An Object-Oriented Software Development Framework for Autonomous Decentralized Systems*

Stephen S. Yau†
Department of Computer Science and Engineering
Arizona State University
Tempe, AZ 85287-5406, USA

Keunhyuk Yeom, Bing Gao, Ling Li and Doo-Hwan Bae
Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611-6120, USA

Abstract

Developing software for distributed computing systems is challenging due to lack of good software development methodologies for distributed computing systems. It is very important to develop reliable, adaptable and expandable application software for distributed computing systems. Autonomous Decentralized Systems (ADS) is a distributed computing system with on-line expandability, on-line maintainability and fault-tolerance capability. In this paper, a framework for developing ADS application software is presented. Our framework consists of object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance. It is based on the object-oriented computation model developed for ADS application software development which supports on-line expandability and on-line modifiability. CASE environments for ADS software development are also discussed.

Keywords: autonomous decentralized system, application software development, object-oriented approach, CASE environment.

1 Introduction

Due to the rapid development of computer and communication technologies, distributed computing systems are used widely in various areas of applications [1]. In some application areas, it is required that the system be continuously operating at any time, including partial system failure, system expansion and system maintenance. Hence, these systems should not only have the fault-tolerance capability, but also on-line expansion and on-line maintenance properties. In order to have these properties, the Autonomous Decentralized System (ADS), which is composed of largely autonomous and decentralized components with the capabilities of on-line expansion, fault-tolerance, and on-line maintenance, has been developed [2, 3]. The ADSs have been applied to several areas, such as the steel production process control systems and train traffic control systems. In order to effectively utilize ADSs, effective application software development methodologies for ADSs are needed.

Developing software for distributed computing systems is more challenging than that for centralized computing systems due to additional complications of interprocessor communication, synchronization, etc.[1, 4] Because the object-oriented paradigm reflects the distributed structure of the problem space and is suitable for representing inherently concurrent behavior, it is suitable to support the software development for large scale distributed computing systems, like ADSs. We have developed an object-oriented computation model [5] for software development for ADSs.

In this paper, we will present an object-oriented software development framework for ADSs based on this computational model. Our framework consists of object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance. We will also discuss our CASE environment supporting these phases and use an Automated Teller Machine (ATM) to illustrate our framework.

2 The Object-Oriented Computational Model

Before we present our software development framework, we will briefly summarize the computational model [5] which is the basis for our framework.

The computation model is based on the object-oriented concept and supports on-line expandability and on-line modifiability at the application software level. In the computation model, ADS software is represented as a set of modules. Each module has its own control thread, the object base, the interface base, and base management mechanism (BMM). An object base contains instances of object classes defined in the module. The objects in the object base of a module is called local objects of this module, and the objects in the object bases of other modules is called remote objects of this module. An interface base contains the object methods and the object names to which these methods belong. There are two types of interface: import interface and export interface. The import interface is a list of methods in other objects invoked by itself. The export interface is a list of methods of its own to be invoked by other objects. The BMM provides functions used to modify the object base and interface base dynamically to support on-line features.

The followings are major features of the computation model:

*This work is supported under the collaborative research agreement between the University of Florida and Hitachi, Ltd., and the collaborative reserach agreement between the Arizona State University and Hitachi, Ltd.

†Formerly with Computer and Information Sciences Department, University of Florida, Gainesville, FL 32611-6120

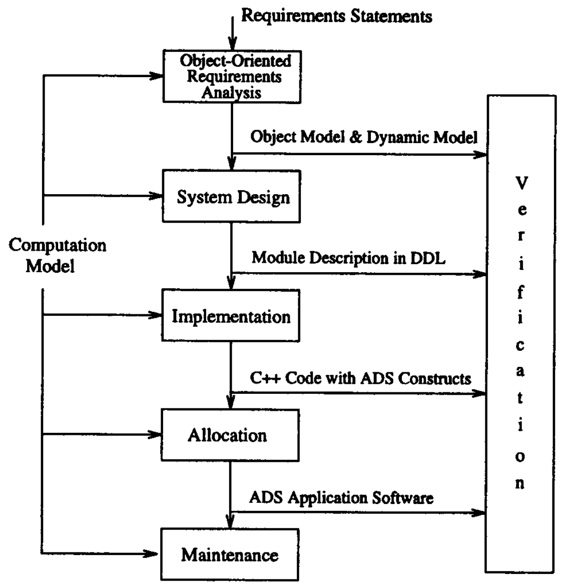

Figure 1: Our object-oriented framework for ADS application software development.

- classes and objects used as bases of software development
- two level encapsulation: module level and object level
- asynchronous remote object method invocation and synchronous local object method invocation
- full logical location transparency achieved by the location mechanism to determine the local object method invocation and the remote object method invocation
- on-line expandability and on-line modifiability achieved by the BMM to modify the object base and interface base dynamically

3 Our Framework

Our framework has the following phases: object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance. The framework for software development for ADS is shown in Figure 1. We start the ADS software development with a set of requirement statements, which is transformed into the object model and dynamic model using object-oriented requirements analysis (OORA) technique. Our OORA technique is an extension of Object Modeling Technique (OMT) [6]. The OMT is based on constructing two visual projections of the system, called the object model and the dynamic model. The object model shows classes and their structures derived from the knowledge about application domain and the requirement statements. The dynamic model shows a finite-state machine model for each class in the object model. The object model and dynamic model are represented by the module description in Design Description Language (DDL) [5] in the system design phase. Then, each module design in DDL is implemented in C++ and the implemented modules are allocated into processors. At the end of each phase, the adequacy of the results of each phase is verified. The maintenance technique can be applied to manage the ADS application software. In this paper, while we discuss the overall framework for the software development for ADS, our focus will be on the requirements analysis and verification, system design and the CASE tools supporting these phases.

4 Object-Oriented Requirements Analysis and Verification

OORA is rapidly gaining popularity, promising to provide a more understandable specification and to better support object-oriented design and implementation. In this section, we will present our object-oriented requirements analysis and verification.

4.1 Object-Oriented Requirements Analysis

To generate precise, concise, understandable and correct model of a real-world application problem, we must examine its requirements, analyze their implications and restate them rigorously [6]. The analysis model consists of the object and dynamic models.

The object model can be constructed as follows [6]:

1. identify object classes from the requirement statements.
2. identify associations between classes.
3. identify object attributes.
4. organize classes using inheritance to share common structure.

We iterate the above steps until the object model is correct. Finally, we apply top-down and/or bottom-up approaches to build our hierarchical object model.

In most existing OORA methodologies, the results of the requirements analysis only concerns static information such as class definitions, hierarchies, and relationships among classes. However, for distributed computing systems like ADS, more information , such as communication behavior, persistency of objects and role of objects, are needed in the requirements analysis phase and made available to the design phase. The dynamic model in [6] using a state transition diagram (STD) is not suitable for representing communication and checking consistency. We have developed a new dynamic model to enhance the original STD. It can be constructed as follows:

1. prepare scenarios.
   We prepare scenarios of typical interaction sequences. These scenarios show the major interactions, external display formats and information exchanges. The approach to construct the dynamic model by scenarios ensures that important steps are not overlooked and that the overall flow of the interaction is smooth and correct.

2. construct event trace for each scenario.
   An event trace is an ordered list of events between objects. Examine the scenarios to identify all external events including all signals, inputs, decisions, interrupts, transitions and actions.

3. build an STD based on event trace.
   It starts with the event trace diagrams that affect the class being modeled. Every scenario or event trace corresponds to a path through the STD. Each branch in control flow is represented by a state with more than one exit transition.

4. Verify consistency.
   Check for the completeness and consistency at the system level. Every event should have a sender and a receiver. Make sure that corresponding events on different STDs are consistent.

In real-world applications, software systems have a substantial number of object classes and associations. To handle this, we developed a hierarchical object model to "layer" the object model as follows:

After identifying object classes and associations, we group the classes using the following bottom-up guidelines:

- Tightly-coupled classes which frequently communicate each other are grouped together.
- Each association should generally be shown on a single screen, but some classes must be shown more than once to connect different sheets.
- Try to minimize the number of *bridge* classes which are the classes which form the bridge between two screens or subjects (which are logical constructs for grouping classes).
- A *star* pattern is frequently useful for organizing subjects: a single core subject contains the top-level structure of high-level classes. Other subjects expand each high-level class into a generalization hierarchy and add associations to additional low-level classes.

Because it is not realistic to apply existing OORA approach directly to large and complex systems due to the complexity of the problem statement itself, we use the following top-down approach:

1. Identify the core activity areas of the system which require analysis, and provide a basis for work partitioning and parallel development. Domain knowledge plays an important role in identifying these areas.
2. Divide the problem statements according to the core areas identified in **Step 1**.
3. Apply our OORA approach to each of the core activity areas to generate the object and dynamic models for each area.

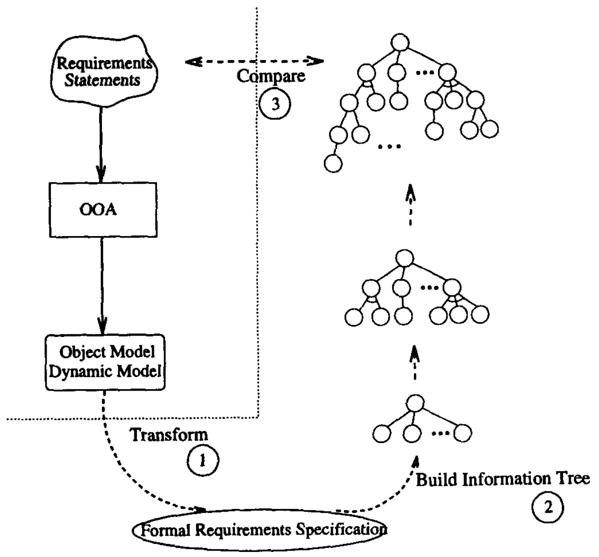

Figure 2: Our verification approach.

4.2 Object-Oriented Requirements Verification

Because the requirement statements are high-level description of a software system in a natural language, and because we need to use the domain knowledge and ignore redundant or unnecessary information from the given requirement statements to derive OORS during OORA, there may exist some differences between requirement statements and OORS. Such differences should be identified and their effects on the system should be evaluated during the verification of OORS. Our verification approach will identify the missing information in the OORS which may be deleted as unnecessary information and the information in the OORS which is not specified in the requirement statements.

To verify the OORS, we check the completeness and consistency between the requirement statements in the natural language and the OORS [7]. To do so, we transform the OORS which is expressed in terms of the object model and dynamic model generated by OORA into a formal specification using a formal specification language, which is then transformed to a graphic form called the information tree. We then verify the completeness and consistency of the requirements specification by comparing the information in the information tree with the given requirement statements in the natural language. The information tree is built to organize the information so that the verification can be systematically performed without increasing the complexity of the problem itself. The overall verification process is shown in Figure 2.

The input of our verification approach consists of OORS expressed in terms of the object model and dynamic model, and the requirement statements in the natural language given by the client. Our verification approach can be summarized as follows:

1. Transform the derived OORS into a formal requirements specification described by a formal specification language.

2. Build an information tree from the formal specification obtained in **Step 1**.

3. Check the completeness and consistency between requirement statements and OORS by comparing each given requirement statement with the information represented by its corresponding path or relationship in the information tree.

5 System Design

During the system design, we determine the structure of the system, which provides the context for more detailed decisions made in later design stages. By making high-level decisions for the entire system, the system designer partitions the problem into modules so that further work can be done by several designers working independently on different modules. In our computation model, ADS software is represented as a set of modules, and each module has its own control thread, a number of local objects and interfaces including import interface and export interface. Based on this structure, we derive the following procedure for the system design:

1. Identify all objects and all inter-object communication from the requirement statements, object model and dynamic model. At the requirements analysis phase, we focused on classes rather than objects in object modeling. However, individual objects which play different roles in the system and their communication behavior should be identified at the system design phase.

2. Cluster the objects into modules. Based on our computation model, a module is not an object nor a function, but a package of objects, associations, operations, events and constraints that are interrelated and include a reasonably well-defined interface with other modules. The formation of the modules or the clustering of objects and related items directly influences the performance of the system being developed. A good clustering algorithm will establish a modular design which will reduce inter-module communication cost, exploit potential concurrency among objects, and achieve fault-tolerance and reliability. For this purpose we use a clustering algorithm to systematically group the objects into modules according to a given set of criteria: minimizing communication, exploiting concurrency, functionality and user constraints. Our clustering approach is a bottom-up heuristic algorithm which optimizes communication with functionality and concurrency as constraints.

3. Identify the control thread of each module. A control thread is a path through a set of state diagrams in which only a single object at a time is active. In our computation model, each module has only one control thread constructing the body of itself, in which objects are initiated and the process activities of the module are defined.

4. Describe each module using DDL defined in our computation model. Our DDL is used to represent software design for ADSs. The DDL representation of the software design reduces the gap between system design and implementation.

5. Decide boundary conditions. The boundary conditions such as initialization, termination and failure, must be taken care of during system design. Constant data, parameters, global variables, tasks and others need to be initialized. In ADS software, content code [8] should be set for each module. On termination, internal objects can simply be abandoned, and each task must release any external resources it had reserved. In case of failure, a plan must be developed for orderly failure, as well as graceful exit on fatal bugs by leaving the remaining environment as clean as possible and recording or printing as much information about the failure as possible before terminating. Exception handling techniques for behaviors like retry, ignore, wait and time out are being developed as defined in our computation model.

6 Implementation and Allocation

In our framework, coding is done using C++ with ADS constructs, such as invocation, receive_result, guard statement, etc., which are based on our computation model. The advantages of using ADS constructs are that the programmer will not have to be bothered by the synchronization, communication and/or location of processors. The ADS constructs can be translated automatically into C++ by a control mechanism in our computation model.

After the implementation, we need to allocate the modules to processors in the ADS system. For the module allocation, we use the criteria of minimizing communication and balancing the load among the subsystems that constitute the distributed system [9]. Our module allocation algorithm is heuristic and suitable for allocating modules onto a distributed system whose subsystems are connected in the form of a LAN and communicating by means of broadcasting such as ADS. In this algorithm we try to achieve load balancing and minimize the communication thereby attempting to increase the concurrency. The algorithm includes fault tolerance aspects such as duplication, that is, no two duplicated modules are allocated to the same subsystem as this would defeat the purpose of duplication. Our approach also allows the user to specify constraints in allocation such as a particular module needs to be allocated on to a specific subsystem.

7 CASE Environment

We have developed a CASE environment to aid the system analysts to generate object model and dynamic models from the given software requirements according to our OORA approach, and to aid the system designer to design the system architecture according to our system design approach. This CASE environment includes a generic drawing editor using Graphical User Interface (GUI), two levels of integration such as common user interface and transferability of data among all the tools in the environment, constraint maintenance which checks

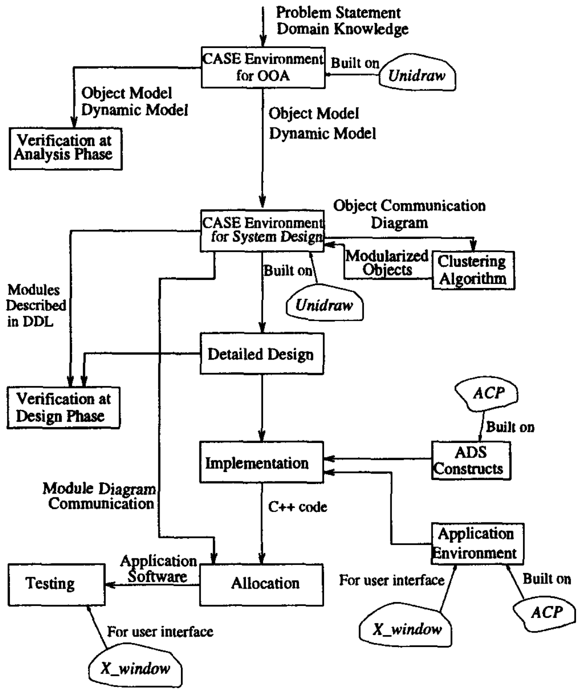

Figure 3: A CASE environment for our framework for ADS application software development.

the connectivity between components, error checking, and automation of output generation from the CASE tools for OORA and for system design

Our goal is to provide a CASE environment for ADS application software development as shown in Figure 3, where two levels of integration are supported. The first level is the common user interface. Commands and tools are accessed uniformly in both the tools for OORA, such as the object modeling tool, dynamic modeling tool and event trace editor, and for system design, such as the object communication modeling tool and module modeling tool. The next level of integration is the transferability of data between the tools, which is another kind of bridge between the tools. The CASE tools need to communicate each other in order to fetch information and check consistency. For example, the CASE tools for the system design may need information from the CASE tools for OORA, such as class definitions and communication behavior.

The design of our CASE tool is based on the framework of Unidraw [10]. We implemented this tool using the object-oriented language C++ and the CASE tools are running on X windows. We also used some objects from the object-oriented graphical toolkit InterViews [11] such as dialog boxes, text editor and structured graphics.

The object modeling tool consists of a viewer, a pull-down menu containing controls that execute a specific command, controls for engaging the current tool, a panner for panning and zooming the viewer and state variable views that display the values of state variables maintained by the object modeling tool. A state variable includes the name of the object model and modification status. The user engages the current tool by clicking on the appropriate control on the tool's left edge. The dynamic modeling tool is very similar to the object modeling tool. It also consists of state variable views, pull-down menu, viewer and panner. We can build the components of the dynamic model such as state, event, class and transition, and manipulate components such as move and select using this tool. In implementation phase, our computation model provides language constructs for ADS software that control communication, synchronization and other ADS properties. These ADS constructs are translated into C++ by a preprocessor. The preprocessor scans the ADS application modules, checks the syntax according to the syntax defined by the computation model and generate source code. The allocation algorithm has been implemented under X window environment. The algorithm consists of three main procedures: *Cluster*, *Spillover* and *Update_Load*. The procedure *Cluster* clusters two nodes to form a compound node. This is done by merging the two nodes, a pivot and a non pivot node into a single node. The procedure *Spillover* is for achieving load balancing. It checks whether a module can be safely allocated to a subsystem. Each time an allocation is performed, the load on the subsystem to which the allocation is performed is updated by a call to the procedure *Update_load*. Finally, we are going to construct CASE environment tools and an integrated comprehensive set of services to support software development for distributed computing systems.

8 An Example

We will illustrate our approach using a simplified Automated Teller Machine (ATM) system. The requirement statements for the system is given as follows:

Develop the software to support a computerized banking system with automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank has its own computer to maintain its accounts and make updates to accounts. ATMs communicate with a central computer of the consortium. An ATM accepts a bank card, interacts with the user, communicates with the central computer to process transactions, and/or dispenses cash.

For simplicity purpose, the system is assumed to have two ATMs, two Banks and one Consortium.

For object-oriented requirements analysis, we obtain the object and dynamic models of the ATM system according to the procedures summarized in Section 4.1. The object model is shown in Figure 4. The state transition diagram of class *ATM* is shown in Figure 5.

For object-oriented requirements verification, we first transform OORS into formal specification and build an information tree from the formal specification. Figure 6 shows the complete information tree of an ATM system. For each statement in the requirement statements, we check the consistency. For instance, *ATM dispenses cash*. The corresponding path of this statement in the information tree is identified by selecting nodes such as ATM, dispense_cash, cash and ≤ 200 as shown with the arrows in Figure 6. We compare this path and the given

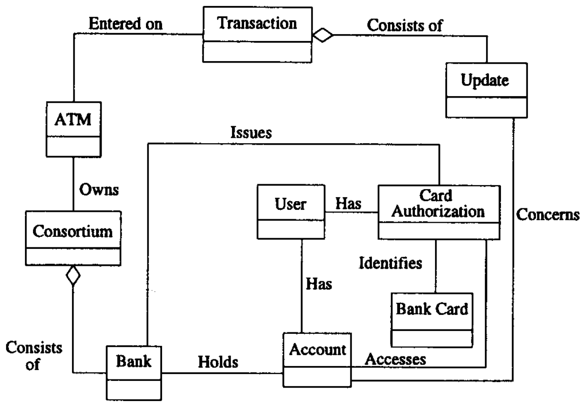

Figure 4: The object diagram for the ATM system.

statement from the requirement statements for the consistency. As a result, we find an inconsistency, ≤ 200, in the OORS. We also consider the path in the information tree, *ATM*, *dispense_cash*, *cash*, and *less than or equal to \$200*, as an example. We make the statement or the enumeration of words for that path as follows: ATM dispenses cash (less than or equal to \$200). We search a statement in the requirement statements matched with the statement represented by that path. We find the statement in the original requirement statement, "ATM dispenses cash". We determine whether *less than or equal to \$200* is from the domain knowledge or a mistake made during OORA. Comparing it with the domain knowledge added during OORA, we find that "less than or equal to \$200" is from the domain knowledge.

For system design, we first identify objects and interobject communications. In this example, objects are ATM1, ATM2, Bank1, Bank2, Consortium, Bank-Cards, Card-Authorizations, Accounts, Transactions, Updates and Users, and interobject communications are "Consortium communicates with ATM1, ATM2, Bank1 and Bank2," "Bank-Cards, Card-Authorizations, Accounts, Transactions and Updates communicate with corresponding Bank," "Updates communicate with Accounts and Transactions" and "Users communicate with ATMs". Next step is clustering. The module communication diagram after clustering is shown in Figure 7. Then, identify control thread of each module and describe each module by DDL. Finally, we should handle boundary conditions. First, we initialize the database using all available account data and set content code for each module. The second condition is termination, but we do not have termination conditions because the system will operate forever. The last one is failure condition. For instance, if a user enters wrong password three times continuously, the card is automatically ejected.

9 Discussion

In this paper, we have presented an object-oriented framework for ADS application software development. Our framework contains object-oriented requirements analysis, system design, implementation, allocation, verification and maintenance. We have introduced a hierarchical object modeling for managing large complex systems. We have also constructed a dynamic model from the event traces automatically using a CASE tool.

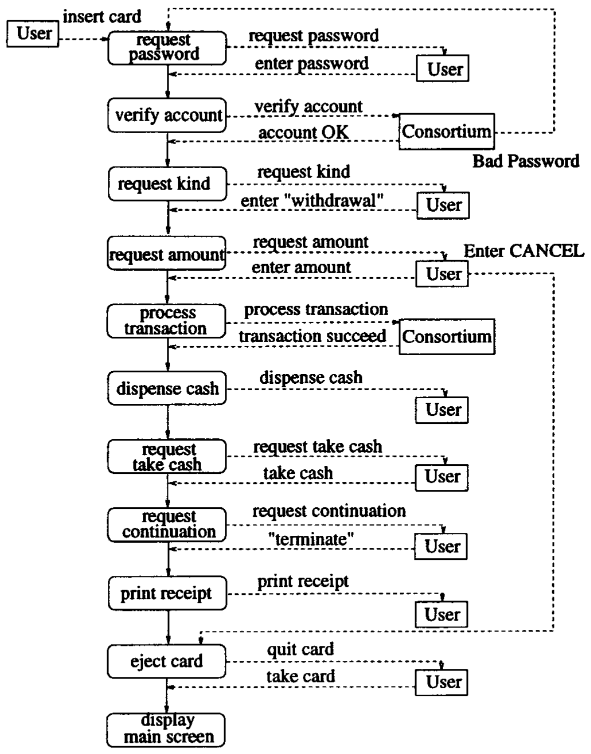

Figure 5: The state transition diagram for class *ATM*.

Since the requirement statements written in a natural language often contain ambiguities which lead to diverse interpretations of the same requirements and cause serious problem in requirements analysis, one of future research is to deal with the ambiguities in the requirement statements for requirements analysis. In design verification, we deal with more specific design issues, such as concurrency, communication and deadlock in software development for ADS. We also need to deal with the testing of application software on the ADS environments. We have developed an allocation algorithm based on the criterion of minimizing communication and load balancing. On-line modification of existing distributed software may degrade the performance of the system unless the allocation is done all over again which may be expensive in terms of time and cost. Hence, we need to develop an efficient reallocation algorithm due to on-line modication of distributed software. To standardize our framework for distributed application software development, we need to extend our computation model to open system environments to achieve the unification of system management. We will also consider the object-oriented software development framework using CORBA. Since ADS can also be operated in the multiple loop environment as well as the single loop environment, we also need to extend our

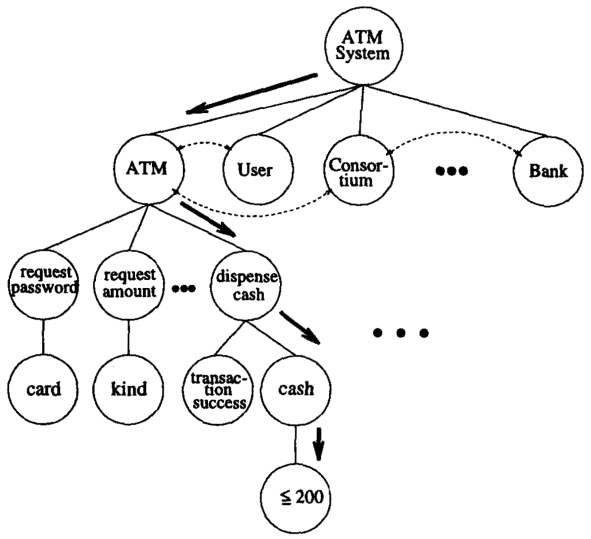

Figure 6: The information tree for the ATM system.

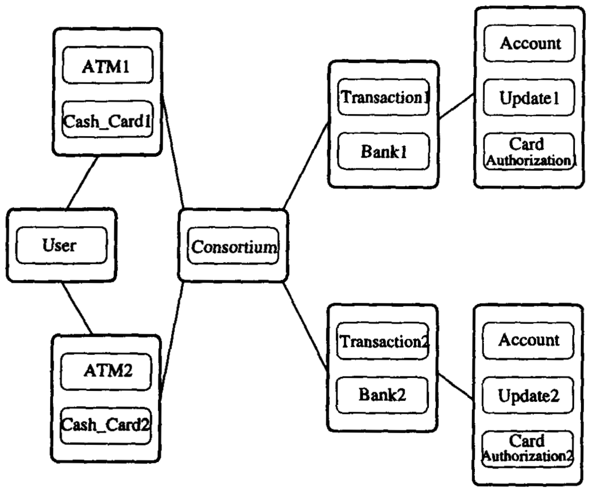

Figure 7: Module communication diagram of the ATM system after clustering.

computation model to the multiple loop environment.

Acknowledgment

The authors would like to thank K. Mori, K. Kavano, H. Suzuki, M. Orimo, C. Sakai and T. Yamamori of Hitachi, Ltd. for valuable discussions on various aspects of the ADS systems.

References

[1] S. S. Yau, X. Jia, and D.-H. Bae, "Trends in Software Design for Distributed Computing Systems," *Proc. Second IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990, pp. 154–160.

[2] H. Ihara and K. Mori, "Autonomous Decentralized Computer Control Systems," *IEEE Computer*, Vol. 17, No. 8, 1984, pp. 57-66.

[3] K. Kawano, M. Orimo and K. Mori, "Autonomous Decentralized Systems: Concept, Data Field Architecture and Future Trend," *Proc. Int'l Symp. on Autonomous Decentralized Systems*, 1993, pp. 28-34.

[4] S. S. Yau, X. Jia and D.-H. Bae, "Software Design Methods for Distributed Computing Systems," *Journal of Computer Comm.*, Vol. 15, No. 5, May 1992, pp. 213–223.

[5] S. S. Yau, and G. -H. Oh, "An Object-Oriented Approach to Software Development for Autonomous Decentralized Systems," *Proc. Int'l Symp. on Autonomous Decentralized Systems*, 1993, pp. 37–43.

[6] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[7] S. S. Yau, D.-H. Bae and K. Yeom, "An Approach to Object-Oriented Requirements Verification in Software Development for Distributed Computing Systems," *Proc. 18th Int'l Computer Software & Applications Conf.*, November 1994, pp. 96–102.

[8] K. Mori, H. Ihara, Y. Suzuki, M. Koizumi, M. Orimo, K. Nakai, and H. Nakanish, "Autonomous Decentralized Software Structure and Its Application," *Proc. ACM-IEEECS Fall Joint Computer Conf.*, 1986, pp. 1056-1063.

[9] S. S. Yau and V. R. Satish, "A Task Allocation Algorithm for Distributed Computing Systems," *Proc. 17th Int'l Computer Software & Applications Conf.*, September 1993, pp. 336–342.

[10] J. Vlissides, "Generalized Graphical Object Editing," Technical Report: CSL-TR-90-427, Computer Systems Laboratory, Stanford University, Stanford, CA, June 1990

[11] Mark A. Linton, Paul R. Calder, John A. Interrante, Steven Tang and John M. Vlissides, "InterViews Reference Manual, Version 3.1," Stanford University, Stanford, CA, 1992