Concurrent Engineering Through Interoperable Software Agents

by

Taha Khedro, Michael R. Genesereth, and Paul Teicholz

CIFE Working Paper
Number 33

May, 1994

STANFORD UNIVERSITY
Copyright © 1994 by
Center for Integrated Facility Engineering

If you would like to contact the authors please write to:

c/o CIFE, Civil Engineering,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 94305-4020
Concurrent Engineering Through Interoperable Software Agents

Taha Khedro, Michael R. Genesereth, and Paul M. Teicholz
Center for Integrated Facility Engineering
Stanford University
Stanford, CA 94305

Abstract
This paper presents an approach to the development of concurrent engineering software applications based on a knowledge sharing technology. In this approach, individual programmers write their programs in the form of separate modules called Software Agents, which interoperate with their peers using an expressive communication standard called Agent Communication Language. The runtime activities of the individual software agents are coordinated by task-independent programs called Facilitators. Facilitators perform a wide variety of tasks, including the automated selection of agents to accomplish subtasks, the mediation of terminology, the buffering of partial knowledge in communication between agents, and the management of communication with other facilitators on other machines. Once set in operation, software agents interoperate to accomplish the overall task in a heterogeneous distributed architecture called Federation. The paper discusses the major aspects of this approach and briefly describes two concurrent engineering applications developed in light of this approach.

1: Introduction

The development of engineering software applications is a labor-intensive activity that requires designers' formal knowledge and expertise in a particular engineering discipline into a set of computer programs linked together to perform specific design tasks. Traditionally, this kind of development is decomposed into pieces and assigned to programmers, who have to agree on data structures and subroutine calling conventions, and their individual efforts must jointly be coordinated to accomplish the original development task. While this approach has produced independently developed robust engineering software applications, it has created a class of applications that do not work together without additional significant efforts to integrate them in ad hoc ways. This kind of integration has focused on data and information transfer among software applications and has been generally limited in supporting the increasing needs of concurrent engineering applications.

Recently, many researchers have recognized that the way towards integratable engineering software applications is to enable better software interoperability and modularity among software programs. The concepts of better modularity and interoperability of software programs have long been the ideals of the software engineering community, and much work has been done on the design of programming languages and environments to support these ideals. Solutions to better software interoperability have included parts of technology such as standard communication languages, subroutine libraries to assist programmers in writing interoperable software, and system services to facilitate interoperation at runtime. All of these parts have been aimed at transferring much of the burden of interoperation from the developers and users of programs to the programs themselves. Unfortunately, the current technology is still inadequate to support the ideal of automated interoperation, and has not affected the development of concurrent engineering software applications. Existing standards are not sufficiently expressive to allow the communication of the definitions, theorems, and assumption that are often needed for software to interoperate. Current subroutine libraries provide little support for increased expressiveness. Directory assistance programs and brokers are limited by the lack of expressiveness in the languages used to document software and by their lack of inferential capability [1].

Recent research on the ARPA-sponsored Knowledge Sharing effort suggests that it may be possible to extend the software engineering technology in significant ways through the development of interoperable software agents [2]. In this approach, individual programmers write their programs in the form of separate modules called Software Agents, which interoperate with other software agents using an expressive standard communication language, called Agent Communication Language (ACL). The runtime activities of the individual software agents are coordinated by task-independent system programs, called Facilitators, that perform a wide variety of tasks, including the automated selection of software agents to accomplish subtasks, the mediation of terminology, the buffering of partial knowledge in communication between agents, and the

management of communications with different machines in a heterogeneous distributed architecture called Federation. This paper discusses various aspects of the knowledge sharing approach to software interoperation, including the concept of software agents, the standard of agent communication language, the federation architecture, and the implementation strategies of this architecture. Then, we briefly describe two concurrent engineering experiments based on this technology. Finally, we give a brief summary of the paper and discuss future research directions.

2: A Knowledge Sharing Technology for Software Interoperability

The basis for knowledge sharing technology for software interoperability is a highly expressive communication language, called Agent Communication Language (ACL). In the approach presented here, programmers agree in advance on the uniform ACL-based interface for their programs. Within their applications, programmers are free to use whatever data structures and algorithms they find appropriate. However, all interaction among applications must abide by the agreed-upon ACL-based interface [2]. The existence of the ACL interface reduces the need for close coordination. Although there is a cost in supporting this standard interface, the overall cost is often lower than that of the traditional approach since there is only one standard to support. An additional benefit of this approach is that the programs it produces can be easily combined with other applications written by different programmers at different times for different tasks. In this way, every program developed will be capable of interoperating with other programs independently developed by other programmers.

In this approach to software interoperation, programs are called software agents. Software agents use ACL to communicate their information and knowledge via task-independent system programs called facilitators, which coordinate their interaction among each other. At runtime, a software agent surrenders its autonomy to its facilitator by connecting to it and by supplying machine-processable documentation, which characterizes its activity. This is called the registration phase. Agent documentation takes the form of a characterization of the ACL requests that the software agent can handle and the events in which it is interested. In addition, an agent can specify background knowledge that includes the definitions of concepts, theorems, assumptions about the world, and so forth. Once it is registered with a facilitator, a software agent is compelled to satisfy the facilitator’s requests whenever it can.

At runtime, facilitators assume the burden of interoperation among software agents forming a federation architecture. Figure 1 illustrates an example of this architecture in which there are just three machines, one with three agents and two with two agents apiece. As suggested by the diagram, agents do not communicate directly with each other. Instead, they communicate only with their local facilitators, and facilitators communicate with each other. As with other broker services, messages from servers to facilitators are undirected, i.e., they have content but no addresses. It is the responsibility of the facilitators to route such messages to agents able to handle them. This architecture is similar to other open distributed architectures appearing on the mainstream of computing platforms, such as the publish/subscribe mechanism of the Macintosh System 7, the Dynamic Data Exchange mechanism of Windows 3.1, and the Common Object Request Broker Architecture (CORBA) by the Object Management Group [3]. The difference between our approach to software interoperation and previously developed ones lies in the sophistication of the processing done by these facilitators. In some cases, facilitators may have to translate the messages from the sender’s form into a form acceptable to the recipient. In some cases, they may have to decompose the message into several messages, sent to different agents. In other cases, they may have to combine multiple messages [1].

3: Agent Communication Language

Agent Communication Language (ACL) provides the basis for the knowledge sharing approach to software interoperation. ACL centers around the Knowledge Interchange Format (KIF) developed by the ARPA Knowledge Sharing Initiative [4,5]. KIF is a prefix version of the language of first-order predicate logic, with various extensions to enhance its expressiveness [6]. It provides a format for expressing design information and knowledge unambiguously. Here is a typical ACL exchange in which agent chipper, representing an electrical engineer or an intelligent engineering program, asks the agent pricer, representing an integrated circuit database, for the price of 486 microprocessors:

```
(package
    :content (evaluate (chip-price 486-chip)))
    :sender chipper
    :receiver pricer
    :reply-with 1234)
```

```
(package
    :content (reply (dollars 200)))
    :sender pricer
    :receiver chipper
    :in-reply-to 1234)
```

The syntax of ACL closely resembles LISP, with expressions built up from lists of individual terms. This simple example reveals many of properties of the language. chipper has put forward a request using the term evaluate, a performative that asks for the value
of a function expressed in KIF, in this case (chip-price 486-chip). It is imperative that both chipper and pricer agree on the exact meaning of this performative evaluate as well as every vocabulary element used, like chip-price, dollars, 486-chip, and 200. For instance, both chipper and pricer need to agree that the term 486-chip refers to a specific variety of computer chips, and not to anything else.

One of the distinctive features of KIF is its ability to encode knowledge about knowledge. Using this feature, a software agent can document its capabilities and interests by providing a list of KIF sentences. Parts of this list define new vocabulary elements in terms of an understood, explicitly shared vocabulary. Others directly specify the extent of what the agent wants or can do. Here are prototypical examples of capabilities and interests,

\[
\text{(handles pricer)} \\
\quad \text{'(evaluate (chip-price ,?x))}\\
\text{(interested chipper)} \\
\quad \text{'(tell (price 486-chip ,?p))}\\
\text{(interested editor)} \\
\quad \text{'(tell (= (pos ,?x) (point ,?r ,?c)))}
\]

The first sentence declares that pricer can provide chip pricing information. Any agent that understands ACL can read this documentation to decide whether its and third sentences declare that chipper and editor are respectively interested in knowing the price of 486 chips and the position of any chip. Any agent that has such information can read this documentation to learn that chipper and editor are interested in this information.

In addition, in writing agent documentation, agents can define new vocabulary elements that express their view of the information. These kinds of definitions can be expressed in the form of KIF axioms as follows:

\[
(\leftrightarrow \text{ (price ?x ?cost)} \\
\quad \text{ (= (chip-price ?x) ?cost)})\\
(\leftrightarrow \text{ (= (pos ?x) (point ?r ?c))} \\
\quad \text{ (and (= (row ?x) ?r)} \\
\quad \text{ (= (column ?x) ?c)))}
\]

The first axiom defines the vocabulary price in terms of chip-price while the second one defines a pos vocabulary and its relationship to other vocabularies, row and column.

After this informal introduction to the use of ACL, let us formally describe it. An ACL message is an expression that starts with the symbol package followed by a list of keyword/value pairs. The general form of a package is as follows:
The content of an ACL message is an expression typically consisting of a KIF description of an action, which includes a message type and a KIF expression. The message type in the expression is often called a performative and is based on Knowledge Query Manipulation Language (KQML) message types [7]. For the above example, evaluate is one kind of performative and (price 486-chip) is one kind of KIF expression. The KIF expression can be a term, a symbol, or a sentence. Each ACL message must include a sending agent and a receiving agent in the :sender and :receiver fields of the message format. The agent name is assumed to be unique across the entirety of any system. The :reply-with field of a request gives the receiver an identifier to use in sending responses to the request. The :in-reply-to field in a response gives the identifier of the request to which the current message is a response. The :content field is the answer to the request. Finally, the :commode field indicates the type of communication for the message. The value of this field must be: single, stream, or sorry respectively indicating one answer, a stream of answers, or no answers.

The KQML performatives offer a wide range of message types for knowledge manipulation. A stash message directs the recipient agent to assert the truth value of the specified sentence. A flush message directs the recipient agent to retract any former assertion of the specified sentence. An ask-if message directs the recipient agent to determine whether it has the truth value of the specified sentence asserted for some values of sentence variables. An achieve message directs the recipient agent to achieve a new state reflected in new values for sentence variables. Finally, a reply message is used by an agent to transmit sentences that answer previous messages. There are other messages, such as ask-one, ask-all, and ask-about, as well as numerous other special cases.

4: An Architecture for Interoperable Software Agents

At the heart of the federation architecture is a collection of facilitators providing a set of runtime services that manage the existence and execution of software agents as well as the interaction between them. In the federation architecture, software agents acting as clients can issue requests in the environment, without regard for their own location in the network or in their host machine. The facilitators transparently perform the agent location, message passing and information translation services required to make agent requests and receive responses. In doing so, the facilitators provide interoperability between software agents on different machines in heterogeneous distributed environments and seamlessly interconnect multiple services. The facilitators are responsible for all mechanisms required to find the other software agents that can fulfill a given request, to forward appropriate requests to those software agents, and to communicate necessary information for making the requests. Depending on what type of request a software agent is issuing, the facilitators will either obtain a response for the request or will merely announce the request. An example of the latter case is that a software agent can send or request information without knowing who and where the recipients are.

The runtime agent library provides access to a communication module for software agents. The communication module provides the basic plumbing needed to connect software agents throughout the underlying network and operating system. In the facilitators, the communication module includes facilities for message delivery, agent location, and software connections and disconnections. Via the communication module, the software agents have transparent access to information. To accomplish this location transparency, every agent supported by the facilitators uses agents' documentations and a dynamic link address. These two components provide the complete identification of the recipient of a request and are the only information needed.

Under the umbrella of the federation architecture, software agents can be developed on programming platforms, which provide for standard dynamic communication and ACL interface. This interface provides standard libraries for developing software agents that exchange information consistently through the standard interface and via the facilitators at runtime. In addition, the architecture provides interfaces to adapt existing software applications to software agents. For programmers, a key aspect of such an architecture is the standard dynamic communication and ACL interface, which insulates an agent's implementation from any system dependencies in the underlying services. Thus programmers can develop and add their own distributed agents without regard to the underlying network communication mechanisms.

5: Facilitation Services of Architecture

In the federation architecture, facilitators employ automated mechanisms to provide services that coordinate the interaction of software agents in the
runtime environment. These services include content-based routing, translation, synthesis, and compaction. In this section, we briefly discuss each of these services.

5.1: Content-Based Routing

Facilitators provide content-based routing to determine appropriate recipient software agents and forward to them appropriate messages for any received undirected message. For example, consider the three software agents chipper, pricer, and editor and their documentations described in Section 3. Suppose that the three agents are communicating via a facilitator which captures their documentation and background definitions. Consider the case when the facilitator receives from chipper the following request for the price of 486 chips:

(evaluate (chip-price 486-chip))

In order to determine which agents can handle this request the facilitator forms the following query:

(handles ?agent ' (evaluate (chip-price 486-chip)))

It then uses its automated mechanisms to find a binding for variable ?agent. In this case, there is just one, pricer. Consequently, the facilitator sends pricer the following message:

(evaluate (chip-price 486-chip))

5.2: Translation

The second important service provided by the facilitator is translation, which is the transformation of messages from one form to another. There are two aspects to the translation process: vocabulary translation and logical translation. The need for vocabulary translation arises because of differences between the abstractions inherent in the implementations of different agents. For example, one agent may work with rectangular coordinates, while another works with polar coordinates. The need for logical translation arises because of limits imposed by agents on the logical structure of messages in which they are interested. Some agents are capable of accepting any message in ACL. Other agents are more selective. As an example of translation, consider again the case of the three agents chipper, pricer, and editor, in which the facilitator receives the message shown below from pricer about the price of the 486 chip.

(tell (= (chip-price 486-chip) 250))

In this case, the facilitator uses its inferential capabilities to deduce the sentence (price 486-chip 250) from the above sentence, sent by pricer and from the first axiom in Section 3 since the incoming sentence does not have the form specified in this interest. In other words, the facilitator translates from one form to the other. It then checks whether any agent is interested in this information and finds out that chipper is. Then it sends the following message to chipper:

(tell (price 486-chip (dollars 250)))

5.3: Synthesis

The translation example described in the preceding subsection is simple: one incoming message leads to one outgoing message. In some cases, an incoming message can result in multiple messages being sent to one or multiple agents. In order to handle such messages, the facilitator must be able to synthesize a multi-step communication plan to handle the incoming message. As an example of this type of message, consider again the three agents and the following message coming from editor indicating the position of a particular chip, chip1.

(tell (= (pos chip1) (point 10 16)))

In this case, the facilitator uses (= (pos chip1) (point 10 16)) and the second position axiom and deduces the following two messages:

(tell (= (row chip1) 10))
(tell (= (column chip1) 16))

According to the current lists of agents' interests shown in Section 3, no agent is interested in this information. However, suppose that agent chipper has already added the following sentence to its list of interests:

(<= (interested chipper ' (tell (= (?term ,?c) ,value))
(member ?term (listof 'row 'column))))

With this list of interests, the facilitator finds that chipper is interested in both messages, which are forwarded to it.

5.4: Compaction

Another important issue in translation is buffering. In some cases, it may not be possible to transform a message into a form that is acceptable to any agent yet. It is possible to merge the information from two or more messages to form an acceptable result. As an example of how the facilitator handles these cases, consider an incoming message from chipper that involves information about the row position of a chip.

(tell (= (row chip2) 100))
The facilitator, in this case, is unable to complete its translation based on the second axiom since there is no column information, which is needed for the position translation. However, the facilitator buffers the following sentence for future use as more information becomes available.

\[(= \text{(row chip2) 100})\]

Now suppose that the facilitator receives the missing column information for \text{chip2} in the following message:

\[(\text{tell } (= \text{(column chip2) 160}))\]

Putting this information together with the preceding sentence, the facilitator can conclude a position sentence, and it sends the derived sentence to \text{editor}.

\[(\text{tell } (= \text{(pos chip2) (point 100 160)}))\]

6: Concurrent Engineering Applications

The knowledge sharing approach to software interoperation has been developed into a practical technology. The ACL standard is fully defined, there are subroutine libraries that assist in writing software agents, and there are running facilitators. Using this technology, we have developed two concurrent engineering applications that involve a variety of engineering software applications necessitating interoperation. In this section, we briefly describe these two applications.

6.1: Designworld

Designworld is an automated prototyping integrated system for small-scale electronic circuits built from standard parts (TTL chips and connectors on prototyping boards). The design for a product is entered into the system via a multi-media design workstation, and then the product is built by a dedicated robotic cell, in effect, a microfactory. If necessary, the product, once built, can be returned to the system for diagnosis and repair [8]. Designworld consists of 18 programs running on 6 different computers (2 Macintoshes and 4 HP workstations).

In the current form, each of Designworld’s 18 programs is implemented as a distinct agent that communicates with its peers via the facilitator in the form of ACL messages. Any one of these programs can be replaced by an ACL-equivalent program without changing the functionality of the system as a whole. Any agent can be moved to a different machine (with equivalent capabilities). Any agent can be deleted and the system will continue to run correctly, albeit with reduced functionality. In the development of the system, there was virtually no communication between programmers, except at the very end; discussion, when it occurred, was limited to negotiation on message vocabulary; and no reimplementation took place as a result of this negotiation (since the mediation of all disagreements was handled by the system’s facilitator). The Designworld system is a good example of the use of software agents.

6.2: CIFE WORLD

CIFE WORLD is an integrated building engineering environment that consists of eight software applications: an architectural CAD system, a structural design system, a footing design program, a structural CAD system, a cost estimating program, a knowledge-based construction planning system, a scheduling system, and an ORACLE database management system [9]. In the building engineering process, five designers and engineers can concurrently perform some of the design tasks and transparently exchange design information and changes via their software applications in an automated and coordinated fashion.

The eight software applications along with two facilitators of the integrated environment run on seven different computers: two DEC workstations, three Sun workstations, and two Macintoshes. Most of these software applications have been implemented at different times by different people, and there has been no communication among their developers. The applications have been adapted to be software agents by the implementation of agent programs that added agents’ capabilities. As a result, these applications have been enabled to register with the facilitator and to communicate in ACL using appropriate design vocabularies particular to the application’s design task.

7: Concluding Remarks and Future Directions

In experimenting with concurrent engineering applications, we have been able to determine how the concept of interoperable software agents and its architecture can support various aspects of software integration. The architecture allows designers to access and obtain necessary information across the network without the need to know where the information resides. The architecture accommodates existing heterogeneities (hardware, networks, software programming tools, database systems, data representations, etc.). In addition, it is quite flexible and extensible in integrating many design and engineering software applications without losing efficiency or effectiveness in information communication among many applications. Independently developed software applications can
seamlessly be integrated into the architecture without the need to make changes to the already integrated applications and existing environment.

In this paper, we have taken a brief look at the knowledge sharing technology approach to software interoperability. The presentation in this paper has ignored many key problems, such as security, crash recoveries, inconsistencies in agents' documentations, and so forth. Despite the fact that we have developed partial solutions to these problems, further research is needed. This defines a path for our long-range vision in developing a technology that allows the interoperability between systems without the intervention of human users or their programmers. Although many problems remain to be solved, we believe that the presented concepts constitute a first important step toward achieving that vision. One of our future research projects will also focus on supporting cooperative engineering. This support will include the identification of appropriate techniques for allowing software agents to assist designers and engineers in detecting design conflicts, in locally managing received design information and changes, and in enforcing and communicating design constraints. All of these demands will have to be orchestrated through appropriate collaboration strategies for resolving design conflicts by negotiation.

Acknowledgments

The authors wish to acknowledge the contributions of many participants in the ARPA Sharing Initiative and other related projects, most notably Mark Cutkosky, Richard Fikes, Tom Gruber, Reed Letsinger, Narinder Singh, Vishal Sikka, and Marty Tenenbaum. This research was funded by Hewlett-Packard, the Office of Naval Research, and the Center for Integrated Facility Engineering (CIFE) at Stanford University.

References


