Title: Specifying and Proving Object-oriented Programs

Area: Software Engineering

Author: A. C. Fleck
Computer Science Department
University of Iowa
Iowa City, Iowa 52242
Telephone: 319-335-0718
FAX: 319-335-3624
Email: fleck@cs.uiowa.edu

Keywords: object-oriented, algebraic specification, history-sensitive behavior, program proving

© 2004

Specifying and Proving Object-oriented Programs

A. C. Fleck
Computer Science Department
University of Iowa
Iowa City, Iowa 52242

Abstract
This paper develops a new approach to the specification of object behavior. It utilizes an algebraic framework for precise specifications that can provide a basis for the verification of an implementation. These are abstract specifications that express behavior that is dependent on the local store without assuming an explicit configuration of instance variables.

The conceptual basis for this new approach is to focus on behavior exclusively from the perspective of *sequences* of messages rather than that of individual messages. The local store of an object means that the effect of sending the same message may differ from one time to the next. This history-sensitive character is the central feature distinguishing state dependent object classes and purely functional abstract data types. The means by which this sequence outlook can be formally modeled is the main point of the paper.

To substantiate the viability of the technique, an illustrative specification is presented. This specification shows that the essential properties of an object class can be captured without imposing a structure on the object's state representation. To illustrate this, two implementations with highly contrasting state representations are shown, and indications are given of how the specification can be used to prove each is correct.

Introduction

This paper provides conceptual development of a new program specification approach in the context of the object-oriented paradigm. We assume that the reader is familiar with the basic concepts of the object-oriented paradigm as found in, for example, [Bud91, Mey88, Weg90]. Some exposure to the ideas of abstract data types is also assumed (e.g., [GH78]). We develop an algebraic framework that enables precise, abstract specification of object behavior that provides a basis for rigorous deductions. While the formal specifications address the *results* to be obtained not how a computation should achieve them, we strive to develop a formalism that evokes a sense of natural harmony with the object-oriented computing paradigm.

In the programming language context where the character of the encapsulated entities is restricted very little, "objects" have often been characterized as Abstract Data Types [e.g., Mey88, SB86]. However, the fact that an object contains (instance) variables which can be changed, makes the Abstract Data Type (ADT hereafter) interpretation an inadequate formal model for objects, a perception acknowledged previously [Coo90, GM87]. The difficulty is that in the algebraic specification formalism [EM85, MG85], ADTs pertain to *immutable* abstract values and the properties of functions that operate on them. On the other hand, objects which contain local storage may change over time while the object retains a single identity that may have multiple

references. Other work in this direction (e.g., Bre91]) takes the view that whether an operation is realized as a function or a procedure (i.e., through side-effects) is an implementation issue. We adopt the distinctly contrary view that this is an important *design* decision deserving a conspicuous means of expression in the specification.

In the object-oriented setting, computations are expressed in terms of a collection of objects and their interactions. But the operations of each individual object class are described separately. This implies that we distinguish between an individual object and its environment in a clear and systematic way. An *object* bundles together both a (local) state and operations which may change both this state and the environment as well (by sending messages to other objects). However, the specification of the individual objects may be naturally separated from the description of interactions in the environment. It is with this static view of objects that are concerned here. Our goal for object specifications is to use them to reason about properties the results of associated messages must have, assuming that supporting properties of any auxiliary objects can be similarly ascertained.

Abstract Object Classes [AOCs]

There has been wide-spread interest in providing specifications of object behavior, and numerous approaches have been considered. Equational dynamic logic is used in [Wie91] in a database context. Variants of first order logic have been used [BLM91, DKR91]. In [EGS92] the process algebra and algebraic ADT approaches are compared. We pursue an algebraic style of specification in the style of [Bre91], but more along the lines of the abstract machine idea [GM87]. In this paper we are interested in specification of behavior in an *object-based* [Weg90] context. We develop the idea of *Abstract Object Classes* (AOCs) which provide a formal specification of *mutable* objects and the properties of operations on them. We generalize the algebraic ADT model [EM85, MG85] to obtain an intuitive conceptual model for the behavior of objects. The abstract object class model pursued here was first proposed in [Fle95].

We develop an object-oriented counterpart of algebraic ADT specifications that retains the same high level of abstraction for state information. That is, we endeavor to develop the means to provide formal abstract specifications for the behavior of objects which are devoid of specifics of both the arrangement of local storage and algorithms for methods. We seek specifications of precisely what the results should be without introducing bias about how they are to be achieved. Our approach is conceptually similar to the “abstract machine” development of [GM82, MG85, GM87]. Instead of the “hidden sorts” they used to extend ADTs to model object behavior, we incorporate an abstract provision for state descriptions and augment this with “hidden

arguments". We also provide explicit identification of the operations that are intended to cause a state change in the receiver — we regard this as a design decision not an implementation choice.

We follow Wegner[Weg90] in the view that an *object* is a collection of functions that share a state. The technical question is what it means for functions, normally stateless entities, to "share a state". Since the behavior of all the objects in the same *class* is identical (but their states may differ), specifications are given for classes and are referred to as *Abstract Object Classes* (AOCs). Intuitively speaking, our approach is epitomized by the slogan AOC = ADT+State. The state is a characteristic that persists throughout all activations of the methods of the object, and can effect the outcome of those operations even though it is not an (explicit) argument. We formalize a view where the state serves as a "hidden argument" whose value may also be changed through side-effect.

We propose the integration of a mechanism to describe state change into the ADT formalism in a way that is natural to the accepted view of object-oriented programming. To this end, we classify the operations (methods) of an AOC by whether they have one (or both) of the following two state-related properties:

- **state-independent**: a "pure" function whose result depends only on its arguments; the result of a **state-dependent** function may depend on the state of the receiving object as well as the explicit arguments of the message; or
- **side-effect free**: such a function yields only a functional result — its execution never changes the state; a **side-effect inducing** operation (sometimes) changes the object's state as well as yielding a result.

From this perspective an ADT is simply an AOC all of whose operations are state-independent and side-effect free. However in general, an AOC specification involves (up to) four state-related categories of operations.

An **Abstract Object Class** (AOC) incorporates a finite collection of abstract data domains and states, a finite collection of message identifiers, each with a given signature and state-related category, together with a finite collection of (possibly conditional) semantic equations which the operations satisfy.

The class being specified may involve pre-defined classes and ADTs. In the ADT literature, the type that is being defined is referred to as the **type of interest** (TOI). Since we are interested in

classes rather than types and will refer to the **class of interest** (COI). Also, methods that yield results that are members of classes other than the COI will be referred to as **selectors**.

For each state-related category of function, one provides the (visible) signatures as in an ADT. That is, for each message, the type of the arguments and the result are indicated. State-dependent functions are distinguished by having an implicit additional argument — the state. Also, each side-effect producing function f: Domain → Range, has an associated *state transition function* δ_f with signature δ_f: State × Domain → State; that is, δ_f has the state of the receiving object as an argument as well as the arguments of f, and returns a state as its result. All the state transition functions are "hidden" (i.e., explicit invocation by a client is not recognized) — and whenever the (visible) operation f is invoked, there is a corresponding implicit invocation of δ_f. In addition, an AOC may incorporate other hidden functions (in any category) as desired.

We seek to retain the same level of abstractness for the state of an object that is accorded to the data values of an ADT, and to express the effect of operations (methods) without prescribing *any* characteristics of a state representation, that is without making any commitment to either constituent local variables or their values. The behavior of the collection of functions for a given AOC is specified in a manner similar to that of an algebraic ADT [GTW78]. However, in addition we need to deal with the specification of the behavior of the state component of the objects. The most abstract description of the state of an object is implicitly given by the sequence of messages [plus the states of any argument objects] it has ever received. Using such a sequence of messages as a state description allows one to avoid making *any* choices about what variables an object necessarily embeds, or what operations exist to manipulate them. This is a key feature of the approach we take here.

Each object in a class will have a name, and we assume the common "dot" notation for sending a message to an object. For our implicit description of states, the state of each object will be described by a sequence from

$\{<\mu_i\ \pi_{i1}.s_{i1}\ \pi_{i2}.s_{i2} \ldots > \mid i = 1,2, \ldots\}^*$,

where μ_i (i=1,2, …) is one of the side-effect inducing messages of the object and $\pi_{ij}.s_{ij}$ gives the name of the jth argument object (π_{ij}) in the message, and its state (s_{ij}, j=1,2, …).[1] State transition functions associated with each side-effect inducing operation are then specified in terms of these state descriptions in the same way that all other functions of algebraic ADTs are specified, and their invocation (with the same arguments) is assumed to be implicit in the invocation of the corresponding side-effect inducing operation.

---

[1]For set S, S^* denotes the collection of all finite sequences over S.

For the State domain, only state transition operations associated with the (visible) messages are admitted, and their behavior is described entirely in terms of the visible message history — there are *no* explicit state manipulations prescribed in the specification. This approach enables us to avoid any concrete assumptions about the State space in the same way that ADTs keep the data values abstract. Despite this abstractness, state dependent behavior is explicitly acknowledged and precisely described for sequences of messages. We feel this captures an intuitive external view of object behavior.

For brevity, the following **notational conventions** are used in the examples to indicate the categories of the operations of the AOCs. The visible side-effect inducing functions are identified by having a corresponding (hidden) state transition function defined; all other functions are side-effect free. Hidden functions, other than state transition functions, are designated by prefixing a "bullet"(•) to the signature, and their signatures are given in full. For the visible functions, only the "visible" portion of the signature is shown (states are suppressed) so that signatures reflect the form of messages from clients. For the results of a message μ that produces a result in the type of interest(COI), we omit explicit equations if the result returned is to be the receiver (self) in the state determined by δ_{μ}. In a circumstance where another value of the COI is to be returned, an explicit equation would be provided.

Our methodology for the abstract specification of (classes of) objects parallels traditional algebraic ADT specifications in many ways. We believe that our approach clarifies the essential difference between data abstraction and object abstraction. It stresses the connection of the behavior of objects with *sequences* of messages while maintaining the essence of state as an overt but completely abstract entity. While we abstract away from details about state, we do not regard it as a hidden sort as in [GM87] — the presence or absence of state is not transparent. Effective integration of the state concept into formal specification enhances an intuitively natural match of the abstraction with implementation details. The idea of viewing state as a hidden argument captures intuition better since its connection with the visible operations is concealed but not completely severed.

The approach we suggest begins with categorizing the methods according to their dependence and effect on the internal state of an object and making those categories an explicit part of the specification. Specifications of the state transition functions associated with the side-effect inducing methods and provide an abstract description of each of them and the states. Once the state behavior is established, descriptions of the state-dependent methods can be pursued.

Because of the role of states in overall behavior, our specifications are intended to distinguish objects only if they have observable behavior which differs. Thus we prefer the final algebra interpretation [GH78, Kam83, Wan79] over the initial algebra view [GTW78]. Of course, the state-independent methods constitute an ordinary ADT and can be specified by well established means.

Implementation Correctness

Since one of the primary goals of a formal specification is to provide a basis for gauging the correctness of an implementation, it is unsatisfactory to leave what it means for an implementation to conform to a specification unstated and informally understood. A correct AOC implementation must possess all the essential information guaranteed by the specification but should not have its configuration of instance variables predetermined. Each implementation of a (visible) method must operate in a way that is consistent with the specification, but in doing so may permit auxiliary information in any way deemed helpful. Hidden functions are not required to be explicitly present in an implementation.

We wish to describe object behavior and hence seek to avoid differentiation of objects based on their internal structure. Hence two states are **equivalent** provided that if we start with two objects, one in each of these states, then for every sequence of messages ending with a selector message, both objects return the same result — that is, the two objects are indistinguishable by any external means. In the ADT literature this is known as the “final algebra” view.

Since our approach results in an ADT-style description of a collection of functions, ADT analysis concepts generally apply. For instance, “sufficient completeness” for terms whose primary operation is a selector remains a property of interest. The primary difference to be accounted for is with the State domain which is represented by sequences of side-effect inducing messages. For the model of an AOC specification we take an “abstract machine” view [MG85, GM87]. The states are the indistinguishability classes of the state representations occurring in our specification, with transitions given by the collection of next-state functions. Any implementation equivalent to the minimal state machine is regarded as acceptable.

We assume that we are interested here in object-oriented implementations of AOCs. That is, an implementation will consist of a class incorporating suitable instance variables to model desired local storage, and methods defined to realize the behavior described by the AOC — that is, a **concrete class**. A correct implementation is required to be able to perform concrete computations that will accurately “realize” the results promised by the abstract specification. The

nature of the realization is captured by describing the way in which the concrete objects represent the abstract objects.

A concrete class C is a **correct implementation** of an AOC provided there are **representation mappings**

α: visible methods of C → visible AOC methods,

ρ: instance variable value configurations of C → AOC states

that have the following properties:

(i) α is injective and surjective (one-to-one and onto) and corresponding methods accept the same number and types of arguments,

(ii) for every AOC state s, there is an equivalent AOC state t and a configuration χ of instance variable values of C so that $\rho(\chi) = t$ (ρ is onto equivalence classes of AOC states), and

(iii) every concrete object o and sequence of its messages $\mu_1(v_{11}, \ldots)\ \mu_2(v_{21}, \ldots) \ldots$, has the property

$\rho(o.\mu_1(v_{11}, \ldots)\ \mu_2(v_{21}, \ldots) \ldots) = \rho(o).\alpha(\mu_1)(\rho(v_1), \ldots)\ \alpha(\mu_2)(\rho(v_{21}), \ldots) \ldots.$

The orientation of the representation mappings from the concrete to the abstract class is chosen so that the abstract values and states may have multiple representations in the implementation (i.e., excess information is tolerated). Property (i) simply requires the specification and implementation to have the same message protocols. Property (ii) insists that every AOC state have at least one equivalent counterpart in the implementation. Property (iii) requires that each computation in the implementation yield a result that is a valid representation of the outcome determined by the specification.

We next present an example incorporating state-dependent behavior and illustrate the message history description of states. After presenting the specification we consider implementations and the nature of the correspondence of concrete and abstract state.

Example — Histogram AOC.

Correctness issues will be illustrated using a simple example. This example is a familiar device — the histogram, a commonly used aggregation of values that may be inspected, graphed, etc. For illustration we assume methods of the class with informal descriptions as follows:

empty — the initial, empty histogram,

tally(v) — add value v to the receiving histogram,

high — return the greatest value in the histogram,

low — return the least value in the histogram,

average — return the average of the values in the histogram,

sampleSize — return the number of observations in the histogram,

frequency(v) — return the number of times value v appears in the histogram.

class Histogram

Signatures — state-dependent operations

tally: Value → Histogram

•δ_{tally}: State, Value → State

high: ε → Value

low: ε → Value

average: ε → Value

sampleSize: ε → Natural

frequency: Value → Natural

State-independent operations

empty: ε → Histogram

The results of sending messages to a Histogram object are specified by the collection of equations below. For the 'tally' message we want the result to be the receiving object in its new state. This is implicitly indicated by providing no equation for the 'tally' operation itself and interpreting this omission as the signal for this common option. The relevant equation is that provided for the associated state transition function. Equations use conditional expressions as is typical in ADT descriptions. In addition to the COI (Histogram), there are two pre-defined classes (ADTs actually) assumed in this specification. Natural is thought of as the familiar natural numbers, and Value is assumed to provide a domain with suitable numeric operations. Details of these classes are omitted.

Equations — for each s∈State and v,w∈Value

δ_{tally}(s,v) = s**^**'tally'(v)

high(s**^**'tally'(v)) = **if** s ≡ 'empty' **then** v **else** max(v, high(s))

low(s**^**'tally'(v)) = **if** s ≡ 'empty' **then** v **else** min(v, low(s))

sampleSize('empty') = 0

sampleSize(s**^**'tally'(v)) = **if** s ≡ 'empty' **then** 1 **else** 1 + sampleSize(s)

frequency('empty', w) = 0

frequency(s**^**'tally'(v), w) = **if** v = w **then** 1 + frequency(s, w) **else** frequency(s, w)

average(s**^**'tally'(v)) = **if** s ≡ 'empty' **then** v
else (v + sampleSize(s)*average(s)) / (1 + sampleSize(s))

Note that the state of a Histogram object is assumed to be determined by the sequence of all the side-effect inducing messages it has received since its creation. This message history representation of states consists of a sequence of message identifiers and their arguments (including their state information). Comparisons of states written as "≡" denote state equivalence. Actually the above equations are the "ok-equations". For brevity the error cases such as sending the 'high' message to the 'empty' Histogram are omitted.

We will next consider two implementations of the Histogram AOC. We adopt Smalltalk[GR89] as the implementation language since it provides a conceptually clean object-oriented language.

Figure 1 shows an implementation (in Smalltalk/V) based on the pre-defined class OrderedCollection. The representation mappings for this implementation use the identifiers to indicate the method correspondence. There are no instance variables augmenting the OrderedCollection object itself, and the storage representation mapping associates an OrderedCollection

with the AOC state empty^tally(v1)^tally(v2)^ … ^tally(vk).

Of course, to judge correctness, we need to know about the properties of the pre-defined class OrderedCollection. Space does not permit developing this in detail, but this implementation uses only basic *sequence* characteristics of this class. With just a little knowledge of Smalltalk, it is a matter of inspection to see that this implementation is correct. The program in Figure 1 deliberately ignores practical aspects of implementing the Histogram AOC. It was constructed to emphasize the naturalness of this specification method for the object-oriented programming paradigm. As such the defining equations of the AOC were adopted as method definitions with as little change as possible. The method implementations exhibit only minor syntactic variation from the specification equations. The one exception to this is the 'tally' method. However it should also be clear that with the indicated storage representation mapping, the concrete 'tally' method conforms perfectly to the 'tally' transition function δ_{tally} of the specification.

```
OrderedCollection subclass: #Histogram1
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !

!Histogram1 class methods !
empty
        "create histogram with no entries"
    ^super new! !

!Histogram1 methods !
average
        "return average value for non-empty histogram"
    | s v n |
    s := self copy.  v := s removeLast.
    n := s sampleSize.
    ^(n = 0) ifTrue: [v] ifFalse: [((n * s average) + v)/(n+1)]!

frequency: aValue
        "same result as superclass occurrencesOf method"
    | s v |
    ^(self = Histogram1 empty)
       ifTrue: [0]
       ifFalse: [s := self copy.  v := s removeLast.
                 (s frequency: aValue) +
                 ((v = aValue) ifTrue: [1] ifFalse: [0])]!

high
        "return largest value in non-empty histogram"
    | s v |
    s := self copy.  v := s removeLast.
    ^(s sampleSize = 0) ifTrue: [v] ifFalse: [(s high) max: v]!

low
        "return smallest value in non-empty histogram"
    | s v |
    s := self copy.  v := s removeLast.
    ^(s sampleSize = 0) ifTrue: [v] ifFalse: [(s low) min: v]!

sampleSize
        "same result as superclass size method"
    | s |
    ^(self = Histogram1 empty)
       ifTrue: [0]
       ifFalse: [s := self copy.  s removeLast.  1 + s sampleSize)]!

tally: aValue
        "add next value at the end of the list"
    self addLast: aValue! !
```

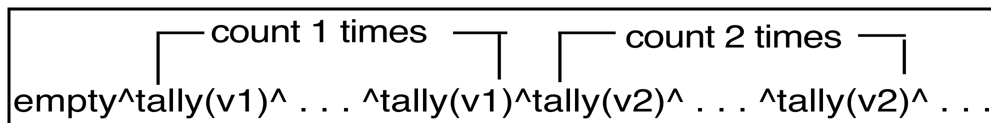
Figure 1: OrderedCollection Implementation of Histogram Class.

As our second implementation of the Histogram AOC we consider a more realistic approach. Instead of a simple sequence representation, a table organization is used so that each value is retained just once, together with a count of its number of occurrences. In addition, this implementation uses instance variables that are revised by the tally message to retain continually up-to-date summary values, and results in an entirely different internal structure than the first implementation. This implementation uses the pre-defined Dictionary class.

The storage representation ρ for the class definition in Figure 2 associates a storage configuration of the instance variables

n	high	low	sum	self	
w	x	y	z	v1	count 1
				v2	count 2
				...	...
				vk	count k

with the abstract state



These message histories do not include all the states of the Histogram AOC. However, with a little reflection, it can be seen that every AOC state is indistinguishable from a state of this form, so the criterion for a representation function is met. Of course, the visible method correspondence is the obvious one given by the identifiers.

The complete formal verification of correctness of the Histogram implementation in Figure 2 is much more difficult to demonstrate than the first version. Again we are dependent on the properties of the underlying (Dictionary) class, but in addition there is a more complex representation (notice that this implementation makes use of redundant information stored in the instance variables). The method implementations look nothing like the specification equations, and the hidden state transition function in the AOC has no functional counterpart in the implementation since its role is simulated by the manipulation of the instance variables. The verification of all the methods essentially reduces to verifying the correct operation of the 'tally' method.

```
Dictionary subclass: #Histogram2
  instanceVariableNames:
    'high low n sum '
  classVariableNames: ''
  poolDictionaries: '' !

!Histogram2 class methods !
empty
        "create an empty histogram with n and sum 0"
    ^Histogram2 new initialize! !

!Histogram2 methods !
average
    ^sum/n!

frequency: aValue
    ^self at: aValue ifAbsent: [0]!

high
    ^high!

initialize
        "set n and sum for empty histogram"
    n := 0.  sum := 0.!

low
    ^low!

sampleSize
    ^n!

tally: aValue
        "Dictionary keys are values, entries are frequencies"
        "instance variables are updated with each new entry"
    self at: aValue
         put: (1 + (self at: aValue ifAbsent: [0])).
    n := n + 1.  sum := sum + aValue.
    (n = 1)
      ifTrue: [high := aValue. low := aValue]
      ifFalse: [(aValue>high) ifTrue: [high := aValue].
                (aValue<low) ifTrue: [low := aValue]].! !
```

Figure 2: Dictionary implementation of Histogram Class.

To formally justify the correctness of this implementation, we are essentially faced with a general program proving task. The strategy would be to show that if the implementation is applied to 'empty', instance variables are initialized correctly, and that the property of having the instance

variables contain the up-to-date entry count (n), sum, high, and low values is an invariant of the execution of the 'tally' method. Elaboration of these details is not included here.

Conclusions

Our presentation has concentrated on the conceptual development of the approach. Specifications are comprised of the descriptions of a collection of functions and hence retain most of the characteristics of ADT specifications, while integrating a perspective of the state concept in a unique and intuitive way. Because of the persistent nature of the state attribute, an AOC specification describes behavior for an arbitrary finite sequence of messages, rather than behavior of single invocations. This is the key characteristic which differentiates AOCs and ADTs. While given the same arguments an ADT function always yields the same results, different instances of the same message to an object may yield different results. Unique problems for providing specifications are created by this difference. One of these problems is devising a specification that relates to sequences of messages rather than individual operation behavior. Another is to maintain states at a completely abstract level. Adopting state descriptions that consist of sequences of side-effect inducing operations is a key and contributes to the solution of both of these problems — states remain completely abstract, and the behavior of objects with respect to sequences of messages is captured even though only the behavior for the “last” message is explicitly provided.

A formal definition of the vitally important concept of an implementation “conforming to” a specification has been provided. The abstractness of our specifications readily admits a variety of implementations and a correctness concept must be general enough to encompass them all. The formulation we have suggested for specifications is suitable for this approach to organizing program proofs and judging implementation correctness. The viability and advantages of the new approach to object specification have been illustrated.

Several important issues remain to be explored. As noted a the outset, inheritance has not been considered. Inheritance has been treated in the algebraic ADT style elsewhere [Bre91]. While the model used in that work differs in several important ways from ours, it would appear that similar treatment is possible with the specification strategy advocated here. Also the idea of “class variables” (storage shared by all the objects of a class) is not considered in our development. Our specifications address behavior determined by the message sequence received by a single object, and it is not (directly) influenced by messages received by other objects of the same class. To model behaviors with such dependencies requires treatment at the level of the environment,

not individual objects. But then the class variable idea really corrupts the encapsulation goals of the object-oriented approach and perhaps should be expected to resist uniform treatment.

References

[BLM91] A. Brogi, E. Lamma & P. Mello, "Objects in a logic programming framework", *First Russian Conf. on Logic Programming* A. Voronkov (ed.), Springer-Verlag, Lect. Notes in AI, V. 592, 1991, 102-113.

[Bre91] R. Breu, *Algebraic Specification Techniques in Object Oriented Programming Environments*, Springer-Verlag, Lect. Notes in Comp. Sci. V. 562, 1991, 228 pp.

[Bud91] T. Budd, *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991, 399 pp.

[Coo90] W. R. Cook, "Object-oriented programming versus abstract data types", *Foundations of Object-Oriented Languages* (J. W. de Bakker, W. P. de Roever & G. Rozenberg, eds.), Springer-Verlag, Lect. Notes in Comp. Sci. V. 489, 1990, 151-178.

[DKR91] R. Duke, P. King, G. Rose & G. Smith, "The Object-Z specification language, version 1", Tech Rpt. 91-1, Software Verification Research Centre, Univ. of Queensland, Australia, May 1991, 61 pp.

[EGS92] H. Ehrig, M. Gogolla & A. Sernadas, "Objects and their specification", Eighth Workshop on Abstract Data Types (M. Bidoit & C. Choppy, eds.), Springer-Verlag, Lect. Notes in Comp. Sci., V. 655, 1992, 40-66.

[EM85] H. Ehrig & B. Mahr, *Fundamentals of Algebraic Specification 1*, Springer-Verlag, 1985, 321 pp.

[Fle95] A. C. Fleck, "Algebraic specifications of objects", Math. Modelling & Sci. Comput. 6 (1996), also presented at Tenth Inter. Conf. on Mathematical Modelling and Scientific Computing, Boston MA, 1995.

[Gog78] J. A. Goguen, "Abstract errors for abstract data types", *Formal Descriptions of Programming Concepts* (E. J. Neuhold, ed.), North-Holland, 1978, 491-522.

[GM82] J. A. Goguen & J. Meseguer, "Universal realization, persistent interconnection and implementation of abstract modules", Proc. 9th Inter. Conf. on Automata, Languages and Programming (A. Poigne' & D. Rydeheard, eds.), Springer-Verlag, Lect. Notes in Comp. Sci., V. 140, 1982, 313-333.

[GM87] J. A. Goguen & J. Meseguer, "Unifying functional, object-oriented and relational programming", *Research Directions in Object-Oriented Programming* (B. Shriver & P. Wegner, eds.), MIT Press, 1987, 417-477.

[GM92] J. A. Goguen & J. Meseguer, "Order-sorted algebra I: equational deduction for multiple inheritance, overloading, exceptions and partial operations", *Theor. Comput. Science* 105,2(1992), 217-274.

[GTW78] J. A. Goguen, J. W. Thatcher & E. G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types", *Current Trends in Programming Methodology, Vol. IV: Data Structuring* (R. T. Yeh, ed.), Prentice-Hall, 1978, 80-149.

[GR89] A. Goldberg & D. Robson, *Smalltalk 80: the language*, Addison-Wesley, 1989, 585 pp.

[GH78] J. V. Guttag & J. J. Horning, "The algebraic specification of abstract data types", *Acta Informatica* 10(1978), 27-52.

[Kam83] S. Kamin, "Final data types and their specification", *ACM Trans. Prog. Lang. & Sys.* 5,1(1983), 97-123.

[MG85] J. Meseguer & J. A. Goguen, "Initiality, induction, and computability", in *Algebraic Methods in Semantics* (M. Nivat & J. C. Reynolds, eds.), Cambridge University Press, 1985, 458-541.

[Mey88] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988, 534 pp.

[SA89] G. Smolka & H. Ait-Kaci, "Inheritance hierarchies: semantics and unification", *J. Symbolic Computation* 7(1989), 343-370.

[SB86] M. Stefik & D. G. Bobrow, “Object-oriented programming: themes and variations”, *The AI Magazine*, 1986, 40-62.

[Wan79] M. Wand, “Final algebra semantics and data type extensions”, *J. Comput. & Sys. Sci.* 19,1(1979), 27-44.

[Weg90] P. Wegner, “Concepts and paradigms of object-oriented programming”, *OOPS Messenger* 1,1(Aug. 1990), 7-87.

[Wie91] R. J. Wieringa, “A formalization of objects using equational dynamic logic”, DOOD'91 (C. Delobel, M. Kifer & Y. Masunga, eds.), Springer-Verlag, Lect. Notes in Comp. Sci., V. 566, 1991, 431-452.