Engineering Intelligent Information-Processing Systems with CAST

Nick Hawes and Jeremy Wyatt

Intelligent Robotics Lab, School of Computer Science, University of Birmingham, Birmingham, UK

Abstract

The CoSy Architecture Schema Toolkit (CAST) is a new software toolkit, and related processing paradigm, which supports the construction and exploration of information-processing architectures for intelligent systems such as robots. CAST eschews the standard point-to-point connectivity of traditional message-based software toolkits for robots, instead supporting the parallel refinement of representations on shared working memories. In this article we focus on the engineering-related aspects of CAST, including the challenges that had to be overcome in its creation, and how it allow us to design and build novel intelligent systems in flexible ways. We support our arguments with example drawn from recent engineering efforts dedicated to building two intelligent systems with similar architectures: the PlayMate system for tabletop manipulation and the Explorer system for human-augmented mapping.

Key words: intelligent robotics, artificial intelligence, information-processing architectures, middleware

1. Introduction

The ultimate aim of many current projects in the field of cognitive or intelligent robotics is the development of a robotic system that integrates multiple heterogeneous subsystems to demonstrate intelligent behaviour in a limited domain (home help, guided tours etc.). Typically the focus in this work is in developing state-of-the-art components and subsystems to solve an isolated sub-problem in this domain (e.g. recognising categories of objects, or mapping a building). In the CoSy[1] and CogX[2] projects, whilst being interested in state of the art subsystems, we are also motivated by the problems of integrating these subsystems into a single intelligent system. We wish to tackle the twin problems of designing information-processing architectures that integrate subsystems in a principled manner, and implementing these designs in robot systems. The alternative to explicitly addressing these issues is ad-hoc theoretical and software integration that sheds no light on one of the most frequently overlooked problems in AI and robotics: understanding the trade-offs available in the design space of intelligent systems [30].

The desire to tackle architectural and integration issues in a principled manner has led us to develop the CoSy Architecture Schema Toolkit (CAST) [21]. This is a software toolkit intended to support the design, implementation and exploration of information-processing architectures for intelligent robots and other systems. The design of CAST was driven by a set of requirements inspired by human-robot interaction (HRI) domains and the need to explore the design space of architectures for systems for these domains. This has led to a design based around a particular computational paradigm: multiple shared working memories.

2. Background and Motivation

A common approach to designing and building an intelligent system to perform a particular task follows this pattern: analyse the problem to determine the sub-problems that need to be solved, develop new (or obtain existing) technologies to solve these sub-problems, put all the technologies together in a single system, then demonstrate the system performing the original task. This "look ma no hands" approach to intelligent system design has a number of problems, but we will focus on the "put all the technologies together" step. If the number of component technologies is small, and the interactions between them are strictly limited, then arbitrarily connecting components may suffice. However, once a certain level of sophistication has been reached (we would argue that once you integrate three or more sensory modalities in an intelligent robot, or would like your system to be extensible, you have reached this level), then this approach lacks the foresight necessary to develop a good system design.

To counter this approach, we follow a *design-based methodology* (cf. Dennett's design stance [12]). In this methodology any work should start from a set of *scenarios* which serve to define the desired behaviour, or *niche*, of the target system (e.g. an intelligent robot). These scenarios should be analysed to determine the *requirements* they place on the design and implementation of the system. For a discussion of the relationship between design space (i.e. the space of possible designs) and nice space see [30]. Given a list of requirements for the target

Email address: `{nah,jlw}@cs.bham.ac.uk` (Nick Hawes and Jeremy Wyatt)

URL: `http://www.cs.bham.ac.uk/go/irlab` (Nick Hawes and Jeremy Wyatt)

[1]Cognitive Systems for Cognitive Assistants: http://cognitivesystems.org

[2]Cognitive Systems that Self-Understand and Self-Extend: http://cogx.eu

system, the designer must then produce a design which satisfies these. The target system can then be implemented from the design, yielding an artifact capable of demonstrating behaviour that can be compared to the initial requirements. By iterating through this procedure, and performing suitable analysis at each step, we argue that the system development procedure is more likely to produce a system with the required behaviour, and is less likely to be influenced by external factors (e.g. already having a particular implementation of a component that must be included, or using a certain legacy representation)

This methodology represents an ideal for AI research in general. Whilst it has been successfully used in small projects where a single person has control over the whole process (e.g. Ph.D projects [31, 16]), applying it in large or multi-site projects (such as EU Integrated Projects) has proven difficult. There are many reasons for such difficulties, including the need to encompass many different research agendas in a single project, and long development cycles which leave little opportunity for evaluation before the next iteration of the cycle begins. However, we must also consider the underspecification of the methodology itself as a factor that exacerbates this problem. For example, how much detail of what kind should be included in a scenario, what level of description should a requirement cover (the consequences of a behaviour, the behaviour itself, its design or even how it's implemented), and how can a design be synthesised from a set of requirements in a systematic fashion? All of these are open questions in the science of engineering intelligent systems.

In this article we will present our solution to one part of this problem. In brief, we are concerned with the steps that take the methodology from a design to a functioning artifact. In terms of a scientific understanding of intelligent systems it is an artifact's design that we are primarily concerned with. This is because a design can represent some generalisation of functionality that can be reused in many situations, rather than a single, task- or platform-specific, implementation. As we are concerned with the designs of whole systems (rather than a single element of a system), we shall focus on *architecture* designs.

In the field of intelligent artifacts, the term "architecture" has been used to refer to many different, yet closely related, aspects of a system's design and implementation. Underlying all of these notions is the idea of a collection of units of functionality, information (whether implicitly or explicitly represented) and methods for bringing these together. In typical intelligent robotics applications it is possible to separate at least two levels of architecture: an information-processing architecture (alternatively referred to as a functional or computational architecture, or a cognitive model) and a software architecture. For example, Spartacus (a robot that attended the AAAI 2005 conference) [27] can be said to have architecture designs at three levels:

- An information-processing architecture (referred to as a "computational architecture" in the paper) called the Motivated Behavioral Architecture (MBA). This design specifies six high-level modules in the system (motivations, dynamic task workspace, behaviour producing modules (BPMs), two BPM processing modules, and system know-how). This design provides a coarse subdivision of the system, allowing particular bits of functionality to be placed in the correct module. This level of architecture says nothing about how the system can solve a particular instance of a problem (e.g. getting from A to B, or building a map), but it provides a framework in which any solution must be couched.

- Within Spartacus there is also an *instantiated* information-processing architecture[3]. This is the architecture MBA, but tailored to the specific problems Spartacus has to solve. At this level the various components have task-specific content, e.g. Spartacus has motivations to plan, survive and interact, and BPMs for docking and avoiding. At this level the architecture describes how the robot should be able to perform any required task using the MBA architecture.

- The final level presented in [27] is Spartacus's software architecture. This level shows how the instantiated information-processing architecture is actually implemented in the component-based software framework MARIE [10]. At this level the design specifies precisely how the robot architecture is decomposed into separate processing units, and how these units are connected.

Given these three levels of architectural description, we must ask which level we should care about when working through our design-based methodology. If our study is intended to produce general-purpose results, then we must focus on level of the information-processing architecture (the most abstract of the three presented levels). This is because it is this level that provides general architectural insights that can be reused across systems. However, if we wish to empirically evaluate our information-processing architectures using our artifacts as exemplars of the theory they represent, then we can only explore designs at the level of the software architecture, as this is all that is present in the final systems. It is this final point that has proved a stumbling block in the systematic study of architectures for intelligent systems. There is often a major disconnect between a system's information-processing architecture and its software architecture. This is the case in the Spartacus system, where the software architecture, whilst containing many of the elements of the instantiated information-processing architecture, is not isomorphic with its higher-level counterparts. It is only possible to use Spartacus as an example in this case because its developers have been thorough enough to separate out the three layers described above. Many other systems are developed with reference to only one of these three layers[4].

Any disconnect between information-processing architecture design and software architecture design poses a problem for our

[3]This is not a term used in [27], but one we are using to refer to the addition of task-specific design content to an information-processing architecture.

[4]Often this is enough for single projects intended to demonstrate one instance of functionality.

design-based methology. If we cannot evaluate our final systems as instantiations (or implementations) of an abstract theory, we cannot argue that we are exploring design space in a systematic fashion; our implementations are actually instances drawn from theories which differ from those we have chosen to explore. Although these differences may be small in some cases, the point is still important: for us to be seriously contributing to a science of engineering intelligent systems we must be able to study the same designs in our systems that we study on paper.

We have spent the last four years exploring architecture designs in integrated robot systems produced collaboratively by a number of universities. To allow us to make progress towards deploying our design-based methodology in this context, we have taken the following steps:

- Following a study the requirements of a number of robotic scenarios, we produced the **CoSy Architecture Schema** (CAS) [19]. This is a schema, or a set of rules, for constraining the design of information-processing architectures.
- From this we produced the **CoSy Architecture Schema Toolkit** (CAST), a middleware toolkit that allows us to directly implement CAS instantiations without resorting to an additional (and possibly disconnected) software architecture [21].
- Within the constraints of CAS we designed an information-processing architecture for a set of requirements drawn from two scenarios (table-top manipulation and human-augmented mapping) [17].
- This information-processing architecture was then implemented in CAST and used in two different robotic systems (one for each of the two scenarios). Whilst these different systems had additional task-specific architectural elements, they both used the same basic information-processing architectural design and implementation.

In this article we focus on the novel engineering properties of the CoSy Architecture Schema Toolkit in the context of the science of building intelligent systems. In Section 3 we present the design of CAS in more detail. In Section 4 we present a technical description of the toolkit based on this design, and then in Section 5, using examples of our design and engineering practises, demonstrate how it has allowed us to design and build intelligent robots with novel properties in novel ways. Finally, in Section 6 we evaluate the contributions of CAST to the process of engineering intelligent systems, and in Section 7 we discuss how our work compares with existing frameworks.

3. The CoSy Architecture Schema

Before we can present the toolkit we must first present the schema it implements. The CoSy Architecture Schema is a design that constructs a single architecture from a collection of *subarchitectures*, where a subarchitecture collects together some related subset of processes of the whole system (see Figure 1). Each subarchitecture contains any number of concurrently active *processing components*. These components can represent functionality at any level of granularity (from a single rule or neuron, to a parser, planner or object recognition system), but are typically considered to be comparable to components in any standard component-based architecture. The schema does not allow these components to communicate directly. Instead they can only exchange information via a shared *working memory* present in each subarchitecture. The schema allows for restrictions to be placed on which components can access which working memories. By default a component can read information from any working memory in the architecture, but can only write to the working memory in its own subarchitecture. In practise this default will often be overridden to exchange information and control signals between subarchitectures. The only additional structure specified by the schema is that each subarchitecture contains a *task manager*, which can specify when processing components are allowed to be active. Components are therefore either *managed* (if the task manager can influence their behaviour) or *unmanaged* (if it cannot). Typically unmanaged components are used for frequent, cheap processing, whereas managed components are used for on-demand, expensive processing that could occupy some of the system's limited resources.

In terms of design CAS falls between two traditional camps of architecture research. On one hand its shared working memories draw influence from cognitive modelling architectures such as ACT-R [1], Soar [25], and Global Workspace Theory [28]. On the other hand it grounds these influences in a distributed, concurrent system which shares properties with robotic architectures such as 3T [5]. Systems built within the CAS schema typically function like distributed blackboard systems (e.g. [13]). Further comparisons to existing work are presented in Section 7.

To support the following discussion we will quickly introduce an example subarchitecture for a visual subsystem of a robot. This example is a simplified version of the visual subarchitecture presented in [18]. In this subarchitecture we will have a video server which sits outside of the subarchitecture (see the "Input" arrows in Figure 1), an unmanaged change detector component, a managed segmentor component and a variable number of managed components that can perform visual processing tasks. The change detector constantly pulls images from the video server, and overwrites a `SceneChanged`[5] struct on working memory when the camera image has changed significantly then become static again (e.g. when an object has been placed in front of the robot). The segmentor waits for this struct then pulls an image from the video server and runs a segmentation algorithm (e.g. some kind of background subtraction) on it. This results in a number of `ROI` structs, indicating regions of interest (ROIs) where the scene is significantly different from the background (i.e. regions where objects might be found). Following this, the additional managed components

[5]In this article, class and method names that you can find directly in our code will be presented in this font.

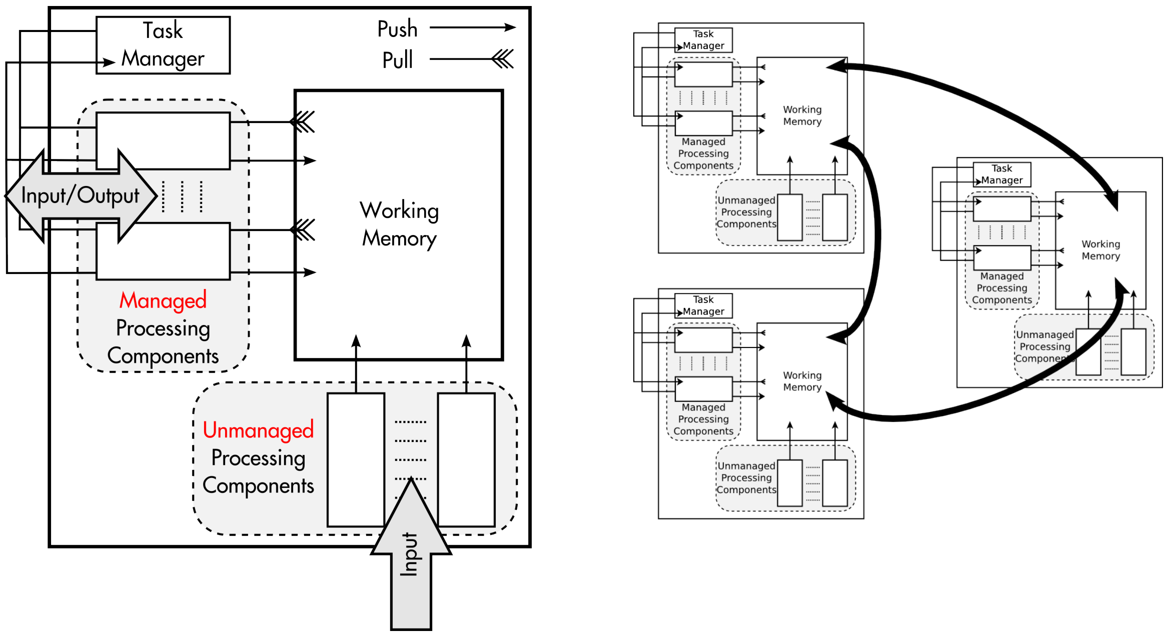
Task Manager
Push
Pull
Input/Output
Working Memory
Managed Processing Components
Unmanaged Processing Components
Input
Task Manager
Working Memory
Managed Processing Components
Unmanaged Processing Components

Figure 1: Two views of the CoSy Architecture Schema. The figure on the left is the schema at the level of a single subarchitecture. The figure on the right shows how a system is built from a number of these subarchitectures. Note that all inter-subarchitecture communication occurs only via connections between working memories.

process these structs in parallel, performing various feature extraction, recognition and classification tasks. The end result of this processing should be that the working memory contains the robot's best hypothesis of what it sees in front of it.

4. The CoSy Architecture Schema Toolkit

The proceeding section provided a very high-level overview of the schema. In this section we will look at how the schema is translated into software as the CoSy Architecture Schema Toolkit. We have previously presented an initial look at CAST [21], but here we will go much deeper into the functionality it provides, and the engineering issues connected to this functionality.

At the highest level of abstraction CAST is a component-based software framework with abstract classes for managed and unmanaged components, task managers and working memories. By sub-classing these components, system builders can quickly and easily create new architectures that instantiate the CAS schema. CAST is built on top of a component-based communication toolkit called BALT (the Boxes and Lines Toolkit). BALT uses CORBA to provide typed, cross-language (C++ and Java) push and pull connections between components located on the same machine or across a network. All the functionality in CAST builds on these connections to be neutral of components' language and location. Thus in the following discussions we will ignore these properties.

4.1. Core Functionality

Before discussing the technical contributions of CAST, we will first provide an overview of its core functionality.

4.1.1. Working Memory Contents

In CAST the working memory is central to all processing, and will therefore occupy much of our presentation. A CAST working memory (WM) is an associative container (much like a hashtable) that maps between working memory *addresses* (WMAs) and working memory *entries*. A WMA is a pair containing a subarchitecture string (which uniquely identifies the subarchitecture containing the working memory) and an identifier string (which uniquely identifies the entry within the working memory). A working memory entry is a templated class that contains the entry object itself, plus information about the object's address, type, and version. The object it contains can be an instance of any struct that can be described in CORBA's interface definition language (IDL). In effect this is any struct that can be written in C++ or Java without using inheritance. Defined in this manner a CAST WM is a way of sharing arbitrary objects across languages and machines via a single interface. From the example presented in Section 3 the visual working memory would contain entries which contain instances of structs like `SceneChanged` and `ROI`.

4.1.2. Working Memory Write Access

Working memory entries are inserted into working memories by the processing components in the system. The initial interface for doing this is the `addToWorkingMemory(_id, _subarch, _object)` command. This command allows any processing component to write an object to a working memory in any subarchitecture. Once the object exists in working memory there are similarly parametrised `overwriteWorkingMemory` and `deleteFromWorkingMemory` commands to alter and delete the stored object respectively. From the example presented in Section 3 the change detector would initially use `addToWorkingMemory` to write an instance of `SceneChanged` to the visual working memory. It would then use `overwriteWorkingMemory` to update this as the scene changes. The segmentor would use `addToWorkingMemory` to add `ROI`s to working memory every time the scene has changed, and the other components would use `overwriteWorkingMemory` to update these `ROI`s with features, recognition results and other bits of visually important information.

As the component-based nature of CAST makes it inherently concurrent, there is a chance that a component performing any of these commands could encounter an error due to another component operating on WM content in parallel. At the WM level, all operations (reads and writes) are sequenced so that no two operations can overlap[6]. However, beyond the WM level component operations are not explicitly synchronised, so it could be possible that component A deletes an entry before component B tries to overwrite it. In error cases like this an exception is thrown in the latter component. This is the error handling model employed by CAST in general. Because the *semantics* of working memory operations (i.e. the role they play in information processing) cannot be second-guessed by the framework, it does not attempt to correct errors at a lower level (e.g. by allowing the overwrite above to be interpreted as an add). Instead an informative exception is produced, and the system designer is then expected to correct the design of the interacting components, or otherwise handle the situation. This is also the case for designer-enforced restrictions on which subarchitectures a component can access. If the access conditions are violated at run-time (which is determined by checking the `_subarch` argument to the write command), then an exception is thrown by the operation.

4.1.3. Working Memory Read Access

Once an entry exists on working memory it is available for reading by other components. Components can retrieve entries from working memory using two access modes: address access and type access. For address access the component provides the entry's WMA and then retrieves the entry associated with it. For type access the component specifies an entry type (e.g. `ROI`) and retrieves a vector containing all of the entries on working memory that are instances of this type. When an entry is read, it is returned along with all of the information (address etc.) it was associated with on WM.

4.1.4. Working Memory Change Notification

Whilst the two previously described retrieval modes provide the basic mechanisms for accessing the contents of working memory, they require information about what is already on working memory (e.g. that an entry of a particular type exists at a particular address) in order to be deployed successfully. This is where the notion of a CAST *change event* becomes important. When any write operation is performed by a component, the working memory that has been operated on generates a change event. This event is a struct that describes the operation that has just been performed. Change events contain the address and type of the changed entry, the name of the component that made the change, and the operation performed to create the change (i.e. whether the entry was added, overwritten or deleted).

To access these change events a component must register *change filters* that can match against arbitrary combinations of fields from the change event. Each filter is associated with a callback function that is called with a change event struct whenever the filter matches an event. Using this mechanism, most component-level processing in CAST can be characterised by a model in which a component waits until a particular change has occurred to an entry on working memory, then retrieves this entry and processes it (or related entries) in some way before writing back to working memory. This creates synchronous processing within an inherently asynchronous system.

From the previous example, the segmentor will register a change filter to listen for change events referring to an `OVERWRITE` of a `SceneChanged` struct. When a matching event triggers this filter, the segmentor will perform the segmentation and write out the appropriate `ROI`s. The rest of the processing components will have change filters listening for change events referring to an `ADD` of a `ROI` struct so that they can process the regions once they're written to working memory.

The change event is perhaps the most fundamental aspect of CAST in terms of design and operation. These events provide the only general-purpose mechanism in CAST for informing components about the processing being carried out elsewhere in an architecture. Events and callbacks provide flexibility in the composition of an architecture. If a particular change does not occur at run-time (if the component that causes it is not present in the system for example) then components waiting on this change will not produce errors, they will just not be triggered. This is much like the flexibility provided by a service-based or publish-subscribe paradigm. The role of change events and filters at an information-processing level is discussed in Section 4.2.3.

4.1.5. Architecture Configuration

As our ultimate motivation is to use CAS to explore a limited region of design space, it is important that CAST allows us to quickly produce different systems from a set of components. This is achieved via a text-based configuration interface. To run a CAST system the designer must provide a text file describing the structure of the system. This text file is divided into sections that each describe a subarchitecture. Each subarchitecture section contains a list of components (including working memory and task manager instances). Each component line describes the language the component is written in, the hostname of the machine which will run the component, the classname of the component, and the instance name of the component. In addition to this, each component can be given configuration options

[6] This is a conservative strategy. A more complex strategy which only serialises operations based on the content (e.g. WMA or type) could be employed as a later optimisation.

that are passed to the component object at run time. Producing different architectures at run-time is as simple as providing a different text file. This allows you to run subsets of components or subarchitectures, or change the relationship between components and working memories, without changing a line of code.

4.2. Advanced Functionality

Below the surface of the core functionality, CAST has a number of interesting properties that relate to both the schema design and the purpose to which it is put.

4.2.1. Working Memory Consistency

Although we have already discussed the problem of ensuring invalid working memory operations are not allowed (see Section 4.1.2), this is only one type of inconsistency that can occur in a shared memory system such as CAST. Another inconsistency that can occur is when component A reads, processes and then overwrites a working memory entry, only for the entry to have changed by process B between the read and write. In this case the intermediate changes to the entry by process B will be lost when process A overwrites it. In the example from Section 3 if there are two components waiting for the addition of `ROI`s then processing and overwriting them, the component that overwrites a `ROI` first will have its changes overwritten when the second component performs a subsequent overwrite on the same entry.

This type of consistency problem has been encountered previously in the study of distributed shared memory (DSM) systems. Although these systems typically operate at a lower level of abstraction (maintaining the consistency of variables within distributed processes) the fundamental problem of maintaining memory consistency is the same. The DSM literature describes a number of different approaches to maintaining memory consistency. The one we have chosen to use is the "invalidate on write" model [11]. Using this approach, every working memory entry is assigned a version number which is incremented on each operation. Components maintain a list of the version numbers of the working memory addresses they have read and written, and optimistically assume that they have the most recent version of the working memory entry until they are informed otherwise. Version checking is only performed when memory is overwritten. At this point the component checks whether the version number for the data it is about to overwrite is the same locally as it is on working memory. If this is not true then the overwrite will violate working memory consistency, so an exception is thrown from the overwrite method.

We chose this approach because, although it allows components to process entries which are out of date (therefore wasting resources), it is cheap to implement and easy to understand. Other methods which attempt to merge or manage inconsistent changes require a much more complex management structure behind the scenes (including communication about variable states between components). The invalidate on write model also fits into the CAST design philosophy for error handling: errors are quickly reported, but then should be fixed by redesigning the component interactions (rather than through extensive exception handling).

4.2.2. Locking

Whilst the event-driven programming style encouraged by CAST allows many processing approaches to be implemented with little overhead, not all approaches fit into this scheme. In particular any processing that requires explicit synchronisation between components is hard to tackle just using the contents of working memory and change events. Given the example where two components both process a WM entry then overwrite it, we require some method to allow a component to claim ownership of a working memory entry until it has finished processing it.

There are actually two related issues here: how do you design a subarchitecture so that components can operate in parallel on shared data in a way that preserves all changes, and what software-level tools are necessary to support this design. The former question requires a much more general discussion than this article intends to present, so we will focus on the latter. After examining the way users design component interactions with CAST, we decided to provide support for the locking of working memory entries by components. Our approach combines two frequently-related aspects of memory access: entry ownership and access permission. For entry ownership we effectively provide a *mutex* lock for each working memory entry. A lock is obtained by issuing a `lockEntry(_id, _subarch, _permissions)` command. Only one component can hold the lock for an entry at any time, so the lock command blocks until the entry is free to lock (e.g. when another component issues the complementary `unlockEntry` command). As with POSIX mutexes, a `trylockEntry` command is also available. For access permissions we provide a three level hierarchy denying the following access modes: overwrite; overwrite and delete; and overwrite, delete and read[7]. These permission settings allow components to protect the working memory entries they have locked from interference by other components. If a component attempts to violate an overwrite or delete restriction an exception is thrown from the appropriate access method. If a read restriction is violated, the calling component blocks at the read command. This provides a method for introducing implicit synchronisation into components that do not use locking.

In the visual subarchitecture example, locking could be deployed as follows. Before any visual component attempts to read and process a `ROI` it must first obtain a lock on it, blocking read access for other components (i.e. the strictest lock available). Once a component has the lock it can then read, process then overwrite the `ROI`, before unlocking it. If all processing components obey this protocol, then their parallel processing becomes serialised and working memory consistency is maintained. If a component doesn't obey the protocol, then the read lock will prevent it accessing the `ROI` while the other components are processing it. Although this approach is safe, it is also inefficient if the components can run their processing in paral-

[7]This is inspired by C++ const variables.

lel, but only need to serialise their final overwrites. In this case, a more fine-grained protocol is required.

By providing this functionality CAST satisfies two processing needs. By default component-based processing occurs in parallel. This means components are free to react to asynchronous changes in information (either on working memory or obtained via sensors) and to act independently, without waiting for other components to run (which would be the case if components ran in serial or were time-sliced in some way). By adding the locking mechanism, CAST caters for conditions where dependencies between components, made explicit via shared information, are necessary. In such conditions the ability to block within an algorithm until a particular condition occurs is a powerful tool. Without a locking mechanism to support this, component designers would have to resort to waiting on separate change-triggered callbacks for these conditions. Such an approach would be both error-prone (as synchronisation code is distributed further across the component code), and unintuitive.

4.2.3. Filtering

Although we can consider the change event and filtering system on a purely technical level, it is informative to consider the relationship between the implementation and its purpose. CAST is a tool for building information-processing architectures for intelligent systems. Change events represent one possible mechanism for allowing one part of such an architecture to track what information is actually available to it. As we have argued elsewhere [32], this is a fundamental problem in information-processing architectures in general.

By posing the problem in terms of events and filters, we can examine the costs and benefits of performing filtering in different ways, and how changes in a system's architecture influence these costs and benefits. A change event impacts upon system performance in two ways: communication overhead and filtering overhead. When a change occurs on a working memory, that working memory broadcasts the change to all the components within its subarchitecture. It then broadcasts the event to all of the other working memories in the system, which in turn re-broadcast the event to all of the components in their subarchitectures. Each time a single event is broadcast it uses up some system resources (at least communication bandwidth and time), and so when events are broadcast unnecessarily (i.e. they are sent to subarchitectures and components that don't require them) we can assume that this has a negative impact on the performance of the system. This is the communication overhead of a change event. When a component only needs a subset of all the events it receives, it must expend some resources (at least computational effort and time) to select (i.e. filter out) these events from the stream of events it receives. This the filtering overhead of a change event. These two types of overhead are closely related: if you reduce the number of unnecessary events a component receives, you will minimise both overheads.

In CAST it is important for us to minimise the number of unnecessary change events sent to components as the effort of broadcasting and processing them can make a system slow and unresponsive to important changes (in the case when the receipt of an important change event is delayed by the processing of other events). In early versions of CAST we required every component to locally filter all of the changes generated and broadcast by all the of the other working memories in the system. This approach was inefficient and did not scale well to systems with many components, or with just one very active component. To overcome this problem we now employ *distributed filtering*. When a component creates a change filter, it also informs its local working memory of the change conditions the filter matches against. The working memory then broadcasts this information to the other working memories in the system. In this manner each working memory collects information about which components are listening for which events via which working memories. Every time a working memory is asked to forward a change event to another working memory, it checks the event against the collected filter information, and only sends the event to subarchitectures which contain a component that is filtering for it. By distributing filtering information across a system in this manner an increase in processing cost (the maintenance and checking of filters) and communication cost (sending filter information from components) at the working memory level has greatly decreased the same costs at a component level.

Distributed filtering demonstrates one of the benefits of a modular system design. In CAST, modules (i.e. subarchitectures) provide cues for filtering information based on system structure, and a centralised communication point in each module (i.e. the working memory) provides the easiest place to implement this kind of filtering. We have used CAST to explore how the subdivision of a system into modules of various sizes affects communication and filtering overhead [20]. In this work we looked at these measures in systems with n components divided into either 1 subarchitecture, n subarchitectures or m subarchitectures, where $1 < m < n$. This work would not have been possible without CAST, as no other tool both links software and information-processing architectures, and provides the flexibility to rearrange the architecture design without altering the system in ways that might make comparisons between systems invalid.

5. Working with CAST

To demonstrate both the kinds of systems you can build with CAST, and how they can be built, we will now present a case study of the use of CAST to develop two robotic systems using an overlapping design and codebase. In the CoSy project we grounded our studies in two demonstrator scenarios: the PlayMate scenario and the Explorer scenario. The PlayMate scenario describes a stationary robot with a manipulator that can interact with a human over a collection of objects on a tabletop. The PlayMate robot can move these objects about in response to commands from a human (including game playing instructions), and can answer questions about their visual and spatial properties. The Explorer scenario describes a mobile robot that can explore and map an indoor environment in collaboration with a human. The Explorer robot can perform simple movement and exploration tasks, and can answer questions about the objects and rooms in its map. Whilst these scenarios provide separate challenges for sensing and action (including behaviours with different spatial and temporal extents), they have similar requirements for planning, decision making, and human-robot interaction. These overlapping requirements entail an overlapping design and development strategy. Initially we developed the PlayMate system (bearing in mind the Explorer requirements), then reused the core architecture in the Explorer system. This order dictates the order in which we will present the systems.

5.1. The PlayMate System

Our approach to designing intelligent architectures with CAS is to adopt a functional decomposition, where each major sensory or processing modality (e.g. vision, language, planning etc.) has a dedicated subarchitecture. Whilst functional decompositions are not appropriate for all scenarios (cf. behaviour-based robotics [9]), the level of detail of the representations we focus on (representations of objects, scenes, utterances, etc.) are well suited to processing in a functional manner. The CAS schema is well-suited to functional decompositions: a subarchitecture working memory can be dedicated to a particular form of representation, and all the components within that subarchitecture are designed to process that type of representation. The filtering restrictions imposed at subarchitecture boundaries mean that this basic design incurs a low overhead in terms of filtering and communication costs (see Section 4.2.3). This is because grouping processes together with the representations they process means that change events do not have to be broadcast throughout the architecture [20].

5.1.1. System Input & Output

The PlayMate robot is iRobot B21r with a Neuronics Katana HD6M arm mounted on its front. Processing for the PlayMate is distributed across multiple (usually between 2 and 4) machines of various specifications via a wireless network. In the PlayMate we have four subarchitectures that are dedicated to representations coupled to sensors or effectors: vision, qualitative spatial reasoning (QSR), communication and manipulation. Each of these subarchitectures contains task-specific representations which are optimised (at design-time) for the kinds of processing necessary for problem solving in that modality. For example, the communication subarchitecture working memory contains instances of packed logical forms which efficiently represent multiple alternative interpretations of input utterances [24], whilst the vision subarchitecture working memory contains ROIs from an image (including the bitmap of the region), features extracted from the regions, and 3D object models.

Both the vision and communication subarchitecture draw on CAST functionality to process information via refinement of shared data structures. The vision subarchitecture operates much like the example presented in Section 3: change detection and segmentation produce region-of-interest and object representations, and these representations are then processed in parallel by multiple components which add information (e.g. edges, visual features, object type hypotheses etc.) into these shared structures. Some of this information represents the final results of a chain of processing (e.g. object type hypotheses), whilst other information (e.g. edges, visual features) represents intermediate steps in such chains. The communication subarchitecture formalises this type of approach by basing a multi-level processing technique on shared data structures [24]. As the communication subarchitecture processes an utterance in a left-to-right manner, its components (an OpenCCG-based parser, and components for reference binding, dialogue move interpretation, dialogue structure management and linguistic saliency), a single representation (the packaged logical form described previously) is incrementally extended "horizontally" in a similar fashion. At each step (i.e. after each word) this logical form is then also extended "vertically" by components that are influenced by both the previous iterations of processing, and by the information available in the rest of the system (e.g. the visual scene or the current action plan). This extension stage allows components to add or remove possible interpretations based on the content of the local working memory, or the working memories in other subarchitectures.

At a software level the representations are encoded as classes described in an interface definition language (CORBA's IDL) which are written and read using the methods described in Section 4. The components only know that this is happening via subscriptions to change events based on the class type and operation. Using CAST to design and implement the processing of both these subarchitectures provides a number of advantages. The interaction between the components in the subarchitectures occurs via data (i.e. the objects on working memory) rather than in a direct fashion. This allows components to change independently of both the data and the other components, allowing alternative components to be substituted without meeting a particular interface specification. Additionally, this method of component interaction design is robust to missing components. Whereas direct connections between components could crash or stall a system, the absence of component from a CAST system (or any system with similar publish/subscribe semantics) will only harm the system at run-time if it provides an intermediate step in a processing chain.

Although this demonstrates that CAST shares the strengths of other engineering approaches based on decoupling information exchange from component interfaces, its support for shared working memories provides an additional advantage. Whereas most communication toolkits in robotics treat the information exchanged between components as something transient that is lost (or consumed) when received (i.e. it is only relevant when it is being communicated), in CAST information is present on working memories until a component decides to delete it. This allows components to share information both with each other, and with a previously undefined collection of other components that can access the information on working memory. In the CAST design it is the change events that are subscribed to and flow throughout a system, acting as triggers for component activity. Information on working memory is then accessed on demand, meaning that the cost of communicating working memory entries is only incurred when a receiving component determines it is necessary. In this way CAST couples the on-demand reads of an remote procedure call-based system with the aforementioned subscription-based communication channels.

One of the main advantages of making information available via a working memory is that it allows information to be used as an input to other processes without its creators triggering, or even being aware of these processes. This allows us to build up subarchitectures, and then collections of subarchitectures, incrementally from groups of components and their shared representations. Within the vision subarchitecture we used this practise to extend the static scene representations produced by the segmentor with mobile objects updated by a tracker. As described previously the segmentor produces representations of objects when the scene is static. This approach is fine for doing static scene analysis for dialogue etc., but is not able to support behaviours such as action recognition which require object trajectories over time. To overcome this we added an additional object tracking component. This uses the ROIs and object representations produced by the segmentor as input, then tracks colour histograms generated from the ROIs and updates the position of the associated object using the result of this tracking. The tracking occurs in parallel to other refinements of the ROIs and objects, and can be added and removed from the system without interfering with these other operations. To make this possible all working memory transactions must be performed based on an access protocol that maintains working memory consistency (see Section 4.2.1 and Section 4.2.2).

Whilst this example demonstrates how sharing information via working memory allows additional functionality to be added within a subarchitecture, this approach also allows us to add additional subarchitectures in a similar manner. The QSR subarchitecture uses potential field models of spatial relations (left-of, near etc.) to abstract away from the metric detail in the vision subarchitecture's 3D model of the world to produce a graph-like model of objects on a tabletop [7]. This model is used to ground referring expressions from the communication subarchitecture (e.g. "the ball to the right of the box"), and generate target positions for manipulation commands (e.g. "put the cup near the box"). As this qualitative model is an abstraction of the vision-generated model of the world, it requires input from vision as a starting point. Rather than wait for input from a particular visual component, the components in the QSR subarchitecture listen for change events signalling either that an object has been added to vision working memory (when it's created), that an object has been overwritten (when it's potentially been moved by the tracker), or that an object has been deleted (when it's no longer visible to the tracker or segmentor). Based on these triggers the QSR components (one component for each relationship type) read all of the objects from vision working memory and update a list of relationships stored on QSR working memory. With this approach any components could be used in the vision subarchitecture, and, providing they use the same representation of objects as the current system, the QSR subarchitecture's behaviour should be unchanged.

Output from the PlayMate system comes either via speech generated by the communication subarchitecture or from pick and place actions generated by the manipulation subarchitecture. In both cases these actions are triggered by another subarchitecture writing an entry to one of these working memories containing a description of the action to be performed. To provide a general purpose interface for triggering actions, each action entries has two linked parts: a generic action entry that is common to all actions, and a subarchitecture-specific entry describing the actual action to be performed (e.g. a logical form describing the content to verbalise, or an object to pick up and the position to place it in). This allows all components that perform actions to listen for the generic entry and then use this to access the task-specific part if appropriate.

5.1.2. Binding

To do reasoning and planning on a model of the world that contains all of the data from its input subarchitectures, the PlayMate system requires a mechanism to abstract and fuse information across subarchitectures. We have previously proposed a design for a *binding subarchitecture* which does precisely this [22]. This design requires that each subarchitecture which can contribute to the system's knowledge about the world (e.g. vision, communication and QSR) contains a *binding monitor* component which translates information from the representations stored on its local working memory (e.g. visual objects or packed logical forms), into a language of *binding proxies* and *binding features*. In our design a feature is a property that the system can know about (e.g. position, colour, shape, a spatial relation etc.) represented in a form that all subarchitectures can process. A proxy is a set of features used to describe a particular thing the system knows about (e.g. an object, a location, a person etc.). We use the term "proxy" because this set of features can be used as a proxy for the original (subarchitecture-specific) information from which it was translated. When a binding monitor creates (or updates) a proxy, it is written to the working memory of the binding subarchitecture. It is then read by a collection of components that work in parallel to *bind* this proxy with all of the other proxies (generated from other information in the system) that refer to the same thing. The result of this binding process is a structure called a *binding union* which is simply a union of all of the features from the bound proxies.

As pictured on the left of Figure 2 the binding process produces a hierarchy of representations ranging from (mutually incompatible) subarchitecture internal representations to an abstracted shared representation which spans the knowledge of the system. The advantage of fusing information via a single representation rather than via n pair-wise translations is that this approach only requires a single new translation for each new subarchitecture, rather than $n - 1$ translations for the pair-wise case. In addition, the union structures provide a dynamically-generated symbolic representation of the system's best hypothesis of the current state. This representation can then be used as input to a deliberative process such as planning. The advantage of generating symbolic descriptions via this hierarchy of representations is that the symbols remain tethered (via links maintained from unions down to proxies down to subarchitecture internal data) to the representations they were generated from. It is essential to maintain this tethering to allow the robot to act in its world, as physical action requires representations

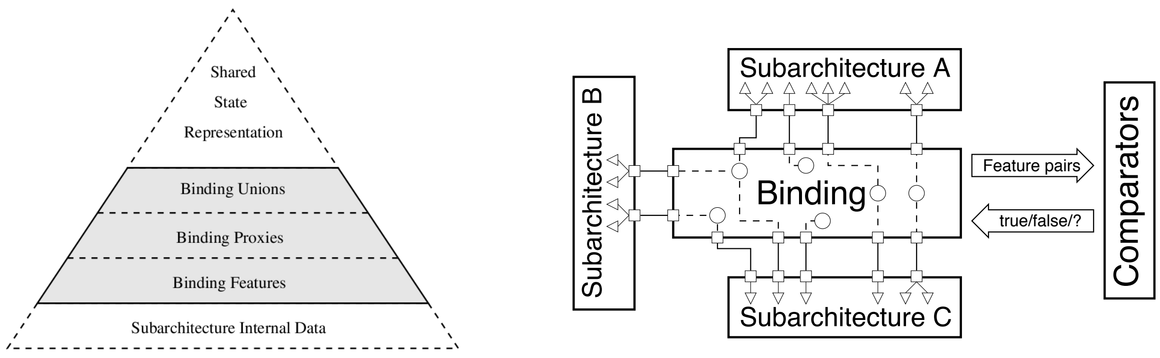

Figure 2: Binding looks like this

closely tied to sensor data (such as the metric information generated by vision).

We have considered the theoretical properties of the binding subarchitecture elsewhere (see [22]). In this section we will concentrate on how its implementation is achieved within CAST. At a first pass, the addition of binding monitors into a subarchitecture can be done in the same way as the additional functionality described in Section 5.1.1: all information available on working memory is accessible to new components so a monitor can just read the information it needs independently of other processing and without adding dependencies onto existing components. Each monitor (the PlayMate system contains four of them) is designed to perform a (currently hard-coded) translation from subarchitecture-local information to proxies and features. As subarchitecture level processing is typically happening incrementally and in parallel, binding monitors can function in this way too. As information is incrementally added to a local representation it can be translated and added to the corresponding proxy in parallel to any other (further) processing occurring in the subarchitecture.

To trigger the binding process, a monitor must signal the binding subarchitecture with a list of new or updated proxies. On receiving this signal three components start processing the proxies. The *binding scorer* component starts comparing the features in the proxies with the features associated with any pre-existing unions. Each comparison assesses whether the features are the same or not, yielding either true, false or unknown, which is stored on working memory. The results of these comparisons are collected for each union and are then turned into a score for binding the input proxy to that union. The *binding judge* collects these scores and determines what process should be run on the proxy. This will either be that the proxy should be bound into a new union, that a previously-bound but updated proxy should be left in its union, or that a previously-bound proxy should be rebound with another union. Given this decision the *binder* then acts accordingly, updating the proxy and union entries on binding working memory.

One reason for dividing the binding process up in this way is to allow the monitors to update their proxies and features asynchronously and not have them waiting for the entire binding process to complete. The process of feature comparison in particular can take a variable amount of time. This is because the design of the binding system allows for feature comparisons to be forwarded to other subarchitectures when domain-specific processing is required. Such behaviour is triggered by the binding scorer writing a description of the comparison to binding working memory. It then adds a pointer entry to the other subarchitecture's working memory containing the WMA of the comparison. When the comparison is complete the comparison entry is overwritten with the result. The binding scorer is listening for changes to this entry, and so can collect the result and add it to the accumulated score for the proxy and union the features belong to. This approach is commonly used in CAST systems to generate remote procedure call-like functionality where one component invokes some processing in another.

In the PlayMate, the binding system acts a little like an abstraction layer that allows processes requiring a single symbolic world model to operate on top of a multiple, concurrently refined, more detailed models. The key to successfully engineering this approach is to tame the inherent complexity of having so many interdependent processes operating in parallel. For example, if a monitor attempts to update a feature whilst the feature is involved in a comparison, or if a proxy is deleted whilst the binder is adding it to a union, there is a risk that inconsistencies are introduced both in single working memory entries and in the overall system. To combat this, the implementation of the binding subarchitecture uses CAST's locking functionality (described previously in Section 4.2.2) to prevent deleterious conditions. For each major phase of the binding process (proxy updating and binding) there is a token entry on working memory. For any component to perform an operation that could influence one of these phases (e.g. deleting a proxy) it must obtain locks on the tokens for the relevant phases. In this manner behaviours that would introduce inconsistencies are made mutually exclusive, with representational consistency coming at the cost of the delays incurred obtaining the necessary locks.

5.1.3. A System Run

Although we have only introduced a subset of the PlayMate's functionality, we will now present an overview of a simple system run to provide a flavour for the way our engineering efforts contribute to the system's behaviour. We will consider an example in which the PlayMate is presented with a blue square, and then is told that "this is a blue square". The contents of a

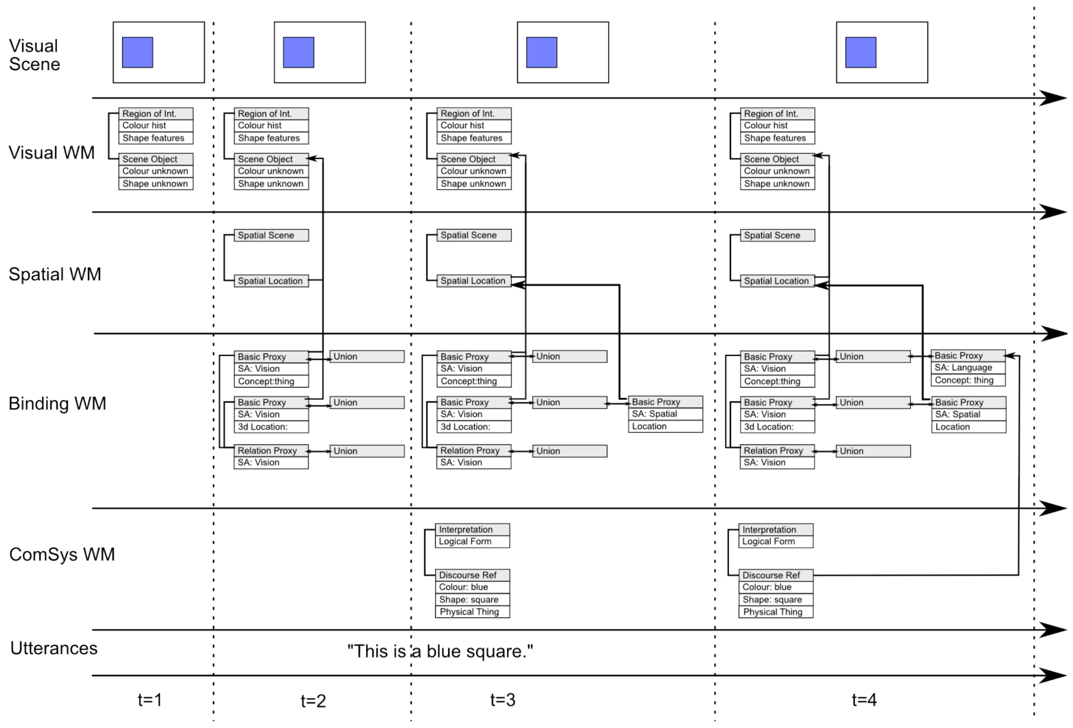

Figure 3: A timeline presenting a simplified and discretised view of the contents of the PlayMate when being shown and told about a blue square. A graphic of the scene is used instead of a camera image for clarity.

subset of the system's working memories are presented in Figure 3. This figure discretises the example into four stages. This is to allow easier presentation: the behaviour of the PlayMate, and CAST itself, is not run in discrete cycles. At $t = 1$ the robot is presented with a blue square. The vision subarchitecture responds to the change in the scene, segments a ROI containing the object, and generates a new object representation which is paired with the ROI. In this example we assume that the system has no knowledge about colours or shapes (it is in the process of being taught), so although the ROI contains some features, the object representation does not contain any recognition results from these features. At $t = 2$ two subarchitectures become active in parallel in response to the appearance of objects on the visual working memory. In the QSR subarchitecture, a spatial scene is created containing a single spatial location with no relations (as no other objects are visible). In addition to this the binding monitor from the vision subarchitecture writes three proxies to the binding working memory. These describe, in binding features, the object, it's location in 3D space, and the fact that the object is at this location. The separation of the object and its location allows changes in object location to occur independently of changes to properties such as colour, shape or category. This means that processes which only care about the latter properties do not need to update their representations when an object changes position (e.g. as it is tracked), and vice versa. As the binding working memory contains no other proxies or unions, these new proxies are each assigned a new union. At $t = 3$ we again have two processes in parallel. The binding monitor in the QSR subarchitecture creates a proxy for the single location in the current scene. The binding subarchitecture determines that this location is similar to the location generated by vision (which it should be as they represent the same position but with different precision) and therefore binds the new location proxy into the union with the existing location proxy. In parallel with this, a human tells the robot "this is a blue square". This triggers speech recognition and incremental interpretation in the communication subarchitecture, yielding a packed logical form and a set of of discourse referents describing the utterance. At $t = 4$ the binding monitor for the communication subarchitecture writes a proxy description of "a blue square" to binding working memory (the "this is" aspect is written elsewhere, but we will ignore that here) where it is compared to the existing unions. As both this proxy and the object proxy from vision have been determined (by their respective binding monitors) to be instances of the super-type *thing*, they are bound together. At this point we have linked together multiple different representations of the world, and have a structure with which we can do some further processing (e.g. learning what a blue square looks like [29], or asking for a clarification [23]). In the PlayMate, further processing is planned and triggered by the motivation subarchitecture. Due to space restrictions we will not provide further details on this subarchitecture, except that it provides components that wrap the MAPSIM planner [6], and trigger actions in other subarchitectures (see [17] for more information).

This example demonstrates the power of engineering a system with CAST. Multiple collections of processes can be active in parallel, collaborating over shared data structures. The use of working memories ensures that the information being processed in these interactions is available at all times, rather than being locked up inside components. This allows other components to operate on the intermediate and final results of processes independently of the generation of the results. This loose coupling also means that subarchitectures can be added and removed over time as the system is designed, prototyped and engineering in various ways. The result of this is that CAST is ideal for supporting scientific and exploratory engineering efforts into intelligent systems, where systems need to be extensible and flexible in various ways (both at run-time and design-time), rather than fixed and special purpose (as may be the case for a commercial product such as an intelligent vacuum cleaner). For further information on the PlayMate system, including videos and additional publications, see http://cognitivesystems.org/playmate.asp.

5.2. The Explorer System

For the Explorer system we used the communication, binding, and motivation subarchitectures from the PlayMate system but replaced the other input and output subarchitectures with ones appropriate for mobile robotics. The Explorer was also run on a different physical platform: an ActivMedia Peoplebot. The additional subarchitectures required for the Explorer are a navigation subarchitecture including processing components for simultaneous localisation and mapping (SLAM), navigation graph construction and robot movements; a conceptual mapping subarchitecture that builds semantically labelled topological maps on top of the navigation graph; and an object search subarchitecture that can perform visual searches for objects in a room, storing the results on working memory. This majority of the algorithms and technology in the Explorer system were ported from a previous system (presented in [33]) to work within CAST. The overall operation of the Explorer system is similar in flow to the PlayMate. Subarchitectures build up task-specific representations (in this case mostly different types of maps) through concurrent collaboration between groups of components. These representations are then made available via the binding system. At this level they are used as input into planning processes, and planning generates actions which are sent back to the appropriate subarchitectures for execution.

It is the use of shared representations that allowed us to reuse the communication, binding and motivation subarchitectures in the Explorer. In particular, the use of binding features and proxies as an abstraction language for information, serving as the input to motivation (as symbols for planning) and communication (as content to be communicated) subarchitectures, allowed the PlayMate subarchitectures to be used independently of the subarchitectures that were originally contributing to the binding process. Once the navigation, conceptual mapping and object search subarchitectures were given binding monitors (a process that was simplified by the availability of task-specific information on working memory) they were able to contribute towards the shared state knowledge of the system, and the Explorer could then make plans for behaviour (such as searching for objects or moving to a different location). The only other addition beyond this was that the Explorer-specific subarchitectures had to provided components that listened for actions posted to their working memories, in order to generate the behaviours when necessary. For further information on the Explorer system, including videos and additional publications, see http://cognitivesystems.org/explorer.asp.

6. Evaluation

Although we have not been able to perform any quantitative evaluation on the use of CAST in the engineering of our two demonstrator systems, we can highlight particular aspects of our approach that have helped or hindered our design and implementation work.

Using CAST as the basis for both the Playmate and Explorer shows that our approach of parallel refinement of shared representations is effective in domains with very different temporal and spatial properties. In the PlayMate, representations (mostly from vision) tend to be quite small scale (representations of between 1 and 5 objects) and can change a few times per second (based on the framerate of the object tracker). In the Explorer the representations are of a much larger scale (line and node maps of three or more connected rooms) but change at a slower rate (as the robot moves slowly and SLAM smooths out sensor noise). Coping with both these cases demonstrates that CAST is applicable to a wide range intelligent robotics domains.

Being able to port existing technologies from previous systems to the Explorer demonstrates that the processing styles supported by CAST are enough to cover at least those styles that were used in these previous systems. So, although the working memory-based processing of CAST disallows direct message passing between components (the fundamental approach of most other robotic implementation frameworks), this does not mean that systems originally built with message passing in mind cannot be restructured to work with shared information.

At multiple points in the descriptions of our systems we have seen how sharing information via working memories, rather than encapsulating it in components, supports both loose coupling between components and subarchitectures, and incremental design and development. This allows a designer to start with an initial set of components to perform some basic processing tasks (e.g. change detection and segmentation in the previous example of a visual system) that populate working memory. Following this, CAST allows the designer to incrementally add components which process the entries on working memory without altering the initial components. As long as these additional components do not introduce dependencies at the information-processing level (e.g. one component must always wait for the results of another component), then they can be added and removed as required by the task.

The nature of CAST has allowed us to generalise this model to whole system development. Our integrated systems feature subarchitectures developed in parallel across multiple sites. Each site follows the above approach to subarchitecture design, gradually adding components to a subarchitecture in order to add more functionality to it. The integrated system then emerges in a similar way: we start with a small number of subarchitectures containing a few components, and then add functionality in two ways. Individual subarchitectures can be extended with new components as before, but also complete subarchitectures can be added to the system (again without recompilation or restructuring) to provide new types of functionality. Because the information produced by existing subarchitectures is automatically shared on their working memories, any new subarchitecture has immediate access to the knowledge of the whole system. As with components, the restriction that communication occurs only via working memories means that additional subarchitectures can be removed without altering the initial system. In [18] we demonstrated this incremental system development model by taking an existing system for scene description and extending it with the ability to plan and perform manipulation behaviours.

Perhaps the biggest problem engineering systems in CAST is also tied to the restriction of only communicating via working memories. When one component wants to request some information from another component or trigger some processing (i.e. perform a remote procedure call), it must do this via working memory. This leads to engineers writing working memory entries that perform the role of procedure calls. These entries are written by the calling component and read and acted on by the called components. Performing processing in this manner in an event-driven system is unintuitive and means that one of the basic tools available to a programmer is not available by default in CAST. Overcoming this, without losing the benefits of information sharing, is a major challenge in the future development of CAST.

The use of CAST also challenges designers and engineers to focus on the interaction protocols used to facilitate the concurrent refinement of shared information. As we saw in sections 4.2.1 and 4.2.2, allowing multiple components to overwrite working memory entries without introducing inconsistencies requires explicit synchronisation models. This forces designers to consider the parallel nature of the processing that is being carried out in CAST systems. Working with parallel processing models is a relatively advanced skill in terms of programming competence, and is something which can often be avoided in other (e.g. basic message passing) paradigms. Also, whilst the use of change events allow interaction protocols to be based on the information changes on working memory, there is always a danger that change events can be semantically overloaded as a system is extended. For example, in our vision subarchitecture an object being overwritten can signify either that its position has changed or that a visual property has been updated. Discriminating between these two cases (to ensure that redundant or incorrect processing is not triggered) requires components to include condition checking code in their change filters.

Even with these drawbacks, CAST provides one final benefit to our projects: an extremely close coupling between system design and implementation. We presented the theoretical motivation for this close coupling in Section 2, but there is a more pragmatic benefit too. Before using CAST, meetings to consider system designs were often unstructured and vague. We either considered systems at a component level or at an information processing level. Component based discussions often tended to be endless, as it is easy to arbitrarily add an extra component, or fruitless, as there is often no justification for one component-based design over another. Information-processing architecture discussions often tended to be high-level and ultimately confusing, as people had different notions for system modules and decomposition, and had different processes for dividing up overall system behaviour into module behaviours. Once we adopted CAST, its focus on shared representations overcame a lot of these problems. In terms of information-processing design, subdivision into subarchitectures around core representations is an intuitive approach that is free to ignore the exact methods of information processing, generation or communication. In terms of lower-level component designs, using shared representations for input and output focuses the design on clear units of processing which are constrained within the overall subarchitecture, and therefore information-processing architecture, design.

7. Comparisons To Existing work

CAST can be compared to many existing software packages intended for the development of robotic systems. Although the basic unit of functionality in CAST is a processing component, it is markedly different from other component-based frameworks such as MARIE [10], ORCA [8] and YARP [14]. These frameworks provide methods for reusing components and accessing sensors and effectors, but they do not provide any *architectural structure* except in the very loosest sense (that of components and connections). It is this structure that CAST provides, along with supporting component reuse. CAST does not provide access to sensors and effectors as standard, although to date we have successfully integrated Player/Stage, various camera devices and a manipulator engine into CAST components. At this stage in the development of CAST we would ideally like to integrate it with one of these component frameworks to provide our architecture structure with access to existing devices and components.

At the other extreme of software for intelligent robotics is the software associated with cognitive modelling architectures. Such tools includes ACT-R [1] and SOAR [25]. Whilst these systems provide explicit architecture models along with a means to realise them, they have two primary drawbacks for the kind of tasks and scientific questions we are interested in studying. First these systems provide fixed architecture models, while CAST provides support for a space of possible instantiations based on a more abstract schema (allowing different instantiations to be easily created and compared). Second, it is not currently feasible to develop large integrated systems using the software provided for these architectures. This is due to restrictions on the programming languages and representations that must be adhered to when using these models. That said, researchers are now integrating cognitive models such as these into robotic systems as reasoning components, rather than using them for the architecture of the whole system (e.g. [4]).

In addition to these two extremes (tools that provide architectures and tools that provide connections) there are a small number of toolkits that have a similar aim to the work presented in this paper. MicroPsi [3] is an agent architecture and has an associated software toolkit that has been used to develop working robots. It is similar to the cognitive modelling architectures described previously in that it has a fixed, human-centric, architecture model rather than a schema, but the software support and model provided is much more suited to implementing robotic systems than other modelling projects. Our work is similar to the agent architecture development environment ADE [2]. APOC, the schema which underlies ADE is more general than CAS. This means that a wider variety of instantiations can be created with ADE than with CAST. This is positive for system developers interested in only producing a single system, but because we are interested in understanding the effects that varying an architecture has on similar systems, we find the more limited framework of CAS and CAST provides useful restrictions on possible variations.

Perhaps the existing tool most similar to CAST is the Active Memory XML Server from the XCF framework [15]. In this approach an "active memory" plays a similar role to a CAST working memory, except that it also includes various processing and control behaviours (such as entry forgetting and access control via Petri nets). Although there are many similarities in design between this and CAST (the use of change events, and basic query and insert operation), there is a fundamental difference in implementation: the XCF Active Memory is implemented as relational database and is accessed via XML queries. The use of XML provides a uniform representation for all working memory entries. Although this representation is more costly to encode and transmit than CAST's CORBA-based serialised objects, it provides one clear advantage: content-based working memory access. By using the XPATH formalism, arbitrary sets of entries can be retrieved from working memory based on the information contained within the entries. In CAST retrieval is only possible via information which describes the entries (type, creator, write mode etc.). Content-based access makes it a lot easier to write code which is dependent on the content of entries (e.g. to process any new object closer than twenty centimetres, or retrieve all objects with an assigned category with a high confidence), and reduces the overhead related to semantically overloaded change events (see Section 6). An additional difference between the XCF approach and that of CAST is that XCF systems only have a single working memory, whereas CAST is designed to support multiple working memories. We see the modularity provided by multiple working memories (and thus multiple subarchitectures) as essential when designing, and running, a complex information-processing architecture. In the future we will actively explore the trade-offs offered by both systems, and others that work on a shared workspace paradigm (e.g. [26]).

8. Conclusion

In this article we have described the CoSy Architecture Schema Toolkit (CAST) and the theoretical schema (CAS) it is based on. We discussed our motivation for developing a toolkit and also described some of the influences that the toolkit has had on the way we design and build intelligent robot systems. Although CAS does not specify a cognitive model *per se*, it constrains the space of models that can be built with it. This constrained space represents a subset of all possible architecture designs; a subset which we have found to fit naturally with the robotics problems we face on a day-to-day basis. In particular, we have demonstrated in this article how the novel properties of CAS, supported by its implementation in CAST, have allowed us to engineer intelligent systems (the PlayMate and Explorer), in ways which make them flexible and extensible, and supports the reuse of components and subsystems across domains. All of this relies on the paradigm of parallel refinement of representations stored on shared working memories, a paradigm which points to exciting new directions in the science of engineering intelligent systems.

9. Acknowledgements

This work was supported by the EU FP6 IST Cognitive Systems Integrated Project "CoSy" FP6-004250-IP, and the EU FP7 IST Cognitive Systems Integrated Project "CogX" ICT-215181-CogX.

References

[1] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., Qin, Y., 2004. An integrated theory of the mind. Psychological Review 111 (4), 1036–1060.

[2] Andronache, V., Scheutz, M., 2006. An architecture development environment for virtual and robotic agents. Int. Jour. of Art. Int. Tools 15 (2), 251–286.

[3] Bach, J., 2003. The micropsi agent architecture. In: Proc. of ICCM-5. pp. 15–20.

[4] Benjamin, D. P., Lonsdale, D., Lyons, D., 2007. A cognitive robotics approach to comprehending human language and behaviors. In: HRI '07: Proceedings of the ACM/IEEE international conference on Human-robot interaction. ACM, New York, NY, USA, pp. 185–192.

[5] Bonasso, R. P., Firby, R. J., Gat, E., Kortenkamp, D., Miller, D. P., Slack, M. G., 1997. Experiences with an architecture for intelligent, reactive agents. J. Exp. Theor. Artif. Intell. 9 (2-3), 237–256.

[6] Brenner, M., 2008. Continual collaborative planning for mixed-initiative action and interaction. In: AAMAS '08: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems. International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1371–1374.

[7] Brenner, M., Hawes, N., Kelleher, J., Wyatt, J., 2007. 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.

[8] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., Oreback, A., 2005. Towards component-based robotics. In: Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on. pp. 163–168.

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

[10] Cote, C., Letourneau, D., Michaud, F., Valin, J.-M., Brosseau, Y., Raevsky, C., Lemay, M., Tran, V., 2004. Code reusability tools for programming mobile robots. In: IROS2004.

[11] Coulouris, G., Dollimore, J., Kindberg, T., 2001. Distributed Systems: Concepts and Design, 3rd Edition. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Dennett, D. C., 1978. Brainstorms: Philosophical Essays on Mind and Psychology. MIT Press, Cambridge, MA.

[13] Erman, L., Hayes-Roth, F., Lesser, V., Reddy, D., 1988. The HEARSAY-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty. Blackboard Systems, 31–86.

[14] Fitzpatrick, P., Metta, G., Natale, L., 2008. Towards long-lived robot genes. Robot. Auton. Syst. 56 (1), 29–45.

[15] Fritsch, J., Wrede, S., 2007. An integration framework for developing interactive robots. In: Brugali, D. (Ed.), Software Engineering for Experimental Robotics. Vol. 30 of Springer Tracts in Advanced Robotics. Springer, Berlin, pp. 291–305.

[16] Hawes, N., 2004. Anytime deliberation for computer game agents. Ph.D. thesis, School of Computer Science, University of Birmingham.

[17] Hawes, N., Brenner, M., Sjöö, K., 2009. Planning as an architectural control mechanism. In: HRI '09: Proceedings of the 4th ACM/IEEE international conference on Human robot interaction. ACM, New York, NY, USA, pp. 229–230.

[18] Hawes, N., Sloman, A., Wyatt, J., Zillich, M., Jacobsson, H., Kruijff, G.-J., Brenner, M., Berginc, G., Skočaj, D., 2007. Towards an integrated robot with multiple cognitive functions. In: AAAI '07. pp. 1548 – 1553.

[19] Hawes, N., Wyatt, J., Sloman, A., November 2006. An architecture schema for embodied cognitive systems. Tech. Rep. CSR-06-12, University of Birmingham, School of Computer Science.

[20] Hawes, N., Wyatt, J., Sloman, A., 2009. Exploring design space for an integrated intelligent system. Knowledge-Based SystemsIn Press.

[21] Hawes, N., Zillich, M., Wyatt, J., August 2007. BALT & CAST: Middleware for cognitive robotics. In: Proceedings of IEEE RO-MAN 2007. pp. 998 – 1003.

[22] Jacobsson, H., Hawes, N., Kruijff, G.-J., Wyatt, J., March 12–15 2008. Crossmodal content binding in information-processing architectures. In: Proceedings of the 3rd ACM/IEEE International Conference on Human-Robot Interaction (HRI). Amsterdam, The Netherlands.

[23] Kruijff, G.-J., Brenner, M., Hawes, N., August 2008. Continual planning for cross-modal situated clarification in human-robot interaction. In: Proceedings of IEEE RO-MAN 2008.

[24] Kruijff, G.-J. M., Lison, P., Benjamin, T., Jacobsson, H., Hawes, N., 2007. Incremental, multi-level processing for comprehending situated dialogue in human-robot interaction. In: Symposium on Language and Robots. Aveiro, Portugal.

[25] Laird, J. E., Newell, A., Rosenbloom, P. S., 1987. Soar: An architecture for general intelligence. Artificial Intelligence 33 (3), 1–64.

[26] Martínez-Barberá, H., Herrero-Pérez, D., September 2008. Multirobot applications with the thinkingcap-ii java framework. In: Hülse, M., Hild, M. (Eds.), IROS Workshop on current software frameworks in cognitive robotics integrating different computational paradigms.

[27] Michaud, F., Côté, C., Létourneau, D., Brosseau, Y., Valin, J. M., Beaudry, E., Raïevsky, C., Ponchon, A., Moisan, P., Lepage, P., Morin, Y., Gagnon, F., Giguère, P., Roux, M. A., Caron, S., Frenette, P., Kabanza, F., 2007. Spartacus attending the 2005 aaai conference. Auton. Robots 22 (4), 369–383.

[28] Shanahan, M., Baars, B., 2005. Applying global workspace theory to the frame problem. Cognition 98 (2), 157–176.

[29] Skočaj, D., Berginc, G., Ridge, B., Štimec, A., Jogan, M., Vanek, O., Leonardis, A., Hutter, M., Hawes, N., 2007. A system for continuous learning of visual concepts. In: International Conference on Computer Vision Systems ICVS 2007. Bielefeld, Germany.

[30] Sloman, A., October 1998. The "semantics" of evolution: Trajectories and trade-offs in design space and niche space. In: Coelho, H. (Ed.), Progress in Artificial Intelligence, 6th Iberoamerican Conference on AI (IBERAMIA). Springer, Lecture Notes in Artificial Intelligence, Lisbon, pp. 27–38.

[31] Wright, I., 1997. Emotional agents. Ph.D. thesis, School of Computer Science, The University of Birmingham.

[32] Wyatt, J., Hawes, N., 2008. Multiple workspaces as an architecture for cognition. In: Samsonovich, A. V. (Ed.), Proceedings of AAAI 2008 Fall Symposium on Biologically Inspired Cognitive Architectures. The AAAI Press, pp. 201 – 206.

[33] Zender, H., Jensfelt, P., Óscar Martínez Mozos, Kruijff, G.-J. M., Burgard, W., July 2007. An integrated robotic system for spatial understanding and situated interaction in indoor environments. In: Proc. of the Twenty-Second Conference on Artificial Intelligence (AAAI-07). Vancouver, British Columbia, Canada, pp. 1584–1589.