Self-Controlling Architecture Structured Agents

Mieczyslaw M. Kokar (contact author)
Department of Electrical and Computer Engineering
360 Huntington Avenue, Boston, MA 02115
ph: (617) 373-4849, fax: (617) 373-8970
e-mail: kokar@coe.neu.edu http://www.coe.neu.edu/˜kokar

Kenneth Baclawski
College of Computer Science
Northeastern University
360 Huntington Avenue, Boston, MA 02115
e-mail: kenb@ccs.neu.edu

Abstract

In this paper we propose a new paradigm for software. It is an extension of the agent based computing approach. The main components of this paradigm include: the use of formal methods for specification of agent functionality; inter-agent communication using a common syntax with ontology defined semantics; goal driven fusion of agents, services and ontologies; control theory based architecture of embedded agents in which software itself is treated as a controlled plant.

1 The Problem

The ubiquity of software is becoming a defining characteristic of the modern world. Yet software is still largely developed as a craft, and the level of quality, robustness and reliability of software varies considerably. Furthermore, software is largely developed in relatively isolated projects, and nontrivial interactions between applications (especially legacy systems) is difficult or impossible.

A further complication to meaningful application interoperability is the sheer size of the problem. Information is now available in enormous quantities, at increasing speeds, with bewildering complexity and diversity. Software applications need to be scalable if they are to cope with this onslaught. However, software itself is being developed at an increasing pace, and the quantity, diversity and lack of interoperability of software is adding to the problem more than contributing to a solution.

The inability of applications to interoperate suggests that software is at a stage similar to that of networks prior to the development and acceptance of the Internet. Prior to the Internet there were many competing proposals for the basic infrastructure for communication between local networks: the Arpanet was just one of many. A similar situation is occurring today with respect to software applications. There are many frameworks being proposed to make it possible for software components to interoperate, such as DCOM, CORBA, Agent-Based Computing, Model-Driven Architecture, etc. It is too soon to say which of these, if any, will be the winner.

Any solution must also deal with the rapid increase in the number and complexity of embedded systems. Embedded systems already outnumber the traditional interactive systems, and the current trend toward connecting them to the Internet will make them the dominant style of system on the Internet. Such systems function autonomously, and their behavior is more closely related to the physical world than is the case with conventional systems. The lack of any direct human interaction makes formal validation much more important. Software component frameworks that were designed primarily for informally specified components, linked together using informally specified languages, function adequately for conventional interactive systems, but not for embedded systems with real-time constraints and stability requirements.

2 Outline of a Proposed Solution

While the precise nature of a solution will depend on many decisions that will be made by the software community as a whole, it is possible to outline some characteristics that are desirable, if not essential, for software development to cope with the problems it now faces. We suggest the following as being important:

- Distributed systems.
- Formal methods and ontologies.
- Formal reasoning as a means of making formal models executable.
- Self-awareness of software.
- Dynamic systems and control-theory.

We are convinced that the agent-based paradigm will be an important, possibly the primary, software system paradigm in the future. In other words, software should be organized as a community of components (agents) able to communicate with each other. Communication requires that agents understand each other. It is hopeless to expect that there could ever be a single simple communication language with fixed semantics. There are far too many specialized domains that have unique requirements. Accordingly, it is much more realistic for there to be agreement on a communication infrastructure that provides a common syntax, while the semantics is determined by *ontologies*.

The next important aspect of our vision concerns formal methods. While the application of formal methods to software engineering is not new, we propose that formal methods will involve much more than just the traditional specification and refinement of specifications to code, such as in the Specware approach [1]. We propose that formal methods can be used for expressing specifications that are executable. First, each agent would include a formal specification of its own functionality. Each agent would also have access to formal ontologies that provide a framework for inference (i.e., theorem-proving). The interaction between two agents involves steps such as the following:

1. Agreement on one or more common ontologies.
2. Sharing of common facts expressed in these ontologies (e.g., contracts and invoices).
3. Actions are performed (executed) as they are proved based on the shared common facts using the common ontologies. The proofs are obtained using theorem-proving or model-checking techniques.

While theorem proving is not feasible in general, the efficiency (the smarts) of the agents would come from the repertoire of theorems they could prove. Justifications (proof traces) can also be shared by agents as a means of improving efficiency. This technology for agent communication provides a reliable and robust mechanism for agents to access the services of other agents and to mediate activities such as commercial transactions.

One difficulty of this paradigm is that agents must be able to find the agents that furnish the services that are needed. While there are agent platforms that can deal with this problem for relatively small numbers of agents, it is much more difficult to deal with hundreds of millions of agents that represent embedded systems connected to the Internet. A high-performance, scalable platform that can support such a large number of agents has been developed [2], but commercial products are not yet available.

Another difficulty is the problem of combining the many kinds of information involved in these activities into a single coherent structure. To deal with these difficulties, we propose the *information fusion* approach, described in [6, 3]. In this approach, theories of particular software components are combined (fused), and then this new combined structure is used to reason about incoming information. In other words, the process of fusion takes place at the theory level, rather than the data level.

We assume that most of the agents would be embedded in a real-world environment. Consequently, agents must be able to deal with inputs from the environment, which are not necessarily the formal linguistic inputs received from other agents. This leads us to the next aspect of our vision: dynamical systems. We have proposed a general architecture for self-controlling software shown in Figure 1. In this architecture, each embedded agent would consist of these main components: *Plant, Controller, QoS, Evaluator, Controller Designer, Reconfigurer*. The description of this architecture is provided in [5]. The main idea here is that software being controlled is treated as the Plant, while the environment is that part of the world with which

the embedded system interacts. The rest of the components in Figure 1 enable the system to control itself. This point of view leads naturally to an expansion of the software industry to include not only traditional tools such as compilers, interpreters and operating systems, but also new classes of tools that function as controllers, controller designers, QoSs, Evaluators and Reconfigurers. Reconfigurers are especially interesting because they require that the system be self-aware and have the ability to reason about its own structure using theorem-proving techniques. As a result, reconfigurers would have many of the same characteristics as the agents discussed earlier, so much so that they would likely be essentially the same kind of software artifact.

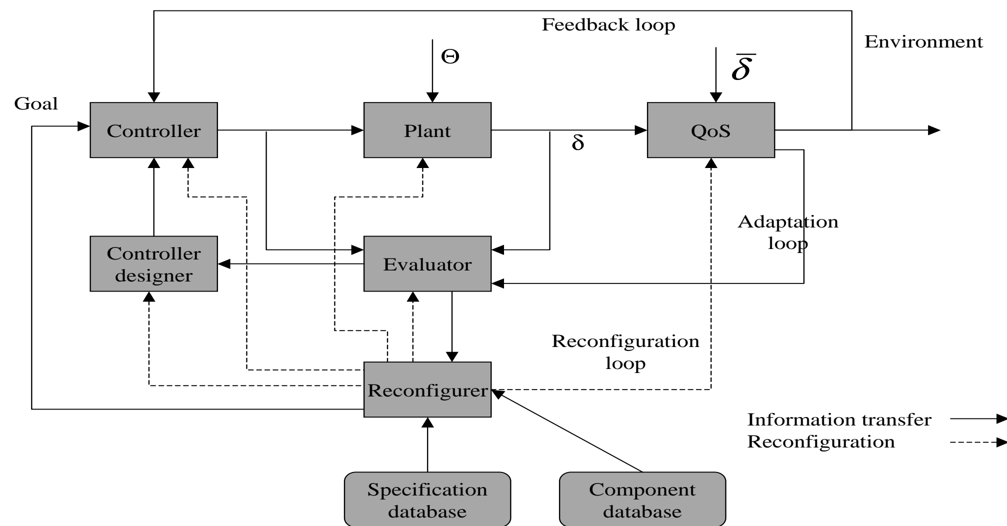

Figure 1: Self-Controlling Software Architecture

3 What Is Needed to Implement this Vision?

Many elements that are needed to implement this vision are either under development or in research labs. Among them we can list the following:

- Agent architectures and underlying communication mechanisms.
- Theorem provers and model checkers.
- Intelligent hybrid controllers.
- A common syntax for information exchange (XML).
- Languages to express ontologies (e.g., DAML: DARPA Agent Markup Language).
- Languages to express software specifications (UML, MOF).

- Formal languages, like Slang, in which fusion of information can be expressed.

Nevertheless, many elements that would make such a vision possible are still missing or incomplete:

- Standards for exchange of specifications.
- Formal semantics for languages like UML, MOF and DAML.
- Efficient theorem provers and model checkers.
- Efficient methods for fusion of information structures.
- Methods for goal-driven on-the-fly restructuring of software.
- Libraries of ontologies capturing domains and environments.
- Efficient implementations of the components of the self-controlling software architecture.

References

[1] Anonymous. Specware: User guide, version 2.0.3. Technical report, Kestrel Institute, 1998.

[2] K. Baclawski. Distributed computer database system and method, February 20 2001. United States Patent No. 6,192,364 on Assigned to Jarg Corporation, Waltham, MA.

[3] K. Baclawski and A. Simeqi. Ontology-based component composition. In *Tenth OOPSLA Workshop on Behavioral Semantics*, pages 1–11, October 2001.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[5] M. M. Kokar, K. Baclawski, and Y. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, May/June 1999:37–45, 1999.

[6] M. M. Kokar, J. A. Tomasik, and J. Weyman. Data vs. decision fusion in the category theory framework. In *Proceedings of FUSION 2001 - 4th International Conference on Information Fusion, Vol. 1*, pages –, 2001.