, *filtering*, *processing management* and *action fusion*. We have gone on to detail our work within CAST to create solutions to each of these problems. Finally we have tried to show that an empirical science of architectures is possible. The work in this aspect of CoSy has been exceptionally useful in allowing us to integrate the work of many of the other chapters. In particular we believe that any true systems approach to AI must include an architectural theory. We believe that CAS represents a significant step forward in architectures for embodied cognitive systems because of the way that it combines the parallelism and incrementality of behaviour based systems with the use of representations that should lie at the heart of any cognitive system.

References

1. J. Anderson, D. Bothell, M. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, 2004.
2. Ronald C. Arkin. *Behavior-Based Robotics. Intelligent robots and autonomous agents.* MIT Press, 1998.
3. M. Brenner, N. Hawes, J. Kelleher, and J. Wyatt. Mediating between qualitative and quantitative representations for task-orientated human-robot interaction. In *Proc. of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI)*, Hyderabad, India, 2007.
4. M. Brenner and B. Nebel. Continual Planning and Acting in Dynamic Multiagent Environments. In *The International PCAR Symposium*, 2006.
5. Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, (47):139–159, 1991.
6. R. Clouard, A. Elmoataz, C. Porquet, and M. Revenu. Borg: A knowledge-based system for automatic generation of image processing programs. *PAMI*, 21, 1999.
7. E. Gat. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*. 1997.
8. Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42:335–346, 1990.
9. Nick Hawes, Aaron Sloman, and Jeremy Wyatt. Towards an empirical exploration of design space. In *Proc. of the 2007 AAAI Workshop on Evaluating Architectures for Intelligence*, Vancouver, Canada, 2007. To appear.
10. Nick Hawes, Aaron Sloman, Jeremy Wyatt, Michael Zillich, Henrik Jacobsson, Geert-Jan M. Kruiff, Michael Brenner, Gregor Berginc, and Danijel Skocaj. Towards an Integrated Robot with Multiple Cognitive Functions. In *The Twenty-second National Conference on Artificial Intelligence (AAAI)*, 2007.
11. Jorg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
12. Ian Horswill. Polly: A Vision-Based Artificial Agent. In *AAAI*, pages 824–829, 1993.

13. Henrik Jacobsson, Nick Hawes, Geert-Jan Kruijff, and Jeremy Wyatt. Cross-modal content binding in information-processing architectures. In *HRI '08: Proceedings of the 3rd ACM/IEEE international conference on Human robot interaction*, pages 81–88, New York, NY, USA, 2008. ACM.
14. D. Kirsch. Today the earwig, tomorrow man? *Artificial Intelligence*, 47:161–184, 1991.
15. Geert-Jan M. Kruijff, John D. Kelleher, and Nick Hawes. Information fusion for visual reference resolution in dynamic situated dialogue. In Elisabeth Andre, Laila Dybkjaer, Wolfgang Minker, Heiko Neumann, and Michael Weber, editors, *Perception and Interactive Technologies: International Tutorial and Research Workshop, PIT 2006*, volume 4021 of *Lecture Notes in Computer Science*, pages 117 – 128, Kloster Irsee, Germany, June 2006. Springer Berlin / Heidelberg.
16. J. E. Laird, A. Newell, and P. S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(3):1–64, 1987.
17. M. F. Land and M. Hayhoe. In What Ways do Eye Movements Contribute to Everyday Activities. *Vision Research*, 41:3559–3565, 2001.
18. D. N. Lee. The Optical Flow-field: The Foundation of Vision. *Philosophical Transactions of the Royal Society London B*, 290:169–179, 1980.
19. L. Li, V. Bulitko, R. Greiner, and I. Levner. Improving an Adaptive Image Interpretation System by Leveraging. In *Australian and New Zealand Conference on Intelligent Information Systems*, 2003.
20. D. McDermott. PDDL: The Planning Domain Definition Language, Technical Report TR-98-003/DCS TR-1165. Technical report, Yale Center for Computational Vision and Control, 1998.
21. Marvin Minsky, Push Singh, and Aaron Sloman. The st. thomas common sense symposium: Designing architectures for human-level intelligence. *AI Magazine*, Summer 2004, 2004.
22. S. Moisan. Program supervision: Yakl and pegase+ reference and user manual. Rapport de Recherche 5066, INRIA, Sophia Antipolis, France, December 2003.
23. N Nilsson. Shakey the robot. Technical Report Tech Note 323, AI Center, SRI International, April 1984.
24. M. P. Shanahan and B. J. Baars. Applying global workspace theory to the frame problem. *Cognition*, 98(2):157–176, 2005.
25. D. Skočaj, G. Berginc, B. Ridge, A. Štimec, M. Jogan, O. Vanek, A. Leonardis, M. Hutter, and N. Hawes. A system for continuous learning of visual concepts. In *International Conference on Computer Vision Systems ICVS 2007*, Bielefeld, Germany, 2007.
26. Aaron Sloman and Matthias Scheutz. A framework for comparing agent architectures. In *In UK Workshop on Computational Intelligence*, pages 169–176, 2002.
27. Mohan Sridharan, Jeremy Wyatt, and Richard Dearden. HiPPo: Hierarchical POMDPs for Planning Information Processing and Sensi ng Actions on a Robot. In *Proceedings of the 18th International Conference on Planning and Sch eduling, ICPAS-2008*, 2008.
28. M. Thonnat and S. Moisan. What can program supervision do for program reuse? *IEE Proc. Software*, 2000.
29. Frank van der Velde and Marc de Kamps. Neural blackboard architectures of combinatorial structures in cognition. *Behavioral and Brain Sciences*, 29:37–70, 2006.

30. J.M. Wolfe and K.R. Cave. The psychophysical evidence for a binding problem in human vision. *Neuron*, 24(1):11–17, 1999.