NORTH-HOLLAND

Verification of Object-Oriented Systems: Domain-Dependent and Domain-Independent Approaches

Nils A. Kandelin
George Mason University, School of Business Administration, Fairfax, Virginia

Daniel E. O'Leary
University of Southern California, School of Business, Los Angeles, California

This article investigates verification tests for an object-oriented accounting system. Those tests exploit the structure available from both the domain and application, and the structure of the knowledge representation (the nature of objects). The verification tests are implemented, in part, within a prototype object-oriented accounting system. Tests are included for objects and their instances. In addition, the system includes objects whose sole duty is to verify the system.

1. INTRODUCTION

Adrion et al. (1982) defined verification as "the demonstration of the consistency, completeness and correctness of the software." In artificial intelligence (AI) systems, many verification efforts are aimed at verification of the "stored intelligence" or knowledge of the system. In knowledge-based and expert systems, such stored intelligence typically is referred to as a knowledge base. Frequently, efforts are concentrated on the knowledge base because design and development efforts treat knowledge base acquisition as a specific and identifiable task, and because available computer software shells often require that the user only furnish the system with the appropriate knowledge.

If assumptions are made about both the nature of the problem being solved (e.g., the domain is financial and accounting systems) and the structure of the storage of the intelligence (e.g., objects or rules), then the pursuit of verification can take greater specificity. As a result, verification efforts in AI generally have taken either of two primary approaches. First, they have concentrated on using meta knowledge about the domain in the investigation of the consistency, completeness, and correctness of the knowledge. Second, they have assumed an underlying structure or type of knowledge representation, e.g., a knowledge base of rules.

The approach in this article is to develop verification approaches that use both sets of assumptions: domain and knowledge structure. In particular, this article discusses verification of object-oriented AI systems using ACTOR, an object-oriented programming language, while using meta knowledge about finance and accounting data bases. Although focused on a specific domain and developed in a specific language, the systems discussed in this article are generalizable to other domains and object-oriented languages. This article extends previous research in verification, which has focused primarily on rule-based systems, and work done on the verification and validation of intelligent accounting systems (O'Leary, 1987).

1.1 Verification

Verification is concerned with building the system right (O'Keefe et al. 1987). As a result, verification is concerned with building the system in concert with the underlying technology. For example, in the case

Address correspondence to Prof. Daniel E. O'Leary, School of Business, University of Southern California, 3600 Trousdale Parkway, Los Angeles, CA 90089-1421.

J. SYSTEMS SOFTWARE 1995; 29:261–269

655 Avenue of the Americas, New York, NY 10010

0164-1212/95/$9.50
SSDI 0164-121294(00109-Z

of rule-based systems, verification is concerned with ensuring that there are no sets of rules that have cycles in them, or that each rule is in the system no more than once (Nguyen et al. 1987).

Validation is more concerned with the quality of the decisions made by the system. Validation is concerned with building the right system (O'Keefe et al. 1987). As noted by Adrion et al. (1982), "validation is the determination of the correctness of the final program or software produced from a development project with respect to the user needs and requirements."

As noted above, Adrion et al. (1982) defined verification as the demonstration of the consistency, completeness, and correctness of the software. Thus, we base our analysis on those three characteristics. In addition, a fourth characteristic, redundancy (e.g., multiple versions of the same rule), an aspect of completeness, is treated independently. Redundancy is used as an additional basis for a number of reasons. First, a number of researchers of rule-based systems have treated redundancy as a separate category (Nguyen et al., 1987; Nazareth, 1989; Preece et al., 1992). Second, there are a number of approaches and issues for checking redundancy in object-based systems, and thus it is useful to address those methods separately. Third, redundancy per se is not a problem in most cases. Instead, the difficulties occur with the maintenance of a knowledge base with redundancies. For example, with system maintenance, one representation may be changed but another may not.

There are a number of definitions of consistency, completeness, correctness, and redundancy (Adrion et al., 1982; Nguyen et al., 1987; Nazareth, 1989; Preece et al., 1992). Consistency refers to the treatment of similar structures and terms in the same manner. Completeness is concerned with examining the issue that knowledge is missing. Correctness is concerned with determining whether there are any ascertainable errors in the represented knowledge. Redundancy addresses the issue of duplication of represented knowledge.

1.2 Domain-Dependent and Domain-Independent Approaches

Previous verification investigations have concentrated on two basic approaches: domain dependent and domain independent (Nazareth, 1989). Domain-dependent verification uses meta knowledge from the domain to examine the consistency, completeness, correctness and redundancy of the knowledge base (Davis, 1976). Consider the example of verification of knowledge about a living room. It would be inconsistent if we labeled two different items in the living room with the same name, e.g., calling both couch and a chair, a "couch." Meta knowledge could be used to ascertain that knowledge about a specific living room probably would be incomplete if the living room did not contain a couch. In addition, it probably would be incorrect if the knowledge places a bathtub in the living room. Similarly, there may be redundant knowledge if there were two couches in the living room.

Domain-independent approaches have concentrated on the structural characteristics of rules (Nazareth, 1989) and the correctness of the weights and probabilities of the rules (O'Leary, 1990) in the development of those systems. In the case of rules, if an item from a list of input facts is not used in the rule base, then that could signal a missing rule—incompleteness. Similarly, two identical rules would indicate the presence of redundant rules. The existence of rules that lead to circular reasoning or weights that violate the underlying probability structure would indicate incorrectness in the rules. Use of different terms to the same fact or concept would indicate inconsistency. These domain-independent approaches have been summarized in a number of formal approaches and tools, including Nguyen et al. (1987), Chang et al. (1990), and Preece et al. (1992). This article expands the analysis beyond rules to object-oriented systems.

This article proceeds as follows. Section 2 summarizes some critical aspects of general objects and the particular object-oriented software (on which the system developed in this paper is based) ACTOR (Duff et al., 1987). Section 3 provides a brief summary of the object-oriented system for which the verification processes were designed. Sections 4 and 5 discuss the domain-independent verification procedures for object-oriented programming in AI. Section 6 develops a domain-dependent approach for the verification of knowledge in that system. Section 7 briefly summarizes the article.

2. OBJECTS AND AN OBJECT-ORIENTED LANGUAGE

In object-oriented programming, problems are defined by use of a set of objects, which function relatively independently of each other, except for interactions along established communication channels. Experiences of programming with objects and some of the current research in the area are discussed in Stefik and Bobrow (1986), Kim and Lochovsky (1989), Tello (1989), Zdonik and Mayer

(1990), Booch (1991), and Rumbaugh et al. (1991). The object-oriented paradigm includes languages, applications, and design methodologies. In this article, we focus on object-oriented languages and their applications. This section provides background on objects and a particular object-oriented language.

2.1 Selected Characteristics of Objects

Structure refers to the relationship between different objects. Typically, that structure can be represented as either a tree or an acyclic network (Winston, 1984).

Encapsulation is the concept that refers to the fact that objects have private memory and sets of operations. Encapsulation prevents the object from being manipulated by anything other than its own previously defined external operations. Often this is referred to as information hiding. Other objects do not need to know all that goes on in every other object.

Delegation refers literally to the assignment of certain messages to other "lower level" objects (e.g., in the tree). This allows the objects to focus their concern on other internal tasks or other messages.

Objects are communicated with by use of *messages*. Messages are specifications of procedures to be performed on objects. In the system discussed here, they provide financial and accounting information to the objects. *Procedures* (methods) are functions that objects use to respond to messages.

Classes are a description of one or more similar objects. *Instances* refer to objects that are not classes. As noted by Stefik and Bobrow (1986), "For example, in a traffic simulation program there may be one class named truck and hundreds of instances of it representing trucks on the highways of the simulation world" (p. 44). *Attributes* are characteristics that describe an object. For example, in the instance of a particular truck, one attribute may be that the speed of the truck is 60 miles per hour.

Inheritance is the feature whereby an object receives a subset of the characteristics of its immediate predecessor in the tree or network.

2.2 ACTOR — An Object-Oriented Programming Language

The ACTOR language was used to develop the prototype commercial decision application described in Section 3.2 ACTOR is a "pure" object-oriented programming language, as defined by Booch (1991), because all language components, such as integers, are represented as objects (Whitewater Group, 1990). Additionally, all activities within ACTOR conform to the message-sending paradigm.

The ACTOR system comes with a large hierarchy of objects (or a class tree) already defined. Included are frames and a frame representation language capability used to create, modify, and query the frames. The ACTOR class tree can be easily modified and extended by use of the programming tools provided with the system.

2.3 Other Object-Oriented Languages

There have been a number of other object-oriented languages developed and discussed in the literature, including SMALLTALK and LOOPS. A review of LOOPS, in particular, and characteristics of other languages, in addition to a thorough discussion of object-oriented programming, are presented in Stefik and Bobrow (1986). Goldberg and Robson (1989) describe the latest version of SMALLTALK, SMALLTALK-80.

3. THE SAMPLE DOMAIN: CREDIT DECISION MAKING

The focus of domain-dependent verification discussed here is on an object-oriented application built for commercial credit decisions. This is a critical application domain affecting a number of financial and accounting processes. This section summarizes some key aspects about the domain and the particular system.

3.1 Commercial Credit Decisions

The different kinds of credit can be classified according to the debtor's responsibility which could be public or private. Furthermore, private credit can be divided into consumer and commercial credit. Commercial credit allows for the exchange of goods and services between businesses with the promise of future payment. Cole (1984), states that "commercial credit is perhaps the outstanding example of how the economy operates on a self-liquidating credit basis, with an estimated 90–95% of transactions between commercial and industrial concerns being carried through the medium of credit exchange" (p. 14).

The granting of commercial credit involves two processes: credit or financial analysis and the credit decision. Credit analysis attempts to evaluate the risk that the customer will not be able to pay or will pay late by examining financial statements and information available from credit agencies. The result of the credit analysis is the credit limit, i.e., the maxi-

mum amount of credit that the customer is allowed. The other process, credit decision making, uses the results of the credit analysis to determine how much credit should be extended to a customer on a case-by-case basis. The credit decision determines the amount of credit to be extended in a particular situation and the associated credit terms.

An important difference between the two processes, credit analysis and credit decision, is in the frequency they are performed. The credit analysis is performed on a periodic basis, whenever new financial or credit information might be available. The credit decision is made for each order received from the customer. The credit decision must be made often, and speed and accuracy are required to preserve customer good will. Thus, it is a good candidate for automation.

3.2 Application

Kandelin (1990) developed a conceptual model of an accounting information system designed to support a wide variety of financial and accounting decision-making tasks. This system uses an entity-relationship model (Chen, 1976), referred to as an events-based representation system (McCarthy, 1982). The set of applications investigated in Kandelin (1990) includes the credit decision. The conceptual model was tested by developing a computation model that uses the ACTOR object-oriented programming system. That model is used to demonstrate verification techniques for object-oriented programs.

The computational form of the model, shown in Fig. 1, consists of three classes of objects, which receive, process, and react to the events represented by the messages. One set of objects is used to form an event message data base, which provides historical information. Report objects, which represent accounting reports, present synthesized information from the accounting even messages to the third class of objects, which constitute a knowledge base consisting of frames.

The prototype implements a message structure based on the entity relationship and adaptations of it to the finance and accounting data base model (Chen, 1976; McCarthy, 1982). The message describes the resources affected (the economic resource), the activity affecting the resource (the economic event), and the outside entity involved (the economic agents). The economic unit (the involved inside party) can be inferred from the economic event by knowing who is responsible for that activity within the organization; therefore, it is not represented in the message structure used in the prototype system.

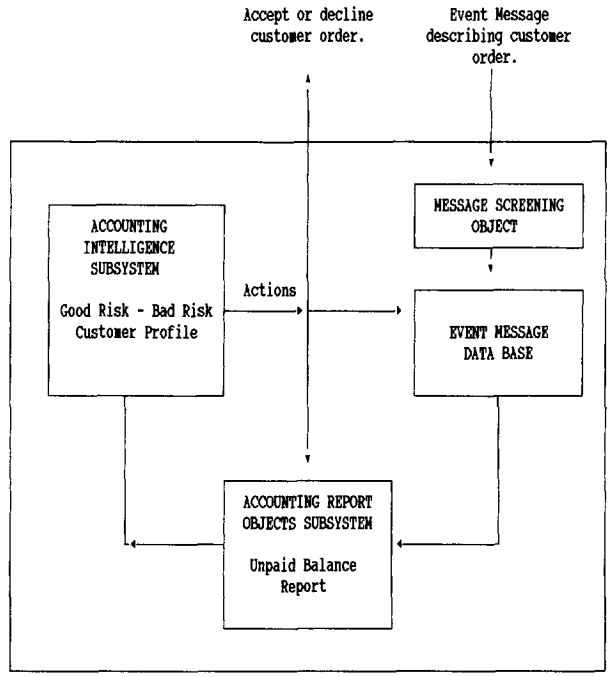

Figure 1. The computational model of an events-based accounting information system to support credit decision making.

An example of a report object would be an "unpaid balance" report (Figure 2). The unpaid balance report is maintained for each customer. This report updates itself with every transaction for that particular customer, such as sale or receipt of payment. The unpaid balance report uses information in the event message to determine if the message requires that the report updates the unpaid balance. If the event message concerns the particular commercial customer, then it processes the messages. If it involves another entity, then the message is disregarded by the report.

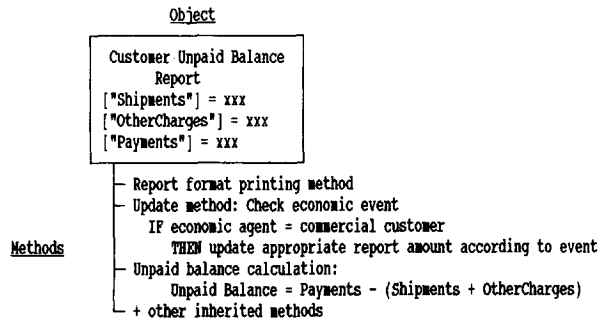

Figure 2. Unpaid balance report.

By use of various report objects to represent synthesized information from the event messages, knowledge from human decision makers can be represented directly in the knowledge base. For example, if a human expert uses a specific piece of information from a certain report, then that knowledge would be elicited from the expert. Using a report object, this information would be synthesized independently from the knowledge base and be available in the optimal format for decision making while permitting a separation of data processing from knowledge procesing.

The frames that make up the knowledge base include representing knowledge about the profile a good (low risk) and a bad (high risk) customer along with each customer's profile. The individual customer profiles contain information on (as well as the method required to calculate) the current credit standing of each customer, which is used for comparison with the credit risk profiles at the time the credit decision is to be made. If the customer matches the bad risk profile, then stricter credit terms might be required. If the customer matches the good risk profile, then credit will be granted. An example of the frames for good and bad risk customers is shown in Table 1. The frames for good and bad risk are ancestors of the known risk frame, which is established through "a kind of" (AKO) slot.

4. DOMAIN-INDEPENDENT VERIFICATION OF CLASSES

Sections 4 and 5 develop some domain-independent verification approaches. Because of fundamental difference between object classes and object instances, the two are discussed separately, with this section focusing on classes and the next section on objects that are instances.

4.1 Consistency

Consistency is necessary to confirm that the same thing (e.g., instance variables and methods) is called by a single name. One approach to consistency is to require that all names be established on a list of feasible names. Then all names in the model would come from that list. This ensures that a single name exists, rather than different names or multiple spellings for the same thing.

Two aspects of objects jeopardize the ability to verify consistency: encapsulation delegation. Encapsulation can facilitate an inconsistent approach to naming conventions resulting from treating each object independently. An environment aimed at facilitating verification is not one that would be consistent with the significantly different approaches and names within different objects associated with encapsulation.

Delegation also can lead to an environment that does not encourage verification, particularly if delegation is handled in an inconsistent manner, delegating some objects but not others. Verification efforts aimed at an object cannot treat the object as an independent entity if there is substantial and inconsistent delegation. As a result, delegation can significantly complicate verification efforts.

Table 1. Frames of Representation of Types of Customers

Frame	Slot	Facet	Value
Known risk	Credit analysis	Default	"Up to date"
Good risk	A K of	Value	Known risk
	Unpaid balance	Value	"Normal"
	Payments	Value	"On time"
	Credit Limit	Value	"Below"
Bad risk	A K of	Value	Known risk
	Unpaid balance	Value	"Large"
	Payments	Value	"Late"
	Credit limit	Value	"Above"

4.2 Redundancy

Redundancy in objects can occur in at least different ways: redundant objects, redundant content, and redundant connections with other objects.

Redundant objects. The same object may accidentally be constructed more than once. In that case, such redundancy can be found by comparing the objects and each of their attributes. An object can be placed in charge of this assessment process as part of the system design.

Redundant content. The content can be redundant in any of a number of cases, including class variables, instance variables, and methods. A verification system can mitigate redundancy, for example, by not allowing the user to put the same method in the object more than once.

Redundant connections. Some languages such as ACTOR eliminate the ability of the developer to inadvertently develop redundant connections by allowing only one connection hierarchically above and one hierarchically below. However, it is still feasible to cycle from one object to another object and back to the original object. Thus, it is important to ensure that the overall tree of object relationships does not have any such cycles or redundant connections. If

there are redundancies, then it may be impossible to establish appropriate rules for trait inheritance.

4.3 Completeness

The issue of inheritance, particularly from multiple parents, raises concern that a complete set of traits has been inherited. In some cases, by only inheriting a subset of traits, critical traits can be lost, or traits that are incomplete without other traits might be inherited, or subsets of inherited traits may conflict, because of a missing (but previously present) trait. An object may be described as a boat without a paddle. The same issues of completeness occur whether there are single or multiple parents.

Another issue in completeness is the completeness of the set of objects from which messages can be received and to which messages can be sent. Verification requires that there are two such sets of feasible objects associated with the object: source and target objects. In ACTOR, a single target object can be specified.

The list of names discussed in the section on consistency also furnishes a completeness test; if a name is on the list, then it should be used by at least one object. If it is not in some object, than that means that, possibly, a part of the design of the system has not been included. Thus, the system would be incomplete.

4.4 Correctness

As noted earlier, the desired structure in object-oriented programs typically is either a tree, as in ACTOR, or an acyclic network. Thus, one verification check is to ensure that the resulting structure of object has no cycles, if it is network, or that it is a tree structure.

An approach to verification of these concerns is to allow a single precedence node, as in ACTOR. However, that approach alone does not guarantee that no cycles are present or that a tree structure is maintained.

Another correctness check is to ensure that the object is not connected to itself. This could lead to logical difficulties and impede inheritance. For example, a frame object can be specified as AKO (or a child of) another frame and inherit slots and values of the parent frame. However, then the parent frame can become AKO the child frame, which results in a circular reference.

In the case where an object has multiple parents, an important correctness issue is what characteristics it inherits. In some languages, there are default orderings on inheritance (Stefik and Bobrow, 1986). As a result, it can be critical to denote sets of conflicting traits that would not be desirable in combination with each other. For the credit decision system, inconsistent or conflicting inherited characteristics or processes can either be prevented by systems design or detected by an object that analyzes the other objects in the system.

5. DOMAIN-INDEPENDENT VERIFICATION OF INSTANCES

Verification of instances is the verification of various aspects of specific occurrences of the objects. A number of different tests can be made to verify the instances. The data within different instances may include either numeric or nonnumeric data. The existence of different data types (numeric and nonnumeric) suggests that tests of correctness include tests of data type. In addition, we would expect that in most instances the attributes be completed in their entirety for the entire set of attributes. Finally, tests of redundant information can limit the extent to which duplication of data is present in the instances.

5.1 Consistency

One aspect of consistency, in cases of numeric data, is that the same attribute for each of a set of instances may be "about the same" (not unusual). The expectation that those instances are about the same indicates that "sameness" can be used to examine those instances to confirm sameness, with a message being sent to an object when it is found that those expectations are violated. Thus, a consistency verification object can periodically check each, or a specified attribute of a set of instances, to ensure that sameness holds.

In the case of nonnumeric data, where a code or a name is captured as an attribute, humans may have a tendency to make errors by using different names or misspellings for the same attribute. Thus, if there is a limited set of attributes, then it likely is appropriate to have the developer specify those attributes in list form, a priori. The list can be used to confirm that the same name (e.g., spelling) is used consistently.

Given such a list, the system would then require that the developer make choices from that list, and then the system would include the attribute in the object. The use of such system-implemented lists makes multiple spellings of the same attribute virtually impossible, thus putting the system in the position of implementing the developer's choice.

Requirements on consistency in one object carry through to other objects through the property of inheritance. If the object is only specified in one object, then that limits concern, with consistency, for that attribute in subsequent objects. As noted by Stefik and Bobrow (1986), "Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place (p. 41).

5.2 Redundancy

Redundancy could occur, in the development of instances, if the system would allow the user to put the same information in an object more than once. Redundancy can occur with the duplication of instances or the duplication of slots in instances (assuming it is inappropriate to have multiple identical slots or instances). This duplication could lead to the possibility that, during maintenance, only one occurrence may be updated, thus leaving the other in the object unchanged and potentially misleading.

A verification check that can be used by the system during operation ensures that the data are put in the system only one time in one place. However, if no such check is made on data input, then an object can be built to check instances for redundancy. Such an object would review the slots in the instances for duplication and instances for overall duplication.

5.3 Completeness

It is critical to guarantee that an instance is completed with the appropriate data before the system accepts it as a complete instance. Otherwise, partially completed instances will be embedded in the system or possibly inappropriate default values will be inherited. As a result, a verification issue that can be built into the system to ensure that the appropriate data are included is to require the user to specify "Default" when the user is planning on using the default values.

In those situations where not all slots will be filled in, another test can be used to ensure completeness. The developer can specify the number of slots in which the specific instance will include data or specify the actual slots that will be required to be completed for each instance.

5.4 Correctness

A number of approaches can be used to assess the correctness of instances. Some object-oriented programming environments, such as SMALLTALK, are "weakly typed" and do not require formal type of declarations, whereas "strongly typed" languages, such as C + +, PASCAL, and Ada, do. (Booch, 1991; Rumbaugh et al., 1991). Tesler (1981) noted that strong typing has several important advantages, including greater opportunity for optimization of the complied program code and improved program reliability by avoiding accidentally typed mismatches between variables. To benefit from the advantages of strong typing, a specific data type for values attached to an instance should be explicitly stated. This would be based on problem domain knowledge of the appropriate data types for that instance. In addition to checking the data type, the magnitude of the instance may also be tested via lower and upper bound checks. For example, a payment amount is not likely to be 100 times greater than an unpaid balance; if it is, then there is possibly an error.

5.5 Specification

This section has noted a number of different specification activities that can be used to ensure a verifiable system. This set of specifications derives from the requirements of building a system by use of an object form of knowledge representation. Lists of attributes can be specified to ensure completeness. Data types can be specified to ensure that appropriate numeric or nonnumeric data can be specified for specific slots or instances.

These specification activities generate sets of information and knowledge requirements needed as part of the knowledge acquisition and system design processes. These activities, required to ensure a verifiable system, might be viewed as the "minimal required specifications." Without specifying this information, it is not possible to ensure a minimal standard verification effort. Unless the information is specified, errors of the types noted above can occur.

6. DOMAIN-DEPENDENT VERIFICATION

Whereas the two previous sections have drawn on the nature and structure of objects for verification, this section discusses those parts of system verification that are, to some extent, domain dependent. Domain-dependent methods draw on structure in the domain and on the specific application.

The research presented here provides only limited domain-specific verification, primarily to illustrate its use with objects in accounting systems. By their very nature, these approaches are limited to the specific

application and domain, but can be extended to other applications and domains. The domain-dependent verification is accomplished, in part by use of two different objects: a message-screening object and an inspection object. The first is responsible for verifying the messages. The second is responsible for doing some verification of the objects in the system.

6.1 Consistency

The consistency of the messages transmitted between objects is critical because messages contain the necessary financial and accounting information on events. In this model, the structure of those messages is based on the entity-relationship financial and accounting data model. Thus, the system uses a method that analyzes each message to ensure that the composition of the messages the theory on which the system is based. For example, certain information in the message must be numeric, whereas other information should be nonnumeric. If there are type conflicts of numeric and nonnumeric information, then that indicates an error.

6.2 Redundancy

When it comes to the analysis of credit and the payment of bills, it is critical to confirm that the message is not sent or processed more than once. A redundancy check is done by use of a message identifier. In addition, the inspection object's knowledge of credit allows it to assess class objects to ensure that there is redundant knowledge being gathered. If there is redundant knowledge, then a report is issued indicating duplication.

6.3 Completeness

As a completeness test, the message-screening object checks the number of elements in the message list to ensure completeness. Each message must contain sufficient information to characterize an accounting event according to McCarthy's (1982) entity-relationship accounting model.

In addition, the inspection object examines the objects to find situations that it thinks are incomplete, based on its knowledge of the objects. For example, the inspection object's knowledge of credit requirements allows it to assess the classes to ensure that the objects contain sufficient information about credit.

6.4 Correctness

The inspection object knows who are eligible agents, what are eligible resources, and what are eligible units within the organization. Thus, that object can ensure that only correct resources, agents, and units are used in the system.

The inspection object also has knowledge and vocabulary about accounting in general and credit in particular. Thus, it is in a position to assess the "reasonableness" of class information. This is done by examining the class information to determine if it is "familiar" with the terms. If the information is not familiar, then the inspection object reports that either a change in class tems or a revision in its vocabulary is necessary.

6.5 Specification

This section generated a number of different specification activities that can be used to ensure a verifiable system. These specification activities, when combined with the specification activities for instances (section 5.5), provide a broad base of required information necessary to ensure a verifiable system. As in the previous discussion, these requirements derive from the use of an object representation.

7. SUMMARY AND EXTENSIONS

This article has presented both domain-independent and domain-dependent verification tests for object-oriented financial and accounting systems. Those tests were generated based on the structure of the object and the particular system. The tests were accomplished by use of a variety of approaches, including the development of objects that perform verification tasks and the development of methods within objects to verify information.

The verification portion of the system can be extended. More domain and meta knowledge can be incorporated into the system. An increase in the amount of both domain and meta knowledge would result in an increased extent of verification of object classes. However, the addition of meta knowledge could also lead to a system whose verification efforts would become increasingly limited to the specific domain.

ACKNOWLEDGMENT

We acknowledge the extensive comments of the anonymous referees and Robert Plant.

REFERENCES

Adrion, W., Branstad, M., and Cherniavsky, Validation, Verification and Testing of Computer Software, *Comp. Surv.* 14, 159–192 (1982).

Booch, G., *Object-Oriented Design*, Benjamin/Cummings, Redwood City, California, 1991.

Chang, C., Cowles, J., and Stachowitz, R., A Report on the Expert Systems Validation Associate, *Exp. Syst. Appl.*, 1, 217–230 (1990).

Chen, P. P., The Entity-Relationship Model, *ACM Trans. Database Syst.* 1, 9–36 (1976).

Cole, R., *Consumer and Commercial Credit Management*, 7th ed., Richard D. Irwin, Homewood, Illinois, 1984.

Davis, R., Use of Meta Knowledge in the Construction and Maintenance of Large Knowledge Bases, Ph.D. Thesis, Stanford University, Palo Alto, California, 1976.

Duff, C., et al., *Actor Language Manual*, The Whitewater Group, Evanston, Illinois, 1987.

Goldberg, A., and Robson, D., *Smalltalk-80: The Language*, Addison-Wesley, Reading, Massachusetts, 1989.

Kandelin, N., Integration of Data and Knowledge in an Events Accounting Information System. Ph.D. Thesis, University of Southern California, Los Angeles, California, 1990.

Kim, W., and Lochovsky, F. H., eds, *Object-Oriented Concepts, Databases and Applications*. ACM Press, New York, 1989.

McCarthy, W., The REA Accounting Model: A Generalized Framework for Accounting Systems in a Share Data Environment, *Accoun. Rv.*, 57, 554–577 (1982).

Nazareth, D., Issues in the Verification of Knowledge in Rule-Based Systems, *Int. J. Man-Machine Stud.*, 30, 255–271 (1989).

Nguyen, T. Perkins, W., Laffery, T., and Pecora, D., Knowledge Base Verification, *AI Magazine*, 69–75 (1987).

O'Keefe, R., Baloci, O., and Smith, E., Validating Expert System Performance, *IEEE Expert*, 81–90 (1987).

O'Leary, D., Validation of Expert Systems, *Decis. Sci.*, 18, 468–486 (1987).

O'Leary, D., Soliciting Weights or Probabilities from Experts for Rule-Based Systems, *Int. J. Man-Machine Stud.*, 32, 293–302 (1990).

Preece, A., Shinghal, R., and Batarelsh, A., Verifying Expert Systems: A Logical Framework and a Practical Tool, *Exp. Syst. Appl.*, 5, 421–436 (1992).

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

Stefik, M., and Bobrow, D., Object-Oriented Programming: Themes and Variations. *AL Magazine* 6, 40–62 (1986).

Tello, E. R., *Object-Oriented Programming for Artificial Intelligence*, Addison-Wesley, Reading, Massachusetts, 1989.

Tesler, L., The Smalltalk Environment, *Byte* 6, 142 (1981).

Whitewater Group, Actor User's Manual, The Whitewater Group, Inc., Evanston, Illinois, 1990.

Winston, P., *Artifical Intelligence*, 2nd ed., Addison-Wesley, Reading, Massachusetts, 1984.

Zdonik, S. B., and Mayer, D., eds., *Readings in Object-Oriented Database Systems*, Morgan Kaufman, San Mateo, California, 1990.