BALT & CAST: Middleware for Cognitive Robotics

Nick Hawes, Michael Zillich and Jeremy Wyatt
School of Computer Science, University of Birmingham
Birmingham, UK

***Abstract*—In this paper we present a toolkit for implementing architectures for intelligent robotic systems. This toolkit is based on an architecture schema (a set of architecture design rules). The purpose of both the schema and toolkit is to facilitate research into information-processing architectures for state-of-the-art intelligent robots, whilst providing engineering solutions for the development of such systems. A robotic system implemented using the toolkit is presented to demonstrate its key features.**

I. Introduction

Many researchers are working on middleware software for robotics, e.g. MARIE [1] and YARP [2]. These projects focus on supporting the connection of separate, reusable, software components, into a complete systems. To date the majority of robotics middleware has been designed to form branching pipelines of processing components that process sensor data in various ways to ultimately generate some behaviour. As we move towards robots that support a wider variety of more complex behaviours (i.e. cognitive or intelligent robots), we must consider how all the processing components in these systems communicate and interact; this problem involves studying *information-processing architectures*. In this paper we present a software toolkit based on a particular architecture design. The architecture has been designed from requirements taken from a limited set of robotic scenarios. The toolkit is intended to simplify the process of implementing systems based on this architecture, and to support a clear distinction between the scenario-specific content of the system and its information-processing architecture.

II. Architectures

The study of information-processing architectures will become more and more important as researchers aim to endow robotic systems with a wider range of more complex and integrated behaviours than is currently possible. The design for the information-processing architecture (referred to as just "architecture" for the remainder of the system) defines exactly how components are connected, controlled, monitored, and exchange information; it defines the internal structure of the system. As such it places limits on the space of possible behaviours that a system is capable of generating.

If we wish to advance the science of designing and building intelligent robots (or any system with many interconnected processing components), then it is vital that we study the architectures used to create them. To allow us to do this, it must be possible to separate the effects of the architecture on the finished system from the effects of other aspects of its design and implementation (such as the components in it, the algorithms they use, the robotic platform it is embodied within etc.). To this end we present a software toolkit which allows researchers to concentrate on developing particular types of robotic functionality, whilst the architecture used to structure this functionality is kept distinct within the system. This means it can be modified independently of this functionality and the effects of doing so can be studied [3].

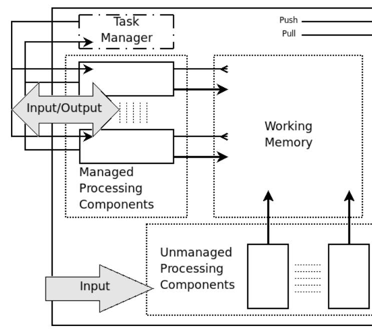

Fig. 1. The CAS Subarchitecture Design Schema.

Our approach to designing architectures is based on the following methodology. From analysis of detailed *scenarios* we derive *requirements* that lead to *design principles* for architectures that can be expressed in terms of *architectural schemata*. These define a (large) *design space* containing many specific designs: the *architectural instantiations*. The toolkit we present is based on a previously developed architecture schema. This schema is presented in Section III. Because it is based on a schema, the toolkit defines a space of possible instantiations (implemented systems) that can be developed with it. The toolkit has two parts: a layer of software that acts as connection middleware, and a layer of structure on top of this that implements our architecture schema. These are presented in Sections IV & V respectively. Following this in Section VI we discuss related work, in Section VII we present an architecture instantiation implemented using our tools, and follow this with discussion and conclusions.

III. CAS: The CoSy Architecture Schema

Rather than base our integration work around a single architecture, our work is based on an architecture *schema*. An architecture schema is a task and implementation independent set of rules for structuring processing components and information, and controlling information flow. To produce a concrete design for a system to solve a particular set of problems, it is necessary to produce scenario-specific instantiations of this schema containing all of the components for a particular scenario (e.g. parsers, segmenters, arm controllers etc.). We take this approach to provide a clear separation between our integration and architectural principles (the schema), and designs and implementations derived from these principles (the design and implementation of the instantiation of the schema). This follows the work of Sloman on niche-space and design-space [4], and is necessary to allow us to study and compare information-processing architectures without being tied to problem-specific details [3].

Analysis of a number of human-robot interaction scenarios led to three design requirements, namely support for concurrent modular processing, structured management of knowledge, and dynamic control of processing. These are met by the CoSy Architecture Schema [5]. 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 perform relatively simple processing on high-bandwidth 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 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 write 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 or across 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 (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. Instantiations of the SA schema function as distributed blackboard systems as used, for example, in Hearsay-II [6].

IV. BALT: Boxes and Lines Toolkit

An architecture is essentially a set of components, the connections between them, and some assumptions about how the connections are made and how the components behave. In our toolkit we separate the component and connection model from the other elements as this is independent of the remainder of the architecture. Doing so maintains the flexibility of the overall toolkit. In addition to this, the problem of connecting processing components is relatively well understood, and a number of software solutions exist already.

Our requirements for the connection layer are as follows: that components can run concurrently; that these components can be connected so they can share information; that this connection process should be the same if they are on the same machine or are on different machines on a network; that component connections are not hardwired into the code (to allow architectures to be changed without recompilation); that component connections can be altered at run time (to support dynamic reconfiguration of architectures); and that components can be written in C++ or Java and can be connected regardless of language. We also required the software to be fairly easy to learn and use.

We tested a variety of existing middleware solutions against this list of requirements. We looked at both robotics-focused middleware such as MARIE [1] and YARP [2], and more general solutions such as CORBA. We found that although the robotics middleware offered more succinct solutions to our problems, they generally only supported a single programming language. On the other hand the multi-language support of the more general middleware solutions was typically accompanied by a rise in the complexity of working with the software.

As a result of this we decided to implement our own middleware software to satisfy all our requirements. This produced the Boxes and Lines Toolkit (BALT), a name inspired by the ubiquitous architecture diagrams drawn by intelligent systems researchers. In brief, BALT is built on top of CORBA to provide (compile-time) typed push and pull connections between components that run in individual threads. It has native support for C++ and Java. Components can be interconnected across language and machine barriers with no change to the structures used within the component code. Connections are optimised for their end points, so same-machine same-language connections use native constructs for data exchange, whilst cross-language and cross-machine connections make use of CORBA's translation mechanisms. Because of this, non-primitive information to be exchanged must be stored in structs automatically generated from IDL by an IDL compiler. It is worth noting that CORBA has a number of limitations (e.g. IDL generated classes not support inheritance) that have led us to consider replacing it in the future.

A BALT system is run by launching a *process server* on each required machine. A configuration process is then launched which sends details of the components and connections to the necessary process servers. The configuration process can also be run as part of the start-up of a process server, so a separate step is not always necessary. The configuration information is provided via a configuration file that describes components and connections along with command-line-style configuration options.

Components in a BALT system can interact in two ways: via push connections and via pull connections. A push connection is a 1-to-N connection in which a sender component transmits data objects which are delivered to the receiver components via a parameter in member function call. A pull connection is a 1-to-1 connection in which a sender component obtains data objects from a receiver component as the return value from a member function call. All of these connections are based on typed interfaces so the validity of the connection can be checked during compilation.

V. CAST: The CoSy Architecture Schema Toolkit

On top of BALT we have built a software toolkit that allows researchers to easily implement instantiations of the previously described architecture schema, CAS. This software is the CAS Toolkit (CAST). CAST provides abstract C++ and Java classes for the key components of CAS: managed and unmanaged components (processing components), subarchitecture taskmanagers and subarchitecture working memories. CAST extends the BALT configuration interface to provide a mechanism to combine the components into subarchitectures, and for these to be combined into complete architectures.

Rather than interacting directly (as BALT components do) processing components in a CAST instantiation share information via working memories. A processing component can write data to its working memory via a number of mechanisms, all of which require that the calling component provides the data along with an ID and some type information. The data is associated with the ID in working memory, and this ID can be used in the future to access the data. The type information describes the *ontological type* of the data, rather than its run-time or compile-time type. This allows data classes to be used for different purposes within a CAST system whilst distinguishing these purposes. When data is written to a working memory, *change objects* are propagated to all subarchitecture managed components and all connected working memories which forward the objects to the managed components in their subarchitectures. These change objects contain information about the ID and type associated with the change, the component that caused the change, and the change operation type (add, overwrite or delete). Components can then use the information contained within these change objects to access the changed data.

Change objects generated as a result of a change to working memory are the primary mechanism for distributing information through the architecture. To reduce the amount of redundant information that is broadcast to all components, change objects can be filtered by both components and the working memories they are attached to. Coarse grained filtering is provided at the level of working memories which can be configured to forward changes to and from other subarchitectures or not. Finer grained filtering is provided at the component level based on the ontological type and memory operation of the change, and whether it originated in an external subarchitecture.

The task-based control mechanism provided by the architecture schema is realised in connections between managed components and a subarchitecture task manager. Components must propose a task that they wish to execute. This proposal can then be accepted or rejected by the task manager. Currently this is not strictly binding, as components can read and write to working memory without having a task proposal accepted.

In order to provide a rough idea of how the toolkit performs we have created a simple benchmarking subarchitecture that consists of a single working memory and pairs of components. These pairs consist of a writer component that writes an array of 1000 bytes to the working memory and a reader component that reads this from the working memory and then deletes it. When the writer receives notification of the deletion the processing cycle is complete and another is started by the writer writing another array to working memory. One of these cycles is intended to represent a typical subarchitecture interaction. It features three working memory operations (an add, a get and a delete) and two change objects being generated (one for the add, one for the delete). To benchmark the basic CAST system we counted the number of cycles a pair of components could complete in a second. This was done for the various combinations of C++ and Java CAST elements (where an element can be a reader, writer or the working memory connecting them). Space does not permit a detailed presentation of these results, but in general when the components were all written in the same language the number of cycles per second was in the range of 7000 to 10000 cycles per second. When the components were a mix of languages (on the same machine) the range was around 100 to 300 cycles per second. In future work we will evaluate the performance of CAST in more detail, in particular investigating the speed difference between same- and different-language communication. Currently we suspect the translation mechanisms in CORBA to be part of the reason for this discrepancy.

VI. Related Work

The work presented in this paper can be compared to two main existing areas of research. The first of these is is the work on cognitive architectures that also provide toolkits for implementing systems using these architectures. Such work includes ACT-R [7] and SOAR [8]. 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.

The second area of research that our work can be compared to is that of robotic middleware. Such work includes MARIE [1] and YARP [2]. These systems provide the means of connecting processing components in a distributed manner, and they also typically provide a collection of components to use in developed systems. Although BALT is comparable to the connection aspects of these tools, CAST's support for a space of possible architecture instantiations sets it apart from connection-orientated middleware. This is both a strength and a weakness; it is possible to implement the same architectures and more (i.e. those that fall beyond the CAS schema) with these middleware tools, but the time taken do so would be greater than if you were using a dedicated tool such as CAST.

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 [9] 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 these projects. Our work is perhaps most similar to the agent architecture development environment ADE [10]. 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.

VII. An Example System

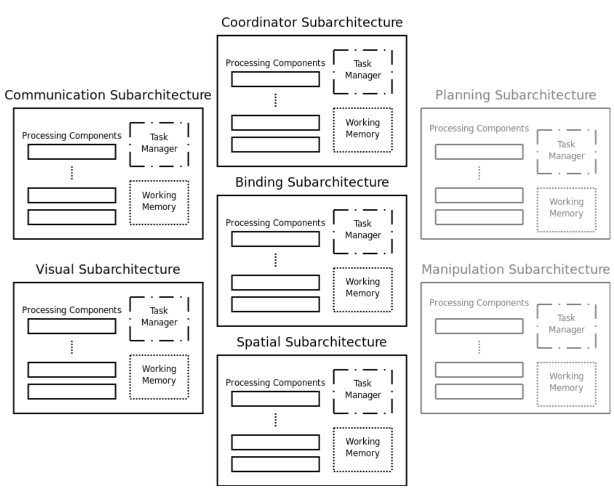

Fig. 2. The example system subarchitectures.

We have used CAST to develop an integrated system for a European integrated project [11]. The system is a linguistically-driven tabletop manipulator that combines state-of-the-art subarchitectures for cross-modal language interpretation and generation [12] and visual property learning [13] 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 [14] to allow it to carry out manipulation commands that feature descriptions based on learnt visual properties.

A	Red object placed on table.
B	Tutor (T): "This is a red thing."
C	Red object replaced with blue object.
D	Robot (R): "Is that red?"
E	T: "No, this is a blue thing."
F	Blue object replaced with red object.
G	Blue object placed to right of red object.
H	Blue object placed to left of red object.
I	T: "Put the blue things to the left of the red thing."
J	R moves right hand blue object to left of red object.

Fig. 3. Events from the example system run

The implemented system features seven subarchitectures. This includes seven working memory components (two in C++, five in Java), seven task manager components (two in C++, five in Java), three unmanaged components (two in C++, one in Java) and twenty-eight managed components (ten in C++, eighteen in Java). All of these are sub-classed from CAST classes. In more detail the subarchitectures and components are as follows: the communication SA containing components for speech recognition, parsing, dialogue interpretation, dialogue production and speech synthesis; the vision SA (VSA) containing components for change detection, segmentation, and visual property learning; the binding SA which provides mediated information exchange between the other subarchitectures (cf. [12]); the spatial SA containing components for representing the current scene, and components for adding spatial relationships to the current scene; the planning subarchitecture containing components for planning, problem generation, plan execution monitoring; and the manipulation subarchitecture containing a single component for translating planned actions into arm behaviour and visual servoing; and the control subarchitecture containing components for motive generation and management. A typical interaction with the implemented system is documented in Figure 3, with internal timing data from the system shown in Figure 4.

The ability to easily reconfigure CAST instantiations allowed the development of the system to occur in stages across four implementation sites. Each site typically developed a subarchitecture as a standalone system which was then added into an instantiation via the configuration interface. New components were simply added to subarchitectures in a similar manner: by adding lines to a configuration file. Using this approach our integrated system was developed in two stages. The first stage was to develop a system that could learn about the objects in its world at answer questions about them. This system is represented by the dark lines in Figure 2. Once this was complete we added in the subarchitectures and extra components necessary to generate and follow plans for manipulating objects. This system is represented by the lighter lines in Figure 2.

One of the strengths of the architecture schema implemented by CAST is its support for the parallel development of representations across multiple subarchitectures. This is

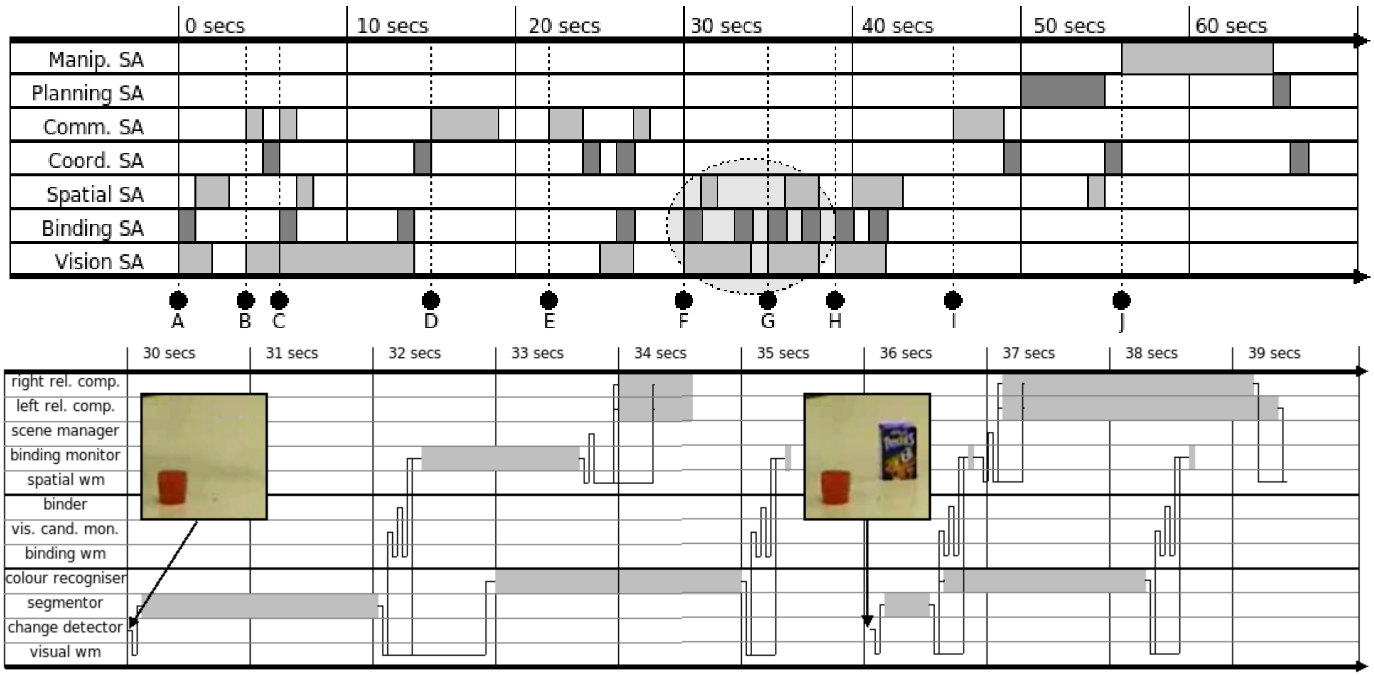

Fig. 4. How processing occurs across SAs in the example. The lower diagram contains a detailed view of the circled area from the upper one. Grey areas represent processing. Black lines on the lower diagram represent data exchanged via WMs.

illustrated by the highlighted region of the upper timeline in Figure 4, which is shown in more detail in the lower timeline. In this example one object is placed in front of the system, then another object is placed to the right of it. When the first object is placed in front of the system the change detection component is triggered, causing a scene changed fact (which also contains information about the time of change, camera id etc.) to be added to the visual working memory (VWM). The appearance of this fact causes the segmenter component to run on the changed scene, which results in a new region-of-interest (ROI) being generated along with a related proto-object (PO). Once these data structures are in working memory then the other components in the architecture start to process them in parallel. In the VSA the property learning components extract features from the ROI which are then added into the data structure. The presence of these features trigger the recogniser which add any recognised property data. In the wider architecture, the presence of the PO causes a binding candidate (BC) to be generated for it in the binding subarchitecture. This is then bound into an instance binding (IB) with other BCs from other subarchitectures. The presence of a new, visible, bound candidate in the binding working memory triggers the generation of a new spatial reference object in the spatial working memory.

When the second object is added to the scene to the right of the blue object, this triggers the change detection and segmentation components to generate an ROI and a PO. Due to the inherently concurrent nature of the design, this can happen in parallel with the processing being performed on the initial blue object. The creation of data structures for the second object in VWM triggers further processing (feature extraction etc.). As long as these processing components are not currently processing other data-structures they are free to process the second object's data structures. When an IB is generated for the red object, the spatial subarchitecture extends its scene to include a new SO. The presence of more than one SO causes the spatial relationship components to start annotating the current scene representation with contextually appropriate spatial relationships. In this case relationships are added to state that the red object is to the right of the blue object and the blue object is to the left of the red one. Relationships are added in parallel to the further processing occurring in the VSA, allowing an understanding of the scene to be built up incrementally in parallel across the architecture.

This example demonstrates the importance of the support for parallelism and incrementality in both the architecture schema and the toolkit. An architecture that ran components in serial would require approximately four seconds longer to process this example. Of course this is more a demonstration of the power of parallelism than it is the schema, but a crucial aspect of integration is managing this parallelism in terms of control and the concurrent changing of information. The latter problem is tackled by enforcing serial access to working memories and through change objects. Control is handled in the schema by subarchitecture task managers preventing components from processing at particular times. Further examples of our system's behaviour can be found in [11].

VIII. Discussion

Using CAST provides a number of advantages for researchers interested in engineering and understanding intelligent robots. As with any middleware choice these advantages come at the cost of limits on exactly what can be implemented in what manner. The principal scientific advantage of using the toolkit is that the architecture of the system is explicitly represented and is based on a number of previously researched design principles [5]. We plan to empirically explore these principles in the future by using the toolkit to vary the internal structure of instantiations of an advanced version of the system presented in Section VII whilst maintaining the same external functionality [3]. Possible variations may include placing all processing components into a single subarchitecture, placing each processing component into a subarchitecture on its own, or legioning instantiations in various ways. Comparing these variations will allows us to study the effects of architectural variation on a state-of-the-art robotic system.

On a more technical level, CAST allows researchers to quickly generate CAS architecture schema instantiations in a distributed fashion using a mix of C++ and Java with an API that remains constant regardless of programming language or component location. The ability to configure architecture instantiations via a separate interface means that systems can be composed out of different combinations of components as appropriate. The fact that these components communicate via working memories (rather than using direct connections) means that components can access information provided by new components without the need for recompilation or component-level reconfiguration. We took advantage of these configuration features of CAST to develop the previously presented system in an incremental manner.

Because the architecture of a system structures its internal connections, it necessarily limits the ways in which programmers can pass information between components. Whereas most middleware software (including BALT) is designed to allow direct communication between components, CAST deliberately eschews this in favour of communication via shared working memories. This means that some processing models are easier to implement than others. It is at this point where the science of intelligent systems and engineering of intelligent systems meet, and potentially conflict. It is therefore necessary to have a strong vision of the design and purpose of the complete system during its implementation.

One of the most critical design decisions involved in using CAST is deciding what types of information should be shared via working memories. For example, should raw sensor data (e.g. laser scans and images from cameras) be shared via working memories, or should only the results of processing such data be shared? For our current system we decided on the latter approach and connected unmanaged processing components to sensors via BALT connections (e.g. push or pull connections). This meant we avoided the overhead of having large data objects transmitted unnecessarily (when they may get written to working memory but not read) and frequently updated information written into working memories (consequently generating a large number of change objects), at the cost of allowing such information to exist *outside* of the schema. We are currently working on a design for a "robot layer" which will integrate such hardware connections (both sensors and effectors) into CAST as far as possible. We hope to build this layer on top of existing robotic middleware to exploit existing work in this field. Because such frameworks use alternative communication software we may have to replace BALT with this to limit overall software dependencies.

IX. Conclusion

In this paper we presented CAST, a toolkit for implementing architecture instantiations for intelligent robotic systems. This toolkit is based on an architecture schema, CAS. The purpose of both the schema and toolkit is to facilitate research into information-processing architectures for state-of-the-art intelligent robots, whilst providing engineering solutions for the development of such systems. The applicability of our tools to this problem was demonstrated with an example implementation featuring a robot that can learn about objects in its world and act on commands to manipulate them.

Acknowledgements

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

References

[1] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky, and F. Michaud, "Robotic software integration using MARIE," *Int. Jour. on Adv. Robot. Sys.*, vol. 3, no. 1, pp. 55–60, March 2006. [Online]. Available: http://marie.sourceforge.net/

[2] G. Metta, P. Fitzpatrick, and L. Natale, "YARP: yet another robot platform," *Int. Jour. on Adv. Robot. Sys.*, vol. 3, no. 1, March 2006.

[3] N. Hawes, A. Sloman, and J. Wyatt, "Towards an empirical exploration of design space," in *Proc. AAAI '07 Workshop on Evaluating Architectures for Intelligence*, Vancouver, Canada, 2007.

[4] A. Sloman, "The "semantics" of evolution: Trajectories and trade-offs in design space and niche space," in *Proc. IBERAMIA '98*, 1998, pp. 27–38.

[5] 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, 2006.

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

[7] J. R. Anderson, D. Bothell, M. D. Byrne, S. Douglass, C. Lebiere, and Y. Qin, "An integrated theory of the mind," *Psychological Review*, vol. 111, no. 4, pp. 1036–1060, 2004.

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

[9] J. Bach, "The micropsi agent architecture," in *Proc. of ICCM-5*, 2003, pp. 15–20.

[10] V. Andronache and M. Scheutz, "An architecture development environment for virtual and robotic agents," *Int. Jour. of Art. Int. Tools*, vol. 15, no. 2, pp. 251–286, 2006.

[11] N. Hawes, A. Sloman, J. Wyatt, M. Zillich, H. Jacobsson, G. Kruijff, M. Brenner, G. Berginc, and D. Skocaj, "Towards an integrated robot with multiple cognitive functions," in *Proc. AAAI '07*, 2007.

[12] G.-J. Kruijff, J. Kelleher, and N. Hawes, "Information fusion for visual reference resolution in dynamic situated dialogue," in *Proc. PIT '06*, E. Andre, L. Dybkjaer, W. Minker, H. Neumann, and M. Weber, Eds., 2006, pp. 117 – 128.

[13] D. Skočaj, G. Berginc, B. Ridge, A. Štimec, M. Jogan, O. Vanek, A. Leonardis, M. Hutter, and N. Hawes, "A system for continuous learning of visual concepts," in *Proc. ICVS '07*, 2007.

[14] M. Brenner, N. Hawes, J. Kelleher, and J. Wyatt, "Mediating between qualitative and quantitative representations for task-orientated human-robot interaction," in *Proc. IJCAI '07*, 2007.