CAST: Middleware for Memory-Based Architectures[*]

Nick Hawes and Marc Hanheide
Intelligent Robotics Lab,
School of Computer Science,
University of Birmingham, UK
{n.a.hawes, hanheidm}@cs.bham.ac.uk

Introduction

This short paper provides a high-level overview of the CoSy Architecture Schema Toolkit (CAST) (Hawes and Wyatt 2010; Hawes, Zillich, and Wyatt 2007). It discusses the goals the development of CAST has tried to achieve, its design, and the philosophy which underlies it.

Goals

CAST was developed to satisfy two related needs in a large-scale integrated systems project[1]: the need to rapidly develop robot architectures using a large number of heterogeneous components (often written in different languages) working across multiple modalities; and the need to maintain a close relationship between the design of an information-processing architecture (or cognitive model) and its implementation in software. Whilst the latter of these motivates CAST from a scientific perspective, it is the former which is of more interest to the robotics middleware community. However CAST is not general-purpose middleware. Instead it naturally supports a number of tuple-space-inspired component interaction patterns with a particular focus on knowledge-intensive (as opposed to sensory-motor) scenarios. As such it has seen most use in scenarios where system behaviour requires high-level processing such as planning, reasoning, language processing etc. in addition to the more traditional robotic subsystems (such as localisation, navigation, vision, manipulation etc.).

Design

The basic architectural unit in CAST is a *component*. Components are not attached to each other directly. Instead they are connected to *working memories*. Components use working memories to exchange information in the form of objects. This is done in a manner comparable to the tuple-space paradigm (e.g. (Gelernter 1985)). Each component is able to add new objects to working memory, and overwrite and delete objects which already exist. These working memory operations generate *change events* which are broadcast

[*]This research was supported by the EU FP7 IP "CogX: Cognitive Systems that Self-Understand and Self-Extend" (ICT-215181)

[1]CoSy: `http://cognitivesystems.org`

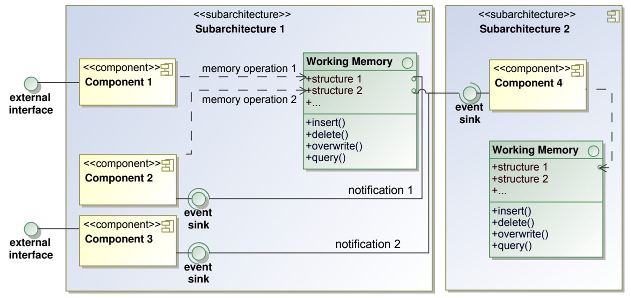

Figure 1: CAST's basic processing schema

throughout the architecture. Components can *subscribe* to events based on the operations they describe (e.g. what the operation was, who performed it, what object and datatype it was performed on etc.). On receiving a change event, a component can use the information it contains to access the referenced object on working memory and perform further processing. All components run in parallel and working memory operations are asynchronous (although change events can provide synchronisation within processing chains).

A working memory and its attached components are referred to as a *subarchitecture*. A single CAST system can be composed of one or more subarchitectures. Communication between subarchitectures is mediated by the working memories. In CAST systems subarchitectures typically (but not necessarily) group together components which process information for a single modality (e.g. vision) or system function (e.g. planning). This produces working memories which only contain certain types of information, thus providing the potential for domain-specific memory-based processing. In general, subarchitectures allow design- and run-time modularity to be enforced in a system, and provide the basis for run-time optimisations (cf. (Hawes, Wyatt, and Sloman 2009)). The relationships between the parts of a CAST system are pictured in Figure 1.

By eschewing direct connections between components, and instead following an event-based approach[2], CAST sys-

[2]For an overview of event-based systems in robotics see: `http://projects.ai.techfak.uni-bielefeld.de/ebs-ro/`.

tems are inherently *loosely coupled*. This provides flexibility when developing systems: components can be developed and debugged in isolation before being added to a subarchitecture, subarchitectures can be developed and debugged in isolation before being combined into larger systems.

Philosophy

There are two key ideas that inform the development and usage of CAST. The first is that the most important part of an integrated system is the *information* it processes. Focusing system design and development on evolution of information on working memories raises the level of abstraction of integration to a level which is well-suited for to the development of complex intelligent systems (as opposed to, e.g, financial software). On an engineering level, the focus on information and working-memory operations allows us to jointly manage data- and control-flow between components. Overall, using CAST enables *information-driven integration*, the benefits of which are explored by Wrede et al. (Wachsmuth et al. 2005; Wrede 2008).

The second key idea behind the development and usage of CAST is that many processing tasks that must be carried within an intelligent system are best designed and implemented as processes where multiple components *collaboratively refine shared information in parallel*. In this approach the result of processing by one component is available to all other related components, allowing them to behave accordingly. This is in contrast to a pipeline model where information is passed from one component to the next which then processes it in isolation.

We find that these two keys ideas are a natural fit with a system design approach that follows a *memory metaphor*. CAST working memories provide an interface for information storage, retrieval and change notification. These mechanisms define the language programmers use to develop CAST systems and provide a common view on all operations that take place within the system. Following this, we see CAST as a member of the class of *memory-based architectures*. This class also includes middleware such as Active Memory (Wrede 2008), blackboard architectures, and arguably production systems-based cognitive modelling tools.

Features

The CAST software framework supports the creation and deployment of distributed, memory-based architectures from components in both C++ and Java. It is built on top of Zero C's Ice middleware[3], allowing communication between languages and machines to occur transparently. All object types shared on working memory must be defined in an interface definition language prior to use. Objects are identified within CAST using a unique key, and working memory access is comparable to using a hash-table. Objects on working memory are protected from corruption using an "invalidate on write" consistency model (Coulouris, Dollimore, and Kindberg 2001). Access to them can be controlled and synchronised using an ownership and locking mechanism. System configuration is performed via a description file which is processed to instantiate components and connect them to working memories. CAST is available under LGPL license, and has recently passed 1000 downloads on Sourceforge. It is built using industry-standard tools such as Ice, Boost, Log4J and Log4CXX. For more information visit `http://www.cs.bham.ac.uk/go/cast` and see previous work e.g. (Hawes and Wyatt 2010; Hawes, Zillich, and Wyatt 2007).

[3]`http://www.zeroc.com/ice.html`

Conclusion

In conclusion, CAST is a middleware framework which is well-matched for problems inherent in developing intelligent systems to perform high-level reasoning using a collection of heterogeneous subsystems. Its memory-based, information-focused processing model supports the creation of loosely coupled, flexible systems and facilitates the development of complex integrated systems for use in both robotics and intelligent systems research. To date CAST has been used to develop robot systems capable of a variety of functions including table-top manipulation and human-robot interaction (Hawes et al. 2007), and self-motivated exploration and mapping (Hawes et al. 2010).

References

Coulouris, G.; Dollimore, J.; and Kindberg, T. 2001. *Distributed Systems: Concepts and Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., third edition.

Gelernter, D. 1985. Generative communication in linda. *ACM Trans. Program. Lang. Syst.* 7(1):80–112.

Hawes, N., and Wyatt, J. 2010. Engineering intelligent information-processing systems with CAST. *Adv. Eng. Inform.* 24(1):27–39.

Hawes, N.; Sloman, A.; Wyatt, J.; Zillich, M.; Jacobsson, H.; Kruijff, G.-J.; Brenner, M.; Berginc, G.; and Skočaj, D. 2007. Towards an integrated robot with multiple cognitive functions. In *AAAI 2008*, 1548 – 1553.

Hawes, N.; Hanheide, M.; Sjöö, K.; Aydemir, A.; Jensfelt, P.; Göbelbecker, M.; Brenner, M.; Zender, H.; Lison, P.; Kruijff-Korbayov, I.; Kruijff, G.-J. M.; and Zillich, M. 2010. Dora The Explorer: A motivated robot. In *AAMAS 2010*. Demo track. To appear.

Hawes, N.; Wyatt, J.; and Sloman, A. 2009. Exploring design space for an integrated intelligent system. *Knowledge-Based Systems* 22(7):509 – 515.

Hawes, N.; Zillich, M.; and Wyatt, J. 2007. BALT & CAST: Middleware for cognitive robotics. In *IEEE RO-MAN 2007*, 998 – 1003.

Wachsmuth, S.; Wrede, S.; Hanheide, M.; and Bauckhage, C. 2005. An active memory model for cognitive computer vision systems. *KI-Journal, Special Issue on Cognitive Systems* 19(2):25–31.

Wrede, S. 2008. *An Information-Driven Architecture for Cognitive Systems Research*. Ph.D. Dissertation, Bielefeld University.