An Approach to Object-Oriented Requirements Verification in Software Development for Distributed Computing Systems*

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

Doo-Hwan Bae and Keunhyuk Yeom
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 effective software development methodologies and tools. In particular, because many errors in the source code can be traced to the errors in the requirements specification, it is especially important to have effective verification techniques for the requirements specification. In this paper, an approach to verification of object-oriented requirements specification (OORS) in software development for distributed computing systems is presented. In our approach, the requirements specification generated by object-oriented analysis is described using a formal specification language, which is transformed into an information tree. Then, the completeness and consistency of the requirements specification expressed in terms of the information tree is verified by comparing it with the original requirements statement.

Keywords: *distributed computing systems, object-oriented requirements verification, object-oriented software development, requirements verification.*

1 Introduction

As VLSI and communication technologies advance, distributed computing systems become more cost effective for various applications. Developing software for distributed computing systems is more challenging than the centralized computing systems due to the additional complications of interprocessor communication, synchronization, etc.[1, 2] Object-oriented software development [3 – 5] is rapidly gaining popularity because it is more understandable to consider a software system as a set of cooperating objects and the concept of classes and objects and the organization of objects naturally reflect the structure of software systems, especially for distributed computing applications. Object-oriented analysis (OOA) [6 – 8] is the process of generating object-oriented requirements specification (OORS) from the requirements statements in a natural language to support object-oriented software development. Since distributed computing systems are used in a wide variety of critical applications, the development of reliable distributed computing systems has become a very important problem [4, 5]. Verification is an essential part in the overall software life cycle. It is closely tied to the individual steps in the software development. The verification executed in each phase of the software development must assure that the software requirements specification is implemented in the design expressed in the software design description and further in the code. This should include compliance with any standards or codes of practice which have been adopted [9]. The overall verification process determines whether the software behaves in accordance with its requirements specification. Considering that many errors found in the source code stem from inconsistent or incomplete requirements specification, the requirements verification is a very important part in developing reliable software for distributed computing environment.

Although much research in the requirements verification has been done [10 – 13], the verification of OORS for distributed computing systems has not been studied. In order to realize the full potential of object-oriented software development, we need to develop an effective verification approach for OORS. In this paper, we will present an approach to verification of the OORS in software development for distributed computing systems. In our approach, we transform OORS into formal requirements specification with a graphical representation expressed in terms of an information tree, and then check its completeness and consistency with the original requirements statements. We will illustrate our approach using an Automated Teller Machine (ATM) example.

2 Overview of our approach

In our approach, the verification of requirements specification is done by checking the completeness and consistency between the requirements statements in a natural language and the OORS, which is expressed in terms of the object model [6] and dynamic model [6] generated

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

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

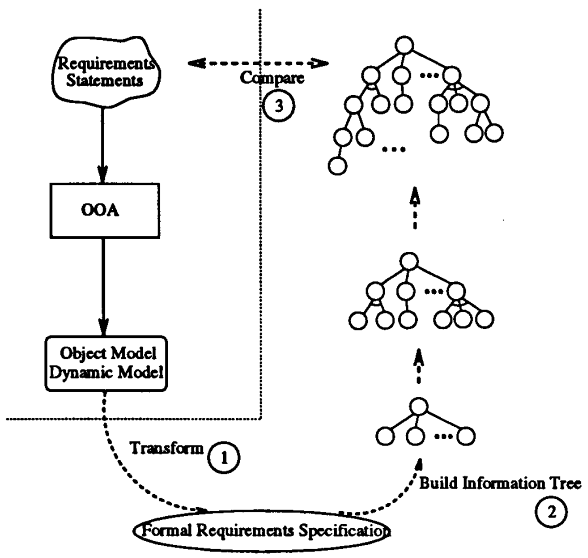

Figure 1: Verification approach using information tree

by OOA. To do so, we transform the OORS into a formal specification using a formal specification language, which is then transformed to a graphic form called the information tree, and then verify the completeness and consistency of the requirements specification by comparing the information in the information tree with the given requirements statements in the natural language. Our overall approach can be depicted as shown in Figure 1.

The input of our approach is 1) OORS expressed in terms of the object model and dynamic model, and 2) the requirements statements in the natural language given by the client. The object model is the static model which specifies the objects, classes, and their various relationships. It is represented by the object diagrams with classes and their structure which are derived from the given requirements statements and domain knowledge. It can be generated by identifying object classes, associations between classes, attributes and inheritance. The dynamic model specifies class states, state transitions, class behavior and objects interaction. The dynamic model is represented graphically with the state diagrams. Each state diagram shows the state and event sequences in a system for one class of objects. Each model describes one aspect of the system, but contains references to the other models. The object model describes the data structure that the dynamic model operates on. The operations in the object model correspond to events in the dynamic model. The dynamic model describes the control structure of objects. It shows which decisions depend on object values and which actions change object values.

The goal of our approach is to identify the inconsistency and/or incompleteness between the OORS and the requirements statements, if any. The requirements statements are high-level description of a software system in a natural language. The OORS describes a software system's external behavior – what the system will interact with its outside world. During the OOA, we expand the OORS using the domain knowledge and ignore redundant or unnecessary information from the given requirements statements. There may exist some differences between requirements statements and OORS because these are at different levels of details in specifying the requirements of the system. Thus, such differences should be identified and their effects on the system are 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 requirements statements. Such information may be created from the domain knowledge or as a result of mistakes or features introduced during the OOA.

Our verification approach can be summarized as follows:

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

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

Step 3 Apply the top-down and bottom-up approaches to check the completeness and consistency between requirements statements and OORS.

3 Formal requirements specification

The formal specification language to describe OORS must have the following characteristics:

- It supports an abstract data type. That is, a class represents an abstract data type.
- It supports the inheritance mechanism. In order to fully support an object-orientation, it can specify the inheritance relationship among classes. It allows single and multiple inheritances. For instance, if we write

  class spec C **is** ... **inherits** D ...

  we mean the operation "add all features of D to the features of C". These features comprise attributes and methods.
- The attributes of a class are generic with their given types as parameters. That means that a class can serve as a template for other classes, in which the template may be parameterized by other classes, objects, and/or methods. A generic class must be instantiated (its parameters filled in) before objects are created. The attribute consists of constants, variables and their types. Attributes also are represented with constraints which restrict the value or the range of that attribute.
- The specification of input and output parameters for each method includes local variables as well as class variables.

$m : S \times I \longmapsto S \times O$

S: class variable (representing class state)

The appropriate output may not be just a function of the input; it may also be a function of the current state of the class. In other words, the method must establish a complete mapping $S \times I \longmapsto S \times O$.

The formal specification language we use for describing OORS is an extension of that by Breu [14]. In order to support parameterized classes, we extend the **attribute** part in [14] with genericity features. The specification of input and output parameters for each method includes class variable to represent the class state. It combines algebraic specifications and object-oriented specifications such as classes.

The OORS $Spec[Sys]$ of a software system Sys can be described by the specification $Spec[C_i]$ of classes $C_i, i = 1, 2, \cdots, n$. Let a software system Sys consist of classes $C_i, i = 1, 2, \cdots, n$. We have

$$\begin{aligned} Spec[Sys] &= Spec[C_1, C_2, \ldots, C_n] \\ &= Spec[C_1] + Spec[C_2] + \ldots + Spec[C_n] \end{aligned}$$

A class specification $Spec[C_i]$ is defined as follows:

class spec C_i **is**
  inherits $I_1, I_2, \ldots, I_l$
  attribute $A_1, A_2, \ldots, A_m$
  methods $M_1, M_2, \ldots, M_p$
  axioms $Ax_1, Ax_2, \ldots, Ax_q$
end class spec

where

$C_i, 1 \leq i \leq n$ represents the name of a class.

$I_i, 1 \leq i \leq l$ represents the name of a superclass.

$A_i, 1 \leq i \leq m$ represents the class variable and its type, the constant, the local variable and its type.

$M_i, 1 \leq i \leq p$ represents the name of a method, its type of parameters and operation of its method. **create** method is also involved.

$Ax_i, 1 \leq i \leq q$ represents the axiom based on the dynamic behavior showing the events the object receives and sends.

Our formal specification language uses an algebraic specification to describe the behavior of the system. The axiom parts in the class specification represent the algebraic specification. The algebraic specification describes a data type from a purely external viewpoint by stating the properties of their operations *methods*. In other words, it does not contain any internal representation. In the algebraic specification, an object is specified in terms of the relationships among the operations that act on that object. The algebraic specification has two kinds of operations: *constructor operations* that create or modify entities which are defined in the specification and *inspection operations* that evaluate attributes which are defined in the specification. Each axiom may have a *guard* which is a predicate. When the guard is true, the axiom is said to be *enabled* for them. When the guard is false, its operation is delayed until it is true. The guard is given after the keyword **when**.

4 Transformation of OORS into formal specification

In this section, we will present the transformation technique from the OORS to the formal specification written in the formal specification language. The technique can be described in the following steps.

Step 1.1 Specify class name, inheritance, and local variables.
The object model contains classes, including class name, attributes and operations in the class, and their relationship. The class name can be written in the following format.

class spec *Class_name* **is** **end class spec**

Inheritance classes are put in the *inherits* part in the formal specification. We convert the attributes of each class from the object model into local variables of a class and identify their types. Their types are generic.

Step 1.2 Specify methods of each class and constraints of attributes.
We determine the domain and range types of a method, and define constraints of each attribute if necessary.

Step 1.3 Define axioms.
We define the relationship between every pair of methods from the dynamic model, and then make axioms to represent all the sequences of methods of a system that are concerned with time. That is, all the sequences of events that occur during the execution of a system are represented.

5 Building the information tree

We build an information tree from the formal specification written in the formal specification language. We can explore all information in OORS by traversing the information tree. However, the current notations used in the information tree does not represent all dynamic behavior, most of which come from the domain knowledge.

The root of the tree represents the system. The system consists of classes and each class has functions. Each function has attributes and constraints. A path in the information tree starts from the root node and traverses the tree through the class, function, attribute and constraint nodes. A relationship among classes represents a physical or conceptual connection among classes. The relationships among classes in the information tree are represented by dashed lines, that is, the invocation of a function in another class is represented by a dashed arrow line from the invoking class to the invoked class. A typical structure of an information tree is shown in Figure 2.

The information tree can be built as follows.

Step 2.1 Identify classes and their communication relationship from the formal specification. Classes are identified from the class name in the formal requirements specification. We can identify the communication among classes from the method invocation of a class to other classes.

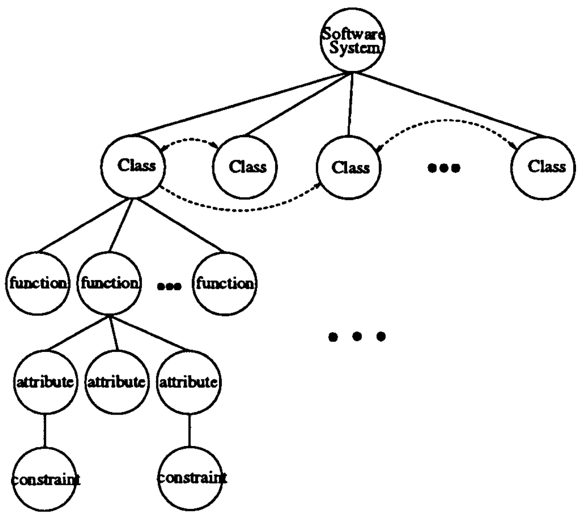

Figure 2: The structure of an information tree

Step 2.2 Identify the functions for each class from the *methods* part of the formal specification. Functions are "what" of a product, describing what the product is to accomplish. We capture all functions and understand those functions and make them children of the class in the information tree.

Step 2.3 Identify the attributes of each function from the *attribute* part in the formal specification. The attributes are characteristics which are required by the user or the client, and are data values held by each object. They are represented as the local variables or as the formal parameters of a function. There may be a relationship between different attributes. Thus, different attributes may be interrelated and dependent. The attributes of each function are attributes of the entire software system and the same attribute may qualify more than one function.

Step 2.4 Identify the constraints of an attribute if exist. We identify the constraints of attributes from the *attribute* part in the formal specification. The constraints of an attribute appear between the parenthesis following an attribute variable in the formal specification. Constraints represent mandatory or boundary conditions placed on the attribute. The constraints of an attribute must be satisfied in the software development.

6 Checking the completeness and consistency of the requirements specification

In this section, we will discuss how to apply the top-down and bottom-up approaches to verify the completeness and consistency between the given requirements statements and the OORS. The OORS is represented as an information tree with two kinds of information: operations in the class and relationships among classes.

Step 3.1 Top-down approach
For each statement in the original requirements statements,

Step 3.1.1 Identify the statement either as an operation statement or a relationship statement.

Step 3.1.2 Find the component in the information tree corresponding to each requirements statement. For an operation statement, we find a path starting from the root of the information tree and search for the corresponding class, function, attribute and constraint. For a relationship statement, we search for the relationship among classes represented by the dashed lines among the class nodes.

Step 3.1.3 Check for consistency and completeness. We compare the given requirements statement with the information represented by its corresponding path or relationship for consistency. If we do not find the corresponding path in the information tree, then we conclude that there is a missing function in the OORS. If we do not find the corresponding relationship among classes in the information tree, then we conclude that there is a missing relationship among classes in the OORS. If we find a path or relationship in the information tree, but it is not equivalent to the given statement, then we find an inconsistency. We also verify the completeness by repeating the above steps for all statements in the requirements statements.

Step 3.2 Bottom-up approach.

Step 3.2.1 Construct the statement or the enumeration of the words representing a path from the root to leaf node or a relationship among classes in the information tree. When we construct the statement for a path from the root to a leaf node, the root node represents the software system we develop, the class node can be a subject, the function can be a verb, the attribute can be a noun and the constraint can be an adjective or adverb. However, sometimes we cannot make a statement in English, but may still make an enumeration of the words from the class name, function name, attribute name and constraint name.

Step 3.2.2 Check for consistency and completeness. We search for a statement in the original requirements statements corresponding to the statement made in Step 3.2.1. If we do not find the equivalent statement in the requirements statements, we determine whether this information is from the domain knowledge by checking against the domain knowledge added during OOA. Otherwise, we have an inconsistent error in OORS. We repeat the above steps for all paths and relationships among classes in the information tree to verify the completeness of the OORS.

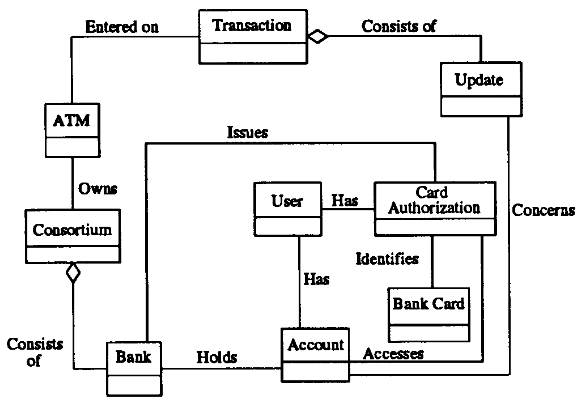

Figure 3: The object diagram for the ATM system

7 An example: an automated teller machine (ATM) system

In this section, we will use a simplified ATM system as an example to illustrate our verification approach. Our approach starts from the given requirements statements and the OORS expressed in terms of the object model and dynamic model generated by OOA. We will use the OOA approach developed by Rumbaugh et al [6].

The requirements statement of the ATM 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 a consortium. An ATM accepts a bank card, interacts with the user, communicates with the central computer to process transactions, and/or dispenses cash.

The object model shows the static structure of the anticipated software system and organizes it into workable pieces. It also describes the object classes and the relationship among them. The object diagram of the ATM system is shown in Figure 3. The dynamic model shows the time-dependent behavior of the system and its objects. It is implemented by preparing a state diagram for each object class with dynamic behavior showing the events the object receives and sends. For example, the state transition diagram of the class *ATM* is shown in Figure 4. Once we have the object model and dynamic model, we show how to apply our verification approach by the following steps.

Step 1: Transform OORS into formal specification

The first step of our verification approach is to transform the OORS into the corresponding formal specification. The following shows the transformation for the class *ATM*.

Step 1.1 Specify the class name, inheritance, and local variables as follows:

```
class spec  ATM  is
```

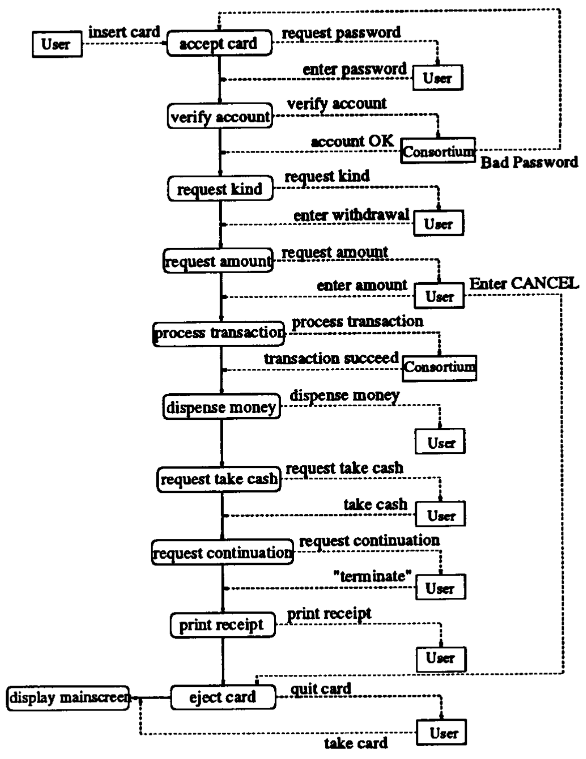

Figure 4: The state diagram for class *ATM*

```
   attribute
     cash: cash_type
     transaction_success: boolean
 end class spec
```

Step 1.2 Specify the methods of each class and the constraints of local variables. For the class *ATM*, we have

```
class spec  ATM  is
   attribute
     cash: cash_type (≤ 200)
     transaction_success: boolean
   method
     request_password ( ATM × card → ATM )
     verify_account ( ATM × password → ATM )
     request_kind ( ATM → ATM )
     request_amount ( ATM × kind → ATM )
     process_transaction ( ATM × amount
                      → ATM × transaction_success )
     dispense_cash ( ATM × transaction_success
                      → ATM × cash )
     eject_card ( ATM → ATM × card )
     display_mainscreen ( ATM → ATM )
 end class spec
```

Step 1.3 Define the axioms for each class. For the class *ATM*, we have

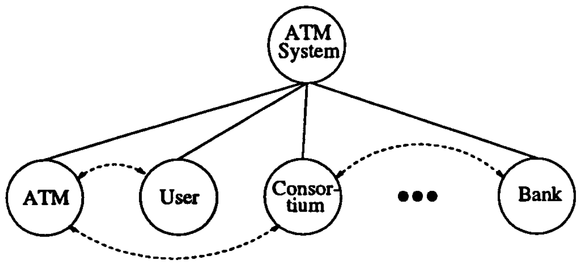

Figure 5: The information tree after adding class information (step 2.1)

```
class spec ATM is
  attribute
    cash: cash_type (≤ 200)
    transaction_success: boolean
  method
    request_password ( ATM × card → ATM )
    verify_account ( ATM × password → ATM )
    request_kind ( ATM → ATM )
    request_amount ( ATM × kind → ATM )
    process_transaction ( ATM × amount
                   → ATM × transaction_success )
    dispense_cash ( ATM × transaction_success
                   → ATM × cash )
    eject_card ( ATM → ATM × card )
    display_mainscreen ( ATM → ATM )
  axioms
    request_password ( ATM, insert_card(User) )
    verify_account ( ATM, enter_password(User) )
    request_kind ( ATM, account_ok(Consortium) )
    request_amount ( ATM, enter_kind(User) )
    process_transaction ( ATM, enter_amount(User) )
    dispense_money ( ATM,
                   transaction_succeed(Consortium) )
    eject_card ( ATM )
    display_mainscreen ( ATM )
end class spec
```

Step 2: Building the information tree

Next, we build an information tree from the formal specification. When we build the information tree, we add information such as object classes, functions for each class, attributes of a function and constraints for attributes as sequences. The information tree after including such information for the object classes is shown in Figure 5, and after including the information for the function in each class is shown in Figure 6. We add the information for the attributes of each function and constraints for the attributes if any as shown in Figure 7.

Step 3: Checking the completeness and consistency

For each statement in the given requirements statements, we apply the top-down approach. For instance, *ATM dispenses cash.*

Step 3.1.1 This statement is classified as an operation statement.

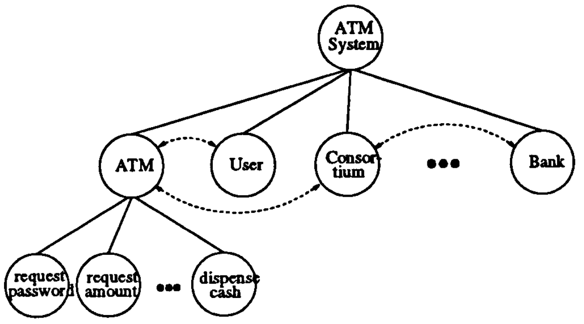

Figure 6: The information tree after identifying functions in each class (step 2.2)

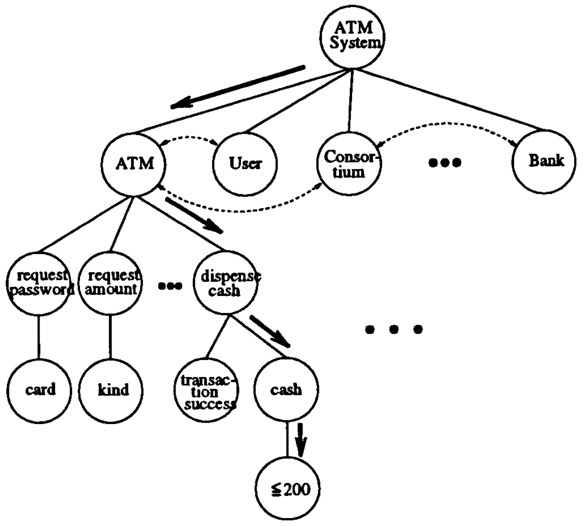

Figure 7: The information tree for the ATM system

Step 3.1.2 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 arrows in Figure 7.

Step 3.1.3 We compare this path and the statement for the consistency. As a result, we find an inconsistency, ≤ 200, in the OORS

We apply the bottom-up approach as follows:

Consider the path in the information tree, *ATM, dispense_cash, cash,* and *less than or equal to $200,* as an example.

Step 3.2.1 We make the statement or the enumeration of words for that path as follows:
ATM dispenses cash (less than or equal to $200).

Step 3.2.2 We search a statement in the requirements statements matched with the statement represented by that path. We find the statement in original requirements statement, "ATM dispenses cash". We determine that *less than or equal to $200* is from the domain knowledge or a mistake made during OOA. Comparing it with the domain knowledge added during OOA, we find that "less than or equal to $200" is from the domain knowledge.

8 Discussions

We have presented an approach to object-oriented requirements verification in software development for distributed computing systems. In our approach, we verify the completeness and consistency between the original requirements statements and OORS. By representing the OORS as an information tree, it is very convenient for the software developer to understand and check the requirements specification, and it is easy to visualize the structure of the software system under development. Our approach can also be automated because we can systematically compare the requirements statements with the OORS through the steps in our approach. The software tools and environment to support our approach, such as automatically transforming the OORS to the formal specification and graphically representing the information tree, will be developed in the future along with an interactive software development environment.

Currently, we assume that the requirements statements are unambiguous. Ambiguities in the requirements statements may lead to different interpretations of the software system, thus making the verification more difficult. Further research is needed to deal with the verification of requirements statements containing ambiguities.

The current notations used in the information tree can be used to represent the simple relationship among the nodes in the information tree such as an operation with single attribute in a class and communication relationships among classes, and does not describe all the dynamic behavior, most of which come from the domain knowledge. The information tree can be extended to represent more complex control information such as *and* and *or* relationship among the nodes in the information tree so that our approach can also represent complex software systems and describe all information in OORS, especially the dynamic behavior of the system . In addition, by extending the information tree with hierarchical structure, our approach can be applied to the requirements verification of large-scale software for distributed computing 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] S. S. Yau, X. Jia and D.-H. Bae, "Software Design Methods for Distributed Computing Systems," *Journal of Computer Communications*, Vol. 15, No. 5, May 1992, pp. 213–223.

[3] G. Booch, "Object-Oriented Development," *IEEE Trans. on Software Engineering*, Vol. 12, No. 2, Feb. 1986, pp. 211–221.

[4] S. S. Yau, and G. -H. Oh, "An Object-Oriented Approach to Software Development for Autonomous Decentralized Systems," *Proc. International Symposium on Autonomous Decentralized Systems*, 1993, pp. 37–43.

[5] S. S. Yau, D.-H. Bae and M. Chidambaram, "Object-Oriented Development of Architecture Transparent Software for Distributed Parallel Systems," Journal of Computer Communication, Vol. 16, No. 5, May 1993, pp. 317–326.

[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] P. Coad and E. Yourdon, *Object-Oriented Analysis*, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[8] S. Shlaer and S. J. Mellor, *Object-Oriented ystems Analysis*, Yourdon Press, Englewood Cliffs, NJ 1988.

[9] W. J. Quirk, *Verification and Validation of Real-Time Software*, Springer-Verlag, Berlin, Heidelberg, 1985.

[10] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends," in *Current Trends in Concurrency*, ed. Bakker, Roever, Rozenberg, Lecture Notes in Computer Science, Vol. 224, Springer-Verlag, 1986, pp. 510–584.

[11] B. Berthomieu and M. Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets," *IEEE Trans. on Software Engineering*, Vol. 17. No. 3, Mar. 1991, pp. 259–273.

[12] R. Gerber and I. Lee, "A Layered Approach to Automating the Verification of Real-Time Systems," *IEEE Trans. on Software Engineering*, Vol. 18, No. 9, Sep. 1992, pp. 768–784.

[13] C. J. Fidge, "Specification and verification of real-time behaviour using Z and RTL," *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Lecture Notes in Computer Science, Springer-Verlag, 1991, pp. 393–410.

[14] R. Breu, *Algebraic Specification Techniques in Object Oriented Programming Environment*, LNCS, Springer-Verlag, Berlin, Heidelberg, 1991, pp. 65–111.