# Towards an Empirical Exploration of Design Space

**Nick Hawes** and **Aaron Sloman** and **Jeremy Wyatt**
School of Computer Science, UK
University of Birmingham
{N.A.Hawes, A.Sloman, J.L.Wyatt}@cs.bham.ac.uk

**Abstract**

In this paper we propose an empirical method for the comparison of architectures designed to produce similar behaviour from an intelligent system. The approach is based on the exploration of *design space* using similar designs that all satisfy the same requirements in *niche space*. An example of a possible application of this method is given using a robotic system that has been implemented using a software toolkit that has been designed to support architectural experimentation.

## Introduction

Although many researchers design architectures for intelligent systems, very few evaluate the influence the architecture has on the system, instead preferring to evaluate the overall system performance on the tasks it was designed for. If the science of building intelligent systems is to advance beyond the current state-of-the-art it is essential that architectures are evaluated as key system components in their own right. Due to the typically complex nature of intelligent systems such evaluation is not easy to do; systems are not traditionally designed and built to allow the influence of the architecture to be isolated from that of the other (interconnected) elements of the system. This paper presents a sketch of a methodology that could be used for comparing architecturally different versions of a single system in order to explore how these variations affect the performance of the whole system.

## Design Space & Niche Space

The study of architectures for intelligent systems is an exploration of *design space*, the space of possible designs for intelligent agents (Sloman 1998). Architecture designs are usually produced to satisfy a set of constraints and requirements. This set defines a particular *niche* that could be satisfied by many designs, and a single niche can be considered as a point in *niche space* (the space of all possible constraints and requirements). In most AI work architecture designs are ultimately implemented in hardware and software. As with the previous mapping between design space and niche space, there is a space of possible implementations that may realise a single design. This could be as simple as implementing a design in different programming languages, or it could involve different robot sensor and effector configurations providing the same gross functionality. This characterisation of intelligent systems has been explored in greater detail in previous work, e.g. (Sloman 1998; 2000).

Given that architectures for intelligence can be described in terms of niches, designs and implementations, there are a number of different ways architectures can be evaluated. For example, designs for architectures for neighbouring points in niche space could be compared to study how the changes in requirements result in a change in design, or the interaction between a single niche and design could be explored in increasing levels of detail (cf. (Hawes 2004; Scheutz & Logan 2001)). For this paper we will concentrate on the evaluation possibilities provided by comparing a number of closely related designs for a single niche.

### 1 Niche, *n* Designs

When looking to draw an informative comparison between architectures, it is important that some common aspects exist between them. This allows the comparison to be drawn with reference to these fixed elements. In the most extreme case of comparing designs for the same niche, only the fact that they satisfy the same niche may be common. For example, a comparison could be drawn between two museum tour-guide robots that must solve the same problems to function correctly but are based on different hardware platforms and have run different software. Another example is a comparison between a design where all possible contexts in that niche have been analysed in advance by the designer and suitable (e.g. reactive) responses Dre-coded versus a design where the system has some general capability which can be used to work out what to do. Typically the former will respond much faster, and the latter will be able to cope with a wider variety of circumstances, though speed differences could be masked by hardware differences. Another difference might be between a neural-net based system that learns by adjusting weights in a network and one that uses symbolic rules and learns by compiling or modifying rules (including rules using probabilities in the conditions or in the actions). Although such comparisons between very differ-

ent designs may provide some information about the ways both systems solve the same problems, it will be difficult to separate out the effects of their different architectures from the effects of their many other possibly different software and hardware elements. This is a special case of the 'credit assignment problem' which bedevils many AI learning systems (Minsky 1995). It can also be a problem for a research community.

In a comparative study, in order to obtain information about the effects a system's architecture has on its overall behaviour it is important to be able to control the variation between the compared systems to just variations in architecture. In terms of software, this means that the systems should be composed of software modules based on the same designs. The variation should occur in the connection and communication patterns that exist between the software modules in the design, i.e. at the level of the system's architecture. In terms of system-level behaviours, this means that the systems must be compared when performing the same tasks (e.g. answering a question or giving a tour). This type of exploration of design space draws on problem-specific knowledge about possibly useful architecture designs, allowing us to explore the design space for a single niche using very small steps between designs (as the majority of their parts are based on the same designs).

In the remainder of this paper we discuss an approach to putting this methodology into practice using a software toolkit designed for experimenting with architectures without altering their processing components.

## The CoSy Architecture Schema & Toolkit

We have recently designed and implemented a robotic system that can manipulate simple objects and engage in simple linguistic interactions with a human about its world (Hawes *et al.* 2007). The design of the system is based on the CoSy Architecture Schema (CAS) (Hawes, Wyatt, & Sloman 2006), which was developed from requirements derived from scenarios for a self-extending household assistant robot with 3-D manipulative capabilities and some understanding of what it is doing. In addition to representational, algorithmic and learning requirements derived from the complexity, variability, ontological richness and unpredictability of domestic environments, the scenarios require support for architectural features such as: concurrent modular processing, structured management of knowledge, and dynamic control of processing.

To satisfy these requirements, the schema allows a collection of loosely coupled *subarchitectures* (SAs). As shown in Figure 1, each contains a number of processing components which share information via a specific working memory, and a control component called a task manager. Some processing components within an SA are *unmanaged* and some *managed*. Unmanaged components continuously perform relatively simple processing. This may include processing high-bandwidth data. They run constantly and reactively, pushing any significant results onto their working memory. Managed processes, by contrast, monitor the changing working memory contents, and suggest possible processing tasks using the data in the working memory.

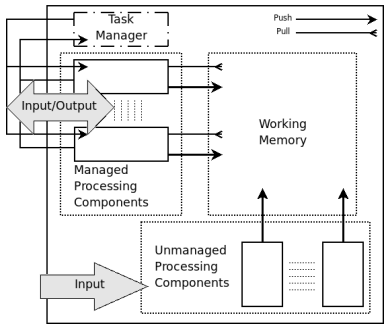


Figure 1: The CAS Subarchitecture Design Schema.

Such processing tasks may include deliberative mechanisms concerned with controlling, monitoring, planning, and learning from actions in the environment, and meta-management tasks concerned with controlling, monitoring planning and learning from internal processes and their consequences, including the deliberative processes. 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 SA, but is *writable* only by processes within its own SA, and by a limited number of other privileged SAs. Components within privileged SAs can post instructions to any other SA, allowing top-down goal creation.

If there are several goals in a subarchitecture they are mediated by the SA's task manager. This mixes goal-driven (top-down) and data-driven (bottom-up) processing and allows goals to be handled that require coordination within one SA or across multiple SAs. At one extreme the number of write-privileged SAs can be limited to one (centralised coordination), and at the other all SAs can have write privileges (completely decentralised coordination). Our scenarios appear to favour a small number of specialised, privileged coordination SAs.

We work from the requirement that processing components must operate concurrently to build up shared representations. SAs work concurrently on different sub-tasks, and components of an SA work on different parts of a sub-task. This is required, for example, to allow the results of one component's processing to influence the concurrent processing of other components working with the same information. This may be the case with components working at different levels of abstraction on the same task (as in POPEYE (Sloman 1978)), or with one component predicting the results of another (as in modular motor learning (Wolpert, Doya, & Kawato 2003)). In general, instantiations of the SA schema function as distributed blackboard systems as used, for example, in Hearsay-II (Erman *et al.* 1988).

To support these requirements in implemented systems we developed the CoSy Architecture Schema Toolkit (CAST) (Hawes, Zillich, & Wyatt 2007), a software toolkit

based on CAS. The toolkit provides interfaces for developing managed and unmanaged components in C++ and Java (and could be extended for other languages) and implementations of task managers and working memories. Components can be connected into subarchitectures across machine and language barriers without recompilation. The toolkit allows researchers to experiment easily with different instantiations of the schema, and to separate the effects of the architecture (i.e. the toolkit) from that of the components software modules. Both of these aspects of the toolkit are important when evaluating an architecture and they will be exploited in the following evaluation methodology.

Although this paper is about ways of comparing different implementations of architectures providing very similar functionality, the main point of both the architecture schema and the toolkit is to allow *significant* changes to an architecture to be made relatively easily during research. This may be necessary where the required architecture is not obvious, or where new components or competences must be integrated into a system. Examples of such cases include allowing new high level visual information to be passed to a natural language subsystem which may be able to use the new information to resolve syntactic or semantic ambiguities, or allowing new information from language to be passed to a visual system to help with the interpretation of ambiguous or obscure visual input (e.g. a sentence drawing attention to an object that is in deep shadow and therefore not very visible).

We have used CAST to develop an integrated system for a large European integrated project (Hawes *et al.* 2007). The system is a linguistically-driven tabletop manipulator that combines state-of-the-art subarchitectures for cross-modal language interpretation and generation (Kruijff, Kelleher, & Hawes 2006) and visual property learning (Skočaj *et al.* 2007) to produce a system that can learn and describe object properties in dialogue with a tutor. In addition to this, the system features subarchitectures for planning, spatial reasoning and manipulation (Brenner *et al.* 2007) to allow it to carry out manipulation commands that feature object descriptions based on the previously learnt visual properties in combination with a collection of 'innate' (directly programmed) visual, motor and learning capabilities.

We wish to evaluate the system we have built on at least two levels. We want to evaluate it on how well it fits its niche (i.e. how good it is at performing the tasks it was designed to do). This will involve benchmarking it over a wide range of visual scenes, dialogues and commands. We also want to evaluate how suitable CAS is as an architecture design for such a system. This requires us to separate the effects of the system's architecture from that of its other elements. The following sections propose a methodology for doing this empirically. However, any empirical results will also need to be explained analytically.

## Comparing CAS Instances

To separate the effects of our CAS instantiation's architecture from the effects of its other components we need other similar instantiations to compare it to. As stated previously, to understand the influence of the architecture on the overall system we must be able to compare the system to other systems that are nearby in the design space that satisfies a single niche. There are many ways to do this, but in CAS and CAST there are a small number of ways to move through design space that are easy to implement with the provided tools[1]. The following sections present the simplest ways of altering CAS instantiations for comparative experiments. These suggestions are followed by a selection of proposed metrics that could be used to empirically compare the effects of the architecture in the various instantiations.

### Variations in Control

Each CAS subarchitecture features a control component called a task manager. The component controls when managed components can process data. Part of designing and implementing a CAS instantiation is writing the rules that each task manager will use for its subarchitecture. In our current system most of the subarchitecture task managers allow components to act whenever they can, in parallel if necessary. The exceptions to this are the spatial and communication subarchitectures which use finite state machines to coordinate the actions of their managed components.

The first way in which we could consider creating easily comparable CAS instantiations is to enforce particular patterns of control in different instantiations. There are three control patterns that sample the possible spectrum of parallelism in an architecture: fully parallel, subarchitecture parallel, fully sequential. A fully parallel architecture is what we currently use, in which each component can act when it is able to. A fully sequential architecture would only allow a single component to act at a time, and no other component could act until it had finished. A subarchitecture parallel architecture would sit between these extremes, allowing only a single subarchitecture to be active at one time but with components within the subarchitecture able to act in parallel. The notion of 'parallelism' allowed by CAS includes both the parallelism that exists when processes are distributed across multiple machines and that which occurs in multi-threaded single CPU processes. Our current system uses a mixture of both these types of parallelism (multiple machines running multiple processes in parallel).

We view the support for parallelism as an essential aspect of architectures for systems that must operate in the real world. As parallelism is not explicitly covered by all architectures for such problems (cf. (Laird, Newell, & Rosenbloom 1987)) it is important to be able to demonstrate the effects of parallelism on a complete system in a principled manner. By varying the amount of concurrency supported by a CAS instantiation (without varying any other aspect of it) we hope to separate out the effects of the schema's support for parallelism from the other aspects of the instantiation.

[1]Ease of implementation is important because few researchers can spare the time to create multiple systems to perform the same task purely for comparative purposes, except where 'rival' teams happen to be using different architectures and/or tools for the same purpose.

### Variations in Connections

Our current system uses 31 processing components divided between 7 subarchitectures. The main role of the subarchitecture grouping is to strategically limit the flow of information between components. Components can subscribe to change information about working memories and use this information to read from working memories. Subarchitectures can be configured to send change information to other subarchitectures or not, and to receive change information from other subarchitectures or not. Components within subarchitectures can then apply additional filtering to this change information, choosing to receive information from a particular source component or subarchitecture; of a particular change type (add, delete or overwrite); or about a particular datatype. Components can choose to discard change information that arrives when they are engaged in processing, or to buffer this information for later access. The buffers used for this are not fixed in size, but are limited by the memory available on the system. In practice this has not become an issue, as change information structures are typically very small (although in some scenarios and systems they could be numerous). The amount of processing a component has to do to receive relevant change information is an indicator of how much redundant information it is receiving as a result of its connections into the wider architecture.

The grouping of components into subarchitectures also controls which components have access to working memories. In the schema, access to working memory is serial, so that when more components require access to a single working memory the greater the chance that an access request will have to be queued while another request is carried out. On the other hand, when a greater number of working memories are used there is a greater overhead in locating a particular item of data.

By varying which components in are in which subarchitectures it is possible to explore the performance trade-offs present in these configuration options. There are a number of possible configurations of components and subarchitectures that could be explored in CAS: a 1-subarchitecture instantiation, an $n$-component $n$-subarchitecture instantiation, and an $n$-component $m$-subarchitecture instantiation. In a 1-subarchitecture instantiation all the components would be in a single subarchitecture, attached to a single working memory. At the other end of the connection spectrum, an $n$-component $n$-subarchitecture instantiation would feature a subarchitecture and working memory for every component. Between these extremes is an $n$-component $m$-subarchitecture instantiation where $n > m$ and $m > 1$. This is the case for our current system, where components are grouped into subarchitectures based on a functional decomposition of the system's desired behaviours. These connection variations could be combined with the previously presented control variations to explore design space in two dimensions of architectural variation.

Architectures tend to be viewed almost exclusively as approaches to grouping and connecting processing components. There are many different possibilities for subdividing systems, (e.g. compare the functional subdivision in (Hawes *et al.* 2007) to the behaviour based subdivision favoured in (Brooks 1986)) but there is not a great deal of objective evidence about the benefits of one approach over another for a particular task. Although it is clear that the above variations in CAS connection patterns will not shed light on the bigger questions of modularity, we hope that exploring a subset of the issues in a controlled way will allow us to make some progress in this direction.

## Experimental Metrics

Using the CAS toolkit it is possible to take our current system and create the various instantiations described above just by changing the way the system is configured at startup time (i.e. with no changes to compiled code). Given that we can produce these implementations of particular points in design space it is important to be able to compare them objectively. To this end we need to measure aspects of the performance of each instance. The following are suggestions for possible metrics we could employ to compare architecture instances:

- The number of successfully completed system-level behaviours in a particular scenario (e.g. answering a question). It would be expected that this would not change across instantiations.
- The time it takes for a system-level behaviour task to be completed. Increased parallelism should reduce this, and the time taken for lower-level tasks will have an influence.
- The number of component-level tasks (e.g. segmenting a scene) that are completed in a given amount of time across the instantiation. This is an indication of how much concurrency is being exploited in the system. The time taken to write to and retrieve from working memory may have a small influence.
- The amount of time taken to write to or read from working memory. This should be influenced by the number of components attached to a working memory, and the levels of concurrency allowed in the system (concurrent processing should cause more access attempts to co-occur).
- The time taken for a component to complete a task (e.g. parsing a sentence or segmenting a scene). This will be influenced by the CPU load of the component's system and the time taken to interact with working memory.
- The CPU load of the machines running the instance. This is a coarse way of judging the complexity of the processing being done by the system. This metric provides no real measure of instance performance in isolation, but should do so when measured over multiple architecture instances running the same components. This assumes that the components always place the same load on the CPU and it is the change in architecture that creates any load variation across instances.
- A ratio of required working memory change information to redundant change information received by a component. This reflects the overhead placed on a component due to the architecture's connectivity.

These metrics are obviously interdependent in various ways and should fluctuate in coherent patterns across system be-

haviours and architecture instantiations. There are likely to be many more aspects of a system's performance that could and should be measured in order to assess the influence of the architecture and how this varies across instantiations.

In addition to these 'object-level' metrics it is also possible to examine 'meta-level' metrics for architecture instances. For example, we could investigate metrics for measuring how easy it is to debug, maintain or extend a system; how well a design supports the addition of various kinds of learning or self-monitoring; how easy is it to give a mathematical specification of the the system; or how many aspects of human-like or animal-like functionality a design supports or impedes. Because we are interested, in this paper, in providing a method for evaluating designs that are close in design space, these issues become less important (as such metrics will probably vary very little across considered designs). In the long-term they are critical issues that must be addressed by the science of building intelligent systems.

## Conclusion

In this paper we proposed a methodology for a task-independent empirical method for evaluating architectures for intelligent systems. The methodology is based on comparing data taken from multiple similar systems that vary only in architectural terms. These systems will all exist at neighbouring points in the design space for a single point in niche space. The data taken from the systems will allow us to evaluate the architectures by understanding how moving through design space in particular ways affects the overall performance of systems built on the architectures.

In the next few months we will apply the methodology to evaluate the CoSy Architecture Schema by understanding its position in design space. This exercise will also allow us to further develop the methodology itself. One particular focus will be developing the metrics that allow informative comparisons to be drawn between implemented robotic systems.

## Acknowledgements

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

## References

Brenner, M.; Hawes, N.; Kelleher, J.; and 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)*.

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

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

Hawes, N.; Sloman, A.; Wyatt, J.; Zillich, M.; Jacobsson, H.; Kruijff, G.; Brenner, M.; Berginc, G.; and Skocaj, D. 2007. Towards an integrated robot with multiple cognitive functions. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*.

Hawes, N.; Wyatt, J.; and Sloman, A. 2006. An architecture schema for embodied cognitive systems. Technical Report CSR-06-12, University of Birmingham, School of Computer Science.

Hawes, N.; Zillich, M.; and Wyatt, J. 2007. BALT & CAST: Middleware for cognitive robotics. In *Proceedings of the 16th International Symposium on Robot and Human Interactive Communication (RO-MAN 07)*. IEEE. To appear.

Hawes, N. 2004. *Anytime Deliberation for Computer Game Agents*. Ph.D. Dissertation, School of Computer Science, University of Birmingham.

Kruijff, G.-J. M.; Kelleher, J. D.; and Hawes, N. 2006. Information fusion for visual reference resolution in dynamic situated dialogue. In Andre, E.; Dybkjaer, L.; Minker, W.; Neumann, H.; and Weber, M., eds., *Perception and Interactive Technologies: International Tutorial and Research Workshop, PIT 2006*, volume 4021 of *Lecture Notes in Computer Science*, 117 – 128. Kloster Irsee, Germany: Springer Berlin / Heidelberg.

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

Minsky, M. 1995. Steps toward artificial intelligence. In *Computers & thought*. Cambridge, MA, USA: MIT Press. 406–450.

Scheutz, M., and Logan, B. 2001. Affective vs. deliberative agent control. In *Proceedings of the AISB'01 Symposium on Emotion, Cognition and Affective Computing*, 39–48.

Skočaj, D.; Berginc, G.; Ridge, B.; Štimec, A.; Jogan, M.; Vanek, O.; Leonardis, A.; Hutter, M.; and Hawes, N. 2007. A system for continuous learning of visual concepts. In *International Conference on Computer Vision Systems ICVS 2007*.

Sloman, A. 1978. *The Computer Revolution in Philosophy*. Hassocks, Sussex: Harvester Press (and Humanities Press). http://www.cs.bham.ac.uk/research/cogaff/crp.

Sloman, A. 1998. The "Semantics" of Evolution: Trajectories and Trade-offs in Desi gn Space and Niche Space, (Invited talk). In Coelho, H., ed., *Progress in Artificial Intelligence, Proceedings 6th Iberoamerican Conference on AI (IBERAMIA)*. Lisbon: Springer, lecture notes in Artificial Intelligence,. 27–38.

Sloman, A. 2000. Interacting trajectories in design space and niche space: A philosopher speculates about evolution. In M.Schoenauer, *et al.*., ed., *Parallel Problem Solving from Nature – PPSN VI*, Lecture Notes in Computer Science, No 1917, 3–16. Berlin: Springer-Verlag.

Wolpert, D. M.; Doya, K.; and Kawato, M. 2003. A unifying computational framework for motor control and social interaction. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences* 358(1431):593–602.