Leveraging Service-Oriented Architecture to Extend a Legacy Commerce System

James. J. Mulcahy
Florida Atlantic University
Computer Science & Engineering
Boca Raton, Florida, 33431, USA
jmulcah1@fau.edu

Shihong Huang
Florida Atlantic University
Computer Science & Engineering
Boca Raton, Florida, 33431, USA
shihong@fau.edu

Andrew B, Veghte
Florida Atlantic University
Computer Science & Engineering
Boca Raton, Florida, 33431, USA
aveghte@fau.edu

Abstract - Among the many challenges faced by businesses that maintain legacy software systems is the integration of those systems with external data sources, which are often other software systems developed and maintained by otherwise unrelated entities. Leveraging service oriented-architecture (SOA) provides a method by which the business value of a legacy system can be extended without the expensive alternative of reengineering or of complete replacement of the software. This paper describes a particular implementation that used a SOA approach to enhance and extend the functionality of a mature, complex multi-channel commerce enterprise system. The implementation linked the legacy enterprise system of a multi-channel merchant with that of an international shipping company that provides a service for electronic shipping label requests. The goal was to further automate the process of shipping and tracking of purchased products returned by customers of the merchant.

Keywords – systems of systems, service-oriented architecture (SOA), software maintenance, systems integration, software reuse, legacy software systems

I. INTRODUCTION

Gone are the days when individual software applications or complete software systems were commonly expected to be rewritten or replaced every few years. More time was once available to both analyze the usefulness of particular implementations and to plan the next generation of replacement software. The emergence of the Internet as a commerce platform introduced a greater need for attention to the "time to market" demands of software development and evolution. Businesses now have the ability to rapidly develop relationships with their partners, establishing higher levels of connectivity between disparate and geographically dispersed software systems. These relationships increasingly require complex software systems to be adapted rather than replaced or completely reengineered.

Software obsolescence is perhaps itself becoming obsolete, at least within the context of large enterprise resource planning (ERP) systems. It is often far too expensive to completely rewrite or replace these complex systems. Replacing otherwise stable and familiar applications is also riskier than continuing to maintain existing software to satisfy evolving business needs. Reengineering and encapsulating, also known as "wrapping" [25], legacy software systems and then exposing them to external systems via a common exchange technique is becoming a more feasible approach. This allows diverse systems to share data and services in a "loosely coupled" manner, integrating legacy system functionality with more modern systems [7].

Service-oriented architecture (SOA) is one method of accomplishing the integration of legacy and modern systems. SOA is a collection of services that allow separate entities to communicate with each other without explicit language or system architecture dependency. This definition can be more generalized to include the overall technique of loosely coupling applications or systems of applications by establishing a common information exchange between the components. Examples of the implementations can range from SOAP (Simple Object Access Protocol) servers that trade files in XML formation to a simpler transfer and retrieval of data between two parties using FTP (File Transfer Protocol).

This paper describes how SOA techniques were used to modify a multi-channel merchant's legacy ERP system to share data with a well-known shipping company. The shipper had exposed a service that provided a process for the exchange of returns-related information between the two parties. The merchant engineered changes to the software in their own enterprise system to access and communicate with this new service. Information exchanged through the service included package pickup and destination information as well as tracking numbers generated by the shipper. Automated interaction with this type of service was intended to increase efficiency and accuracy in the handling of merchandise returned or exchanged by customers.

The remainder of this paper is structured as follows. Section 2 describes the requirements that motivated the use of SOA in this particular implementation. Section 3 discusses the choice of architecture and design of the solution. Section 4 details the actual software reuse, engineering and reverse engineering activities. Finally, Section 5 summarizes the implementation and offers conclusions as to the efficacy of the resulting solution with respect to the techniques that were used.

II. LEGACY SYSTEM BACKGROUND

In this case study, the requirement for SOA implementation was an opportunistic one. A service offered by United Parcel Service (UPS), a well-known shipping company [1], drove the desire for a change to the merchant's legacy enterprise system. The resulting software implementation would increase the automation and control of a vital part of the supply chain of the retail process.

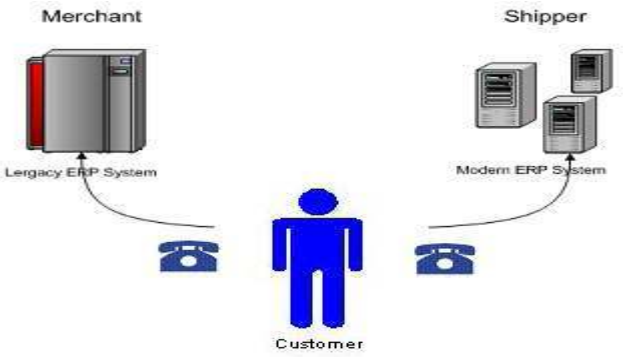

Figure 1. Original data flow.

The procedure to be replaced relied on the customer to be the focal point of certain critical points of the merchandise returns process (Fig 1). The return of a damaged, incorrect, or undesired product is initiated by contact between the customer and the merchant from whom the merchandise was purchased. This may take the form of an e-mail or a phone call originated by the customer.

The merchant responds with a Return Merchandise Authorization (RMA) number, information that is manually recorded by the customer and transcribed to the returned package(s). Sometimes, merchandise must be returned to a specific warehouse. A damaged guitar may be sent to a different warehouse than a garment of the wrong size or fabric, for example. The address for the correct warehouse is also relayed to the customer, who must then transcribe it onto the relevant package(s) prior to shipment. In this manual process, the customer is also often responsible for contacting the shipper and arranging for the pickup of the returned goods.

Automating much of this currently manual procedure is intended to reduce potent for human error at this point in the returns process. An error in supplying or transcribing the return address or RMA number could lead to lost merchandise or accounting headaches for both the customer and merchant. Upon implementation of the proposed software solution, the merchant would instead electronically transmit a specific warehouse location directly to the shipper, along with additional details about the shipment, such as the expected weight and dimensions of individual packages. The shipper responds to the electronically transmitted request with one of its own. In this implementation, the acknowledgement of the return request would contain the package tracking number(s), which improves the ability of both the merchant and the customer to track the shipment progress once it is en-route to the merchant. The shipping label itself is then e-mailed to the customer, and can be subsequently printed and affixed to the package(s) by the customer prior to pickup by the shipper.

An important requirement of the implementation was fault-tolerance. In this case study, the expected software solution would operate as a scheduled background process in a largely unmonitored environment. Major failure points of the software solution needed to be identified and mitigated from the onset of development.

Fault tolerance would be incorporated into the design throughout the development process, rather than as a separate "detail" phase near the end of the project. The affected parts of the system would be designed to handle most likely hardware and software failure conditions, such as the inability to connect to a database, or to a remote FTP site. Some faults would be logical in nature, requiring them to be trapped and reported in an appropriate manner, but without halting any processes.

III. CHOOSING THE ARCHITECTURE

A critical requirement in this implementation was that both the shipper and merchant enterprise systems would be loosely and asynchronously coupled. It was likely that either or both systems could be temporarily unavailable to the other at some future time.

Each business involved in this integration operates as a separate entity, and the unavailability of one should never directly affect the normal daily operation of the other. All software systems are periodically offline due to scheduled maintenance activities and other planned outages. Unexpected hardware or software failures also contribute to intermittent outages, as do connectivity problems between the systems and other conditions not directly controllable by either party .

The architectural model chosen for this implementation was based on existing infrastructure owned and operated by the merchant. SOA techniques were already being used to collect and confirm online web-based purchases from one of the merchant's internet sales channels. The SOA approach would satisfy the loose coupling requirement of the new implementation, allowing the legacy software to operate independently on a completely different platform than the shipper. The shipper's system architecture was, in fact, unknown and not directly visible to the developers producing the solution [16].

A SOAP/XML/FTP implementation had been previously deployed by the merchant to integrate the legacy ERP system with an external third party internet retail channel. This existing architecture would serve as a basis for the new merchant-shipper SOA integration.

A new service would be added to the merchant's existing SOAP server, operating alongside other SOA services already deployed on the server. The new service would be the gateway to the primary data link between the merchant and shipper. It would handle the delivery and retrieval of request files from the merchant and response files from the shipper, while also archiving inbound and outbound files for historical and recovery purposes. The implementation would largely reduce the responsibility of the consumer in the returns process, eliminating the need for direct contact by the consumer with the shipper (Fig 2). This paper will give more attention to the legacy component of the solution and the steps taken to assure its loose coupling and fault tolerance.

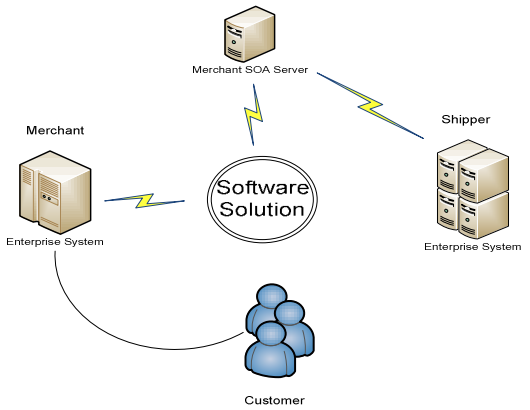

Figure 2. Proposed data flow.

IV. IMPLEMENTING THE SOLUTION

The solution implemented has several major components. The first is the new service that handles the actual exchange of request and response files with the shipper via an external IP address. This component was developed by a third party specializing in service architecture and design, and subsequently installed on one of the merchant's existing servers.

The second component is the customer relationship management (CRM) module that handles product returns. This component is part of the larger ERP system. The module was modified to produce a special type of database record when an electronic shipping label for the return was requested. It was also modified to display related information from the database. This information included data extracted from the shipper response – the shipment tracking number, the date the returned merchandise was picked up by the shipper, and its destination.

The third component is the new application developed to extract the return requests and process the shipper responses. This application transforms return requests generated by the CRM component into the data format required by the shipper. It also processes the inbound shipper responses as they arrive. From these responses, the merchant's ERP database is updated with tracking information and other relevant data supplied by the shipper, along with the current date and time.

The fourth component of the solution is the collection of "listeners" (programs and/or jobstreams that poll for information) and command files required to orchestrate the flow of data files between the legacy system of the merchant and that of the shipper. The listeners and command files are written in the native job control language (JCL) of the legacy ERP platform.

Before any components were designed or implemented, reusable components were identified. This included software from existing legacy software repositories, processes and techniques already being used, hardware (servers), and SOA techniques already in use (SOAP).

A. Identifying reusable components

Reusing hardware, architectures, source code, and software applications from the merchant's existing resources was an important activity in this implementation. Reusing processes not only reduces development time, but also reduces the risk associated with introducing newly coded and untested software.

The first reusable component identified was a server owned by the merchant that was capable of communicating with the target shipper service. The server was already host to several other SOA-type services, including a SOAP implementation that handled origination of web-based internet orders.

Several dozen software routines from the legacy software system were identified as reusable for this implementation, and were subsequently integrated into the data extraction and transformation (DET) component, which is described in more detail later in this paper.

Two "listener" processes were also required by the solution to handle data transfer between the legacy mainframe of the merchant and the server that would be tasked with communicating the data with the shipper. Similar processes had been previously implemented by the merchant in other contexts of their enterprise system, and were therefore used as models for some components of the new implementation.

B. Defining the data exchange format

Requests for shipping labels by the merchants and tracking/pickup confirmation responses by the shipper are communicated between the parties using a format predefined by the shipper.

Pickup information is expected by the shipper in a particular "pipe-delimited" sequential file format. Each record contains a value identifying the merchant, contact information identifying the customer, and the warehouse location receiving the returned merchandise – including details about the actual package(s) to be shipped.

Two types of responses are produced by the shipper. This first type is confirmation of the original merchant returns request. The confirmation is received as a sequential file. Data content includes the merchant-generated RMA, original order number and a tracking number generated by the shipper for the pending return.

A second type of response file produced by the shipper is sent less frequently than the confirmation files. This file contains a summary of all returned merchandise that has been picked up by the shipper over a period of time (e.g., the last day). Any package(s) included in these summary files are assumed to be en-route to the merchant's warehouse.

Communication between the DET component and the enterprise system was accomplished by extending the use of an existing ERP dataset containing order-related comment records. These records were already structured to vary by type and content. One type of comment could contain notes from the customer service representative, for example. Another type could contain a timestamp indicating when an order has been processed by the warehouse. The ability to store and search for comment records by type would allow the CRM returns management module to be extended to write a special type of comment record that would be detected only by the DET module. This would in turn trigger the module to appropriately generate electronic shipping label requests for the shipper.

C. Engineering the transformation process

At the core of the merchant-shipper integration effort is the component considered the pivot point of the entire integration process. This module is responsible for transforming data to the required format exchanged by the two parties. It is also responsible translating response data from the shipper and applying changes to the enterprise database with details received from the shipper. The DET module can be scheduled to run at regular intervals, but can also be executed "on demand." It is capable of running in one of three modes, and each of these modes can be scheduled to run at different times, or with a different frequency.

The first mode searches the ERP database at regular intervals for unprocessed merchandise returns for a particular shipper (UPS). The records are identified by the special type of comment record mentioned earlier (created by the CRM returns management module). The DET module gathers date related to the order and merchandise return from various datasets in the enterprise database and produces a file containing a shipping label request for each order. There may be one or more items to be returned to the merchant for any given order. At this time, another order-related comment record is added to the database indicating the date and time that the electronic request has been generated.

A second mode handles one of the two types of shipper responses. Once the shipper has processed a returns request previously sent by the merchant, it produces a response that includes the same RMA originally sent by the merchant, along with a unique tracking number generated by the shipper. When executed in this mode, the DET module parses the incoming file and extracts each RMA and the tracking number that is associated with it. The enterprise database is subsequently updated with the tracking number, along with new order-related comment record containing the date and time of the processed response.

The third mode processes another type of shipper response – the summary response file. This file lists all merchandise picked up from customers since the last time a response file was generated. The file contains more than one hundred fields describing each of the packages involved. In this implementation, the DET module is only concerned with the mere existence of a record for each particular return authorization. The authorization number is the only value extracted from each record and used to update the enterprise database with another order-related comment record that indicates that the returned package is en-route as of the current date and time.

Each of the cases described above will result in the creation of a new order-related comment record. This record is stored in the merchant's enterprise database, and the CRM module automatically displays these new returns-related comments along with other pre-existing comments, creating a seamless integration of the user interface with the new automated implementation. Leveraging existing functionality and legacy source code helps accomplishment this with very little newly engineered coding.

For all modes, the DET module produces a log of each of the processed transactions. This log is stored in a flat file to facilitate that is easy to review by humans for historical and manual or automated troubleshooting purposes. Any logical errors, database errors, or trappable operating system errors are logged and the appropriate action is taken. This action would vary by circumstance. The most serious error would result in a "graceful" shutdown of the application with a dump of fault details to a unique flat file and to the system console. Minor warnings appear as well, without shutting down the application. In most cases, such warnings would be related to returns requests that could not be processed and were therefore skipped.

The merchant-shipper integration is expected to be largely unmonitored, and it was critical that any faults would be automatically escalated to the attention of human users. It is also necessary for all files produced and consumed by the implementation to be archived or otherwise retained in the event of a failure or unexpected outage. This was accomplished by the creation of jobstreams, "listener" applications, and automated e-mail procedures to provide the controlling structure for the enterprise side of the SOA solution.

D. Combining the components

To combine each of the components into a single, yet loosely coupled system, a series of jobstreams were written in the JCL native to the ERP platform. These jobstreams were created to orchestrate and control the flow of data between the ERP platform and the FTP server, which would in turn control the flow of data between the server and shipper (UPS).

One jobstream was created to execute the DET module in a mode that searches for outstanding product returns that require return shipping label requests. The interval at which the module is executed is controlled by the jobstream (e.g., hourly). The jobstream itself is either launched manually by a system operator or as part of the general "start of day" process executed by the business.

Two other jobstreams were created to execute "listener" programs tasked with monitoring for request and response files that need to be transmitted from platform to another. These listeners behave much like the polling feature in implementations such as SOAP. One jobstream launches an application that "listens" for response files to appear in a particular outbound folder on the ERP platform. Once such a file is detected, the jobstream automatically opens an FTP session and attempts to transmit the file to the merchant's SOAP/FTP server.

If the operation succeeds, the jobstream moves the request file to an archive folder so it will not be picked up and reprocessed during the next polling cycle. If the operation fails, the file is not moved to the archive folder. Instead, it will be automatically reprocessed as part of the next cycle.

This technique allows for additional fault tolerance. If the FTP connectivity is reestablished by the time the next cycle executes, the file(s) that could not be transmitted as part of the previous cycle are transmitted during the current cycle along with any new request files. Any problems transmitting the files are logged and automatically e-mailed to technical support.

The second "listener" jobstream performs similar duties as the first, but with data flowing in the opposite direction. Instead of polling a local folder for outbound request files, it polls a remote folder for pending inbound response files. The remote folder resides on the merchant's SOAP server, where responses are collected from the shipper.

The jobstream uses an FTP session to check the remote folder. When files are detected, they are transmitted to a local inbound folder, where they will be processed by the next DET jobstream cycle. As implemented in the first listener jobstream, any failures are logged and automatically e-mailed to technical support staff.

If transmission of a response file from the FTP server to the ERP platform is successful, the file is moved to an archive area on the remote server to prevent it from being picked up during the next polling cycle. If the process fails or if the FTP site could not be accessed, the file remains unmoved. The next polling cycle will attempt to locate and retrieve the files along with any newly arrived responses.

Both listener jobstreams control the frequency at which the listener polls for new request files. They also control which folders are considered the inbound, outbound and archive folders. The location of the remote FTP site and login information is also stored in the jobstream. Any of these details can be changed at the JCL level without affecting the other components of the larger system. With each of the aforementioned jobstreams in place, electronic shipping label requests are regularly generated. Any responses from the shipper are regularly processed. The final component of the solution is the part that exchanges the requests and responses with the shipper.

E. Implementing the server component

This component of the solution was developed by a third party that specialized in building web services. It was developed autonomously from the rest of the project and later installed on the merchant's existing SOAP server for integration testing. The component's primary function was very similar to the enterprise-side implementation. Request files inbound to the server from the ERP platform were periodically transmitted to the shipper, while response files were periodically retrieved from the shipper and stored in an outbound folder. This folder would be considered an inbound folder to the enterprise-side processes.

Files that could not be transmitted to or retrieved from the shipper were left in place until the next cycle repeated the attempt. Again, any failures were logged and automatically reported to support staff via e-mail. With each component in place, integration and acceptance testing began. While the business logic surrounding the production of requests and consumption of responses was straight-forward, special attention was paid to testing the fault tolerance of the system as a whole.

F. Testing the solution

During the integration testing phase, the shipper exposed a test location where files could be stored and retrieved. This allowed the server-component's data transmission and fault tolerance to be tested. Once it was confirmed that files could be transmitted retrieved, and properly archived, attention turned to the loose coupling requirements. This took the form of modifying the service configuration to point to an incorrect IP address, simulating a connectivity or shipper-originated outage. The service performed as expected, reporting the failed attempts and resetting for the next polling cycle.

The enterprise-side data transmission capabilities were tested in a similar fashion. Once mocked-up files were observed to flow to and from the server-side implementation, and finally archived, the relevant jobstreams were modified to

simulate an intra-network connectivity disruption. The listeners appropriately left unprocessed files "in place," logged and reported the failed attempts, and reset themselves for the next polling cycle.

The link between the CRM module from the ERP system and the DET module was tested by human users, who simulated merchandise returns requests involving the shipper. Database disruption was also simulated by "pointing" jobstreams to the wrong or nonexistent database while a return was being processed. The DET module successfully logged the errant condition, skipped and reported the records it could not process, and ceased execution until its next scheduled cycle.

Once local integration testing was completed, a final round of acceptance testing was performed with "real" orders placed by the merchant's customer support staff. The process was monitored from the point at which the customer service representative confirmed a return request until the shipper responded with the tracking number, and then later with simulated pickup details. The results were documented, release notes were prepared, and the entire implementation was migrated to "live" production.

V. RESULTS AND CONCLUSION

In the case study described by this paper, the release of the software implementation came just after a holiday season. It was put into production just in time to for the typical spike in volume of returned merchandise that follows major holiday shopping seasons. The timing of the release helped test the efficiency and stability of the software under higher-than-normal volume conditions early in its life cycle. The developers monitored the deployed system over the next several months. Most automated error-reporting e-mails observed were related to logical errors such as missing package weights or RMAs, but there were no errors or conditions that caused further modification of the software. Temporary connectivity issues were observed during the monitoring period as well, verifying the fault tolerance and loose coupling of the implementation. None of the outages required human intervention. No missed merchandise returns were observed during the period.

The approach applied to the design and implementation of the software described by this case study was largely responsible for the highly successful and efficient deployment and continued stability of the product. Loose coupling of the software components and systems improved its fault-tolerance. Information reuse at the source code and component levels also contributed to the fault-tolerance, while also allowing for speedier development and release of the implementation. When properly implemented, SOA techniques can extend the value and reach of legacy enterprise systems, even under the "time to market" pressures that drive such development. This paper demonstrated one such example that improved both the business value of the enterprise systems involved, and the scalability of a mature legacy enterprise system. The implementation automated a once previously error-prone and largely manual merchandise returns process. It used the electronic exchange of relevant data between the entities involved in the supply chain, reducing the involvement of the human component, including the customer.

REFERENCES

[1] Ames, Sam; "UPS Unveils Electronic Label Services," *CNET News*, Dec 26, 2001. http://news.cnet.com/UPSunveils-electronic-label-service/2110-1017_3-277411.html

[2] Bass L., Clements P., Kazman R.; Software Architecture in Practice, Addison-Wesley, 1997.

[3] Burner, Mike; "The Deliberate Revolution," *Queue*, vol. 1, no. 1, pp. 28-37, Mar 2003. ACM.

[4] G. Canfora, A.R. Fasolino, M. Tortorella, "Towards Reengineering in Reuse Reengineering Processes," ICSM, pp.147, 11th International Conference on Software Maintenance (ICSM'95), 1995.

[5] Chikofsky, E. J., Cross, J. H.; "Reverse Engineering and Design Recovery: A Taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13-17, Jan 1990. IEEE.

[6] Dholakia, R.R., Zhao M. and Dholakia N.; "Multichannel Retailing: A Case Study of Early Experiences," *Journal of Interactive Marketing*, vol. 19, no. 2, pp. 63-74, Mar 2005. Wiley Periodicals. [7] Grevink, R.; "SOA: The Perennial Legacy Issue," AttachMate corporate web site, January 2006, URL = http://www.attachmate.com/nr/rdonlyres/4b1e1d09-b4bd-414e-aa05-42f940a68e49/0/intl_dev_integration_jan2006.pdf

[8] Jones, Steve; "Toward an Acceptable Definition of a Service," *IEEE Software*, vol. 22, no. 3, pp. 87-93, May/June 2005.

[9] Lehman, M. M. and Ramil, J. F.; "Software Evolution and Software Evolution Processes," *Annals of Software Engineering*, vol. 14, no. 1-4, pp. 275-309, Dec. 2002.

[10] Lehman, M.; "Programs, Life Cycles and Laws of Software Evolution," *Proceedings of IEEE Special Issue on Software Engineering*, 68(9):1060-1076, September 1980.

[11] Leung, H. K. N., White, J.; "Insights into Regression Testing," *Proc. Conf. on Softw. Maint.* Oct 1989, pp.60-69. IEEE.

[12] Lewis, G. and Smith D.; "Systems of Systems: New Challenges for Maintenance and Evolution," *Frontiers of Software Maintenance*, Sep-Oct 2008, pp.149-157.

[13] Lewis, G.A. and Smith, D.B.; "Service-Oriented Architecture and its Implications for Software Maintenance and Evolution," *Frontiers of Software Maintenance*, Sep-Oct 2008, pp. 1-10.

[14] Muschamp, Paul; "An Introduction to Web Services," *BT Technlogy Journal*, *Jan* 2004, pp. 9-18.

[15] Müller, H.A., Jahnke J.H., Smith D.B., Storey, M.A., Tilley, S.R., Wong, K.; "Reverse Engineering: A Roadmap," *ICSE '00: Proceedings of the Conference on the Future of Software Engineering*, 2000, pp. 47-60

[16] Sneed, H. M.; "Encapsulation of Legacy Software: A Technique for Reusing Legacy Software Components," *Ann. Softw. Eng.*, vol. 9, no. 1-4, pp. 293-313, Jan. 2000.

[17] Tilley, S.; "The Canonical Activities of Reverse Engineering," *Annals of Software Engineering*, 9(2000), pp. 249-271. J.C. Blatzer AG, Science Publishers

[18] Tilley, S.R., Smith D.B., Paul, S.; "Towards a Framework for Program Understanding," *Proceedings of the 4th International Workshop on Program Comprehension*, 1996, pp. 19-30. IEEE Computer Society