Benchmarking The Influence of Information-Processing Architectures on Intelligent Systems

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

***Abstract*— The design of the information-processing architecture used to develop an intelligent robot plays a significant role in the behaviour of the final system. In this paper we discuss the possibilities for benchmarking the influence of architecture designs on intelligent robots. We separate this problem into two sub-problems: benchmarking the architecture design and benchmarking the implementation of the design. For each of these sub-problems we list some design- and run-time properties that could be investigated. To further demonstrate these ideas we present some early efforts to benchmark the run-time properties of a previously developed architecture schema.**

I. Introduction

Many current robotic projects are focused on the task of building intelligent or cognitive robots. In such projects the majority of the effort is directed towards developing component technologies such as parsers, planners, map building systems and object recognition techniques. All these technologies are of course vital for any such project, but with the overwhelming effort directed at the components, the study of how these components are *integrated* is often overlooked. We argue that understanding the *information-processing architecture* used to integrate components into a single intelligent system is an important, and oft-overlooked, element of the science of designing and building intelligent robots (and other systems). This paper describes, at a high level, our motivation for wanting to understand architectures, and some proposals for experimental methodologies and metrics for evaluating the influence of an architecture on an intelligent system.

II. Background

A common approach to 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 (some of which this workshop is trying to address), but we will focus on the "put all the technologies together" step. If the number of component technologies are 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 a robot, or would like your system to be generally 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.

Given that there is a huge space of possible architecture designs for intelligent systems (cf. [1, 2]) it is important both that we (as scientists concerned with designing and building such systems) understand this design space, and that we are able to evaluate the influence of the architectures we use on the systems we build. We have argued the former point elsewhere (e.g. [3, 4]), and will focus on the latter point in this paper.

III. The Problem

Informally put, the problem is as follows: imagine that a number of teams of researchers are all asked to build an intelligent system to solve a clearly defined problem (e.g. attending a conference, being a tour guide in a museum or crossing a desert). All the teams are given the same software libraries that provide solutions to the all the problems the target system will need to solve. They are also all given the same hardware (sensors, effectors and processing power) to run the software on. The teams each produce a system to solve the initial problem, and each system displays different behaviour to the others (some may display large differences, some small). The question is: what are the differences between these systems?

There are many different ways to answer this question. One way may focus on the engineering skills of the teams, one on the communication middleware used to wrap the libraries into components, one on the division of the task into sub-tasks and modules etc. To address this, we will assume that each team took the following approach: either adopted an existing

information-processing architecture design (e.g. 3T [5], Soar [6] or the Subsumption Architecture [7]) or invented their own; built the architecture infrastructure in software (or used the software provided by the original architecture designers); and then used this infrastructure to assemble the provided software libraries into the finished system. In other words they used an *architecture toolkit* to build their system. For ease of discussion we will assume that an architecture toolkit is the software realisation of an abstract architecture design. We assume that such software allows the teams in the example to build robots that are direct *instantiations* of the abstract design (which is almost never the case in real applications).

By making the assumption of an architecture toolkit we can separate out two separate aspects of a toolkit which we might want to benchmark or otherwise study in terms on their influence on the design and implementation of intelligent robots. These aspects are the abstract architecture design that the toolkit is based on (i.e. the design of the toolkit), and the software (middleware) which the system engineers have to use to build systems with (i.e. the implementation of the toolkit). We will address these separately in the following sections. However, before that it is worth stressing that all evaluation and benchmarking of architecture designs must be done in relation to a particular problem or task. Architecture designs (and implementations of these designs) are created to satisfy the requirements of a particular *niche* [8]. The use of a particular design (e.g. The Subsumption Architecture) may demonstrate completely difference performance from one task (e.g. rubbish collection) to another (e.g. chess playing). Therefore please consider all evaluation procedures and metrics mentioned subsequently to be *task dependent*. Given this, they will only be meaningful if we compare *different architectures* on *the same task* using *the same procedures and metrics* [4]. A further qualification is that the following sections and suggestions are almost certainly incomplete. We present them here purely as a starting point for the discussion on benchmarking for intelligent systems.

IV. Evaluating Architecture Designs

Using the toolkit metaphor we can separate our discussion of architecture design evaluation into two sections: the influence the abstract architecture design has on the design & build-time requirements of an intelligent system, and the influence the design has on the run-time requirements of an intelligent system.

A. Design & Build Influences

Although the influence of the architecture design can have a massive impact on the overall design & build of an intelligent system, it is very hard to benchmark this impact. To measure the effects of an architecture design on the design & build of system quantitatively you would have to look at the person-months spent on particular aspects of the task. However, such measurements would not be particularly informative on their own (and would be heavily influenced by the individuals in question), without a qualitative description of how the architecture influenced different parts of the problem solving activities involved in the system design & build. Such qualitative factors could include:

- What, if any, representations does the architecture design enforce? How do these relate to the requirements of the tasks? For example, is a single unified representation required (and if so, how is translation to and from this handled), or can multiple representations be used?
- What, if any, system decomposition does the architecture design enforce? Does it only allow behavioural or functionality decompositions, or are arbitrary decompositions allowed?
- What functionality does the design provide? For example, does it include learning mechanisms or action arbitration mechanisms? How do these relate to the requirements of the task?
- Does the architecture design provide a useful tool for system design? For example is it too restrictive to allow particular designs to be used, or is it too general so that it does not constrain the space of possible designs enough to be useful?
- How easy is it to add new functionality to the system? Does the architecture mean that additions require a complete restructuring?

B. Run-time Influences

Once an architecture toolkit has been used to build a system, we can then investigate the effects the architecture design has on the system at run-time. Run-time influences are more amenable to quantitative benchmarking, given appropriate metrics that could be extracted from a running system. That said it could prove hard to separate the run-time influences of architecture design and the run-time influences of the architecture implementation. Some properties that could be measured include:

- How often does the system succeed, fail, make a small mistake etc.?
- How quickly can the system respond to task-relevant changes in its environment?
- What is the overhead of moving information from one component to another in the system (in terms of filtering out relevant information, not purely communications load)? (see Section VIII)
- How well does the architecture scale up to bigger problems or scale out to problems requiring additional functionality? I.e. how does the performance of the architecture change as the requirements of the task change.

V. Evaluating Architecture Middleware

Given a software toolkit that represents an architecture design you can again benchmark, evaluate or investigate it in terms of design & build time influences and run-time influences. As we are now referring to the software aspects of architecture toolkits, we expect that these factors are closely related to the properties you would wish to benchmark for

(non architectural) middleware such as YARP [9] or MARIE [10].

VI. Design-Time Influences

As discussed in Section IV-A, design-time influences are very hard to benchmark or evaluate directly. However, there are questions that should be asked of any software tool used for robotics (or otherwise).

- How well does the toolkit support code reuse?
- How flexible is the toolkit with respect to programming style? Does it allow programmers to solve problems in the way they want, or are they overly constrained by the architecture design.
- What support for debugging does the toolkit provide? Can intermediate results of processing be inspected?
- How well does it integrate with existing software? E.g. does it provide interfaces for using Player/Stage [11], or for accessing standard hardware devices?
- What are the real-time properties of the toolkit? Does it provide performance guarantees?
- What are programming languages does the toolkit support?

VII. Run-Time Influences

When looking to benchmark the software aspects of an architecture toolkit at run-time, we are mostly interested in the aspects of the performance of the system that are independent of the actual architecture design. These include:

- What is the (CPU, time, memory) overhead of using the toolkit?
- How robust is the system with respect to component or communication failures?
- How quickly can data be exchanged between two or more components?

VIII. An Example of Comparing Architectures

In previous work we suggested a method for benchmarking architectures [4]. This method is relatively close to the thought experiment presented in Section III. Following this method we took a robotic architecture toolkit, the CoSy Architecture Schema Toolkit (CAST) [12], and developed a system with it. We then altered the architecture in principled ways whilst making the minimum changes to component code that we could. Instantiations of these altered architectures were then run on the same problem, allowing us measure some of their run-time characteristics. These measurements were then compared to allow us to benchmark each architecture on this problem. For this study we focused on a small selection of the run-time properties of the architectures. These properties relate to the third bullet point in Section IV-B: the overhead of moving information from one component to another in the system.

A. The Experimental System

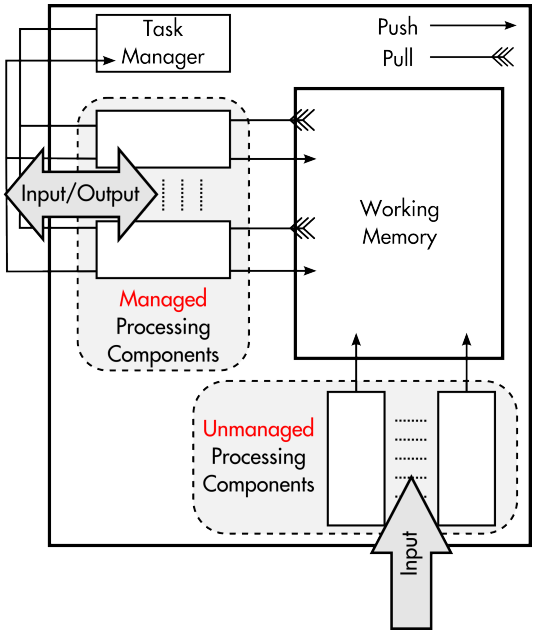
The system which we have used for benchmarking is derived from an intelligent robot cable of object manipulation and human-robot interaction [13]. It is based on the CoSy Architecture Schema (CAS). This schema is pictured in Figure 1. It has a number of features relevant to developing integrated systems, but here we will focus on the approach it takes to passing information between components [1].

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 in Figure 1(a), each subarchitecture contains a number of (concurrently active) processing components which share information via a working memory. When information is operated on (added, overwritten or deleted) in a CAS working memory a *change event* is generated. This event structure contains information on the operation performed, the type (i.e. class name) of the information, the component which made the change, and the location of the information in the system. Components use this change event data to decide whether to perform some processing task with the changed information. To restrict the change events that are received, each component is able to *filter* the event stream based on change event contents. Components typically subscribe to relevant change information by registering callbacks on combinations of the fields in the change event. For example, a vision component may subscribe to additions of regions of interests. This filter would refer to the change event's operation type (addition) and data type (region of interest).

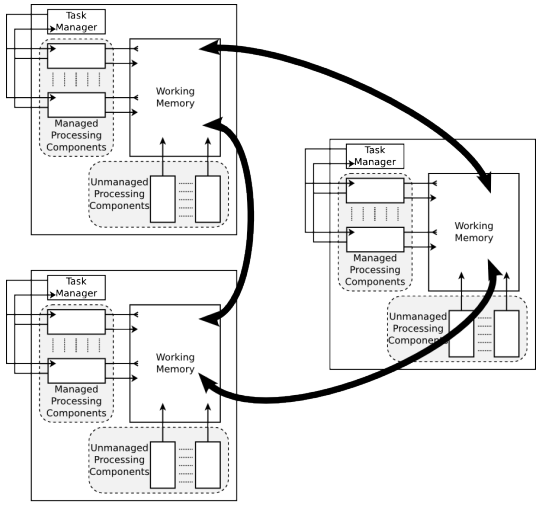
Subarchitecture groupings influence the flow of change events, and thus the flow of information between components. Within a subarchitecture, components are sent all the of change events generated by operations on that subarchitecture's working memory. They then use their filters to select the relevant events from this stream. Change events that describe operations on other working memories (i.e. those outside of the subarchitecture) are first checked against the union of all of the filters registered by components in a subarchitecture. If an event passes these filters then it is forwarded to all of the subarchitecture's components via the same mechanism used for local changes. This reuse of the existing filter mechanism adds redundancy to the change propagation mechanisms, but reduces the complexity of the system.

When a component reads information from, or writes information to, a working memory, or a change event is broadcast, a *communication event* occurs. A communication event abstracts away from the underlying communications infrastructure, hiding whether the information is being moved in memory, over a network or translated between programming languages. Within subarchitectures any operation requires a single communication event. When communication happens between two subarchitectures an additional communication event is required due to the separation (this is equivalent to

[1]For a more complete description of the schema, see our previous work [14, 13, 12].



(a) The CAS subarchitecture design schema. All information passes via the working memory.



(b) A CAS architecture based on three subarchitectures. Cross-subarchitecture communication occurs via connections between working memories.

Fig. 1. Two views of the CoSy Architecture Schema.

the information passing over one of the dark lines in Figure 1(b)).

Change and communication events are implemented as part of the CoSy Architecture Schema Toolkit (CAST) which realises the CAS schema in an open source, multi-language software framework [12]. In CAST the change event system is implemented as a callback mechanism modelled on event driven programming. The communication events are implemented as procedure calls or remote procedure calls depending on the languages and distribution methods used in the instantiation.

Using CAST we have built an integrated intelligent robot which features subarchitectures for vision, qualitative spatial reasoning (QSR), communication, continual planning, binding, manipulation and control [13, 15]. To provide a simpler system useful for exploring the design space of information sharing mechanisms, we reduced this system to a smaller number of subarchitectures: vision, binding, and QSR. This reduction was chosen because it provides a simpler system which still integrates two modalities with distinct representations (quantitative vision and qualitative spatial reasoning). For the experimental runs we replaced some of the components in the visual subarchitecture with simulated components. These not only simulated the results of visual processing, but also the interactions of the components via shared working memories (the important aspect of this study). This allowed us to fully automate interactions with the system in order to perform a large number of experimental runs. Aside from these alterations, the remaining components were taken directly from our original robotic system.

When presented with an object after a change to the visual scene, the system first determines its 3D position and then extracts some visual attributes. This information is abstracted into the binding subarchitecture where it become available in a simplified form to the rest of the system. The presence of object information in the binding subarchitecture triggers the QSR subarchitecture which computes spatial relations between the new object and any other known objects. This relational information is transmitted back to the binding subarchitecture where the relations are introduced between the existing object representations.

B. Methodology

We can use the shared memory-based design of CAS to benchmark the effects of varying information sharing patterns between components in our experimental system. We do this by altering the ratio of components to subarchitectures.

We start with an n-m design where n components are divided between m subarchitectures, where $n > m > 1$. This is our original system described above, in which components are assigned to subarchitectures based on functionality (vision, binding or QSR). We then reconfigure this system to generate architectures at two extremes of the design space for information sharing models. At one extreme we have an n-1 design in which all n components from the original system are in the same subarchitecture. At the other extreme of design space we have an n-n design in which every component is in a subarchitecture of its own. Each of these designs can be considered a schema specialisation of the CAS schema from which a full instantiation can be made.

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

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

To measure the effects of the architecture variations, we require metrics that can be used to highlight these effects. We previously presented a list of possible metrics that could be recorded in an implemented CAS system to demonstrate the trade-offs in design space [4]. Ultimately we are interested in measuring how changes to the way information is shared impacts on the external behaviour of the systems, e.g. how often it successfully completes a task. However, given the limited functionality of our experimental system, these kind of behaviour metrics are relatively uninformative. Instead we have chosen to focus on lower-level properties of the system. We have compared the systems on:

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

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

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

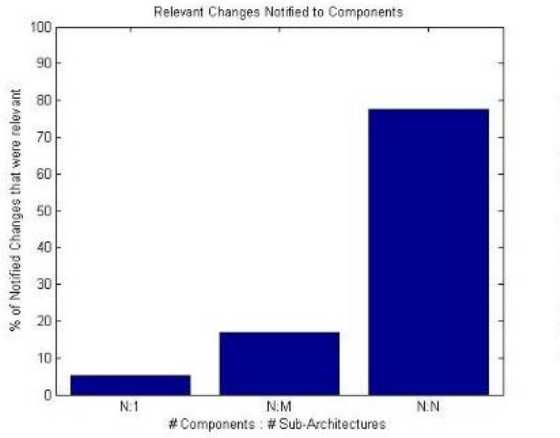


Fig. 2. Average number of relevant change events received per component.

C. Results

The results for the filtering metric are based around the notion of a *relevant event*. A relevant event is a change event that a component is filtering for (i.e. an event that it has subscribed to). Figure 2 demonstrates the percentage of relevant events received per component in each instantiation. 100% means that a component only receives change events it is listening for. A lower percentage means that the connectivity of the system allows more than the relevant change events to get the component, which then has to filter out the relevant ones. This is perfectly natural in a shared memory system. The results demonstrate that a component in an n-1 instantiation receives the lowest percentage of relevant events. This is because within a subarchitecture, all changes are broadcast to all components, requiring each component to do a lot of filtering work. A component in an n-n instantiation receives the greatest percentage of relevant changes. This is because each component is shielded by a subarchitecture working memory that only allows change events that are relevant to the attached components to pass. In the n-n case because only a single component is in each subarchitecture this number is predictably high[2]. This figure demonstrates the benefits of a directly connected instantiation: components only receive the information they need.

However, this increase in the percentage of relevant changes received comes at a cost. If we factor in the filtering operations being performed at a subarchitecture level (which could be considered as "routing" operations), we can produce a figure demonstrating the total number of filtering operations (i.e. both those at a subarchitecture and a component level) per relevant change received. This is presented in Figure 3. This

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

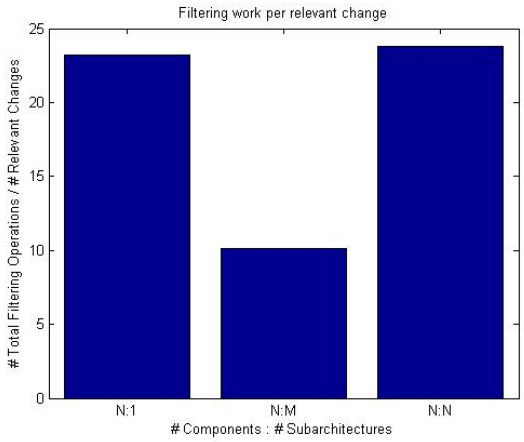


Fig. 3. Average filtering effort per relevant change event received.

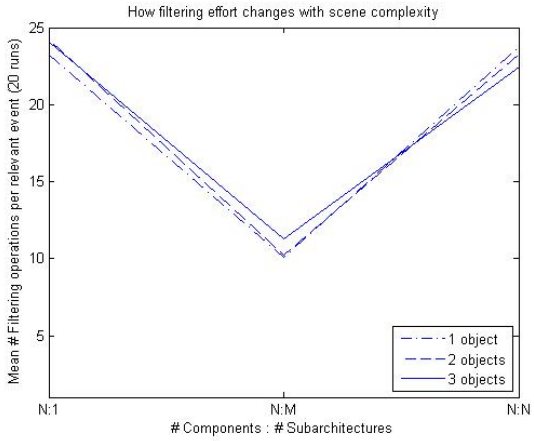


Fig. 4. Average filtering effort per relevant event compared to scene complexity.

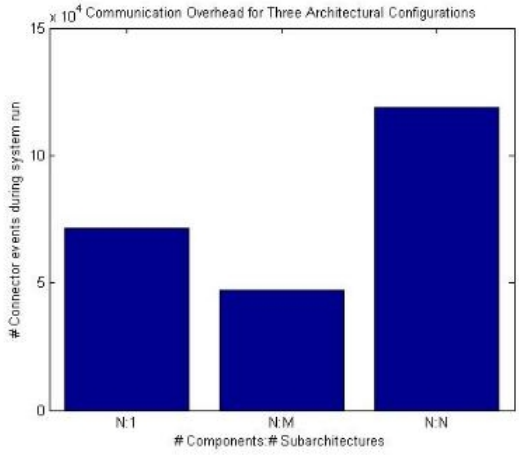


Fig. 5. Average total communication events per instantiation run.

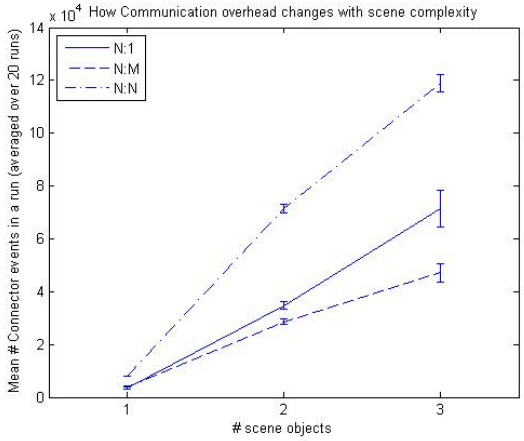


Fig. 6. Average total communication events per instantiation run compared to scene complexity.

shows a striking similarity between the results for the n-1 and n-n instantiations, both of which require a larger number of filtering operations per relevant change than the n-m instantiations. In the n-m systems, the arrangement of components into functionally themed subarchitectures results in both smaller numbers of change events being broadcast within subarchitectures (because there are fewer components in each one), and a smaller number of change events being broadcast outside of subarchitectures (because the functional grouping means that some changes are only required within particular subarchitectures). These facts mean that an individual component in an n-m instantiation receives fewer irrelevant change events that must be rejected by its filter. Conversely a component in the other instantiations must filter relevant changes from a stream of changes containing *all of the change events in the system*. In the n-1 instantiations this is because all of these changes are broadcast within a subarchitecture. In the n-n instantiations this is because all of these changes are broadcast between subarchitectures. Figure 4 shows that these results are robust against changes in the number of objects in a scene. Also, the nature of the results did not change between the systems with vision and binding components, and those with the additional QSR components.

Figure 5 demonstrates the average number of communication events per system run across the various scenes and configurations for the three different connectivity instantiations. This shows that an n-n instantiation requires approximately 4000 more communication events on average to perform the same task as the n-1 instantiation, which itself requires approximately 2000 more communication events than the n-m instantiation. Figure 6 demonstrates that this result is robust in the face of changes to the number of objects in a scene. The nature of the results also did not change between the systems with vision and binding components, and those with the additional QSR components.

This result is due to two properties of the systems. In the n-n system, every interaction between a component and a working memory (whether it's an operation on information or the propagation of a change event) requires an additional communication event. This is because all components are separated

by subarchitectures as well as working memories. In addition to this, the number of change events propagated through the systems greatly effect the amount of communication events that occur. In the n-n and n-1 instantiations, the fact that they effectively broadcast all change events throughout the system contributes significantly to the communication overhead of the system.

IX. Conclusions

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

In this simple example the existence of such a sweet spot, subject to well defined assumptions, could be established mathematically without doing any of these experiments. However, we have shown the possibility of running experiments to test such mathematical derivations, and also to deal with cases where no obvious mathematical analysis is available because of the particular features of an implementation. This experimental work also supports our argument that information-processing architectures for intelligent systems can and should be benchmarked. It is our hope that further work along these lines will provide system designers with a body of knowledge about the choices and trade-offs available in architectural design space, allowing them to build systems that satisfy their requirements in an informed and principled manner.

Acknowledgement

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

References

[1] P. Langley and J. E. Laird, "Cognitive architectures: Research issues and challenges," Institute for the Study of Learning and Expertise, Palo Alto, CA, Tech. Rep., 2002.

[2] 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*, vol. 11, no. 2, pp. 151–180, April 2007.

[3] N. Hawes, A. Sloman, and J. Wyatt, "Requirements & Designs: Asking Scientific Questions About Architectures," in *Proceedings of AISB '06: Adaptation in Artificial and Biological Systems*, vol. 2, April 2006, pp. 52–55.

[4] N. Hawes, A. Sloman, and J. Wyatt, "Towards an empirical exploration of design space," in *Proc. of the 2007 AAAI Workshop on Evaluating Architectures for Intelligence*, Vancouver, Canada, 2007.

[5] R. P. Bonasso, R. J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an architecture for intelligent, reactive agents." *J. Exp. Theor. Artif. Intell.*, vol. 9, no. 2-3, pp. 237–256, 1997.

[6] J. E. Laird, A. Newell, and P. S. Rosenbloom, "Soar: An architecture for general intelligence," *Artificial Intelligence*, vol. 33, no. 3, pp. 1–64, 1987.

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

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

[9] P. Fitzpatrick, G. Metta, and L. Natale, "Towards long-lived robot genes," *Robot. Auton. Syst.*, vol. 56, no. 1, pp. 29–45, 2008.

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

[11] B. Gerkey, R. T. Vaughan, and A. Howard, "The player/stage project: Tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, 2003, pp. 317–323.

[12] N. Hawes, M. Zillich, and J. Wyatt, "BALT & CAST: Middleware for cognitive robotics," in *Proceedings of IEEE RO-MAN 2007*, August 2007, pp. 998 – 1003. [Online]. Available: http://www.cs.bham.ac.uk/ nah/bibtex/papers/hawesetal07cast.pdf

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

[14] N. Hawes, J. Wyatt, and A. Sloman, "An architecture schema for embodied cognitive systems," University of Birmingham, School of Computer Science, Tech. Rep. CSR-06-12, November 2006. [Online]. Available: ftp://ftp.cs.bham.ac.uk/pub/tech-reports/2006/CSR-06-12.pdf

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