Scheduling Blackboards for Interactive Robots

Kristinn R. Thórisson,[1] Thor List,[2]
Christopher Pennock,[4] John DiPirro,[3] Freyr Magnússon[1]

[1]Reykjavík University, CADIA, Department of Computer Science, Ofanleiti 2, 203 Reykjavik, Iceland; thorisson(at)ru(dot)is
[2]University of Edinburgh, IPAB, JCMB, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
[4]New York University, Media Research Lab, 719 Broadway 12th floor, New York, NY 10003 U.S.A.
[3]Communicative Machines, 33 Viewforth, suite 4F1, Edinburgh, EH10 4JE, Scotland, U.K.

Abstract

An increase in systems integration, for example in humanoid robotics and intelligent environments, has called for better solutions to support multi-module integration. Blackboards can simplify construction of systems with large numbers of heterogeneous components requiring a high number of fine-grained interactions. In this paper we describe the use of a scheduling blackboard used for developing interactive robots. Our blackboards, called *whiteboards,* both simplify and extend the blackboard model in a number of significant ways. Chief among their features are: An explicit temporal model; quality of service; publish-subscribe; queries; discrete messaging; streaming data; programming language independence; as well as a number of solutions to practical issues for improving development effort and runtime performance. Whiteboards present a compelling case for the use of scheduling blackboards in robotics, where multiple functionalities and modules need to be integrated into a coherent whole.

Introduction

Blackboards (Selfridge 1959, Englemore 1986, Adler 1992) have proven a useful construct in A.I. for facilitating interaction between heterogeneous components with complex run-time behaviors. The principal idea behind blackboards is modularity: A problem space is broken up into problem areas, each of which has its own component or "expert" working on it, communicating with experts in other areas via the blackboard, posting information to it and reading information from it. Blackboards are thus a general way of enabling multiple logical modules to exchange information in environments with unpredictable events.

Many issues are hindering progress in A.I.; lack of computing power ranks high among them. Another is the lack of powerful development tools and methodologies that help coordinate efforts between researchers and groups: attempting to build human-level intelligence without either is bound to fail. The basic tools need to enable integration yet cannot make too many assumptions about how they are to be used. Past efforts to provide general-purpose blackboards have failed in part because their complexity was too high.

Modularity is a prime candidate as viable methodology when it comes to building large systems that integrate sensing, planning, language and action. Recent examples have confirmed the notion that modular approaches speed up the development of complex systems (cf. Simmons et al. 2003, Bischoff 2000, Martinho et al. 2000). They have also proven to facilitate the collaborations of large teams of researchers (e.g. Fink et al. 1996, 1995). Modularity played a large role in the construction of successful robotic systems such as Bischoff et al.'s HERMES robot (2000) and Simmons et al.'s robot Grace (2003), the latter of which involved the work of 5 institutions and over 20 people. Martinho et al. (2000) describe an architecture designed to facilitate modular, rapid development of theatrical agents in virtual worlds. As in many other similar systems, their approach involves splitting the processing into separate modules, each of which can be developed somewhat independently of the others. This enables parallel implementation, reducing development time. Subsequently combining the pieces to form a full system becomes a simpler task.

Message passing also facilitates the construction of modular systems. Maxwell et al. (2001) used a message-passing architecture with central memory to facilitate modular integration of several key technologies needed for constructing interactive robots. Their robots have sophisticated vision, planning and navigation routines, speech recognition, speech synthesis, and facial expression. With 10 undergraduates working on the project for 8 weeks, constructing three different robots with these features, the application of blackboards or message-based architecture to such problems is given strong support.

While the above examples show the promise of blackboard architectures, it is important to note that none of them explicitly employ blackboards. We believe this is because most blackboards have historically been designed

and used exclusively for discrete messaging. For robotics, computer vision and hearing, an important requirement is that both streaming and atomic data types be directly supported. Computer vision, for example, may require several stages of a video stream, each one being the result of a particular type of processing. The integration with discrete data types is also a necessity if decisions, which are discrete by nature, have to be made based on the visual processing. Such systems can become quite complex, especially during their development stages. It is therefore critical that the development framework support easy integration of the two. However, it is conceptually very difficult to meet this requirement. An ideal system would make it possible to freely mix data types together during the development of a systems without introducing undue runtime overhead, complex architecture or difficulties in use. For example, it is not uncommon that modifications of a computer vision system under development require frequent changes in the flow and stages of video streams, mixing discrete events into the processing and enabling event-driven processing of streams.

A recent A.I. Magazine article about the annual RoboCup competition (Pagello et al. 2004) states that integration is "one of the biggest challenges remaining" in the field. An increased interest in systems integration, for example in humanoid robotics and interactive environments, is calling for better solutions in support for integration.

With roots dating back to the early days of A.I. and the bulk of work done in the 80s, it may seem a step backward to revisit the blackboard idea. However, current practice indicates otherwise. With an eye to the present state and recent developments in the field, this paper describes a type of blackboard called a whiteboard[1] that retains the benefits of blackboards already mentioned but addresses the above deficiencies using ideas from network routing and semantic web technologies. Rather than making detailed assumptions about the nature of their usage in A.I., whiteboards impose only very general design principles on the systems they are used in and assume a generic modular software development methodology. Multiple operating systems, programming languages and the proliferation of cheap computers and networks, makes the need for a general approach to integration of distributed systems in A.I. more important than ever. Whiteboards try to exploit this opportunity and answer the need for basic data exchange mechanisms, while being sufficiently powerful for use in complex integration projects.

We will begin with a review of related work and then describe the main features and benefits of whiteboards.

Related Work

The history of blackboards goes back to Selfridge's Pandemonium system (1959). Since then a host of systems have been built around the concept of agents or daemons with a shared memory, with significant advances being made in the mid- and late 1980s (Adler 1992). The agents in blackboard systems have been referred to by various terms, including "knowledge sources", "daemons" and "experts", the terms chosen being related to their intended role in their respective systems.

A number of past systems were focused on mechanisms and representations particular to specific domains. For example, HEARSAY-II (Erman 1980), one of the earliest blackboard systems, was specifically built for speech recognition. GBB (Corkill et al. 1987) provides general mechanisms data passing and module triggering, but goes to great lengths to provide support for spatial representations and data handling.

Recently a revived interest in agent-based architectures has lead to a focus on distributed systems. Several notable projects have been started in recent years focusing on agent-based communication, among them KQML[2] (Knowledge Query and Markup Language) (Mayfield et al. 1995) and there have been more recent efforts related to grid computing.[3] Distributed systems share many features with blackboards, though with notable exceptions. One of the differences is that in systems with distributed agents the agents know much more about basic communication than the experts in older blackboard systems. This means that distributed agents are in less obvious need of the "smart routing" that scheduling blackboards provide. These can be classified into object-oriented and message-oriented approaches.

Object-Orientation

CORBA[4] is a relatively general technology that allows transparent communication between programs running on multiple computers that are written in different languages. CORBA takes the object-oriented approach: An object makes a request for a service or for information, and this request is brokered by a central server, simulating an extended function call, or Remote Method Invocation (RMI). Java has a similar facility built-in, for bridging between separate Java programs.

The RMI mechanism works well for systems that can assume a larger temporal granularity than the network can provide. In many real-time systems, however, this assumption is insufficient: The clock of the real world keeps ticking on with no regard for delays in memory access, cpu slowdowns or network delays. This calls for additional mechanisms that deal explicitly with time, starting with the basic step of a particular temporal assumptions (e.g. a global clock versus distributed synchronization) and explicit tracking of time. An

[1] Whiteboards are so called because they are the modern descendant of blackboards.

[2] http://www.cs.umbc.edu/kqml/

[3] Cf. http://www.naradabrokering.org/

[4] http://www.omg.org/technology/documents/formal/corba_iiop.htm

extension to CORBA, Real-time CORBA,[5] is meant to address this shortcoming in the original design. However, because CORBA and other object-oriented approaches (e.g. DCOM[6]) try to make the whole system behave like one big computer program, including blocking on remote procedure calls, they can be cumbersome to deploy and debug and they do not support well systems in which real-time performance is paramount.

Message-Orientation

Object-oriented systems like CORBA only support "pull" communication. That is, modules have to poll for data at a set update rate. However, complex information processing systems require both "pull" and "push": Modules cannot always know which messages may be relevant to them, requiring push, but they will sometimes need to ask for messages which only they themselves know that they need, requiring pull. The message-based model subsumes the remote procedure invocation model and can be expected to overtake it.

The alternative to the object-oriented approach is message-based routing. KQML was an initiative that predates most of the current work in this area and provided a framework for message-based machine communication that was modeled after natural language performatives in speech-act theory (Austin 1962, Searle 1969). While KQML provided a boost for the subsequent Semantic Web[7] effort, it suffered from a semantic-pragmatic confusion: The "envelope" representation of messages and the surface representation of their content was not sufficiently cleanly separated. This has been addressed in many subsequent efforts.

Practical Issues

Narada[2] is a system that has solved numerous problems regarding message-based routing, including communication through firewalls. Narada has so far been implemented in Java. A practical problem is that many real-time applications require native C/C++ code, either for speed purposes or to create native drivers for audio and video I/O. A pure Java system thus loses some of its platform independence while at the same time possibly running more slowly than a clean native implementation would. Another issue is the footprint of the system; Narada is in many ways unwieldy – with a goal of solving a huge set of design problems the footprint has become prohibitively large for a number of uses, for example, in its dependency on Xerces,[8] Xalan[9] and about 10 other large external libraries. This limits deployment on platforms with restricted memory sizes and also makes it difficult to port the system to other programming languages. A related problem, which it shares with CORBA, is that it is not simple to set up or use.

Except for temporal accountability (see below), systems such as Elvin[10] and Open Agent Architecture (OAA)[11] (Martin et al. 1999) have, to a smaller or larger extent, addressed the above requirements. Elvin is a content-based semantic router with a central routing station. Elvin has been used in some systems with good results (Johnson et al. 2004), showing that the publish-subscribe approach is a powerful way to construct modular systems. The OAA is a hybrid architecture that relies on a special inter-agent communication language (ICL) – a logic-based declarative language that is good for expressing high-level, complex tasks and natural language expressions. While this is precisely what is needed for many A.I. applications, it requires a special-purpose parser, and makes it necessary for all agents in the system to contain this parser, which makes it harder to integrate heterogeneous components to create a single system.

This shortcoming is also shared with another similar effort from FIPA[12]: The FIPA message and routing specification uses a special syntax for messages, requiring it to use non-standard parsers. A more general way of representing the outermost "envelope" of a routed message would be to use XML. This achieves a higher level of generality in the outermost layer while allowing the content of such messages to be represented in any applicable language, including ICL.

Publish-Subscribe

With the proliferation of peer-to-peer systems, the benefits of message-based, publish-subscribe systems has been in focus recently (Baldoni et al. 2003). In pub-sub systems a module can register for a message type, and any time a message of that type is posted (by anyone in the system) the message will be delivered to the subscribed module. Among the most obvious benefits of this system is that the messages embody an explicit representation of each module's contextual behavior, and carry with it their state.

The Elvin system goes a step further and provides mechanisms for semantically routing messages to modules based on the meaning of the messages' content. This might become an important feature of the coming semantic Web (Berners-Lee et al. 2001) where the routed messages are meant for human consumption.

Explicit Temporal Representation

While all of the above approaches have pros and cons, and many may come close to providing a sufficient foundation for integration in cognitive robotics, interactive applications and other large A.I. systems, the best ones still fall short because time is by and large an

[5] http://www.cs.wustl.edu/~schmidt/TAO.html

[6] http://www.microsoft.com/com/wpaper/default.asp#DCOMpapers.

[7] http://www.w3.org/2001/sw/

[8] http://xml.apache.org/xerces2-j/

[9] http://xml.apache.org/xalan-j/

[10] http://elvin.dstc.edu.au/

[11] http://www.ai.sri.com/~oaa/

[12] http://www.fipa.org/

ignored problem in all of them: Temporal information is managed only within the agents or processing nodes themselves, not in the transmission infrastructure. This means that a receiver of a message cannot know how long ago the message was posted or how old the information is that its content is based on. Message time stamping, as well as quality of service via prioritized scheduling, are functionalities still missing in CORBA and most of the other message-based and publish-subscribe based approaches including EQUIP (Greenhalgh 2002), Elvin, OAA and the FIPA message and routing specification.

Overview of Whiteboards

Whiteboards consist of (i) a general-purpose message type format, (ii) ontologically-defined message and data stream types, and (iii) specifications for routing between system components. Whiteboards differ from prior traditions in blackboard construction in a few important ways. They are not built specifically for supporting reasoning and expert systems but are somewhat closer to network routing technologies. Because they are intended to help connect together systems that may be running on different computers and written in different programming languages, they use an explicit message wrapper layer and employ only lightweight semantics for triggering of routing events. In that aspect they are closer to a classical publish-subscribe information bus than blackboards.

Whiteboards do not contain special areas for data with different priorities; instead they expect that all data be tagged with a priority and requested quality of service. Another important difference is that they are even more independent from their many potential applications than most of the general-purpose blackboards systems, such as HEARSAY-III (Erman et al. 1981) and BB1 (Hayes-Roth et al. 1984). We hope this will make their application relevant to a larger set of problems and, much as the simplicity and transparency of HTML helped with its adoption, will increase their usage as a coordinating mechanism between researchers and research teams.

Whiteboards have been designed to have an explicit representation of time. This supports the development of interactive A.I. systems where data posting and reception must be non-blocking. Time is an integral part of the message format: Interactivity requires that the protocol have clear temporal accountability, which it does. In traditional blackboard systems modules post data to a central server and that data is either delivered to subscribed modules by the blackboard (scheduling blackboard) or the modules poll for it. Whiteboards enable both. They also provide a localized recording of all system events, system history (as far back as needed), and runtime state. This, combined with polling and simple but powerful query semantics, make the whiteboard function as a database of all past events.

Many early blackboards acted as a “global memory” using only one blackboard. We recommend using whiteboards somewhat like packages in object-oriented programming languages like Java, i.e. to isolate logically related information for purposes of easing system construction, and enabling a kind of “wide-area local” memory. The separate message wrapper semantics designed for network routing makes this relatively straight forward. The remote feature of whiteboards has, for example, proven useful in allowing easy transmission for off-board processing of video and audio gathered from the robot’s body.

To summarize, the main features of whiteboards are:

- Direct addressing as well as publish-subscribe
- Ontologically-defined message and stream types
- Simple but powerful message semantics
- Explicit temporal representation
- Message-type mechanism supports easy subscriptions
- Scheduling prioritization built-in
- Streams and messages with common publish-subscribe semantics

Defining a Whiteboard

A whiteboard is defined by a few parameters:

- Name
- Priority
- Maximum number of messages
- Streams & buffer sizes
- Location of whiteboard: Local or remote

A whiteboard has a globally unique name, given to it by the system’s designer. The maximum number of messages is an integer. Priority of a whiteboard can be set relative to other whiteboards to determine which gets to run first, which becomes significant when there is CPU starvation. Based on the assumption that message delivery is always the highest priority in the system, whiteboards automatically have higher priority than all other system components.

Stream buffer size has two limits, a soft and a hard. The soft limit is used when data is not locked. The hard limit is used when someone is using the data; when data on a stream is marked as being 'in use' by a module the whiteboard will not delete the old data even when the size grows above the soft limit, only when it reaches the hard limit. Information in the streams describes what kind of binary data they contain.

A whiteboard can be instantiated either on the machine where it is configured and or it can be configured on one machine but instantiated on another. Having this option as a part of the whiteboard makes it easy to specify systems where multiple whiteboards are used to mediate information flow between various components on a network.

Message & Stream Semantics

The semantics of whiteboard messages are intentionally simple, with few required slots. Messages have a content slot that holds the main body of the message, a GUID,[13] and timestamps that make it possible for both the runtime system (message posters and readers) and human programmers to understand the temporal evolution of the system as a whole. The timestamps are:

- postedTime
- receivedTime

Posted time is the precise point in time when the message left the sender; the received time is the precise point in time at which the recipient (whiteboard or other module) received the message.

Taken together, the GUID and timestamps can be used to track and query the full history of the runtime system. This feature can be used by the human designer or by the system itself (i.e. the modules).

Since the whiteboards are always the receiver of a message, subscriptions are handled through a special type of message called a *wakeup message*, which functions as follows: Upon receiving a message M of type T, the whiteboard will forward it to anyone who is subscribed to messages of type T; message M will be the content of the wakeup message. This arrangement greatly simplifies message semantics because only two timestamps must be interpreted to see what has happened during the transmission of any message.

At transmission time, messages are represented as an XML structure. Messages can also handle binary attachments. By using XML libraries for posting, subscribing and receiving, messages can be implemented in any programming language. Messages have an optional priority which enables whiteboards to decide which messages, if any, it should let take precedence over others when transmitting them to subscribers.

As already mentioned, all messages and data streams have a type. The type allows full input and output definition for the modules, thus embodying an explicit representation of the modules' behavior in context – messages effectively implement the modules' APIs. Types are represented as dot-delimited lists, where each delimited segment is ontologically defined. To take an example, for a message whose content contains words from a speech recognizer based on some processed audio input, the message type would start with `Input`, because the content of the message is a further processing on some input received by the system. A further specification of this message type could be `Input.Perc.UniM.Hear.Voice.Speak.Sentence`, where `Perc` means perception, `UniM` means unimodal (i.e. a single mode like hearing), `Voice` means that the sound was determined to come from a human voice, etc. As can be seen, the specification goes from the most general to the most specific descriptors, left to right, respectively. In this way, modules can subscribe to a message type by specifying it to any level of detail. For example, a module interested in all messages related to human voices would simply subscribe to `Input.Perc.UniM.Hear.Voice.*`. This mechanism makes it easy to incrementally build and debug of complex, interactive systems with multiple levels of abstraction.

[13] Global Unique IDentifier; a string that globally and uniquely identifies the message it is attached to.

The opportunity exists to define explicit ontologies for a large set of concepts that would standardize a wide variety of information channel types, carrying either discrete messages or streaming data. To further facilitate such ontological definitions, namespace indicators may be put at the top of the tree; for example, `RU.S1.Input.Perc` would indicate that the segments following `RU.S1` are defined in a message type ontology made by Reykjavik University called "S1".

This approach to message types is simple to use and easily human-readable while still leveraging the power of ontological definitions. This makes interchange between systems with different message types easier to manage, both manually and automatically. The main principles of whiteboard messages and their transmission are adopted from the OpenAIR[14] specification, an open-source message format and routing specification.

Lastly, messages have a slot for language. Its semantics match those of KQML messages: It is intended to tell the receiving system how the content of message is encoded. Our language tag also has a version slot which can be used to indicate e.g. dialects. For example, if the Language slot holds the string "LISP" the version tag could be used to specify e.g. "CommonLISP".

In addition to handling discrete messages, whiteboards employ a mechanism for handling streams of data such as audio and video. These are circular buffers, supporting both pushing and polling of data, temporal and content-based searches, and alarms. Alarms are notification of events such as buffers starting and stopping or the arrival of time-critical information.

Publishing Messages & Streams

Modules publish messages to whiteboards via a very simple function call. The parameters are: the name of the whiteboard (modules must know the IP address and port of the whiteboard to do this – see discussion on component lookup below), the type of the message being posted, cc slot (if any module should be forced to receive the message, see below), and the content of the message, if any.

Unlike messages, streams must be defined as part of a whiteboard's specification before runtime. This has to do with issues of safety in memory usage, among other things. However, streams also have a type, using the same dot-delimited syntax as messages, and modules can reference this type to publish data to a stream.

[14] http://www.mindmakers.org/openair/

Subscribing to Messages & Streams

To subscribe to a message of a particular type on a particular whiteboard, a system component sends a message of type `AIR.Subscribe`[15] with content in the following form:

```
<triggers>
  <trigger from="Whiteboard-1" type="Input.Perc.*"/>
</triggers>
```

This would register a subscription with a whiteboard of the name "Whiteboard-1" for all messages of type `Input.Perc.*`, where "*" indicates a wildcard. A new subscription overrides any existing one. The tag is called "trigger" because any message of this type coming into the whiteboard will trigger it to dispatch it to the subscribed modules.

Subscribing to streams is very similar to message subscription. The XML for a stream subscription is of the form:

```
<stream name="input" source="Input.Audio.Perception"/>
```

This would register a subscription with the whiteboard that owns the `Input.Audio.Perception` stream, and would allow the registered module to receive a feed from the stream. Streams have global unique types because it makes it easy for the developer to transfer them between whiteboards without having to change the code of the modules that use them.

Matching for Routing Subscribed Data

Most semantic data routing approaches used in publish-subscribe systems use a large set of structures to match on. That is, an incoming (published) data structure is matched against all of the subscription profiles. These structures are ontologically defined and are often represented in XML. This matching requires significant computation, because every incoming XML representation requires both string matching and ontological lookup. This very heavy method has value predominantly in a semantic Web environment where the publishers and subscribers of data are not known a priori. Beyond the significant ontological overhead, this also requires an efficient mechanism to support the discovery of various ontologically defined services.

In most closed development environments, however, such as those found in many A.I. labs, this approach is overkill, since it requires significant development times and introduces runtime overhead. The dot-delimited multiple tree structure used in whiteboards eliminates a significant amount of the overhead associated with the processing time required for matching, yet retains the benefits of well-defined classifications of the data contained in the message and streams. In a small development environment of less than 10 people, and perhaps upwards of 50 to 100 people, this approach presents a favorable alternative.

[15] All messages related to the OpenAIR protocol start with "`AIR.`".

Routing & Priority Model

In the whiteboard model, all messages travel between modules via a whiteboard. A message is therefore always addressed *to* a whiteboard. However, unlike in the FIPA architecture specification, modules can also communicate with each other directly. To satisfy the timing issues and the "bookkeeping" role that a whiteboard must serve, direct routing is done via a *cc* (from the postal mail vernacular "carbon copy"). Any module that is listed on the cc list of a message will get that message, whether it is subscribed to its type or not. This way modules can post "private" messages to each other, for purposes of efficiency, yet a whiteboard will always get a copy of the message, retaining the benefits of local access to all system events.

All modules in a whiteboard system have unique names, and therefore all messages can be traced to a unique source and a point in time.

Implementation & Use in Robotics

We have incorporated whiteboards into an integrated framework (Thórisson et al. 2005) that we are currently using for various projects at R.U. Among the systems for which we have used whiteboards extensively during development is Skundar, a mobile robot platform with a WiFi connection, designed and built at Reykjavik University. The platform's hardware is based on simple, modular components. It includes two motors with wheels for mobility (from motorized scooters) and a motor control interface[16] using the I^2C protocol,[17] a PC motherboard, a USB webcam for video capture and two optical USB mice for tracking movements on the floor, with the mouse for sensing movement, forwards and backwards, being placed between the drive gears, the other sensing rotation being placed at the robot's front.[18] All components are off-the-shelf.

Skundar does local processing of data from the sensors and issues commands to the motors. We use the WiFi connection to do off-board processing, making it possible to expand the robot's "brain" as the need arises. 12 dual-processor Intel processor computers running Linux[19] are used to process data such as audio and video from the robot, and to generate higher-level plans.

Our software implementation includes a mixture of Java and C++ software modules. (In combination with the OpenAIR specification whiteboards themselves are programming language independent.) At first we had five main components (Figure 1) that communicated via whiteboards running the in the Psyclone software. The

[16] MD22

[17] I^2C ("I-squared-C") is a two-line bus that is widely used to connect chips together on a printed circuit board.

[18] The full spec is available in Icelandic at: http://wiki.isir.is/wiki/Skundar_aka_Véldýrið_Pottormur

[19] Linux Gentoo 2.6.

Rob0Mouse module uses the *libgii*[20] code library to monitor the mice and their movements. It reads information from the mice as MouseMotion messages. This module posts mouse data updates at 10 Hz, whether the robot moves or not. Rob0Tracker (Tracker) receives MouseMotion messages and computes the direction and displacement of the robot. It posts posts this information as Track messages.

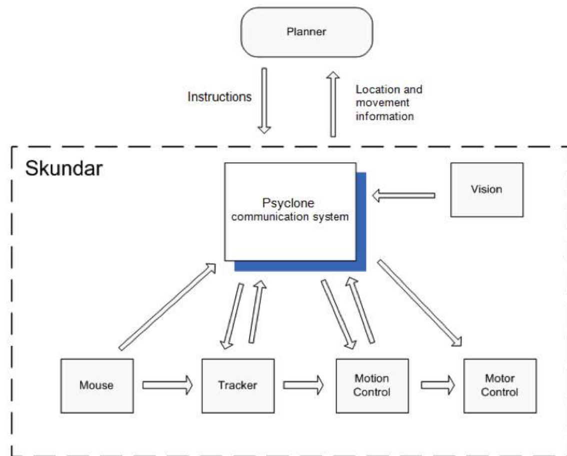

FIGURE 1. The origininal 5 modules implemented in the Skundar robot, with all interaction (except vision) being handled via discrete OpenAIR messages.

The motion control component, Rob0MotionControl, controls the motors. It subscribes to Track message types that contain Skundar's direction and speed, as well as a Cmd2dMove message that contains information about where it should be heading and how fast. The Rob0MotionControl module uses fuzzy logic (Zadeh 1994) to regulate speed and direction. The motor control component, Rob0MotorControl, controls the MD22 through I^2C. It sends instructions to the motors via DiffMotion messages.

The vision component, Rob0Vision, accepts data from video4linux driver and sends them to Psyclone that then sends them to another computer for further processing. It uses libjpeg to compress the photos. Rob0Vision is a library that runs inside the Psyclone server (internal module) and therefore doesn't create any network traffic.

The functionality and number of initial modules was determined by simply breaking the low-level problems of the robot's behavior into intuitive components, taking operational constraints into account such as the limited-bandwidth WiFi link and the limited power of the on-board processor, following the guidelines of the Constructionist Design Methodology.

After the design had been implemented and tested we found that the design created a bottleneck for some of the necessary processing, compromising the the safe operation of the robot. More specifically, the transmission of messages between modules was not being handled as fast as we had wanted. This result was not foreseen – we had not tested the system with any comparable configuration.

Since we had cleanly modularized the functions of the robot this problem was easily remedied through of one of the main benefits of using whiteboards during design and development: We combined the Mouse and Tracker components into a single component, removing the need to pass messages between them. As this new module was written in C++ we also effortlessly converted it into an internal module in Psyclone (Picture 2). By doing this we reduced the message transfer necessary (what was explicit messages before was now handled via variable passing inside the same executable). By making it an internal module all communication with whiteboards is further reduced to pointer arithmetic – much faster than the prior solution, which necessarily relied on TCP/IP messages. On the negative side, however, developers lose the ability to monitor the communication between the Mouse and Tracker modules at runtime. Also, raw Mouse data is lost from the system.

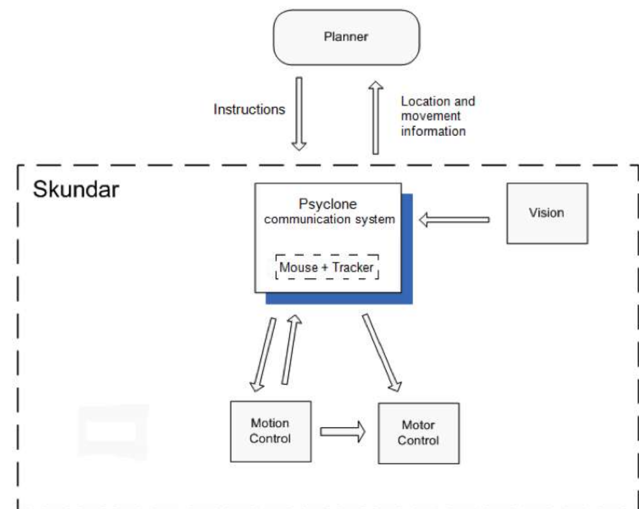

FIGURE 2. The modified robot sensing and control architecture after analysis of the power of the processor onboard the robot was deemed insufficient to handle the data traffic of the initial design. Compared to many alternatives, the use of whiteboards made the change easy, as all interfaces between modules have already been directly externalized through messages.

Whiteboards are also being used in a nation-wide "Garage A.I." movement.[21] Among these is the construction of a robot with an on-board motherboard with WiFi. It is being used to stream data from a microphone, camera and computer mice, and integrate this with motion control, navigation and speech output. Whiteboards have also been evaluated in a comparison to the CAVIAR computer

[20] http://www.ibiblio.org/ggicore/packages/libgii.html

[21] http://ailab.ru.is/projects/garageai

vision architecture, with positive results (List et al. 2005). Mixing binary streams and messages freely makes it significantly easier to build computer vision systems than with more monolithic approaches.

We have incorporated whiteboards into an integrated framework (Thórisson 2005) that we are currently using for various projects at Reykjavík University. One involves the integration of speech recognition and synthesis and agents inhabiting a virtual world. The components are written in various languages. The use of whiteboard constructs significantly simplifies connecting them together into a larger system. The use of semantically tagged messages and streams is proving to be a very powerful mechanism, especially for the A.I. parts of the applications.

Conclusion

We have presented an updated version of scheduling blackboards called *whiteboards*. Whiteboards address several deficiencies of earlier blackboard systems, while also simplifying their functionality.

The whiteboards allow a level of modularity in the construction of complex systems that is unmatched by most other approaches; we believe that increased modularity – even coarse-grained modularity – in current A.I. efforts could help the field tremendously by allowing people to integrate each others' work more easily. Blackboards in general, and whiteboards in particular, can play a role in facilitating this. The kind of modularity supported by whiteboards does not impose a particular school of thought on the architectures that can be built with it. There is for example nothing in the nature of whiteboards that prevents them from being used in behavior-based A.I. (c.f. Brooks 1991), "good old fashioned" A.I., and hybrid systems. Whiteboards embody a tried and tested approach to software development. They present a structured way of thinking about the elements of a system and their interactions that makes them more explicit than most other approaches.

For this reason interface standardization for modules is important. Among the interface issues, standardizing message format is key, along with creation of message type ontologies. An effort is already underway towards this – a specification called OpenAIR.[22] With its lean yet relatively complete message and routing specification, OpenAIR provides uniformity in message APIs, including temporality, and may be used across various A.I. efforts.

The small footprint and overhead of whiteboards, which they owe mostly to the simplicity of their specification, also makes them attractive to employ, even in embedded systems.

Conceptual clarity of the routing model and ease of use are key components for wide adoption of any specification. We believe that whiteboards meet these needs and we hope to see others starting to use modularity and whiteboards to facilitate collaboration and build larger A.I. systems that come closer to representing the complexity found in intelligent systems in nature.

[22] http://www.mindmakers.org/openair

Acknowledgments

The authors would like to thank the members of MINDMAKERS.ORG for their contributions to the OpenAIR specification and Sigrun Gunnhildardottir for the illustrations of Skundar. Thanks also to Communicative Machines Inc. for the free version of Psyclone, which can be found on the Mindmakers Website.

References

Adler, R. (1992). Blackboard Systems. In S. C. Shapiro (ed.), *The Encyclopedia of Artificial Intelligence*, 2nd ed., 110-116. New York, NY: Wiley Interscience.

Austin, J. L. (1962). *How to Do Things With Words*. Cambridge: Harvard University Press.

Baldoni, R., M. Contenti, A. Virgillito (2003). The Evolution of Publish/Subscribe Communication Systems. *Future Directions of Distributed Computing*. Springer Verlag LNCS Vol. 2584.

Berners-Lee, T., J. Hendler, O. Lassila (2001). The Semantic Web. *Scientific American*, May.

Brooks, R. A. (1991). Intelligence without reason. *Proceedings of the 1991 International Joint Conference on Artificial Intelligence*, Sydney, 569-595.

Corkill, D. D., K. Q. Gallagher, P. M. Johnson (1987). Achieving Flexibility, Efficiency, and Generality in Blackboard Architectures. *Proceedings of the Sixth National Conference on A.I.*, Seattle, Washington.

Dodhiawala, R. T. 1989. Blackboard Systems in Real-Time Problem Solving. In Jagannathan, V., Dodhiawala, R. & Baum, L. S. (eds.), *Blackboard Architectures and Applications*, 181-191. Boston: Academic Press, Inc.

Engelmore, R., and A. Morgan, eds. (1986). *Blackboard Systems*. Reading, Mass.: Addison-Wesley.

Erman, L. D., F. Hayes-Roth, V. R. Lesser, D. R. Reddy (1980). The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty. *ACM Computer Surveys*, **12**, 213-253.

Erman, L. D., P. E. London, S. F. Fickas, (1981). The Design and an Example of Use of HEARSAY-III. *Proceedings of the Seventh International Joint*

Conference on Artificial Intelligence, Vancouver, BC, 409-415.

Fink, G. A., Jungclaus, N., Kummer, F., Ritter, H., Sagerer, G. (1996). A Distributed System for Integrated Speech and Image Understanding. *International Symposium on Artificial Intelligence*, 117-126, Cancun, Mexico.

Fink, G. A., N. Jungclaus, H. Ritter, G. Saegerer (1995). A Communication Framework for Heterogeneous Distributed Pattern Analysis. *International Conference on Algorithms and Architectures for Parallel Processing*, 881-890, Brisbane, Australia.

Greenhalgh, C. (2002). EQUIP: A Software Platform for Distributed Interactive Systems. Technical Report Equator-02-002, University of Nottingham.

Hayes-Roth, B. (1984). BB1: An Architecture for Blackboard Systems that Control, Explain, and Learn about Their Own Behavior. Technical Report HPP-84-16, Stanford University, Stanford, California.

Johnson, W. L., S. Marsella, N. Mote, M. Si, H. Vilhjálmsson, S. Wu (2004). Balanced Perception and Action in the Tactical Language Training System. *Workshop on Embodied Conversational Agents: Balanced Perception & Action*, July 20th, 18-25. AAMAS 2004: The Third International Joint Conference on Autonomous Agents & Multi Agent Systems, New York, July 19-23.

List, T., J. Bins, R. B. Fisher, D. Tweed, K. R. Thórisson (2005). Two Approaches to a Plug-and-Play Vision Architecture, CAVIAR & Psyclone. In K. R. Thórisson, H. Vilhjalmsson & S. Marsela (Eds.), *AAAI-05 Workshop on Modular Construction of Human-Like Intelligence*, Pittsburgh, PA, July 10. AAAI Technical Report WS-05-08, pp. 16-23.

Martin, D., Cheyer, A., & Moran, D. (1999). The Open Agent Architecture: A Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, **13**(1-2), 91-128.

Martinho, Paiva, C. A. & Gomes, M. R. (2000). Emotions for a Motion: Rapid Development of Believable Pathematic Agents in Intelligent Virtual Environments. *Applied Artificial Intelligence*, **14**(1), 33-68.

Maxwell, B. A., L. A. Meeden, N. S. Addo, P. Dickson, N. Fairfield, N. Johnson, E. G. Jones, S. Kim, P. Malla, M. Murphy, B. Rutter, E. Silk (2001). REAPER: A Reflexive Architecture for Perceptive Agents. *A.I. Magazine*, spring, 53-66.

Mayfield, J., Labrou, Y., and Finin. T. 1995. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Muller, and M. Tambe, editors, *Intelligent Agents II – Proceedings of the Second International Workshop on Agent Theories*, Architectures, and Languages (ATAL'95), held as part of *IJCAI '95*, Montreal, Canada, August.

Pagello, E., E. Mengegatti, A. Bredenfel, P. Costa, T. Christaller, A. Jacoff, D. Polani, M. Riedmiller, A. Saffiotti, E. Sklar, T. Tomoichi (2004). RoboCup-2003: New Scientific and Technical Advances. *A.I. Magazine,* **25**(2), 81-98. AAAI. Menlo Park, CA: AAAI Press.

Searle, J. R. (1969). *Speech Acts: An Essay in the Philosophy of Language*. Cambridge, UK: Cambridge University Press.

Selfridge, O. (1959). Pandemonium: A Paradigm for Learning. *Proceedings of Symposium on the Mechanization of Thought Processes*, 511-529.

Thórisson, K. R., H. Benko, A. Arnold, D. Abramov, S. Maskey, A. Vaseekaran (2004). Constructionist Design Methodology for Interactive Intelligences. *A.I. Magazine*, **25**(4), 70-93. Menlo Park, CA: American Association for Artificial Intelligence.

Thórisson, K. R., C. C. Pennock, T. List, J. DiPirro (2004). Artificial Intelligence in Computer Graphics: A Constructionist Approach. *Computer Graphics Quarterly*, **38**(1), 26-30, New York, February.

Thórisson, K. R., T. List, J. DiPirro, C. Pennock (2005). *A Framework for A.I. Integration*. Reykjavik University Department of Computer Science Technical Report, RUTR-CS05001.

Zadeh, L. A. (1994). Fuzzy logic, neural networks and soft computing. *Comm. ACM*, Vol. 37, No. 3, pp. 77–84.