Object-Orientation in Requirements, Specifications and Models

TR-2001-12

Thomas A. Alspaugh and Annie I. Antón
Department of Computer Science
College of Engineering
North Carolina State University
Raleigh, NC 27695-7534 USA
+1-919-515-5764
{taalspau,aianton}@eos.ncsu.edu

December 4, 2001

ABSTRACT

Object-oriented requirements and specifications seem desirable for object-oriented design and implementation. However, in this paper we demonstrate that this is not necessarily the case. When we compare the conventionally desired attributes of requirements (e.g., easily modifiable, expressed in the customer's terms, and individual requirements which are separable) with some of the conventional properties of object-orientation (e.g., taxonomic organization), we see that in some ways they are incompatible with each other. While a specification may be expressed as an object-oriented model, there are disadvantages to doing so that have not been adequately considered in this context. We illustrate these distinctions with examples from the Kaiserslautern Light Control Problem and offer strategies to assist practitioners as they address these issues.

0.1 Keywords

Object-oriented requirements, object-oriented specifications, object-orientation.

1 INTRODUCTION

The success of object-orientation as a methodology for organizing and designing software has inspired researchers to adapt it for the organization and specification of requirements. Since the early 1990's a steady succession of publications have appeared in this area [8, 17, 22, 24]. However, several basic questions have yet to be adequately addressed:

- Are the desired characteristics of requirements and specifications consistent with object-orientation?
- Can object-oriented requirements and specifications achieve any of the benefits expected from object-orientation in software development?
- What advantages and disadvantages does object-orientation introduce with respect to the quality of requirements and specifications?

This paper addresses these questions, focusing on the extent to which object-orientation is desirable in requirements and specifications, how it can be most effectively used in this context, and in what situations it is advantageous or disadvantageous for software quality.

This paper is organized as follows: Section 2 examines the conventional roles and characteristics of requirements, specifications, and models, in preparation for considering them from the point of view of object-orientation. Section 3 provides some relevant background as well as an overview of related work. The application of object-orientation to requirements and specifications is examined in Section 4 which considers whether or not requirements are inherently object-oriented and explores the advantages and disadvantages of object-orientation in specifications. Section 5 discusses how object-oriented design methods sometimes obscure the conventional distinctions between requirements, specifications, and designs. Finally, in Section 6 we present our conclusions and discuss our plans for future work.

2 REQUIREMENTS, SPECIFICATIONS, MODELS

In this section, we define a few key terms to establish a common understanding for our discussion of requirements, specifications and models. Requirements concern "what" properties and functionality a system shall provide, whereas design components concern "how" the system shall meet the requirements. More often than not, models reveal units of implementation since models map function to structure. To illustrate the important distinctions between requirements, specifications models, we employ an example from our analysis of the Kaiserslautern University Light Control Problem [11]. The Light Control System (LCS) is intended to address the disadvantages of the university's existing system which is currently energy-inefficient due to the inability to adjust light sources relative to need and daylight (e.g., a room may be lit even when no one is occupying the room) since it is run manually. An informal description of the problem provides a set of functional, user, and non-functional needs [11]; this document served as the basis for our analysis and the examples included herein.

In Figure 1, we provide a Venn diagram to briefly summarize our discussion in this section. The diagram characterizes requirements, specifications and models by showing the relationships that exist among each within the context of object-oriented methodologies.

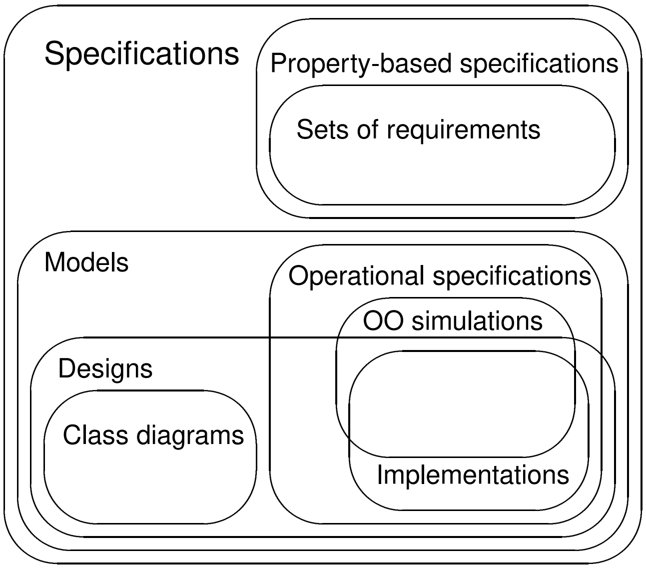

Figure 1: Relationships between kinds of specifications

2.1 Requirements

A *requirement* concerns the environment phenomena in which a system or machine interacts [15]. More simply, a requirement is a desired property of a system. Typically each requirement is separable from other desired properties. Requirements are frequently either invariants of a system, or history conditions on the system's states; they may be classified as safety or liveness properties. They are expressed in terms of externally visible behavior. Requirements evolve over time, so traceability (the ability to trace back to the sources of each requirement) is important. Requirements are frequently best expressed in natural language, although if the stakeholders for the system are mathematically sophisticated a formal notation may be more appropriate. In any case, it is desirable that stakeholders be able to understand the requirements, however they are expressed. Consider the Light Control System requirements provided in Figure 2. These requirements, while informal, provide specific information pertaining to the properties that the system must possess.

2.2 Specifications

A *specification* is a complete and accurate description or definition of the important characteristics of a system. Specifications should be free of "implementation bias", meaning that the description should not be contaminated by consideration of *how* the system might do *what* it must do [15]. More often than not, specifications are expressed formally, in a notation whose semantics are precise and unambiguous. For any specification, the decision of what characteristics are important is a defining one. A specification of a system can be written at any of a range of levels of detail, and with

Id.	Requirement
U13	"When a hallway section is occupied by a person, there has to be safe illumination."
FM1	"Use daylight to achieve the desired light setting of ... each hallway section whenever possible."
FM2	"The ceiling light group in each hallway section has to be off when the hallway section has been unoccupied for at least T2 minutes," where T2 is a value that can be set.

Figure 2: Informal Requirements from the Light Control Case Study

any of several emphases on the kinds of information that are included.

In software development, specifications provide unified pictures of the system, in which everything important is brought together and made consistent. For example, Figure 3 portrays the SCR specification [2, 13] that corresponds to the three informal natural language requirements of Figure 2. This SCR specification provides considerably more detail than the requirements in Figure 2. The specification elaborates the requirements by expressing them operationally.

2.3 Models

A *model* is a construction that expresses important properties of a system by instantiating them. While a specification may merely state an important property, a model possesses it. Object-oriented models purport to model the real world; however, the models really just describe programming objects [15]. Models behave like the system they model; they have analogous components that interact as the system's components do; or they consist of entities whose connections are the same as the connections within the system [7, 15]. For example, consider the state machine in Figure 4. This form of model is prominently used in both object-oriented and more traditional non-object-oriented methodologies. The finite state machine in Figure 4 corresponds to the natural language requirements listed in Figure 2 and the SCR specification in Figure 3.

In software development, a design is a model that is isomorphic to the high-level organization of the system's implementation. An object-oriented design's class diagram is a model containing boxes that represent each class, and lines between those boxes that correspond to relations between the classes. An object-oriented simulation is a model that acts out the interactions between a group of component objects, so that overall the behavior of the group of objects is an approximation of whatever the system is simulating.

We now consider the mutual roles of requirements, specifications and models within the context of software quality.

2.4 The roles of requirements, specifications, and models

There exist obvious similarities and mutually supporting relationships between requirements, specifications and models, and they correspond to the artifacts of software de-

Mode transition table for the mode class `mcStatus`			
Mode Class = `mcStatus`			Trace.
Old Mode	Event	New Mode	
`unoccupied`	`@T(mOccupied)`	`occupied`	U13
`occupied`	`@F(mOccupied)`	`temp_empty`	FM2
`temp_empty`	`@T(DUR(NOT mOccupied > mT2)`	`unoccupied`	FM2
	`@T(mOccupied)`	`occupied`	U13

Type dictionary		
Name	Type	Units
`yLight`	`enum in {off, on}`	none
`yLL`	`integer in [0,10000]`	lux
`yTimer`	`integer in [0,30]`	minutes

Condition table defining the value of cHallLL			
Mode Class = `mcStatus`			Trace.
Mode	Condition		
`unoccupied`	`true`	`false`	FM2
`occupied`	`mHallLL > tSafeLL`	`mHallLL` $\leq$ `tSafeLL`	FM1
`temp_empty`	`mHallLL > tSafeLL`	`mHallLL` $\leq$ `tSafeLL`	FM1
`cHallLL`	0	`tSafeLL - mHallLL`	FM1

Terms used to define `cHallLL`			
Name	Type	Init.	Description
`tSafeLL`	`yLL`	14	Safe illumination

Controlled variables			
Name	Type	Init.	Description
`cHallLL`	`yLL`	0	Intensity level of hall light

Monitored Variables			
Name	Type	Init.	Description
`mHallLL`	`yLL`	0	Level of natural light in hall
`mOccupied`	`boolean`	*false*	*True* when hall is occupied
`mT2`	`yTimer`	15	Time until empty hall

Figure 3: Example specification, in SCR format

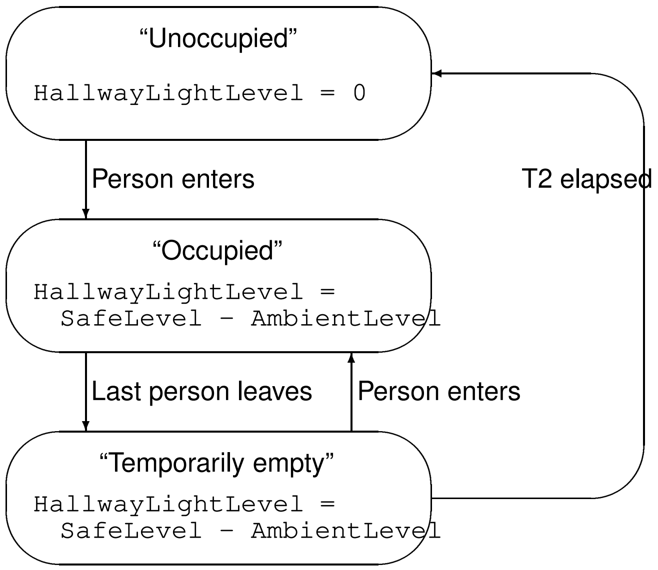

Figure 4: Example model, in the form of a finite state machine

velopment in different ways. A model may act as a specification of a system; the system's implementation is its most detailed specification, and its most exact model; the system's problem domain is frequently expressed as a model that specifies how the problem domain behaves; and a scenario for the system is a rough specification for part of the system's behavior [4], and also a model of an interaction between part of the system and part of its environment.

A set of requirements may form a specification; and a model may constitute a specification; but a model cannot be a set of requirements since a model is not easily separable into component requirements. Similarly, any model is a specification (of everything it embodies). However, there are specifications that are not models; a specification need not be constructive in the sense of implying an example that meets the specification, and a model is in itself an example, Requirements are frequently a set of constraints, as are the three requirements in Figure 2 from the LCP, and are not in any reasonable sense models of the systems whose requirements they give.

Objects are particularly appropriate for models. There is a natural and obvious correspondence between a process to be simulated, and a group of cooperating objects each of which corresponds to a component involved in the process, and all of whom together approximate the entire process. This correspondence holds despite the fact that if we consider requirements of the process as a whole (rather than for some component in the process), it is quite unlikely that any individual requirement will correspond to an individual object or type of object. This is an indication that object-orientation, although useful for specifications (in the form of models), may not be as useful for requirements as discussed in the remainder of this paper.

The following section provides an overview of several existing object-oriented specification approaches.

3 BACKGROUND AND RELATED WORK

Generally speaking, an object-oriented specification consists of a group of cooperating objects, the types of the objects and the relations between those types, with the behavior of the objects of each type expressed in terms of states and transitions between them. Several forms of object-oriented models, specifications, and requirements have been proposed for a variety of circumstances. In this section we provide a brief overview of two kinds of specification techniques: operational and property-based. Operational specifications consist of types, cooperating objects, relationships among types and objects, and use cases. Less common are property-based specifications [14] that directly state a system's required properties. These techniques are briefly summarized in Figure 5. We first examine property-based specifications.

3.1 Operational/Property-Based Specifications

Property-based specifications focus on a system's required properties. Three approaches, in particular, may be characterized as object-oriented: Bryant's object-oriented natural language specifications, Yau et al.'s specifications for distributed systems, and TROLL.

Bryant's object-oriented natural language specifications are expressed formally as a grammar, or less formally in carefully constructed natural language statements that can be translated into a formal notation [8]. The formal representation is operational and is equivalent to a set of class definitions, inheritance relationships, and state transition relationships. The natural language representation is property-based and is reminiscent of Prolog. An analyst begins with a specification in constrained natural language, and via successive refinements produces a formal specification, a design, and an implementation.

Yau et al.'s FORM specifications are intended for distributed systems [24]. Their operational specification consists of a set of classes, objects, the static relationships among the objects, and a state diagram for the behavior of each class. The property-based specification consists of a set of requirements expressed in natural language. Their goal is to compare the operational and property-based specifications for consistency as a means of verifying each of them.

TROLL specifications are intended for general use [17]. The language provides a rich variety of means for expressing behavior, so that many sorts of specifications may be expressed naturally. Object behavior may be expressed as assertions in first-order logic about their data values and in temporal logic about the evolution of their data values over time; as constraints on object states; as restrictions on events and sequences of events that can change object states; and with several varieties of behavior inheritance. The language allows analysts to specify problem domains as formal models (the semantics of their entire language are expressed in temporal logic), providing a structure for organizing the information in the model in natural ways.

Method	**Notation**	**Property Based**	**Operational**	**Salient Characteristic**
Dano, Briand, and Barbier [10]	Use cases	no	yes	Produces two representations of the system's behavior, for cross-checking
FORM (Ye *et al.* [25])	Conceptual Graphs	no	yes	Employs types and their relations to organize the requirements
ROAD/EE (Tamura et al.) [22]	Statecharts	no	yes	Produces a simulation of the final system
OO Natural Language Specifications (Bryant [8])	Two-Level Grammar	yes	yes	Requirements are first expressed in constrained natural language, then transformed into the formal representation
Yau, Bae, and Yeom [24]	State diagrams, properties	yes	yes	Produces two specifications, one operational and one property-based, and compares them for consistency
TROLL (Jungclaus *et al.* [17])	Rich variety of notations	yes	yes	Produces completely formal models while organizing the requirements in a natural way

Figure 5: Object-Oriented Specification Methods: Summary of Salient Characteristics

3.2 Operational Specifications

Operational specifications offer the most common form of object-oriented models. Three, in particular, are most applicable to the discussion in this papaer: Dano *et al.*, Ye *et al.* and Tamura's ROAD/EE.

Dano, Briand, and Barbier [10] take an approach which on the surface is not necessarily object-oriented. They consider an object-oriented specification to be a set of use cases, with links expressing temporal relations between use cases. A set of cooperating objects, their types, and their states are implicit in these use cases but do not appear directly. Two representations of the use cases are produced, one in the form of tables of transitions between abstract states, produced by domain experts, and another more detailed one in terms of Petri nets, produced by analysts. Consistency between the two representations then confirms the analysts' understanding of the user's requirements, and to an extent the consistency and completeness of the specification.

Ye *et al.* represent objects, types, relations between objects and between types, and events that specify behavior [25], all in terms of Conceptual Graphs [21]. Their specifications are intended for systems without concurrency. By using types and relations the information present in a set of requirements may be organized and expressed formally.

Tamura *et al.*'s ROAD/EE specifications are intended for concurrent systems, and especially for embedded real-time systems [22]. The behavior of each type of object is represented using statecharts [12]. Both the system and its environment are specified by groups of objects in this approach. A primary goal is to produce a simulation of the system directly from the object-oriented specification. The audience for this simulation is the users of the proposed system.

3.3 Other OO Specification Approaches

This paper is not intended to simply survey object-oriented specification approaches; we thus refer the reader to Wieringa's survey of an additional 19 object-oriented specification methods [23], such as Shlaer-Mellor, Booch and OMT. Some of the methods, notably Fusion, OMT and UML, provide mechanisms for analysts to produce explicit descriptions of system behavior, usually in some form of scenarios. Most, but not all, of the methods provide some specification of object behavior in various forms of finite state machines, sometimes augmented by use cases or conditions. The cooperative behavior of the group of objects that make up a system thus provide a specification of the system's behavior. Wieringa notes that few of the 19 methods he examines, such as JSD and Syntropy, make a clear distinction between components in the environment of a system and components of the system itself; and that this blurring was already present in the Entity Relationship Diagram [9], of which object-oriented class diagrams are extensions.

In summary, there are numerous approaches for object-oriented specifications. The most common mechanism for expressing behavior is via a group of cooperating objects. These methods are intended for a variety of uses, including: checking for consistency and a restricted notion of completeness; validation of requirements using simulations; reasoning about the specification; and organization of information in the specification, all of which seek to introduce software quality during the early stages of the software development lifecycle.

4 OBJECT-ORIENTED REQUIREMENTS AND SPECIFICATIONS

This section further examines the properties of requirements, specifications, and models. We consider in what ways object-orientation, in light of the characteristics previously discussed, is consistent with and supports these properties.

4.1 Requirements and Object-Orientation

As we discussed in Section 2, requirements should be separable; expressed in natural language; easy to understand and change, both individually and in combination; and individually traceable to earlier requirements and other sources from which they were derived.

4.1.1 Separability of requirements

To the extent that two classes exhibit low coupling, they may be considered separable. Similarly, two objects may be said to be separable to the extent that they encapsulate their states. Thus separability of requirements could be supported by expressing each requirement with a separate class or a separate object.

4.1.2 Natural language

One can hardly help noticing how prevalent diagrams, pictures, and charts are in the practice of object-oriented development. Object-oriented code is in the form of text, and natural language is frequently used to make the code or its design more clear, but understanding the code or the design behind it seems to involve reading or drawing a diagram. Diagrams are a more natural and effective notation for important aspects of object-oriented designs and programs (for example to show the various relationships between classes). However, since that requirements in natural language are advantageous, object-orientation does not appear to provide a benefit.

There is one sort of information in many object-oriented methodologies that is typically expressed in natural-language: use cases [16]. Use cases are also sometimes used as a substitute for requirements. Although use cases are associated with object-orientation, use cases themselves are not object-oriented; they are linear narratives in form and do not have states or methods or display encapsulation or inheritance. Thus object-orientation in requirements is not achieved by expressing them in the form of use cases.

4.1.3 Ease of understanding and change, individually and in combination

Two advantages offered by object-orientation are ease of understanding and ease of change. Both of these are supported by encapsulation, which reduces the amount of information that needs to be understood and can hide the effect of changes, and by inheritance, which provides a way to avoid duplicating common information shared by more than one class or type or common changes made to more than one class or type. Requirements do not encapsulate anything; they do not contain a state or an implementation that can be encapsulated, so encapsulation does not appear to offer ease of understanding or change in the context of requirements. On the other hand, requirements can share common information and some sort of inheritance relationship to express this sharing could prove beneficial.

4.1.4 Traceability

Traceability of requirements is a relation that is traditionally expressed in the form of a matrix. It can also be represented as a directed hierarchy or network, and inheritance among classes and object types can take that form also. Thus an inheritance relation can be used to express the derivation of requirements from earlier requirements and other sources.

In summary, some of the traits of object-orientation may be useful in expressing requirements. Expressing each requirement with a single object or class is consistent with separability, and an inheritance relationship can express information shared among requirements and also traceability. However, we do not believe that expressing requirements using these traits or any combination of them is sufficient to make requirements object-oriented.

4.2 Object-Oriented Specifications

As discussed in Section 2, specifications should be complete and accurate; apropos for the chosen level of abstraction and point of view; and expressed in a formal notation whose semantics are precise and unambiguous.

4.2.1 Completeness and accuracy

We see that there are object-oriented specification languages with formal semantics and a variety of notations for expressing behavior (for example, TROLL [17]). It is interesting that these notations can include logical formulae that express properties, which are arguably not object-oriented, as well as the more ordinary operational notations based on transitions between states. However, the operational notations suffice for a complete and accurate expression of program behavior, even if they are less convenient than properties in some situations.

4.2.2 Level of abstraction and point of view

Section 3 presents several object-oriented specification languages, covering a variety of levels of abstractions and addressed to several points of view. There seems no reason to believe that object-orientation restricts the level of abstraction or point of view of a specification.

4.2.3 Formal notation with precise semantics

Object-oriented specification methodologies usually focus on (in this order) decomposition, communication between objects, behavior of design components, and to a lesser extent behavior of the system as a whole and its environment [23]; most methodologies provide informal or semi-formal notations for behavior, rather than more formal notation with precise semantics.

4.2.4 Other issues

A definite practical advantage of using object-oriented notations for specifications, despite whatever difficulties they may introduce, is that the notation and the concepts behind them are already in use elsewhere in the development process, so there is no need for an additional notation or approach. A disadvantage is that since a similar or identical notation and approach are used for specifications and designs, it is difficult to distinguish the specifications from the designs and easy to contaminate them with information that does not belong there but rather belongs somewhere else where the same notation is used. It is generally not possible to identify a diagram as a software specification, domain specification, or design, without reading the accompanying text.

We note that an object-oriented specification can be particularly appropriate in certain situations. When the behavior of a group of cooperating processes is to be described, such as a simulation or a distributed algorithm, an object-oriented specification can map directly onto the entities whose behavior is being described.

Object-orientation can provide a suitable basis for a specification language, meeting the needs of completeness, point of view, and formality. In addition, object-oriented specifications seem particularly well suited to the specification of simulations and other similar systems.

5 DISTINCTIONS BLURRED BY OBJECT-ORIENTED DEVELOPMENT

The rapid adoption and retrofitting of object-oriented analysis methods to support requirements analysis and specification activities has resulted in a blurring of the distinction between specifications (or even requirements) and designs [18]. Most object-oriented methods bring a powerful organizational technique to bear on a software problem: the organization of the solution is given the same shape as the description of the problem. Many of the conceptual entities that appear in the design (individual types, classes, and objects) typically correspond to conceptual entities in the problem domain and problem statement; and the object-oriented relationships between those design entities (subtyping, inheritance, aggregation, and behavior dependency) also correspond to relationships between entities in the problem domain and statement. The resulting design method is (relatively) easy and direct. However, since this organizational technique produces the analysis and design largely at the same time and by means of the same process, the distinction between the results of requirements analysis and the results of design is blurred.

Researchers have proposed the use of what amounts to object-oriented designs with behavioral information as specifications [17, 22, 24, 25]. This approach has several significant advantages and disadvantages. The advantages are that the techniques for producing object-oriented designs are well-established [6, 20, 16, 19]; and it is possible to reason about the design, notably by simulating its behavior [22, 24], facilitating the verification of the design for correctness and appropriateness. There is also a notable disadvantage, however. If the design is the specification, then there is no separate specification against which to verify the design or the implementation, and the specification is virtually certain to have been strongly influenced by design considerations. In this situation, errors in the design are much less likely to be found (or if found, identified as errors rather than decisions), and the system as a whole is less likely to meet the original needs.

Even if a separate specification and design are made, it becomes difficult to decide how to distinguish an object-oriented specification from an object-oriented design. As previously discussed, the specification and design are often (and typically) expressed in the same notation, and produced by a similar or even (on some level) identical processes. As a result they are each likely to be influenced by considerations that should only apply to the other and are likely to be too similar to be useful.

6 CONCLUSIONS AND FUTURE WORK

Requirements must adequately and precisely state *what* a user desires or needs and they should be expressed in a manner oriented towards this objective. Natural language representations, formal or semiformal representations, or a combination of both can be best, depending on the specific problem, domain and/or environment. Throughout this paper we discuss the distinction between specification and design. Clearly, this is not a problem specific to object-orientation for it has always been an issue with Structured Analysis and Design as well. However, we do observe that it does become somewhat more of a challenge in object-orientation. Thus, practitioners must continue to weigh the advantages and disadvantages of constructing specifications so that they look like designs.

While the concept of object-oriented requirements and specifications is certainly appealing, we have shown that this appeal may be illusory to a certain extent in that it presents challenges for practitioners and researchers alike. Although object-orientation is useful for producing and constructing specifications in the form of models, its benefits for requirements is not as clear. As Jackson states, object-oriented models really describe programming objects rather than the real world [15], and requirements mostly concern the real world, or environment, in which the corresponding system must ultimately operate. In the case of requirements, object-orientation does not clearly support some of the most important properties we desire. In the case of specifications, object-orientation produces certain advantages, most notably the possibility of modelling a specification using well-developed object techniques, with the added benefit of, perhaps, fitting in with object-oriented development methods and notations. However, here object-orientation also introduces challenges: the use of a single notation for more than one distinct purpose makes it difficult to keep separate the kinds of information appropriate for each purpose; and it can be difficult to express a specific behavior using cooperating objects, and to understand the behavior resulting from any group of cooperating objects.

It is clear that "object-oriented" specifications will continue to be used, certainly for some time to come. Object-orientation is advantageous when what is needed is a precise specification of what the desired system must know about its environment for fulfillment of its required tasks; this is clearly more difficult to accomplish with natural language descriptions and yet it may be easily modeled using object-oriented notations. Practitioners are faced with decisions when using object-oriented methodolgies for requirements analysis and specification activities during software development. In this paper we surface some of the issues associated with these approaches in an effort to help practitioners avoid or, at a minimum, reduce some of their ill effects. Object-oriented specifications typically ignore the user/system boundary and neglect the user's viewpoint of the system; and current object-oriented methodologies lack adequate decomposition facilities needed for organizing large specifications.

We are currently investigating other possible forms of object-oriented specification as exhibited in the domain of telephony services. In particular, we are considering the efficacy of employing object-oriented constructs to organize specifications and models within the context of evolutionary and transformational prototyping. We will build upon our previous work in goal-based requirements engineering which demonstrated that goal hierarchies are another useful mechanism for organizing and reasoning about goal relationships (and supporting requirements and associated specifications) in a logical manner [1, 3, 5].

References

[1] T. A. Alspaugh, A. I. Antón, T. Barnes, and B. Mott. An integrated scenario management strategy. In *RE '99: IEEE Fourth International Symposium on Requirements Engineering*, pages 142–149, June 1999.

[2] T. A. Alspaugh, S. R. Faulk, K. H. Britton, R. A. Parker, D. L. Parnas, and J. E. Shore. Software requirements for the A-7E aircraft. NRL Memorandum Report 3876, Naval Research Laboratory, Washington, DC, Aug. 1992.

[3] A. I. Antón. *Goal Identification and Refinement in the Specification of Soft-ware-Based Information Systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, June 1997.

[4] A. I. Antón and C. Potts. A representational framework for scenarios of systems use. *Requirements Engineering Journal*, 3(3–4):219–241, 1998.

[5] A. I. Antón and C. Potts. The use of goals to surface requirements for evolving systems. In *Proceedings of the 1998 International Conference on Software Engineering (ICSE'98)*, pages 157–166, 1998.

[6] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1993.

[7] M. L. Brodie, J. Mylopoulos, and J. W. Schmidt. *On Conceptual Modelling*. Springer, New York, NY, 1984.

[8] B. Bryant. Object-oriented natural language requirements specifications. In J. Edwards, editor, *23rd Australasian Computer Science Conference, 2000 (ACSC 2000).*, pages 24–30, 2000.

[9] P. Chen. The entity relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, Mar. 1976.

[10] B. Dano, H. Briand, and F. Barbier. Progressing towards object-oriented requirements specifications by using the USE CASE concept. In *Proceedings of the 1996 IEEE Symposium and Workshop on Engineering of Computer-Based Systems*, pages 450–456, 1996.

[11] R. Gotzhein. The light control case study : Problem description. Journal of Universal Computer Science, Special Issue on Requirements Engineering, August 2000. http://rn.informatik.uni-kl.de/ recs/problem/pd.ps.

[12] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[13] C. Heitmeyer and R. Bharadwaj. Applying the SCR requirements method to the light control case study. *Journal of Universal Computer Science*, 6(7):650–678, Aug. 2000.

[14] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, Nov. 1998.

[15] M. Jackson. *Software Requirements and Specification: a lexicon of practice, principles and prejudices*. Addison-Wesley, Wokingham, England, 1995.

[16] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. ACM Press, 1992.

[17] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL — A language for object-oriented specification of information systems. *ACM Transactions on Information Systems*, 14(2):175–211, Apr. 1996.

[18] J. Mylopoulos, L. Chung, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, Jan. 1999.

[19] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall Inc., 1991.

[20] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley Longman, Reading, Mass., 1999.

[21] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machines*. Addison-Wesley, Reading, Mass., 1984.

[22] N. Tamura and T. Nakajima. ROAD/EE : A prototyping environment for object-oriented specifications. In *Proceedings, Technology of Object-Oriented Languages and Systems (TOOLS 23)*, pages 176–189, Santa Barbara, California, July 1997. IEEE Computer Society Press.

[23] R. Wieringa. A survey of structured and object-oriented software specification methods and techniques. *ACM Computing Surveys*, 30(4):459–527, Dec. 1998.

[24] S. Yau, D. Bae, and K. Yeom. An approach to object-oriented requirements verification in software development for distributed computing systems. In *Proceedings of the 18th International Computer Software and Applications Conference (COMPSAC '94)*, pages 96–102, 1994.

[25] P. Ye, J. Hughes, I. McChesney, and D. Glass. FORM : A formal object-oriented requirements model based upon conceptual graphs. In *Proceedings of the IEEE International Conference on Developing and Managing Intelligent Systems Projects*, pages 221–228, 1993.