Developing Intelligent Robots with CAST

Nick Hawes and Jeremy Wyatt
Intelligent Robotics Lab
School of Computer Science
University of Birmingham Birmingham, UK
{n.a.hawes, j.l.wyatt}@cs.bham.ac.uk

Abstract— **In this paper we describe the CoSy Architecture Schema, and the software toolkit we have built to allow us to produce instantiations of this schema for intelligent robots. Although the schema does not specify a cognitive model *per se*, it constrains the space of models that can be built from it. Along with descriptions of the schema and toolkit we present the motivation behind our designs (a need to explore the design-space of information-processing architectures for intelligent systems), and a discussion of the kinds of design, implementation and information-processing models they support.**

I. 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 a particular 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 a robotic system. 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 [20], [10].

The desire to tackle architectural and integration issues in a principled manner has led us to develop the CoSy Architecture Schema Toolkit (CAST) [14]. 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 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. Given the topic of this workshop, this paper will expand the motivation behind the design of CAST (Section II), how the computation paradigm of multiple shared working memories is implemented in our software toolkit (Section III), and how this paradigm influences the systems we build and the way we build them (Section IV).

[1]Cognitive Systems for Cognitive Assistants: http://cognitivesystems.org
[2]Cognitive Systems that Self-Understand and Self-Extend: http://cogx.eu

II. UNDERSTANDING ARCHITECTURES AND INTEGRATION

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 two 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. Instead, the initial problem analysis should cover the *requirements* for the system's information-processing architecture design (i.e. the integrating parts) in addition to the component technologies.

In the field of intelligent artifacts, the term "architecture" is still 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. At this level of description there is no real difference between the study of architectures in AI and software architectures in other branches of computer science. However, differences appear as we specialise this description to produce architectures that integrate various types of functionality to produce intelligent systems. Architectures for intelligent systems typically include elements such as fixed representations, reasoning mechanisms, and functional or behavioural component groupings. Once such elements are introduced, the trade-offs between different designs become important.

Such trade-offs include the costs of dividing a system up to fit into a particular architecture design, and the costs of using a particular representation. Such trade-offs have been ignored by previous work on integrated systems, yet these factors are directly related to the efficacy of applying an architecture design to a particular problem. Our research studies architectures for integrated, intelligent systems in order to inform the designers of these systems of the trade-offs available to them.

Given that there is a huge space of possible architecture designs for intelligent systems (cf. [18], [21]) it is important both that we (as scientists concerned with designing and building such systems) understand this design space and the trade-offs it offers, and that we are able to evaluate the influence of the architectures we use in the systems we build. Although we could study architecture designs purely in theory (cf. [15]), the dynamic and complex internal and external behaviours of even the simplest robots situated in the real world means that we may have more success in studying the designs empirically. For this we require implementations of our architecture designs that allow us to separate the effects of the architecture design from the effects of the components being integrated by the design. We have argued this point elsewhere (cf. [11]), but it is worth restating as it motivates our approach to the design of middleware for intelligent robots. To avoid the uninformative, ad-hoc approach to building integrated systems (characterised above as "look ma no hands"), we must not only be able to demonstrate that our system works, but we must also be able to provide some analysis on why the system works the way it does. This type of analysis is almost always performed for the components of an integrated system, but it is rarely, if ever, performed for the architecture design used to integrate the components.

There may be many reasons why researchers do not evaluate the influence their chosen information-processing architecture has on the behaviour of their intelligent system. One reason is the confusion between the types of architectures (at various levels of abstraction) that feature in the implementation of an intelligent system. These can include very abstract information flow architectures, more concrete decompositions into subsystems and functional components, and further, more detailed, decompositions into classes and functions in software. In our experience, it is typically this latter type of architecture (i.e. the system's software architecture) that is the only architecture explicitly present in the final system implementation. Because of this, it is difficult to isolate the information-processing architecture (which we assume is typically at a higher level of abstraction than classes and functions in most systems), from the rest of the system; it is only implicitly present in the implementation. To address this problem we advocate the use of an *architecture toolkit* when building intelligent systems. Such a toolkit fixes the information-processing architecture explicitly in the implementation of the system, and keeps the architectural elements separate from the components in the system.

III. THE COSY ARCHITECTURE SCHEMA TOOLKIT

This reasoning has led us to develop the *CoSy Architecture Schema Toolkit* (CAST) [14], a software toolkit which implements the *CoSy Architecture Schema* (CAS) and thus allows us to build information-processing architectures for intelligent robots based on instantiations of this schema.

A. The Schema

As mentioned in Section II we are interested in understanding the trade-offs available in the design space of architectures for intelligent systems. We designed CAS to allow us study a small region of this space in a systematic manner. CAS is an architecture *schema*, i.e. a set of rules which can be used to design architectures. We refer to the process of designing and building an architecture from the schema as *instantiation*. By comparing instantiations of the schema we can uncover some of the trade-offs that are available within the design space defined by the schema.

CAS is based around the idea of a collection of loosely coupled *subarchitectures*, where a subarchitecture can be considered as a subsystem of the whole architecture. As shown on the left in Figure 1, each subarchitecture contains a number of processing components which share information via a *working memory*, and a control component called a *task manager*. Some processing components within an subarchitecture are *unmanaged* and some are *managed*. Unmanaged components perform relatively simple processing on data, and thus run constantly, pushing their results onto the working memory. Managed processes, by contrast, monitor the changing working memory contents, and suggest possible processing tasks using the data in the working memory. As these tasks are typically expensive, and computational power is limited, tasks are selected on the basis of current needs of the whole system. The task manager is essentially a set of rules for making such allocations. Each subarchitecture working memory is *readable* by any component in any other subarchitecture, but is *writable* only by processes within its own subarchitecture, and by a limited number of other *privileged* components. These components can post information to any other subarchitecture, allowing top-down goal creation and cross-architecture coordination.

If there are several processing goals that a subarchitecture needs to achieve, they are mediated by its task manager. This mixes top-down and data driven processing and allows goals to be handled that require coordination within one subarchitecture or across multiple subarchitectures. At one extreme the number of privileged components can be limited to one (centralised coordination), and at the other all components can be privileged (completely decentralised coordination).

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 [17], and Global Workspace Theory [19]. On the other hand it grounds these influences in a distributed, concurrent system which shares properties with robotic architectures such as 3T [5].

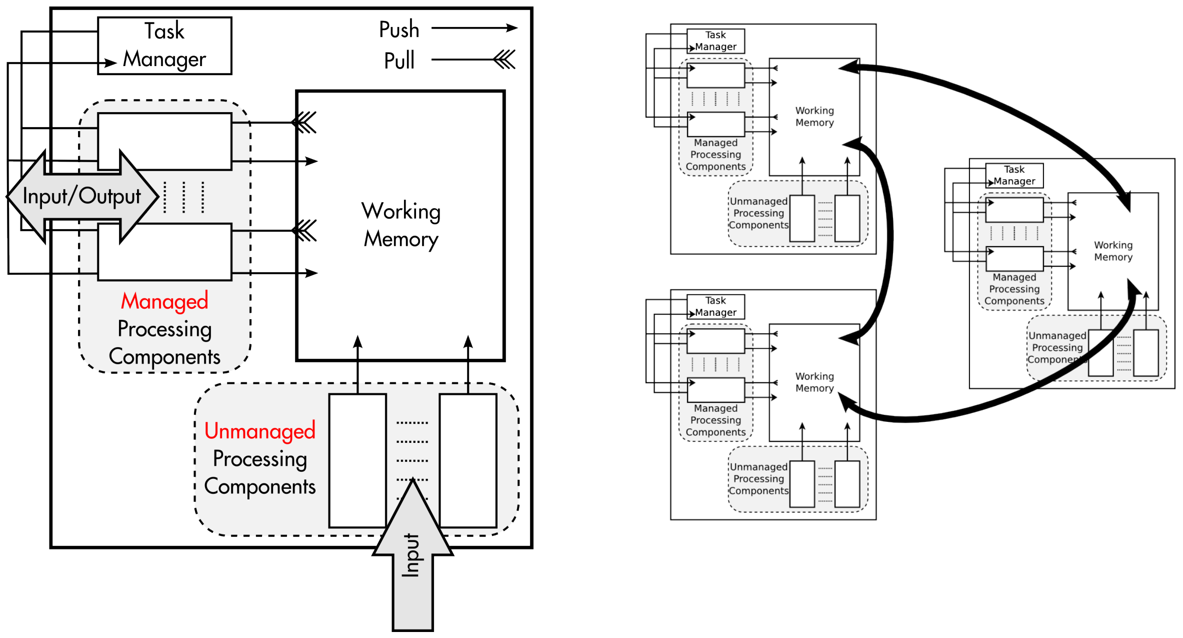
Task Manager
Push
Pull
Input/Output
Working Memory
Managed Processing Components
Unmanaged Processing Components
Input

Fig. 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 via working memory connections.

B. The Toolkit

This schema is implemented in our software toolkit CAST. The toolkit provides 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 provides access to functionality intended to make programming within the schema as simple as possible. Most of this functionality is based around accessing working memories.

In CAST a working memory is an associative container that maps between unique identifiers (IDs) and *working memory entries*. Each entry is an instance of a *type*, which can be considered as analogous to a class. 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 of 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 that are being performed on the entries it contains. These events contain the ID and type of the changed entry, the component that made the change, and the operation performed to create the change (i.e. whether the entry was added, overwritten or deleted).

As is necessary for any robot-centric middleware, CAST can run distributed across multiple machines. It also natively supports both C++ and Java (a requirement for our project's software which we found hard to satisfy using other robotic middleware). We currently use CORBA to provide support for the cross-language and cross-network translation, although we're considering replacing this in the future. Translation is hidden from the programmer, so that there is no difference when a component accesses a working memory written in the same language on the same machine, to when it accesses a working memory written in a different language on a remote machine. To support the exploration of the design space of CAST instantiations, architectures built with CAST can be reconfigured without changing component code or recompilation. This allows us to add and remove subarchitectures, and change the decomposition of components into subarchitectures, without changing a line of code. Finally, CAST is open source, and freely available from http://www.cs.bham.ac.uk/research/projects/cosy/cast/.

IV. THE BEHAVIOUR OF CAST

CAST allows us to design and build a wide range of intelligent systems. These systems will vary in many ways, but because they are CAST instantiations they will all share at least one feature: shared working memories. It is this feature that gives CAST systems their distinctive information-processing behaviour (a behaviour similar to distributed blackboard systems [8]). We now explore this behaviour in more detail.

A. *Concurrent Refinement of Shared Information*

Two of the requirements that influenced the design of CAS are the requirement that information is shared between components, and the requirement that components are active in parallel. These requirements combine in CAS to produce a model in which processing components work concurrently to build up shared representations. For example, the visual working memory in our recent intelligent system [12] contains entries representing regions of interest and objects in the scene. Once these basic structures have been created (via segmentation and tracking) other components (such as feature recognisers) can work in parallel adding information to these entries. As the data is shared, any update to an entry is available to any component that needs it as soon as the entry is changed on working memory. In this manner the results of processing by one component can by asynchronously incorporated into the processing of another as soon as the results are available.

It is not only components within a subarchitecture that are are active in parallel. Subarchitectures themselves are also concurrently active, allowing the concurrent refinement of information in one subarchitecture working memory to influence the processing of components in another. This behaviour allows us, for example, to narrow the space of hypotheses in incremental parsing [16] in parallel to extracting information about the visual scene.

Although this processing model could be implemented in a purely component-based architecture, the amount of concurrent data sharing between subsystems means that it is a very natural fit with the design of CAS.

B. *Incremental Design & Development*

By only allowing components in a CAST system to communicate via information shared on working memories we reduce the interdependencies between them. This allows components to be added to, or removed from, CAST instantiations without the need for the architecture be restructured (or for any code to be recompiled). This freedom allows us to adopt an incremental approach design and implementation.

Due to the subarchitecture schema design, a processing behaviour (e.g. a linguistic or visual interpretation behaviour) can be designed, implemented and tested incrementally. A designer can start with an initial set of components to perform some basic processing tasks (e.g. segmentation in the previous example of a visual system) that populate working memory. Following this, CAST allows the designer to incrementally add components to process the entries on working memory without altering the initial components (unless extensions to them are required). 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. To return to the example of the visual system, this incremental model allows us to have a basic subarchitecture that provides the 3D properties of objects. We can then add additional components (recognisers, feature extractors etc.) to this subarchitecture as required by the application domain, whilst leaving the original components untouched.

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 them 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 [12] 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.

We have also taken advantage of the ability to restructure a CAST system to perform a preliminary exploration of the design-space delineated by CAS. In [13] we took an existing CAS instantiation and systematically varied its ratio of components to subarchitectures. The encapsulation of the CAS-level system components into a toolkit allowed us to benchmark architectural properties of the resulting systems separately to their functional properties.

V. 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 [7], ORCA [6] and YARP [9]. 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 [17]. 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 a fixed architecture model, whilst 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 perhaps most 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.

VI. Conclusions

In this paper 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. It is worth noting that there is a related space of designs that may come from combining our schema with other middleware or architecture approaches (where this combination would typically provide a smaller, rather than larger, space of designs). We are excited to see the other contributions to this workshop and explore possible interactions between their design spaces and ours.

VII. Acknowledgements

This work was supported by the EU FP6 IST Cognitive Systems Integrated Project "CoSy" FP6-004250-IP.

References

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

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

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

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

[5] R. Peter Bonasso, R. James Firby, Erann Gat, David Kortenkamp, David P. Miller, and Mark G. Slack. Experiences with an architecture for intelligent, reactive agents. *J. Exp. Theor. Artif. Intell.*, 9(2-3):237–256, 1997.

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

[7] Carle Cote, Dominic Letourneau, Francois Michaud, Jean-Marc Valin, Yannick Brosseau, Clment Raevsky, Mathieu Lemay, and Victor Tran. Code reusability tools for programming mobile robots. In *IROS2004*, 2004.

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

[9] Paul Fitzpatrick, Giorgio Metta, and Lorenzo Natale. Towards long-lived robot genes. *Robot. Auton. Syst.*, 56(1):29–45, 2008.

[10] Nick Hawes, Aaron Sloman, and Jeremy Wyatt. Requirements & Designs: Asking Scientific Questions About Architectures. In *Proceedings of AISB '06: Adaptation in Artificial and Biological Systems*, volume 2, pages 52–55, April 2006.

[11] 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.

[12] Nick Hawes, Aaron Sloman, Jeremy Wyatt, Michael Zillich, Henrik Jacobsson, Geert-Jan Kruijff, Michael Brenner, Gregor Berginc, and Danijel Skočaj. Towards an integrated robot with multiple cognitive functions. In *AAAI '07*, pages 1548 – 1553, 2007.

[13] Nick Hawes and Jeremy Wyatt. Benchmarking the influence of information-processing architectures on intelligent systems. In *Proceedings of the Robotics: Science & Systems 2008 Workshop: Experimental Methodology and Benchmarking in Robotics Research*, June 2008.

[14] Nick Hawes, Michael Zillich, and Jeremy Wyatt. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of IEEE RO-MAN 2007*, pages 998 – 1003, August 2007.

[15] Randolph M. Jones and Robert E. Wray. Comparative analysis of frameworks for knowledge-intensive intelligent agents. *AI Mag.*, 27(2):57–70, 2006.

[16] Geert-Jan M. Kruijff, Pierre Lison, Trevor Benjamin, Henrik Jacobsson, and Nick Hawes. Incremental, multi-level processing for comprehending situated dialogue in human-robot interaction. In *Symposium on Language and Robots*, Aveiro, Portugal, 2007.

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

[18] Pat Langley and John E. Laird. Cognitive architectures: Research issues and challenges. Technical report, Institute for the Study of Learning and Expertise, Palo Alto, CA, 2002.

[19] Murray Shanahan and Bernard Baars. Applying global workspace theory to the frame problem. *Cognition*, 98(2):157–176, 2005.

[20] Aaron Sloman. The "semantics" of evolution: Trajectories and trade-offs in design space and niche space. In Helder Coelho, editor, *Progress in Artificial Intelligence, 6th Iberoamerican Conference on AI (IBERAMIA)*, pages 27–38. Springer, Lecture Notes in Artificial Intelligence, Lisbon, October 1998.

[21] D. Vernon, G. Metta, and G. Sandini. A survey of artificial cognitive systems: Implications for the autonomous development of mental capabilities in computational agents. *Evolutionary Computation, IEEE Transactions on*, 11(2):151–180, April 2007.