A Specification Language for Object-Oriented Analysis and Design

Ted L. Briggs[†]
Intergraph Corporation
Huntsville, AL 35894
email: tlbriggs@ingr.com

John Werth
Department of Computer Science
University of Texas at Austin
Austin, Texas 78712
email: jwerth@cs.utexas.edu

Abstract

This paper introduces and illustrates the use of ObjLog, an algebraic specification language for Object-Oriented Analysis and Design. ObjLog is fully abstract, i.e., it specifies the message-passing and instantiation of objects without explicit use of state. Object behavior is abstractly defined by traces composed of message send and response events, including instantiation requests. ObjLog extends equational algebraic specification techniques to specify these traces and to reason about state dependent transitions in objects. Unlike most existing specification languages, ObjLog is sufficiently expressive to specify the full range of value-based message-passing and instantiation behavior exhibited by sequential object-oriented programming languages. In this paper, ObjLog is used to specify a simple example which is difficult to fully specify using other specification languages. The resulting ObjLog specification is then refined in three different ways: subtyping by extension, specialization, and aggregation.

Keywords: Specification Language, Object-Oriented Specification, Object-Oriented Analysis, Object-Oriented Design, Message-Passing, Object Types

1 Introduction

Although a number of methodologies [3, 6, 7, 22, 23, 24] for Object-Oriented Analysis (OOA) and Object-Oriented Design (OOD) have been proposed and are currently in use, they all are based on informal models of objects and use informal specification languages. Consequently, the resulting specifications may be ambiguous or inconsistent [13]. In contrast, the use of a formal specification language for OOA/OOD ensures precise specifications, a formal model of objects, and a well-defined notion of abstraction. A formal specification language also provides the ability to state and prove correctness properties, as well as to reason about the system.

This paper introduces and illustrates the use of ObjLog, an algebraic specification

[†] Work done while at the University of Texas at Austin.

language for OOA/OOD. ObjLog is fully abstract, i.e., it specifies the message-passing and instantiation of objects without explicit use of state. Object behavior is abstractly defined by traces composed of message send and response events, including instantiation requests. ObjLog extends algebraic (equational) specification techniques to specify these infinite traces and to reason about state dependent transitions in objects. An ObjLog specification is potentially executable, i.e., terms can be reduced automatically through term-rewriting.

1.1 Specification Languages for OOA/OOD

A specification language for use with Object-Oriented Analysis or Design should reflect the unique and interrelated nature of OOA/OOD. Both OOA and OOD differ from their traditional counterparts in that the resulting specifications are object-based. Because OOA and OOD share a common set of objects, a specification language for OOA/OOD must express *refinment* of a requirements specification into a design specification. In contrast, Structured Analysis, results in a purely functional model which requires *translation* into state-based modules for Structured Design.

During Object-Oriented Analysis, objects are used to denote entities in the problem domain, while in Object-Oriented Design, objects are used to denote software modules. In both cases, interactions are modeled through message-passing. Hence, an object-oriented *requirements specification* defines how entities interact to specify system behavior, while an object-oriented *design specification* defines state-dependent module interfaces and their interactions. In addition, objects are used to model dynamic entities and it is necessary to also specify the instantiation of objects. However, current specification languages, both informal and formal, cannot express the full range of message-passing and instantiation behavior exhibited by sequential object-oriented programming languages.

A specification language for OOA/OOD must also support the use of abstraction during the analysis and design process. In particular, during Object-Oriented Design, module interfaces are represented by the message-passing interfaces of objects and specification of the (imperative) message-passing interface abstracts the module internals and allows design and implementation decisions to be hidden [20, 21]. This ensures that implementation decisions and internal design decisions of a module can be changed later without affecting the system behavior.

1.2 Formal Specification Languages for OOA/OOD

Specification languages which are formal are especially important for OOA/OOD. First, they permit one to prove that a particular correctness property is extensible, i.e., it continues to hold when the system is modified, either by the addition of new objects or by modifications to existing objects. Second, they provides the capacity to formally reason about complex system behavior. This is important because many object-oriented development methodologies are essentially bottom-up: objects are specified and then composed to form systems. Hence, system properties must be deduced from object specifications. Although object methods themselves are seldom complex, object interactions and dependencies often are.

However, at present, most formal specification languages are not well suited for

OOA/OOD (See Section 7). In contrast, a number of useful OOA/OOD methodologies have been developed which are informal. A representative list of popular OOA/OOD methodologies would include Booch [3], Coad and Yourdon [6, 7], Rumbaugh [22, 23], and Shlaer and Mellor [24]. These methodologies typically incorporate three separate modeling techniques: entity-relationship diagrams, state machines or Statecharts [11, 12], and data flow diagrams. Hence, they do not present a unified, coherent model of objects, but rather three interrelated models. Consequently, it is possible that a specification may define an inconsistent set of models [13]. Moreover, due to the collection of models used, it often is not possible to reason effectively about the behavior of the system under development. In an effort to deal with these problems, this approach was refined by Hayes and Coleman [13], who defined a coherent model of objects and a specification language based on Objectcharts [2, 8]. However, this work, like all these specification languages incorporates state machine models, which limits their expressiveness.

1.3 Organization of the Paper

In this paper, a simple example is used to introduce and illustrate the use of ObjLog. The corresponding ObjLog requirements specification is presented and the underlying formal semantics is briefly discussed. Behavioral properties are expressed in ObjLog through the use of equations. ObjLog supports the three common forms of refinement often used in OOA/OOD: subtyping by extension, specialization, and aggregation. The expressiveness of ObjLog is demonstrated by the range of traces which may be expressed. The formal development of the underlying trace model is given in a companion paper [5].

The paper is organized as follows. Section 2 informally defines the underlying notions of objects and types in ObjLog. In Section 3, the example problem is introduced, the analysis model is defined, and the syntax for services is fixed. Section 4 introduces our trace model of object behavior and gives selected traces for the example. This also provides the basis to later discuss the expressiveness of other specification languages. In Section 5, the requirements specification for the example is given using ObjLog. Section 6 illustrates the use of three types of specification refinement in ObjLog. Section 7 compares and surveys related work. Finally, Section 8 concludes the paper.

2 Objects, Messages, and Types in ObjLog

This section informally defines the underlying notions of object, message, and type in ObjLog. It is useful to clarify these notions due to the diversity of related concepts often used in object-oriented development.

2.1 Objects

Objects act both as clients and servers which interact through messages. Each *object* has a label and a set of history-sensitive services. The parameters and return values of services are stateless abstract data type values called *primitive values*. An object is permanently labeled by a unique, immutable *object identifier*. Object identifiers are considered primitive values, but may not be otherwise manipulated except to compare for equality.

A *signature* is a set of service names with their respective parameter and return types. A *subsignature* is any subset of a signature. The *object signature* of an object is the signature consisting of all the services provided by the object. An object *satisfies* every signature which is a subsignature of its object signature.

In contrast to the abstract ObjLog approach, objects in object-oriented programming languages have a label and a state-based implementation. The *implementation* of an object is a set of attributes and related set of methods. An *attribute* is an instance variable declaration, i.e., a name and data type. A *method* is a specific implementation of a service, i.e., it is an n-ary operation on primitive values which may side-effect the instance variables. The value returned by a method is a function both of the instance variables and method parameters.

2.2 Messages

Objects interact only through messages. A *message* is a term $<o, m(x_1, \ldots, x_n)>$ with object identifier o, service name m, and parameters $x_1, \ldots, x_n$. A *message send event* is a request by an anonymous client for a specific service from a specific server. A *response event* to such a request is the value returned by the service. A *trace* is a countable sequence of message send and response events. The *observable behavior* of an object consists of all possible traces of events involving the object as either a client or server. Note that objects often require the services of other objects, i.e., servers may act as clients while evaluating messages. Further, objects may provide services which instantiate other objects.

2.3 Types

An *abstract object type* or simply *type* is the set of objects which satisfy a given signature and exhibit equivalent[1] behavior on that signature. In this paper, two traces are considered equivalent if there exists a permutation of object identifiers which will map one trace into the other. Informally, this means that objects of the same type can be substituted for each other without affecting the behavior of the system. A type T_1 is a *subtype* of T_2, if the objects of T_1 are also of type T_2. These are polymorphic types, i.e., objects have many abstract object types.

Programming languages, in contrast, typically support two different notions of type: classes and signature types. A *signature type* is a weaker notion of type: the set of objects which satisfy a given signature. This is purely syntactic classification which does not enforce any constraints on object behavior. A *class* is a stronger notion of type: a set of objects with a common implementation. Consequently, the objects of a class are always of the same abstract object type, regardless of the notion of equivalence. Conversely, abstract object types are always partitioned into classes, i.e., the same observable behavior may arise from several implementations. A class C_1 is a *subclass* of C_2 if the objects of class C_1 are also objects of class C_2, i.e., they incorporate the same implementation as the parent class.

---

[1] Several notions of equivalence can be considered to address different concerns for object types. At the very least, trace equivalence must be introduced to model instantiation of objects with arbitrary object identifiers. For a discussion of various other possible ideas of equivalence see the companion paper on the formal development of the underlying trace model [5].

3 Example Problem and Initial Analysis

This section introduces the example problem which will be specified using ObjLog. The example problem is defined, an object-oriented model produced, and the object signature defined.

3.1 Requirements Statement

The problem is to develop a simple graphical drawing system which allows a user to draw, connect, and move lines[2]. For simplicity, we will assume that the user interacts directly with the objects, rather than indirectly through an event manager.

Definition 1. (Problem Statement)

Consider an infinite two dimensional plane with integer valued coordinates. A *line* is defined by two distinct endpoints labeled "left" and "right". (The "left" label denotes the endpoint with the least x coordinate and, if the x coordinates of the two endpoints are equal, then it denotes the endpoint with the greatest y coordinate.) The set of lines on the plane is called a *diagram*. Initially the diagram is an empty set. However, a user may create a *new* line for a given endpoint coordinates or may *move* an existing line a given displacement.

A user may also *attach* two lines which share a common endpoint. A line may be attached to at most one other line at each endpoint. Newly created lines are not attached at either endpoint. The attached lines and their endpoints form a graph: two lines are *connected* if they are in the same path. When a user moves a line, all connected lines are also moved. For simplicity, a line cannot be connected to itself.

The services which a line object with identifier *line* provides a user are:

User Services	Description
getX(s)	Return the x coordinate of endpoint s.
getY(s)	Return the y coordinate of endpoint s.
hasAttached(s)	Return whether endpoint s has a line attached.
isConnected(s, l)	Return whether *line* is connected by an endpoint s to a line l.
isSamePt(s_1, l, s_2)	Return whether an endpoint s_1 of *line* is the same point as endpoint s_2 of line l.
move(x, y)	Move *line* and all connected lines by the given displacement.
attach(s_1, l, s_2)	Attach two lines at a common point. Attach the endpoint s_1 of *line* and the endpoint s_2 of line l. Return a boolean indicating whether l was attached to *line*.
connect(s_1, l, s_2)	Connect the endpoint s_1 of *line* to the endpoint s_2 of line l by creating a new line attached to both endpoints. Return a boolean indicating whether the endpoints were connected.

3.2 Object-Oriented Model

An object-oriented analysis produces an object-oriented model which specifies system behavior. The first step is to identify the objects, their interactions and collaborations. This can be conveniently represented in graphical fashion by an interaction diagram. Specifically, an *interaction diagram* is a labeled directed graph in which nodes denote types and edges denote sets of services called *contracts*[3]. An out-edge denotes a set of services which are required and an in-edge denotes a set of services which are provided.

[2] This is an adaptation of the problem used by Hayes and Coleman [13]. It is simplified by the elimination of boxes, but illustrates a wider range of behavior by the inclusion of additional services.

This is similar to the configuration diagrams of Coleman and Hayes [13], the Object Communication Model of Shlaer and Mellor [24], and the collaboration graphs of Wirfs-Bock et.al. [26].

For this simple example, the resulting object-oriented model consists of two object types: *User* and *Line*. It also includes two contracts: *User-Line* and *Line-Line*. The interaction diagram for the drawing system is given below in Figure 1. The *User-Line* contract is defined above in the requirements statement. A object of type *Line* provides this set of services to the user. The user does not provide any services and is shown only to illustrate the services it requires.

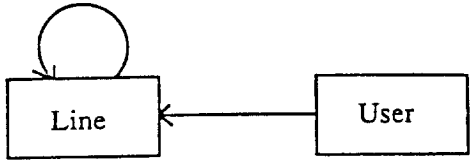

Figure 1.: Interaction Diagram.

The *Line-Line* contract must now be defined to permit line objects to collaborate and provide the required services to the user. Line objects themselves will require a set of services similar to those provided to the user: *getX*, *getY*, *hasAttached*, *isSamePt*, and *isConnected*. However, the *move* and *attach* services cannot be used recursively: hence two additional services must be introduced: *MoveAttached* and *doAttach*. An object with identifier *line* also provides these two additional services to other objects:

Additional Services	Description
MoveAttached(s, x, y)	Move all lines connected by endpoint s to *line*.
doAttach(s_1, l, s_2)	Attach the endpoint s_2 of line l to the endpoint s_1.

The *move* service moves attached lines at both endpoints, while the *MoveAttached* service only moves an attached line at a specific endpoint. The *attach* service involves two objects, while the *doAttach* only affects the state of a single object.

3.3 Signature

The next step is to define the signature of *Line* objects. The set of primitive abstract data types, *Prim*, used in this model contains the following primitive data types: object identifiers, integers, sides, booleans, and void. These are denoted respectively by the following data types names or *sorts*: *ObjId*, *pos*, *int*, *side*, *bool*, *void*. Object identifiers will be denoted $o_1, o_2, \ldots$. A special data type *void* is used to indicate that a returned value will be ignored. The enumerated type *side* = { *left*, *right* } is used to label the endpoints of lines. A function *opp* maps one side into the other.

The signature of *Line* objects is defined in Figure 3 (where services have already been divided by the ObjLog approach into selectors and updates). The parameter and return types are defined with functional notation, similar to the way signatures for abstract data

[3] ObjLog does not introduce explicit notation for contracts. Rather, a contract (sometimes also called a view) is represented simply as a subsignature.

types are often defined. Hence *getX* and *getY* both take a single argument of type *side* and returns an *integer*. The notation *ObjId*(*LineSig*) denotes the set of object identifiers of all objects with signature type *LineSig*. A *constructor* is a special service provided by a special system object, which creates new objects of the given object type. This signature also specifies a signature type *LineSig*.

4 Object Behavior: A Trace Model

This section defines object behavior in general, as well as the behavior of line objects in particular, using a trace model. The trace model adopted here differs from many other models in that it permits nested service. The services provided by objects are not local, atomic actions, but rather correspond roughly to remote procedure calls. Message-passing involves a pair of events: a send and a response. The formal development of traces and the definition of abstract object types is given in a companion paper [5].

4.1 Messages and Traces

The message-passing behavior of objects will be defined by a sequence of message sends and responses. A message send is written as a term "*send*(*o*, *request*)" where *o* is an object identifier value and *request* is a request term which consists of a service name followed by parameter values. A response is indicated by the term "*ret*(*value*)". A sequence of events is written using the ";" operator. For example, given a *Line* object with object identifier o_1, a message send event which requests the *x* coordinate of the left endpoint followed by a response of "4" would be written:

$send(o_1, getX(left))$; $ret(4)$

A *trace* is a countable sequence of message send and response events starting with a message send. In ObjLog, objects are assumed to be sequential, i.e., a message sequence is processed sequentially one message at a time. For example, the message sequence to first move the line o_1 and then return the left *x* coordinate is written:

$send(o_1, move(5, -5))$; $send(o_1, getX(left))$

The resulting trace is written:

$send(o_1, move(5, -5))$; $ret(void)$; $send(o_1, getX(left))$; $ret(9)$.

An object *satisfies* an signature if it responds to messages containing its method names with data values of the correct type. The behavior of an object can be characterized by an infinite set of all possible traces. The ObjLog specification language, however, is based on defining the algebraic properties inherent in such traces.

4.2 Instantiation

Instantiation of objects is treated in ObjLog as a special type of message. The system is regarded as a special object with object identifier *sys* which can act only as a server and does not require the services of any other objects. Objects are instantiated by a *constructor message* to the system object. The constructor is prefixed by a signature or type name to allow overloaded constructor names. For example, the message to instantiate an object of signature type *LineSig* is written *send*(*sys*, *LineSig.new*(0, 0, 2, 3)). The *sys* object guarantees the uniqueness of object identifiers by returning object identifier of previously uninstaniated objects.

4.3 Object Behavior in the Example Problem

The requirements statement provided only an informal and incomplete specification of line object behavior. The behavior of line objects in a specific diagram can now be specified precisely by a selected set of traces. Traces will also be used as a basis for discussing the expressiveness of other specification languages. To this end, the services an object provides will be classified as local, non-local, nested, trigger, and recursive.

A diagram which might be produced by the finished drawing system is shown below in Figure 2. Each line is labeled by its object identifier. Attached lines are indicated by a circle at the attached endpoint.

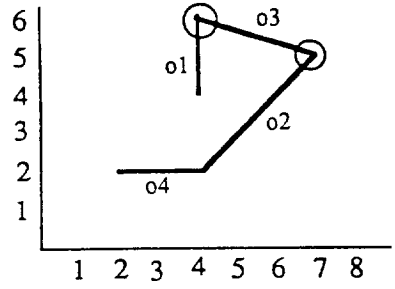

Figure 2.: Example of a diagram.

Assuming that the system allocates object identifiers sequentially, the diagram above could be created by the following sequence of messages:

send(*sys*, *Line.new*(4, 4, 4, 6)); *send*(*sys*, *Line.new*(4, 2, 7, 5));
send(o_1, *connect*(*left*, o_2, *right*)); *send*(*sys*, *Line.new*(4, 2, 2, 2))

A *local service* is the simplest form of service provided by a object: it generates no additional messages and hence does not require the services of other objects. The trace of a local service is a message send followed by a response. A *Line* object has four local services: *getX*, *getY*, *hasAttached*, *doAttach*. An example of a local trace is:

send(o_1, *hasAttached*(*left*)); *ret*(*true*).

An object with only local services can be modeled formally as a state machine where the message send events are interpreted as input and response events are interpreted as output.

In general, services may be *non-local*, i.e., require the services of other objects. In this case, a server, which is has received a message, sends a message to another object. The response to this secondary message must be returned before the server itself can return a response. Hence this results in *nested traces*. For example, the *isSamePt* service can be defined using the *getX* and *getY* services from the line in question.

send(o_1, *isSamePt*(*right*, o_2, *left*));
    send(o_2, *getX*(*left*)); *ret*(4);
    send(o_2, *getY*(*left*)); *ret*(2);
ret(*false*)

The resulting trace is indented to show the nesting level. In this case, the object o_1 generates a sequence of messages to object o_2 before the response is returned.

A *trigger service* is a special case of a non-local service whose trace always contains only *void* responses. The effect is to simply trigger services from other objects without requiring a return value. Trigger services are the most general non-local services which can be expressed by Statecharts or Objectcharts. A *Line* object has two trigger services: *move*, *MoveAttached*. The *move* service triggers *MoveAttached* services in every connected line as shown below:

send(o_1, *move*(2, 2))
    send(o_3, *MoveAttached*(*left*, 2, 2));
        send(o_2, *MoveAttached*(*right*, 2, 2)); *ret*(*void*);
    ret(*void*);
ret(*void*)

The nesting level reflects the fact that object o_1 has two lines connected by its *left* endpoint. Note that each *Line* object must know the object identifier of its attached lines as well as the label of the endpoint of those attached lines.

A *recursive service* is another a special case of a non-local service. The trace length and nesting level are data dependent. This type of service cannot be expressed using Statecharts or Objectcharts. A *Line* object has three recursive services: *move*, *MoveAttached*, and *isConnected*. The *isConnected* service is a recursive service, which recurses over the connected lines until it finds a matching object identifier. For example, the trace for zero attached lines is

send(o_1, *is_connected*(*right*, o_4)); *ret*(*false*)

and the trace for three connected lines is:

send(o_1, *is_connected*(*left*, o_4));
    send(o_3, *is_connected*(*right*, o_4));
        send(o_2, *is_connected*(*left*, o_4)); *ret*(*false*)
    ret(*false*);
ret(*false*)

Space does not permit further characterization of the trace behavior of *Line* objects in general and the *attach*, *doAttach*, and *connect* services in particular.

5 Specification with ObjLog

This section illustrates how the example problem is specified in ObjLog. The ObjLog signature is defined and the distinction between selectors and updates is discussed. Then the ObjLog requirements specification is defined and the informal semantics are discussed. Finally, we sketch how one uses equations in ObjLog to reason about objects.

An ObjLog specification abstracts the internal attributes and state values by utilizing only the elements of the signature to specify object behavior. This requires only a specification of the selector, update, and constructor services, which are often implicit in other specification techniques. Given these services, the syntax used to define object behavior is minimal: message-passing, assignment, and primitive data type operations. This is similar in style, except for the additional data type operations, to a pure object-oriented programming language.

5.1 ObjLog Signature

An ObjLog *signature* is partitioned into selectors, updates, and constructors as shown in Figure 3 below. This partition of services into selectors and updates is, in fact, just a

generalization of the usual division into attributes and methods, respectively, in state-based languages. The set of primitive data types *Prim* is assumed to be formally specified in a many-sorted algebraic specification language, e.g., OBJ3 [10]. Every primitive data type is also assumed to have two infix operators: relational equality, written " == ", and a conditional, written " *if* _ *then* _ *else* _".

Type Signature LineSig =
  Primitive Types: Prim
  Selectors:
    getX, getY: *side* → *int*
    isSamePt: *side*, *ObjId*(*LineSig*), *side* → *bool*
    hasAttached: *side* → *bool*
    isConnected: *side*, *ObjId*(*LineSig*) → *bool*
  Updates
    move: *int*, *int* → *void*
    MoveAttached: *side*, *int*, *int* → *void*
    attach: *side*, *ObjId*(*LineSig*), *side* → *bool*
    doAttach: *side*, *ObjId*(*LineSig*), *side* → *void*
    connect: *side*, *ObjId*(*LineSig*), *side* → *bool*
  Constructors
    new: *int*, *int*, *int*, *int* → *ObjId*(*LineSig*)
  End

Figure 3.: ObjLog Signature for Line.

A *selector* service is an abstraction of the internal object states. It denotes a fixed, *n*-ary function on the primitive data types, for each internal state. An *update* service is an abstraction of the internal state transitions. An update message affects the response to subsequent messages by changing the internal state characterization. For example, a sequence of *getX* selector services each returns the same response, but a *move* update will change this response.

send(o_1, *getX*(*left*)); *ret*(4); *send*(o_1, *getX*(*left*)); *ret*(4)
send(o_1, *getX*(*left*)); *ret*(4); *send*(o_1, *move*(4, 5)); *ret*(*void*); *send*(o_1, *getX*(*left*)); *ret*(9)

5.2 ObjLog Requirements Specification

An ObjLog *specification* consists of an ObjLog signature and a definition of the effect of updates and constructors on selector services. The difference between ObjLog requirements specifications and design specification is one of refinement and not syntax. The ObjLog requirements specification for the example problem is given below in Figure 4. It incorporates the signature *LineSig* and adds two hidden selectors *AttachedLine* and *AttachedLinePt* and a hidden update *doMove*. The two hidden selectors are required to define the object identifiers of the attached lines and the endpoint at which they are attached. The hidden update simplifies the specification of *move* and *MoveAttached*. In addition, some of the selectors in the *LineSig* signature are classified as dependent selectors.

Object Type **Specification** Line =
Primitive Types: Prim
Selectors:
getX, *getY*: *side* → *int*
hasAttached: *side* → *bool*
Hidden
AttachedLine: *side* → *ObjId*(*Line*)
AttachedLinePt: *side* → *side*
Dependent:
isConnected: *side*, *ObjId*(*Line*) → *bool*
isSamePt: *side*, *ObjId*(*Line*), *side* → *bool*
Definitions
isConnected(*s*, *o*) ↦
if not hasAttached(*s*) *then false*
else if AttachedLine(*s*) == *o then true*
else send(*AttachedLine*(*s*), *isConnected*(*opp*(*AttachedLinePt*(*s*), *o*))) *fi fi*
isSamePt(s_1, *o*, s_2) ↦
$getX(s_1) == send(o, getX(s_2))$ *and* $getY(s_1) == send(o, getY(s_2))$
Updates
move: *int*, *int* → *void*
MoveAttached: *side*, *int*, *int* → *void*
attach: *side*, *ObjId*(*Line*), *side* → *bool*
doAttach: *side*, *ObjId*(*Line*), *side* → *void*
connect: *side*, *ObjId*(*Line*), *side* → *bool*
Hidden
doMove: *side*, *int*, *int* → *void*
Definitions
move(*x*, *y*):
MoveAttached(*left*, *x*, *y*);
if hasAttached(*right*)
then send(*AttachedLine*(*right*), *MoveAttached*(*opp*(*AttachedLinePt*), *x*, *y*)) *fi*
MoveAttached(*s*, *x*, *y*):
doMove(*x*, *y*);
if hasAttached(*s*)
then send(*AttachedLine*(*s*), *MoveAttached*(*opp*(*AttachedLinePt*), *x*, *y*)) *fi*
doMove(*x*, *y*):
$getX(left) \mapsto getX(left) + x$, $getY(left) \mapsto getY(left) + y$,
$getX(right) \mapsto getX(right) + x$, $getY(right) \mapsto getY(right) + y$
attach(s_1, *o*, s_2):
if not hasAttached(s_1) *and isSamePt*(s_1, *o*, s_2)
and not send(*o*, *hasAttached*(s_2))*andnotisConnected*(*opp*(s_1), *o*)
then doAttach(s_1, *o*, s_2); *send*(*o*, *doAttach*(s_2, *self*, s_1)); *return*(*true*)
else return(*false*) *fi*
doAttach(s_1, *o*, s_2):
hasAttached(s_1) ↦ *true*, *AttachedLine* ↦ *o*, *AttachedLinePt* ↦ s_2
connect(s_1, *o*, s_2):
if not hasAttached(s_1) *and not send*(*o*, *hasAttached*(s_2)) *and not isConnected*(*opp*(s_1), *o*)
then let line = *send*(*sys*, *Line.new*(*getX*(s_1), *getY*(s_1), *send*(*o*, *getX*(s_2)),
send(*o*, *getY*(s_2))))
side = *if send*(*line*, *isSamePt*(s_1, *line*, *left*)) *then left else right fi*
in *doAttach*(s_1, *line*, *side*); *send*(*o*, *doAttach*(s_2, *line*, *opp*(*side*)));
send(*line*, *doAttach*(*side*, *self*, s_1)); *send*(*line*, *doAttach*(*opp*(*side*), *o*, s_2));
return(*true*)
endlet
else return(*false*) *fi*

Constructors
new: *int, int, int, int* → *ObjId*(*Line*)
Definitions
new(x_1, y_1, x_2, y_2):
if $x_1 < x_2$ *or* ($x_1 == x_2$ *and* $y_1 \geq y_2$)
then getX(*left*) ↦ x_1, *getX*(*right*) ↦ x_2, *getY*(*left*) ↦ y_1, *getY*(*right*) ↦ y_2
else getX(*right*) ↦ x_1, *getX*(*left*) ↦ x_2, *getY*(*right*) ↦ y_1, *getY*(*left*) ↦ y_2
EndSpec

Figure 4.: ObjLog Line Type Specification .

5.3 Semantics

ObjLog extends algebraic specification techniques to specify traces. Space permits only a brief sketch of the formal semantics of ObjLog.

5.3.1 Selector Services

Several kinds of selectors are used in ObjLog specifications: primary, hidden, derived, and dependent. In the requirements specification above, only primary, hidden, and dependent selectors are used. Later examples will use and discuss derived selectors. A selector is an *n*-ary function in the primitive data type algebra. A *selector algebra* is an extension of the primitive algebra. A *universe* is the set of all possible such selector algebras.

A *hidden selector* is a function which may be used only within the body of the specification to simplify expression of interface behavior, to indicate design choices in a design specification, and to permit expression of behavior which could not otherwise be expressed. Similarly, hidden updates are used to simplify the specification, but cannot used outside the body of the specification.

A *dependent selector* is expressed as a function of other selectors and messages to other objects. It denotes a function which is dependent on other objects and is defined by a transition function. In the specification above, *isSamePt* is mapped into the boolean expression which contains two messages. A dependent selector is not directly affected by updates. Hence, the effect of updates and constructors on dependent selectors need not be defined. Dependent selectors are typically required to specify services which involve instantiation or services required from other objects.

5.3.2 Update Services and Constructors

Updates and constructors denote *transition functions* on the selectors. These are defined component-wise on each selector using the notation $u: s \mapsto e_1$ to denote that u is an update which will *map* the selector s into the expression e_1. By convention, the map for any selector s which is not explicitly given is assumed to be the identity map, $u: s \mapsto s$. A transition function maps all selectors simultaneously. If an update also returns a non-void value, the notation "*return*(*value*)" is used. A conditional operation is used for transition functions with the assumption that the " *else* " branch is either defined or it is the identity map. Constructors denote transition functions which define the initial selector values.

A transition function extends to a homomorphism on selector algebras. Hence the set of updates is a monoid under the ";" operator. An ObjLog specification which does not contain messages denotes a *transition monoid* defined over a universe. A *configuration* is a transition monoid and a specific selector algebra.

5.3.3 Messages

The environment is a special object which contains object configurations and allows objects to interact. More precisely, it provides a selector service which maps object identifiers into configurations. A *send* operation is an update service of the environment, which reduces the request term in the appropriate configuration. The order of reduction for expressions is fixed as left to right.

5.3.4 Satisfaction

An ObjLog specification defines an abstract object type, i.e., it defines a class of traces. An object *satisfies* a specification if it exhibits all of the traces defined by the specification.

5.4 Equational Reasoning

ObjLog uses equations to specify object behavior. Thus far, we have used traces to illustrate object behavior. An equation provides a concise representation which specifies how traces are generated. Object behavior has two important aspects which must be specified: the value returned and the effect on subsequent services. ObjLog introduces two kinds of equations to specify and reason about both aspects: data type equations and transition equations respectively.

5.4.1 Data Type Equations

A *data type equation* denotes equality of data type values and is written using the standard equational notation " = ". Because selectors denote functions on the primitive data types, data type equations can be used to define the response of a particular selector. For example, the initial selector value of *hasAttached*(s) for a *Line* object can be written as *hasAttached* = *false*. If $o \in ObjId(Line)$, then this equation represents the trace:

$send(o, hasAttached); ret(false).$

Data type equations can also be used to express invariant relationships between services. (Expressions and parameter lists are assumed to be evaluated left to right.)

$$true = not\ isConnected(s, self)$$
$$true = not\ hasAttached(s) \Rightarrow not\ isConnected(s, o)$$
$$true = hasAttached(s)\ or\ isConnected(s, o)\ or\ not\ isSamePt(s, o, s_2) \Rightarrow not\ attach(s, o, s_2)$$
$$true = hasAttached(s)\ or\ isConnected(s, o) \Rightarrow not\ connect(s, o, s_2)$$

5.4.2 Transition Equations

A *transition equation* denotes equality of transition functions and is written " ≡ " Because updates denote transition functions, transition equations can be used to specify properties of updates. For example, a sequence of two *move* operations is equivalent to a single move for *Line* object:

$$move(x_1, y_1); move(x_2, y_2) \equiv move(x_1 + x_2, y_1 + y_2)$$

Transition functions compose under sequencing. Hence, this transition equation is satisfied because both updates denote the same transition function:

$$getX(s) \mapsto getX(s) + x, \quad getY(s) \mapsto getY(s) + y.$$

Any such sequence of two observations in a message sequence could be replaced by a single observation without affecting the behavior of the remainder of the message

sequence. This represents the behavior of an infinite number of traces each characterized by selector values. Transition equations express invariant properties of objects. ObjLog is unique in that one can specify update behavior directly through transition equations.

A partial set of the transition equations derived from the requirements specification are given below in Figure 5. The identity transition function is denoted *id*. The *move* service is absorptive and commutative and has inverses. Both *attach* and *connect* are idempotent.

$move(0, 0) \equiv id$
$move(x, y); move(-x, -y) \equiv id$
$move(x_1, y_1); move(x_2, y_2) \equiv move(x_2, y_2); move(x_1, y_1)$
$move(x_1, y_1); move(x_2, y_2) \equiv move(x_1 + x_2, y_1 + y_2)$
$attach(s_1, o, s_2); attach(s_1, o, s_2) \equiv attach(s_1, o, s_2)$
$connect(s_1, o, s_2); connect(s_1, o, s_2) \equiv connect(s_1, o, s_2)$

Figure 5.: Transition Equations for Type Line.

6 Refinement of Specifications in ObjLog

This section illustrates three major forms of specification refinement and their expression in ObjLog. A specification may be refined by extension subtyping, specialization, or aggregation. Refinement of specifications may occur during either the analysis or design phases as well as the during transition from analysis to design. Typically, a requirements specification reflects an abstract view of system functionality which must be refined into a design specification that reflects specific system assumption and design choices for objects.

6.1 Refinement by Extension Subtyping

An extension subtype extends the functionality of the parent type with additional selectors, updates, and constructors. Subtype T_1 is an *extension subtype* of T_2, if every implementation of T_1 is also a implementation of T_2, i.e., a class of type T_1 is also of type T_2. Extension is often called abstract implementation or data refinement in the context of abstract data type specification. This is useful in analysis and design when the functionality of the requirements is extended. Design for reuse typically also extends the specification to a wider range of applications.

Object Type **Specification** ExtLine =
  Incremental Refinement of: Line
  Subtype of: Line
  Selectors:
    getMoveCnt: $\rightarrow$ *int*
  Updates:
    reset: $\rightarrow$ *void*
    Definitions
      move(x, y): *getMoveCnt* $\mapsto$ *getMoveCnt* + 1
      reset: *getMoveCnt* $\mapsto$ 0
  Constructors:
    new: *int, int, int, int* $\rightarrow$ *ObjId(ExtLine)*
    Definitions
      new: *getMoveCnt* $\mapsto$ 0
  EndSpec

Figure 6.: Extension Subtype of Line.

An extension subtype *ExtLine* of *Line* is specified above in Figure 6. It adds a new selector *getMoveCnt* and new update *reset*. An additional map is added to the *move* update and *new* constructor to define their effect on the new selector. This is expressed as an *incremental refinement* which specifies update and constructor definitions to add to existing update and constructor maps.

An *ExtLine* object satisfies the same set of transition equations in Figure 5 as a *Line* object, provided we restrict ourselves to the *Line* subsignature. However, if the full *ExtLine* signature is used, *ExtLine* will not satisfy the transition equation: $move(x_1, y_1); move(x_2, y_2) \equiv move(x_1 + x_2, y_1 + y_2)$. The use of the *getMoveCnt* service will distinguish these two message sequences.

6.2 Refinement by Specialization Subtyping

A specialization subtype redefines the parent type specification to specialize its representation. A specialized subtype generally permits a simpler specification than the parent type. For example, a horizontal line is a specialization of a *Line* object which requires only a single *y* coordinate. Another example is a square which is a specialization of a rectangle which is a specialization of a polygon. A polygon must be specified by one point for every vertex, a rectangle requires only two points, and a square requires only one point and a length.

Object Type **Specification** HorizLine =
  Incremental Redefinition of: Line
  Subtype of: Line
  Selectors:
    Hidden
      yPos: → *int*
    Derived:
      getY: *side* → *int*
      Constraints
        getY(*left*) = *getY*(*right*)
        getY(*left*) = *yPos*
  Updates
    Hidden Definitions
      doMove(x, y):
        $getX(left) \mapsto getX(left) + x, getX(right) \mapsto getX(right) + x, yPos \mapsto yPos + y$
  Constructors
    new: *int, int, int, int* → *ObjId*(*HorizLine*)
    Definitions
      new(x_1, y_1, x_2, y_2):
        if $x_1 < x_2$ *or* $(x_1 == x_2$ *and* $y_1 \geq y_2)$
          then $getX(left) \mapsto x_1, getX(right) \mapsto x_2, yPos \mapsto y_1$
          else $getX(right) \mapsto x_1, getX(left) \mapsto x_2, yPos \mapsto y_1$
  EndSpec

Figure 7.: Horizontal Line as a Specialized Subtype.

The *Line* type is specialized to a horizontal line in the specification given in Figure 8. A hidden selector *yPos* is introduced to define the y coordinate of the line. The *getY* selector becomes a derived selector which depends on *yPos*. The constraints express the property of horizontal lines that both endpoints share the same y coordinate. Only the *doMove* update and *new* constructor need be specified in order to define their effects on *yPos*. The other updates used, but did not update *getY*. This is expressed as an

incremental redefinition which specifies update and constructor definitions to replace existing updates and constructor definitions, analogous to overwriting in object-oriented programming languages. In this simple example, the correctness of the subtype is ensured by rewriting portions of the *Line* specification to replace *getY*(*s*) with *yPos*.

A *derived selector* can be expressed as a function of primary selectors, e.g., *getY* is a function of *yPos*. The effect of updates and constructors on derived selectors is implicitly defined and need not be specified. Hence, the use of derived selectors simplifies the definition of updates and constructors, resulting in a more concise specification. Further, by reducing the number of primary selectors, reasoning about the behavior the object is simplified.

A *constraint* is an equation which is always satisfied. Constraints are used to define derived selectors and help to simplify specifications. In the specification above, the values of *getY*(*left*), *getY*(*right*) and *yPos* are always equal. The derived selector *getY* always denotes an integer such that both constraints are satisfied, i.e., *getY* for both sides is equal to *yPos*.

6.3 Refinement by Aggregation

An ObjLog specification can be refined by aggregation in which local services are provided by a component object. This is useful, especially in design, to simplify complicated objects and permit sharing of component objects. The effect of aggregation is to transform primary, derived, and hidden selectors into dependent selectors. This requires that updates and constructors which map these selectors be redefined using messages to the component objects.

In the interest of space, we only sketch the details of this form of refinement. The *Point* object type must specify selectors for *x* and *y* coordinates and a *move* update. The aggregate line type *AggLine* is then parameterized by the *Point* object type, indicating that *AggLine* objects require the services of *Point* objects. Parameterization allows us to build one specification from another. A selector must be added to *AggLine* which specifies the component objects. The selectors *getX* and *getY* then become derived selectors. The local update *doMove* is rewritten to send a *move* messages to the component objects.

7 Comparison to Related Work

The basis of our approach can be summarized by the following four properties which distinguish ObjLog from other specification languages for OOA/OOD:

- **Full Abstraction** ObjLog specifications are property-based and fully abstract, i.e., specify only the message-passing and instantiation behavior of objects. This ensures that internal design decisions can later be changed without affecting the system behavior. This approach is equivalent to the responsibility-driven design approach of Wirfs-Bock et.al. [26]. In contrast, other specification languages are typically model-based, e.g., explicitly use attributes and state machine models. Specifically, the observable behavior of an object is expressed as a trace of message sends and responses, extended to model the instantiation of objects. An ObjLog specification defines the algebraic properties of such traces.

- **Fully Expressive** The *expressiveness* of a specification language is the range of message-passing and instantiation behavior which can be specified relative to sequential object-oriented programming languages. ObjLog fully expresses message-passing based on abstract data type values[4]. However, existing specification languages, both formal and informal, typically express only a limited range of message-passing and instantiation behavior.
- **Equational Reasoning** ObjLog uses equations to specify and reason about interface properties. Specifically, data type equations define the response to a particular service as well as expressing invariant relationships between services. Transition equations express invariant update properties of objects. ObjLog is unique in that one can specify update behavior directly through transition equations.
- **Executable Specifications** ObjLog specifications are potentially executable, i.e., can be reduced automatically through term-rewriting. Equations can then be mechanically evaluated by comparing the reductions of two expressions. Such executable specifications provide the user with a prototype of the abstract system. In contrast, specification languages which require a general first order theorem prover also generally require interactive assistance from the user.

We classify related approaches to specification languages for OOA/OOD into three categories: informal methodologies, algebraic specification, and other specification approaches.

7.1 Informal Methodologies

The informal methodologies discussed in Section 1.2 each provide an informal specification language which permits a complete specification of the message-passing interface. However, a state machine model with special semantics, is typically adopted to model message-passing. This in turn limits the expressiveness of these languages and restricts specification of non-local services to the special case of trigger services. Hence, in the example considered in this paper, recursive services, such as *move*, *MoveAttached*, and *isConnected*, cannot be specified by these languages. Initial values of attributes may be specified, but instantiation cannot be specified and constructors are generally not defined.

For example, the Object-Oriented Modeling Technique (OMT) methodology of Rumbaugh et. al. [22, 23] is comparable in approach and modeling power to other methodologies, including Booch [3], Coad and Yourdon [6, 7], and Shlaer and Mellor [24]. It adopts the Statechart [11, 12] representation of state machines. However, the Statechart model does not adequately express interacting objects because it uses a broadcast model

[4] Some object-oriented programming languages also allow message-passing based on pointers or functions. The use of pointers in a "mixed" language such as C++ typically compromises encapsulation. All "pure" object-oriented languages implicitly use closures (called blocks in Smalltalk) as arguments. A closure is a function plus an environment, i.e., it can be executed outside its original environment and also preserves encapsulation of attributes. Conditionals (if-then-else function) are implemented in "pure" languages as messages to Boolean objects with the two branches as closure arguments. This can be modeled in ObjLog by the use of boolean functions which will later be implemented as messages to Boolean objects.

of communication. Object interactions are modeled by sending events which may be directed to a single object or a class of objects. However, every object which can accept the event, will do so concurrently. Reasoning is difficult because data flow events are used to trigger state machine transitions.

The approach of Coleman and Hayes [13] is based upon formalizations of the three models used in the OMT methodology: the HP-SL type system, Objectcharts, and Hoare Logic. This permits the definition of consistency of models, but proofs involve induction on system structure and hence are not practical. The Objectchart [2, 8] model refines Statecharts by adopting a uni-directional model of message-passing. A transition may generate of a set of messages, but response values cannot be defined.

7.2 Algebraic Specification

A number of algebraic specification languages have been proposed specifically for object-oriented programming. Unfortunately, they are not suitable as OOD specification languages. Each suffers limitations in specifying the message-passing interface or adopts a model of message passing which limits the expressiveness of the language.

The approaches of McKenzie [17], and Breu [4] model message-passing correctly, but specify stateless, functional interfaces, i.e., classes are modeled as abstract data types of objects. Object identity is not modeled. McKenzie supports message-passing with multiple targeting, similar to the CLOS (Common Lisp Object System) language.

FOOPS (Functional and Object-Oriented Programming System) [9] is an algebraic specification language for classes in which classes are modeled as algebras of methods acting on objects. The methods are parameterized by the object identifiers of objects of the class. Message-passing is modeled by the functional composition of services which restricts update services to object identifier return types. Update services with different return types can be modeled, but requires the introduction of additional selectors. Further, the use of functional composition precludes dynamic instantiation. FOOPS cannot express a design which has multiple constructors because only a single constructor with a variable number of parameters is provided. Equational reasoning for data types is supported, but transition equations are not.

Maude [18] is a concurrent, communication based, specification language for objects based on rewrite logic and order-sorted algebra. The object system is represented as a term which evolves under rewrite by messages. However, messages in Maude are uni-directional, i.e., they do not return values. They also include the sender's object identifier, i.e., they are not anonymous. Each visible attribute gives rise to a pair of implicitly generated communications and a rule. The rewrite rules for an update which returns a value must generate an explicit reply. Maude does not provide an mechanism for object instantiation, rather it must be modeled explicitly by instantiating a system or class object. Further, sequencing is implicit in the rewrite rules, and additional, intermediate states may be required. Rewrite, rather than equations, are used for reasoning.

7.3 Other Approaches

A number of other approaches have also been proposed for formal models and formal specification languages. However, there are often limitations in the model of objects and behavior specified.

Several existing specification languages have also been used to specify objects, e.g., VDM [19] and LOTOS [16]. These approaches have the advantage of using existing languages and tools. However, these specification languages do not directly support object abstractions and hence extraneous abstractions must be introduced to model objects. VDM is a model-based language. The use of VDM introduces VDM primitives into the specification. LOTOS is a property-based language based on processes and algebraic specification language. Mayr [16] uses LOTOS to model objects as processes, but this results in the explicit introduction of channels in the specification.

There has also been considerable interest in verification using Hoare Logic. However, this is necessarily based upon explicit representation of state with attributes. The use of first order logic also limits the extent to which reasoning can be mechanized. The Larch language has been extended and used to specify C++ objects [25]. America and de Boer [1] have provided a sound and complete proof system for SPOOL. Leavens and Weihl [15] incorporate message-passing and inheritance, but do not model updates.

Helm, Holland, and Gangopadhyay [14] define an informal specification language for contracts and their composition and refinement. Contracts are specified through type obligations, which define the variables and external interface to be supported, and causal obligations, which define a sequence of message to be sent and a postcondition to be satisfied. Invariants, which the collaborators must maintain, can be specified in first order logic. This is a state-based approach, defined in a high level language based on sending messages and updating variables.

8 Conclusion

In this paper we have informally introduced the ObjLog specification language and shown examples of its use in analysis and design. A simple problem was defined and its behavior characterized with traces. The requirements specification in ObjLog specified this behavior, while other specification languages could only specify a selected subset of traces. Three types of specification refinement were illustrated which are representative of both analysis and design, as well as the transition from analysis to design. The abstract, property-based nature of ObjLog specifications was illustrated in the examples and through the use of equations.

The trace model is useful to describe the behavior of objects and provide a basis for comparing the expressiveness of specification languages. A formal trace model of objects has been defined and used with testing equivalence to define abstract object types [5]. The full, formal algebraic semantics are currently under development. We are also investigating correctness proofs for a variety of systems.

References

[1] P. America and F. de Boer, "A sound and complete proof system for SPOOL," Technical Report 505, Philips Research Laboratories, May 1990.

[2] S. Bear, P. Allen, D. Coleman, and F. Hayes, "Graphical Specification of Object-Oriented Systems," *OOPSLA/ECOOP '90 Conference Proceedings*, Phoenix, Arizona, 1990, pp. 28-37.

[3] G. Booch, *Object-Oriented Design with Applications*, Benjamin/Cummings Series in Ada and Software Engineering, Benjamin/Cummings Publishing, 1991.

[4] R. Breu, *Algebraic Specification Techniques in Object-Oriented Programming Environments*, LNCS 562, Springer Verlag, 1991.

[5] T. L. Briggs and J. Werth, *A Trace Model for Objects*, University of Texas, (Technical Report in preparation).

[6] P. Coad and E. Yourdon, *Object-Oriented Analysis (2nd Edition)*, Yourdon Press, Englewood Cliffs, NJ, 1991.

[7] P. Coad and E. Yourdon, *Object-Oriented Design*, Yourdon Press, Englewood Cliffs, NJ.

[8] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Transactions on Software Engineering*, 18(1), Jan. 1992, pp. 9-18.

[9] J. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented and Relational Semantics with Logical Semantics," in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (editors), MIT Press, 1987, pp. 417-477, (also as CSLI-87-7 July 1987).

[10] J. A. Goguen and T. Winkler, "Introducing OBJ3," Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Aug. 1988.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, 8, 1987, pp. 231-274.

[12] D. Harel, A. Pnueli, J. Pruzan–Schmidt, and R. Sherman, "On the formal semantics of Statecharts," *Proceedings, Logic in Computer Science '87*, 1987, pp. 54-64.

[13] F. Hayes and D. Coleman, "Coherent Models for Object-Oriented Analysis," *OOPSLA '91 Conference Proceedings*, 1991, pp. 171-183.

[14] R. Helm, I. M. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," *OOPSLA/ECOOP '90 Conference Proceedings*, Phoenix, Arizona, 1990, pp. 169-180.

[15] G. T. Leavens and W. E. Weihl, "Reasoning about Object-Oriented Programs that use Subtypes," *OOPSLA/ECOOP '90 Conference Proceedings*, Phoenix, Arizona, Oct. 1990, pp. 212-223.

[16] T. Mayr, "Specification of Object-Oriented Systems in LOTOS," in *Formal Description Techniques*, K. J. Turner (editor), North-Holland, 1989, pp. 107-119.

[17] R. J. McKenzie, *An Algebraic Model of Class, Inheritance, and Message Passing*, PhD Thesis, Univ. of Texas at Austin, 1992.

[18] J. Meseguer, "A Logical Theory of Concurrent Objects," *OOPSLA '89 Conference Proceedings*, Oct., 1989, pp. 101-115.

[19] C. Minkowitz and P. Henderson, "A Formal Description of Object-Oriented Programming Using VDM," *VDM 87*, 1987, LNCS 252, Springer Verlag, pp. 237-259.

[20] D. L. Parnas, "Information Distribution Aspects of Design Methodology," in *Information Processing 71*, C. V. Freiman (editor), North-Holland, 1971, pp. 339-344, (Proceedings of the IFIP Congress 71, Ljubljana Yugoslavia Aug. 23-28, 1971).

[21] D. L. Parnas, "On The Criteria To be Used in Decomposing Systems into Modules," *Communications of ACM*, 15(12), Dec. 1972, pp. 1053-1058.

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

[23] J. Rumbaugh and M. Blaha, "Tutorial Notes: Object-Oriented Modeling and Design," *OOPSLA '91 Conference Proceedings*, 1991.

[24] S. Shlaer and S. J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press Computing Series, Yourdon Press, Englewood Cliffs, NJ, 1988.

[25] J. M. Wing, "Using Larch to Specify Avalon/C++ Objects," *IEEE Transactions on Software Engineering*, 16(9), Sept. 1990, pp. 1076-1088.

[26] R. Wirfs-Brock, B. Wilkerson, and L. Wiener, *Designing Object-Oriented Software*, Prentice Hall, 1990.