https://ntrs.nasa.gov/search.jsp?R=19910011384 2018-03-21T22:50:22+00:00Z

N91-20697

THE AI BUS ARCHITECTURE FOR DISTRIBUTED KNOWLEDGE-BASED SYSTEMS

Dr. Roger D. Schultz and Iain Stobie

Abacus Programming Corporation, 14545 Victory Blvd., Van Nuys, CA 91411

Abstract

The AI Bus architecture is layered, distributed object-oriented framework developed to support the requirements of advanced technology programs for an order of magnitude improvement in software costs. The consequent need for highly autonomous computer systems, adaptable to new technology advances over a long lifespan, led to the design of an open architecture and toolbox for building large-scale, robust, production-quality systems. The AI Bus accommodates a mix of knowledge-based and conventional components, running on heterogeneous, distributed real-world and testbed environments.

This paper describes the concepts and design of the AI Bus architecture and its current implementation status as a Unix C++ library of reusable objects. Each high-level semi-autonomous agent process consists of a number of knowledge sources together with inter-agent communication mechanisms based on shared blackboards and message-passing acquaintances. Standard interfaces and protocols are followed for combining and validating subsystems. Dynamic probes or demons provide an event-driven means for providing active objects with shared access to resources, and each other, while not violating their security.

This work was carried out for the ALS STRESS (Space Transportation Systems Expert Systems Study) ADP 2301 & 2302, and resulted in a prototype implementation of many of the designed objects. It is now being used as the fundamental framework of Abacus' cooperative systems research project, which is examining various problem-solving mechanisms in distributed AI.

1. Introduction

The AI Bus was first developed as an approach to integrating the Space Station software, and more recently has been applied to the Advanced Launch Systems project (ALS). Both applications share requirements of a long life-time with several upgrades and high degrees of autonomy. Since the major cost in large modern software systems is that of maintenance, a major goal of the AI Bus is to provide a toolbox of reusable "plug-compatible" software objects.

In this paper we review the architecture's requirements, design and current implementation using Unix and C++. Since a particular interest is in supporting high-level models of cooperation and problem-solving, we also describe our current experimentation in these areas using the AI Bus facilities..

2. Requirenents

The AI Bus architecture was developed to meet the following requirements:

- Support cooperating, distributed systems
- Support embedded and real-time applications
- Facilitate technology upgrades during long lifetime
- Permit mixed procedural, knowledge-based and off-the-shelf components
- Support cooperation of autonomous components
- Suport control system and sensor-based applications
- Support fault-tolerant approaches
- Facilitate verification and validation

These requirements drove the design, as summarized in the Figure 1.

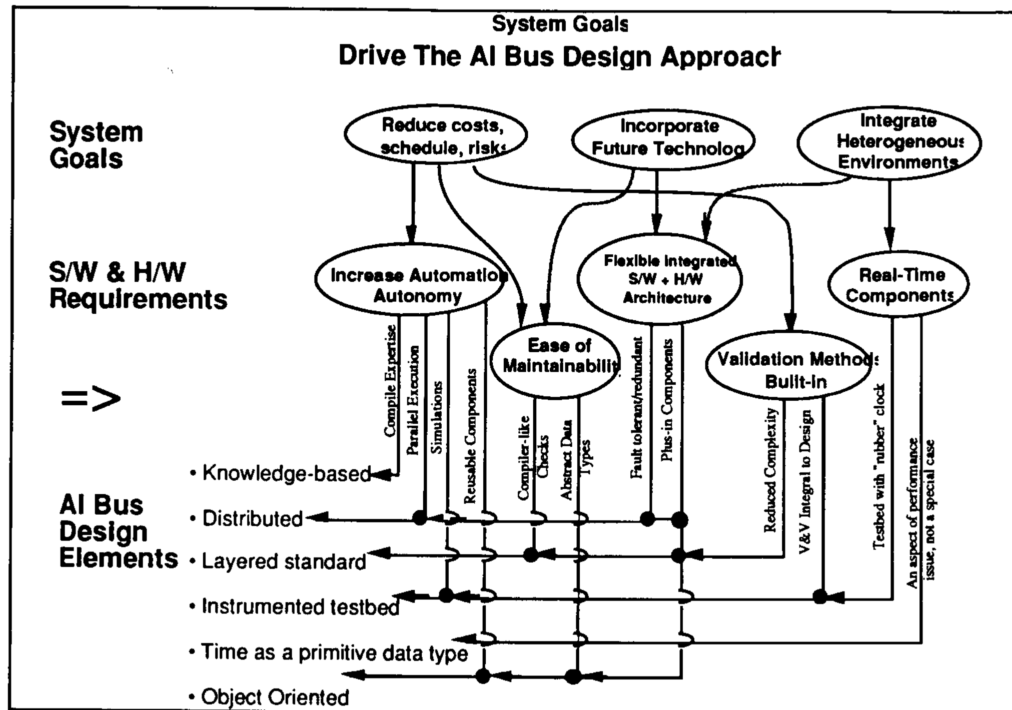

Figure 1—Origin of the AI Bus

3. Design

The design of the AI Bus is specified as an object library. The objects provide both an implied implementation architecture, and a standard interface specification. Unlike a jigsaw puzzle, where the pieces dictate one particular assembly architecture, the building block objects of the AI Bus must support multiple instances of architectures, being more like an erector set. A hardware analogy that comes close to the software framework provided by the AI Bus is that of a kit composed of a computer backplane, some standard function cards such as memory and processors, device interfaces, etc. and user customizable cards which contain the logic necessary to interface to the bus, and accept standard daughter cards, wire wrap, etc. to allow the application designer to construct an application specific function card. Application systems are then built by selecting the proper combination of the highest level cards that meet the application design requirements, and integrating them in a way to solve the application problem.

In addition to this object-oriented approach, a layered specification was developed: at the bottom are the physical entities, then the operating system components, then conventional tools such as databases and user interfaces, followed by knowledge-based tools such as inference engines, and at the top are generic applications such as diagnosis shells which simply need to be customized for a specific application. Services in one layer are insulated from changes in the implementation of services they use in lower layers, because the (public) protocols remain the same despite changes in the (private) implementation. Thus, alternative implementations can be selected and software upgrades can be installed without alteration of higher-level modules. Along with the protocols, a set of static verification tools checks the application

syntax for possible errors. The separation of the representation language from its implementation permits modular V & V, as does the layered approach. Furthermore, dynamic validation is supported by the AI Bus audit probes, which are intelligent demons attached to the service objects, to be used in building an instrumented testbed. These audit probes act like stream transducers which monitor and query not only physical transactions (over the network or a database, for example) but also software invocations.

The layers and representative object classes are illustrated in Figure 2.

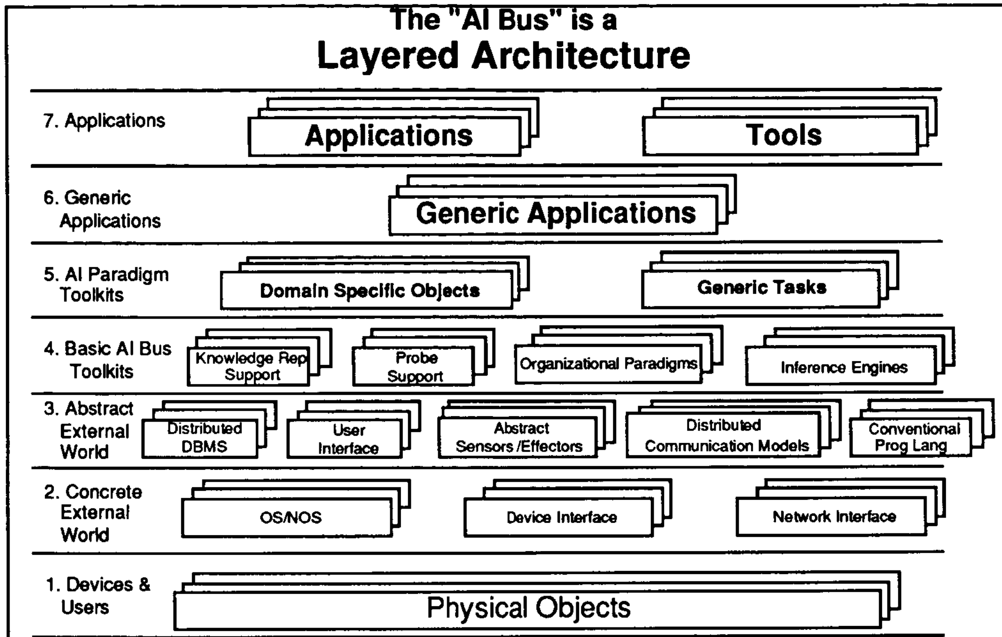

Figure 2. The layers of the AI Bus

4. Implementation

The design of the AI Bus was summarized in a set of abstract-data-type class specifications, intentionally kept language-independent in order to avoid restricting the design. As well as defining the interfaces, the design also specified inheritance between classes. For the implementation, we chose C++ and Unix because of the performance benefits of a relatively low-level language (as opposed to Smalltalk, for example) and its wide availability: a fundamental goal was to build a production quality system, not an experimental testbed. For the common knowledge representation language (layer 4) we chose Clips because it is distributed with source code and hence is amenable to customization. Message passing between distributed Clips systems was easily accomplished by writing three user-defined C++ functions (aibus_ask, aibus_tell, aibus_answer) that are called from the right hand side of a Clips rule. The communication services were built on top of the RPC protocol, and the user interface used X Windows.

We are currently working on decomposing the inference engine into object-oriented modules, so that rules can inherit conditions and actions, and rule-bases can inherit rules from other rule bases, and also incorporating non-linear fact and pattern representations (e.g. Prolog's recursive structures). At the lower layers, we are interested in non-Unix platforms and using commercial distributed operating systems.

5. Support for Cooperative Systems

Although developments in the last fifteen years have taken advantage of hardware advances by distributing data and processing, the control has remained centralized in master-slave relationships. Machines are now "talking" to one another, but the question for cooperative systems is deciding what to say, when, and by whose authority. Just as humans form organizations in order to function more effectively - the whole is greater than the sum of the parts - the promise of cooperative systems is that they can tackle problems beyond the capabilities of current architectures.

The AI Bus follows the distributed object oriented model of interaction between loosely-coupled agents, the fundamental active entities which communicate via messages (Ref. [1,2,3]). An agent is defined as a collection of knowledge sources and an organization; these knowledge sources may be implemented as expert systems (an inference engine and a knowledge base) or a conventional system - just so long as the specified interface is followed. One organizational mechanism is the blackboard, based on the paradigm of agents sharing their problem solving state (Ref. [4]). Each knowledge source has a list of capabilities and interests - which match questions it can answer and information it would like to be told - the agent advertises these attributes with the Finder (a lower layer communication object) and keeps a cache of other agents' capabilities and interests for subsequent communication.

An agent's specification thus permits implementation along several sizes of granularity. Internally, it can be a whole organization of problem solvers, or just a simple procedural program. For efficiency reasons in Unix-like environments where context-switching is costly, a large grain may be preferred, and this can be used at the next layer up as a generic task - an agent which is a specialist in one area of problem solving (Ref. [5]). An example of the internal organization of a complex agent is illustrated in Figure 3.

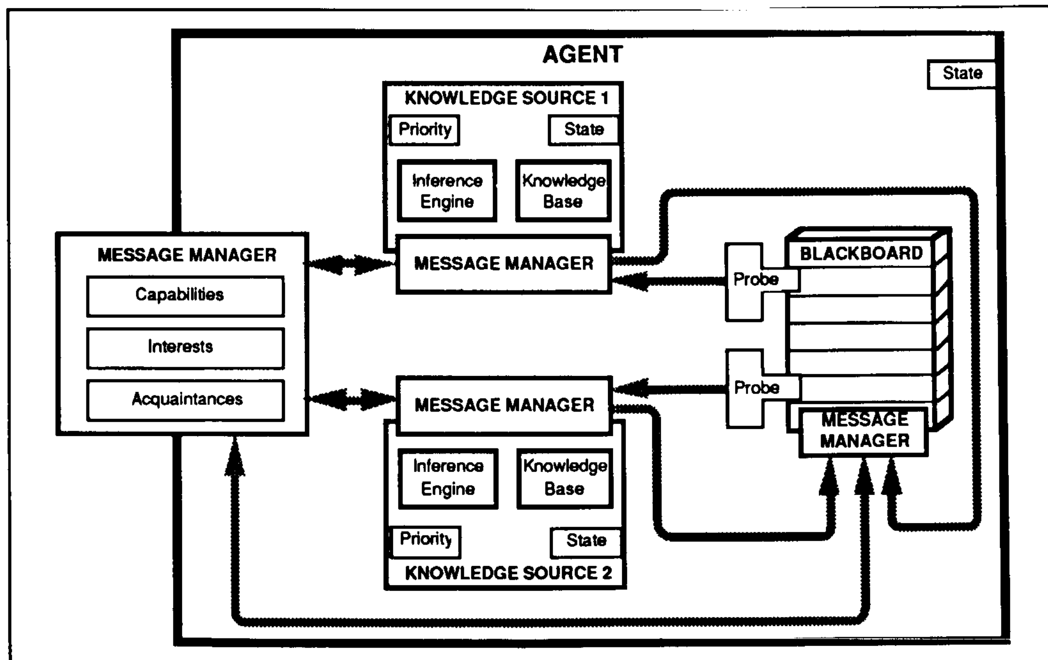

Figure 3. An Example of an Agent Composed of Several Layer 4 Objects

The AI Bus extends the paradigm of event-driven programming in its Probe object (Ref. [6,7]). A probe is activated based on matching patterns of events and conditions and routes information about subsystem activity to interested parties. A probe's history can be used to maintain partial matches for efficiency (e.g. in the blackboard), and has a priority for use in scheduling. A standard event, condition and action language allows the evaluation and interpretation of probes to be implemented by the probed object - a class of probeable objects is specified, and includes databases, network communication, blackboards and agents; there are corresponding subclasses of probes.

Probes can be used to support dynamic validation and to monitor resource usage. A subclass of probes called abstract sensor/effectors can be used in hierarchical process control applications - they provide data, retain state and do some filtering, but in addition they recognize alarm situations and provide direct pathways between each other for fast response.

A blackboard is realized in the AI Bus as a restricted subclass of agent - it is a passive server which is interested in everything (or at least whatever it is programmed for). Agents post information on the blackboard by sending it messages, they install probes on it to gather information resulting from matching events plus several current and historical conditions. A blackboard is thus a semi-permanent communication space, but also acts as a mechanism for loosely-coupled organization whereby several agents can combine partial results without repeated inter-agent communication. It is more than a global database, in that the probes' histories provide a short-term memory and record of partial matches, so that new additions and requests can be processed quickly (in the style of the Rete algorithm for rule-based systems); in contrast, database queries are processed one at a time. This is an object-oriented version of the blackboard concept, and it is important to contrast it with blackboard systems which contain a centralized scheduler in control of the serial execution of agents: in the AI Bus the agents are autonomous. Although logically centralized, a blackboard may be physically distributed for performance reasons: in this case, consistency must be maintained using techniques (e.g. multiple copies, deadlock avoidance) borrowed from distributed databases.

A layer of services exists between the operating system and the programming tools which allows the developers to concentrate on problem-solving rather than worrying about actual physical locations. Each agent has a Post Office object, which queues incoming messages and permits addressing by name, rather than location. The Post Office uses a distributed Finder to map globally unique names to the addresses of active objects. Control is passed via messages which represent remote procedure calls - they are intercepted by an agent's Message Manager, which is responsible for converting messages to procedures, and keeps a queue of questions received together with their askers (for subsequent direction of replies). Remote procedure calls can be asynchronous or synchronous. The question of whether the receiving agent blocks until it processes the request depends on the organization used: if the agent does, it is under the control of the sender (a client-server relationship), if not it is autonomous.

6. Current Direction

Having implemented a subset of the AI Bus objects, we are currently experimenting with using these tools in developing cooperative systems for applications such as air traffic control. In this domain, the decentralized controllers are assumed to be non-hostile, but nevertheless overall coherence and efficiency can deteriorate because of ill-informed local decisions. Facilities such as blackboards and negotiation protocols enable the controllers to make decisions based on more global understanding of the situations, and to cooperate on long-term solutions. Feedback on the usefulness of the tools developed so far has proved to be an esssential driver in the further development of the AI Bus framework.

References

1. Agha, Gul, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.
2. Ishikawa, Yutaka and Mario Tokoro, *Orient84/K: An Object Oriented Concurrent Programming Language for Knowledge Represetnation* in Object Oriented Concurrent Programming, Yonezawa & Tokoro, eds, MIT Press, 1987.
3. Yonezawa, A. , J-P. Briot, E. Shibayama, *Object-Oriented Concurrent Programming in ABCL/1*, in Alan H. Bond and Les Gasser. *Readings in Distributed Artificial Intelligence*. Morgan Kaufmann Publishers, San Mateo, CA, 1988.
4. Nii, Penny, *Blackboard Systems*, AI Magazine Volume 7, nos. 3 and 4, 1986.
5. Chandrasekaran, B., *Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design*, IEEE Expert, Fall 1986.
6. Schultz, Roger and A. Cardenas, *An Approach and Mechanism for Auidtable and Testable Advanced Transaction Processing Systems*, IEEE Transactions on Software Engineering, SE-13 (6), June 1987.
7. Schultz, Roger and A. Cardenas, *An Expert System Shell for Dynamic Auditing in a Distributed Environment*, ACM SIGSAC '87 Conference Proceedings, 1987.