Pergamon

0005-1098(93)E0020-5

Automatica, Vol. 30, No. 8, pp. 1221–1261, 1994

0005-1098/94 $7.00 + 0.00

Survey Paper

Object-oriented Programming in Control System Design: a Survey*

C. P. JOBLING,† P. W. GRANT,‡ H. A. BARKER† and P. TOWNSEND‡

This paper discusses how the object-oriented programming paradigm may help to revolutionize the implementation and use of computer-based environments for the design of control systems.

Key Words—Object-oriented programming; computer-aided control systems design; programming environments; programming methods; software design methods; user interfaces; modelling support; database management.

Abstract—The object-oriented paradigm shows great potential as a means for revolutionizing software development. In the last decade, much research has been directed towards the development of design methods, languages, environments, reusable libraries of software components and database systems to support this paradigm. The first part of the paper presents the terminology of the object-oriented paradigm, reviews the state-of-the-art in object-oriented programming and discusses class libraries and object-oriented design. The second part of the paper discusses its application in the area of computer-aided control system design. It is argued that the adoption of these ideas will increase greatly the productivity of software developers in this field and improve the facilities that will be offered to users.

1. INTRODUCTION

RENTSCH (1982) SUGGESTED that object-oriented programming (OOP) would be in the 1980s what structured programming was in the 1970s. Although this may have been somewhat over-optimistic as an estimate of the rate of adoption of OOP by the software community, the importance of this technology is now more widely recognized, and the prediction may well be fulfilled in the 1990s.

In Section 2 of this paper "a systems view of programming" is used to show that OOP is a useful way of decomposing complex software systems and modelling real-life artefacts. These facts should make the paradigm easier to understand by control engineers and indicate its usefulness in the particular area of computer-aided control systems design (CACSD). In later sections the main benefits that OOP offers for software development, including powerful data-modelling concepts, good encapsulation of data and functions, ease of re-use and savings in maintenance costs, are described. The development will include a brief tutorial introduction to the terminology used and an overview of the languages, environments and software design methods that support the OOP methodology. The application and impact of these techniques to the development of CACSD environments will be examined in Section 3.

Throughout Sections 2 and 3 of this paper, the elegant graphical notation of Rumbaugh and colleagues' object-modelling technique (OMT) will be used. The elements of the notation that are used in this paper are summarized in Appendix A. A complete description of the notation may be found in Rumbaugh *et al.* (1991).

2. OBJECT-ORIENTED PROGRAMMING

2.1. *A systems view of software engineering*

Software systems, like many other engineering artefacts, are complex entities. In common with other systems in engineering, complexity takes the form of a hierarchy. A complex system is composed of related subsystems which in turn have their own subsystems and so on until some lowest level of elementary components is reached (Courtois, 1985). The fact that systems can be decomposed into a hierarchical structure is a major reason why engineers are able to understand, describe and even "see" such systems and their parts (Simon, 1982). In performing such decomposition, use is often made of the fact that interactions between components at the same level are stronger than those between levels. Thus engineers are able to separate the high frequency dynamics of components, which involves their internal structure, from the low frequency dynamics involving the interaction between them. This in turn provides the separation that makes it possible to study components in isolation (Booch, 1990). This systems perspective,

*Received 27 March 1991; revised 29 June 1992; received in final form 15 September 1993. This paper was not presented at any IFAC meeting. This paper was recommended for publication in revised form by Editor K. J. Åström. Corresponding author C. P. Jobling. Tel. +44 792 295580; Fax +44 792 295686; E-mail C.P.Jobling@Swansea.ac.uk.

†Department of Electrical and Electronic Engineering, University College of Swansea, Singleton Park, Swansea SA2 8PP, U.K.

‡Department of Computer Science, University College of Swansea, Singleton Park, Swansea SA2 8PP, U.K.

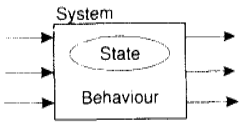

FIG. 1. A dynamic system.

familiar to all control engineers, provides useful insight into the problems of analyzing, designing and implementing software systems.

The classical definition of a dynamic system is that of a collection of matter, parts or components which are included inside a specified, often arbitrary, boundary in which one or more aspects of the system change with time (Shearer *et al.*, 1971). The boundary of a dynamic system encloses some state and behaviour; inputs which cross the system's boundary excite the behaviour and influence its state; outputs from the system reflect the state or behaviour of the system. The block diagram in Fig. 1 illustrates this view of a dynamic system. It represents the encapsulation of state and behaviour and the well-defined interfaces through which external influences impinge on the system. As a model, this view applies equally well to a software system as to a control system.

The ideas that developed into structured programming, and in particular divide and conquer, top down decomposition, or stepwise refinement were first expounded by Dijkstra (1965) and the actual term "structured programming" was also coined by him in a later paper Dijkstra (1969). It was not until the 1970s, however, that structured programming became the paradigm that "every one was in favour of, every manufacturer promoted its products as supporting, every manager paid lip service to, and every programmer practised (differently)" (Rentsch, 1982).

Structured programming is essentially the top–down decomposition of software systems on the basis of the functions, or behaviour, of the system components. As a principle it spawned analysis and design techniques such as structured analysis (DeMarco, 1978) and structured design (Constantine and Yourdon, 1979) and implementation languages like PASCAL (Wirth, 1971). However, as a modelling technique to help software engineers cope with the complexity of very large software systems and thereby generate software to specification, in budget and on time, functional decomposition is seriously flawed because it violates the normal principles of system decomposition by not encapsulating a system's state within a system's boundary.

In engineering terms, a software function is representable by the general formula

$$\mathbf{y}_f = f(\mathbf{u}_f),$$

where $\mathbf{u}_f$ is a vector of input data objects and $\mathbf{y}_f$ is a vector of output data objects. The output values depend only on the input values, so a software function has no state and no memory (Fig. 2). However, as already indicated, most useful software systems have state as well as behaviour. Thus at some high level of abstraction the system obeys equations of

FIG. 2. A software function.

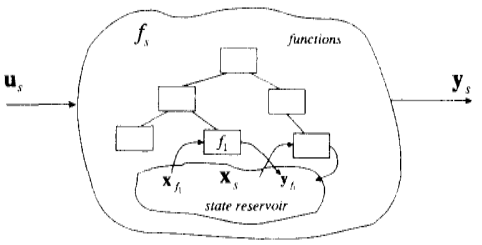

FIG. 3. Functional decomposition.

the form

$$\mathbf{y}_s = f(\mathbf{u}_s, \mathbf{x}_s, t),$$

where $\mathbf{u}_s$ and $\mathbf{y}_s$ are the system inputs and outputs, respectively, $\mathbf{x}_s$ is the state of the software system and t is time. Since a software system has a state whereas a software function has none, in order to model a software system by functional decomposition it is necessary to have the state of the system outside the units of decomposition. As Fig. 3 illustrates, functional decomposition is therefore an attempt to model a dynamic system by a hierarchy of state-less functions and a global state "reservoir" from which, in principle, any function may draw its inputs and deposit its outputs.

This clearly violates the principle of dynamic system decomposition as a control engineer would understand it. To ensure that the boundary principle is not violated, both state and behaviour must be part of each subsystem. We may thus agree with Booch (1986) who states that functional decomposition does not effectively address data abstraction (Liskov and Zilles, 1974; Guttag, 1977) and data hiding (Parnas, 1972) which are essentially software-centred statements of the system boundary principle. Further, functional decomposition as a design principle is not responsive to change in the problem space. The reason for this is that important data end up being passed around a lot of modules through common blocks or as global data. If the structure of the data changes, then so must many of the software components.

In contrast, the object-oriented paradigm enables the basic components of a software system to be viewed as automata, as shown in Fig. 4 (Wegner, 1990). An object encapsulates both state $\mathbf{x}_o$ and behaviour which consists of operations $f_{o_1}, f_{o_2}, \ldots, f_{o_n}$

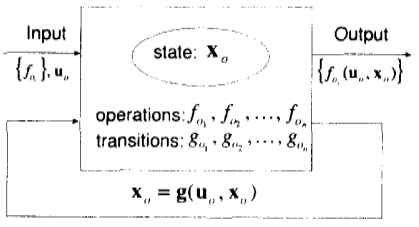

FIG. 4. A component of an object-oriented software system viewed as an automaton.

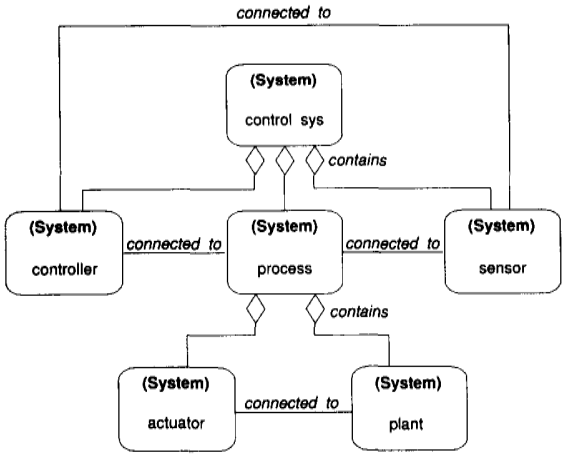

FIG. 5. An object-oriented model of the dynamic system illustrated in Fig. 6.

and state transitions $g_{o_1}, g_{o_2}, \ldots, g_{o_n}$. The input "signal" consists of a stream of messages each of which contains a function name f_{o_i} and (optionally) some data $\mathbf{u}_o$. On receiving a message, the object will perform operation f_{i_o} on data $\mathbf{u}_o$ and state $\mathbf{x}_o$ to generate an output $f_i(\mathbf{u}_o, \mathbf{x}_o)$. It may also generate a state-transition $\mathbf{x}_o' = \mathbf{g}_o(\mathbf{u}_o, \mathbf{x}_o)$. The output stream may itself contain messages to other objects.

In a software system, objects at the same level in a hierarchy are connected together by associations (Rumbaugh *et al.*, 1991) which are the software analogues of signal wires. Levels in the hierarchy are separated by special associations called aggregations which indicate the idea that groups of objects are part of other objects. Figure 5 illustrates how association and aggregation might be used to model the hierarchically structured dynamic system illustrated in Fig. 6. Associations are indicated by lines between objects, aggregations by lines terminated by diamond shapes. This figure serves not only to indicate that object-oriented programming makes software decomposition like dynamic system decomposition, but also to demonstrate that object-oriented programming may usefully be regarded as a way of modelling the real world. Programming as modelling of real-world objects, and software systems as simulations of real-world systems is a very attractive way of looking at the problems in CACSD that are addressed in Section 3 of this paper.

2.2. *Terminology*

To the newcomer, the terminology of the object-oriented paradigm seems strange and bewildering. The cognoscenti pepper their conversations with terms such as "objects", "messages", "methods", "inheritance" and "polymorphism" with little regard for the programmer who has yet to discover data types. The aim of this section is to put over the essence of OOP and to define some of the key terms and concepts. In doing so, the systems view of software will be expanded to illustrate the concepts. The more interested reader is directed to Section 3.7, in which the many tutorial introductions to the topic are reviewed, and to the glossary in Appendix B.

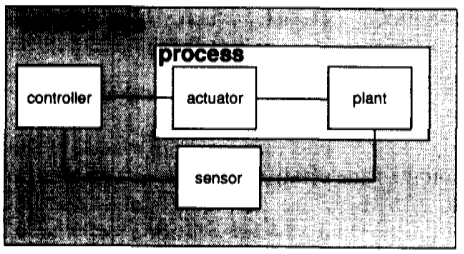

FIG. 6. A model of an hierarchically structured dynamic system.

2.2.1. *Objects.* The object-oriented paradigm is based on a single entity, the object, which represents both data and the functions and procedures that operate on the data. In an object-oriented software system, the state is contained, or encapsulated, within objects. The state of an object, and hence ultimately the state of a software system, may be manipulated only by the *interface functions* that define the outside view of the behaviour of an object.

To take the systems analogy further, it is useful to look at a software object in terms of the familiar state-space model used in control (Fig. 7). Hidden away inside the object is the state vector and the transition matrix. The state may be modified from the outside world only by calling on *controller functions* in the input part of the interface. Some input functions merely request the object to indicate its internal structure, which it does by the use of *observer functions* that return a view of the object's state. Other functions will modify the state and return an output and so act as *controller-observer functions*. Some states may be implementational details that are neither observable nor controllable. It is important to note that the state of an object should only be modified or observed through the interface of the object. This ensures the implementation of the object is immaterial to its input–output behaviour, and is a kind of *similarity* principle. In the OOP terminology, the interface functions are called *methods*, the input and output signal streams are called *messages*.

At this point, a concrete example may help to clarify the discussion. Consider an object that represents a complex number z (Fig. 8). In an object-oriented implementation of z we might expect to find functions such as those shown in Table 1. Some of these functions *observe* the value of z; its real and imaginary parts, its complex conjugate, its modulus (often obtained using the function ABS) and its argument. Depending on what is actually stored in the state of z, some of these functions make requests that may require computation, but none actually change the state, and so they are all *observer functions*. Typical *controller functions* would be constructor functions that actually create a Complex Number object by statements equivalent to the equations:

$$z = x + jy$$
$$z = x$$
$$z = jy$$
$$z = re^{j\theta}$$

and *operator functions* such as +, −, * and / whose

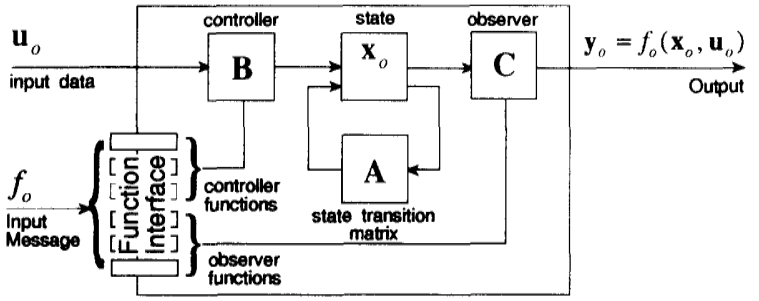

FIG. 7. State-space model of a software object.

FIG. 8. Complex number object z.

TABLE 1. Input messages and corresponding state transitions for the complex number object z shown in Fig. 8

Message	State-transition	Output
$=x+jy$	$z \leftarrow x+jy$	—
Real	—	x
Imag	—	y
Conj	—	$x-jy$
Abs	—	$\sqrt{x^2+y^2}$
Arg	—	$\tan^{-1} y/x$
$+1+j2$	$z \leftarrow (x+1)+(y+j2)$	—

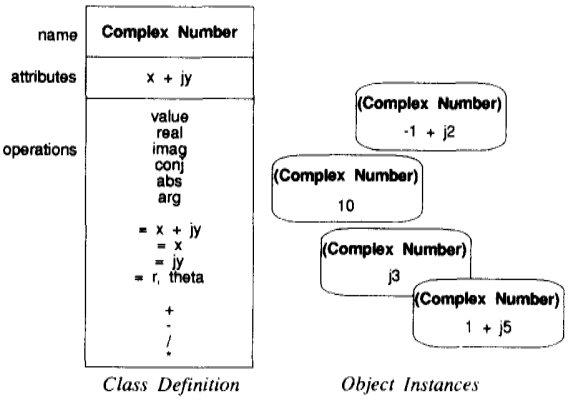

FIG. 9. Diagrammatic distinction of class and object.

action in equations like

$$z \leftarrow z + (1 + j2)$$

results in a modification of the state of z. The actual implementation of the data contained within the Complex Number object is immaterial. For example, it would be usual to see z implemented as a pair of real numbers x and y, but there would be no difference in the outside view, or similarity, of a complex object implemented in polar form with r and θ.

As we have already seen, the state of an object is manipulated by the receipt of *messages* from other objects. Messages usually contain instructions and data. The object which receives the message invokes the *method* or procedure appropriate to the instruction part of the message and uses the data to perform the manipulation required. The set of all possible messages that an object can respond to is known as its *protocol*.

2.2.2. *Classes.* An object in an object-oriented system can be thought of as an instance of a particular data type. The data and the operations that may be performed on objects of this type are specified in an abstract data type definition (Meyer, 1988). Such a type specification describes the common features and behaviour of a group of similar *objects* and in the object-oriented terminology is called a *class definition*. To illustrate the distinction between class and object consider the class definition for a complex number object. The class definition for a Complex Number describes the features that apply to all complex numbers, whereas the Complex Number z is an actual *instantiation* of an object from the Complex Number class. This object has a particular value that makes it unique from all other Complex Number objects.

In talking about object-oriented systems, it is sometimes useful to use the indefinite article for classes (*a* Complex Number) and the definite article for the objects instantiated from the class (*the* Complex Number z). In the OMT notation, this distinction is made by the shape of the box as shown in Fig. 9. Rectangular boxes represent the class definitions, rounded boxes the objects that may exist at a given time during the execution of the software system. In a software system, any object will have its own unique state, but shares the functions defined in the class with all other objects of the same type.

The object-oriented approach to software decomposition makes the data in the system the key to splitting the sytem up into subsystems. This is a "bottom-up" approach which is justified because (Meyer, 1988):

- it is easier to combine actions on data if the data structures are taken into account;
- it is easier to build reusable components if they take account of the data; and
- data structures are the most stable features of a software system; the specification of what is to be done with the data is more often changed than the structure of the data itself.

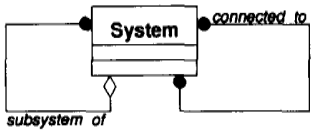

FIG. 10. OMT class diagram for a dynamic system.

In addition, decomposition on the basis of objects is true to the spirit of data abstraction and data hiding. Systems decomposed in this way are resistant to change and easier to maintain because changes tend to be local rather than global.

2.2.3. *Composition and decomposition.* In modelling and therefore in OOP, the need to capture associations between classes is important. Consider the object model (Fig. 5) of the control system shown in Fig. 6. In order to model correctly the class from which System objects are instantiated, we need to be able to indicate that a System may be *connected to* one or more other Systems. The notion of a connection to another System is an *association* of the class System. Similarly a System may contain one or more other Systems. This idea of "having" or *containing* some attribute is what Rumbaugh *et al.* (1991) call an *aggregate association*. An OMT class diagram for System which has these properties is shown in Fig. 10. Note that lines are used to indicate associations, diamonds to indicate aggregate associations and filled circles to indicate zero or more associations (or multiplicity). Thus, this diagram can be interpreted to mean that a System may be associated by a many-to-many association called *connected to* to zero or more other Systems. Also, any System may contain zero or more Systems as indicated by the aggregate association *subsystem of*.

Aggregate relationships produce a *parts hierarchy* which is orthogonal to the *inheritance hierarchy* discussed later. In systems terms, an aggregate hierarchy is used to model how systems are created from subsystems. Associations indicate how components at the same level of a subsystem interact.

2.2.4. *Polymorphism.* Another feature of OOP systems is the idea of *polymorphism*. This means that different objects can respond to the same message with *different* methods. It removes the need for the creation of numerous differently named functions which provide similar services for different data types. To take an example from FORTRAN, consider the generic function ABS. In order to implement this function for the different data types, the programmer must remember to use either ABS, DABS, IABS or CABS depending on whether the data is REAL, DOUBLE, INTEGER or COMPLEX. In an object-oriented system, these data types would be represented as different classes and the developer of each class would have to implement a suitable procedure called ABS that returned the absolute value of a number. However, the user of the class has only to remember that the function named ABS will return the absolute value of any numerical data type. This helps to simplify the use of similar classes and removes a possible source of error. It also simplifies the maintenance of class libraries at the cost of some increased complexity at the design stage.

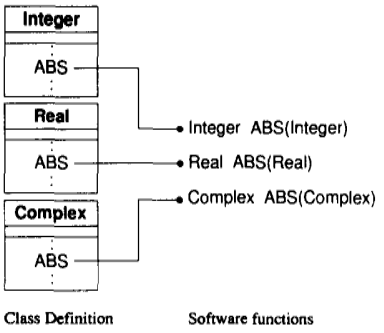

FIG. 11. Binding of object behaviour to software function.

In order to support polymorphism, the software system must provide some means of determining which version of a given function is correct for the arguments provided. This is illustrated in Fig. 11 for the ABS example described above. To determine the absolute value for a Complex Number z the user might program*:

```
Complex z = 0.5 + j0.5, M;
M = z.ABS( );
```

In order to execute this code the software system must know how to create objects of type Complex, how to initialize the complex data and how to respond to the messages = and ABS. To execute `M = z.ABS( )` the system has to decide that since both the argument `z` and the result `M` are Complex it should use the function with *signature*

$$\text{ABS} : \text{Complex} \rightarrow \text{Complex}$$

rather than any other function with the same name but different argument and return types.

The selection of a version of a function according to the signature of its calling and return arguments is called *binding*. Distinction is often made between software systems that provide *static binding* at the compilation stage or *dynamic binding* at run-time. There is also a further distinction that is made for languages with compile-time binding. If function calls must be made explicit at compile time the language provides early binding. If the binding of compiled functions to actual parameters can be deferred until run-time the language is said to support *late-binding*. In general, dynamic binding provides more flexibility than static binding since it facilitates the extension of a class library by its user. If data types must be statically defined, then late binding rather than early binding is to be preferred since it allows the class developer to provide the links that allow easy extension by the user of a class library.

*The programming notation used here is pseudo C++. The expression `z.ABS( )` means send the message ABS to the object z. The operator = can also be interpretated as sending the message = and the data on the right of the = sign to the object on the left. In both cases in this example the result is that the object on the left takes on new values.

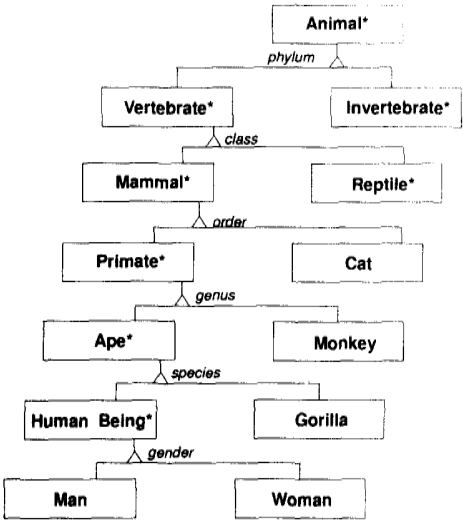

FIG. 12. Inheritance: the animal kingdom.

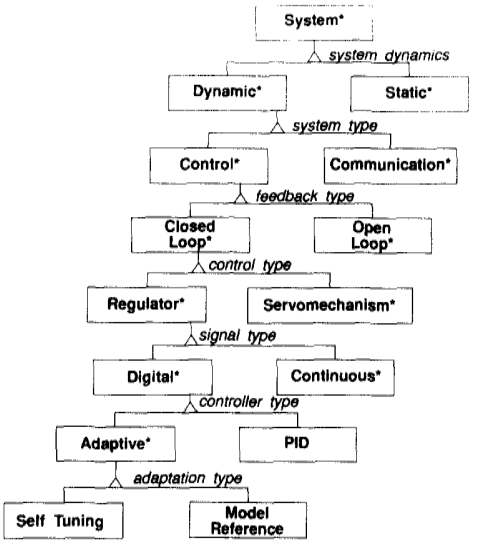

FIG. 13. The is a relationship of adaptive control systems.

2.2.5. *Inheritance.* The object-oriented paradigm presents one further important feature known as inheritance. This is a way of classifying groups of related objects in such a way as to reduce the amount of code which must be written to implement a software system and to encourage and facilitate the re-use of existing software components. In a manner analogous to the way in which zoologists classify the animal kingdom, OOP systems provide a means for arranging related objects in a hierarchy. The root and nodes of such a hierarchy belong to generalized, abstract entities like *animals* and *vertebrates*, and at lower levels more specialized properties are introduced such as *mammal, primate, human being*. At the leaves of the hierarchy are concrete entities such as *cat, gorilla, man* and *woman*. Figure 12 illustrates such a hierarchy using a combination of OMT (Rumbaugh *et al.*, 1991) and the naming convention of Meyer (1990). By Meyer's convention, elements with an asterisk following the name are *abstract classes*, that is classes which have no actual instances, while classes named with no asterisk are *concrete classes* which do have instances.

To illustrate the distinction, a vertebrate is not an identifiable animal but rather a common property of a group of some animals that is used in subclassifying the animal kingdom by *phylum*. In an object-oriented software implementation of a database, say, for classifying the animal kingdom, abstract classes like `Vertebrate` would be used to record those attributes that are common to all vertebrates, whilst concrete classes, such as **Cat** would be used for those animals that would actually have a data record, or object, in the database. In the OMT notation a small triangle is used to indicate that an inheritance relationship exists between classes and the text next to a triangle indicates the primary feature that distinguishes a class from its superclass.

In an inheritance hierarchy, each class inherits all the properties of its predecessor. Thus a **Cat** inherits the properties that distinguish the order **Mammal** from *order* **Reptile** and the *phylum* **Vertebrate** from the other *phyla*.† In OOP, a subclass may make direct use of its inherited properties or alternatively it may refine or ignore them. It may also add new properties of its own.

A good way of looking at inheritance (Rumbaugh, 1993) is to regard it as an abstraction of the notion "*is a*". Consider a possible classification scheme for control systems which is shown in Fig. 13. At the root of this hierarchy is the **System** class whose purpose is to encapsulate such notions as "a system has a boundary", "a system has inputs" and a "system has outputs". Two possible subclasses of **System** are **Static** and **Dynamic**; **Dynamic Systems** may be further subclassed by *system type* into systems for **Control** or **Communication** and so on. Thus, the bottom level item in Fig. 13 labelled **Model Reference** takes on all the properties of its ancestors in the inheritance hierarchy. In other words **Model Reference** *is an* **Adaptive Digital Regulator** which, also, *is a* **Closed-loop Control System**, which *is a* **System** and therefore also has a boundary, inputs, and outputs.

Apart from its useful role in classification, the primary purpose of inheritance in OOP is to provide behaviour abstraction and code and data sharing mechanism.

Figure 14 represents part of the Smalltalk class library for numbers (Goldberg and Robson, 1983; Pinson and Wiener, 1988). Code sharing in this class library is achieved as follows. **Magnitude** and **Number** are *abstract classes*, **Integer**, **Float**, **Fraction**, **Imaginary**, and **Complex**, are *concrete classes*. The abstract classes provide for the sharing of generic code, the concrete classes define the internal

†The use of inheritance in the class diagram shown in Fig. 12 should not be taken too literally. The reader should be aware that all "concrete" classes in the animal kingdom are defined at the level of *species*. Thus, strictly speaking, the class **Cat** is an abstract member of the subclass *family* (which is not shown) which itself is a subclass of *phylum* **Vertebrate**, *class* **Mammal** and *order* **Carnivore**. The *Tiger Panthera tigris* is a species, or concrete subclass of the **Cat** *family*.

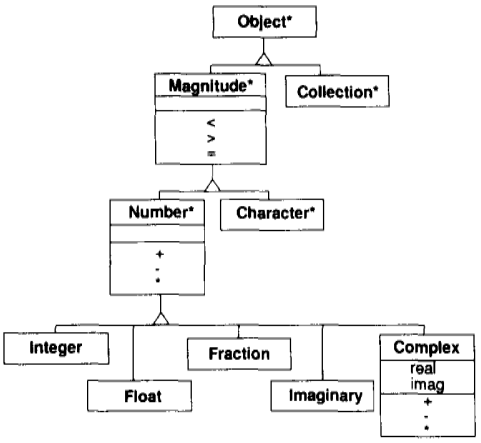

FIG. 14. Code sharing and abstraction in an inheritance hierarchy: numbers in the Smalltalk-80 class hierarchy.

representation of the numeric datatypes and code for specific datatype dependent operations.

The Magnitude class provides general abstractions for datatypes that can be compared in size. To operate, it requires only two basic methods. All other **Magnitude** methods, for example $\leq$, $>$, $\geq$, min, max are all defined in terms of these two basic methods. The comparison methods $<$ and $=$ must be defined by the subclasses of **Magnitude** since the code used to perform these comparisons will obviously depend on the actual types of object to be compared. However, the remaining comparison methods may safely be left to the **Magnitude** class. Thus, subclasses of **Magnitude** inherit all its methods, but must actually re-implement only two of them! The reduction in the new code that must be written for a new subclass of **Magnitude** is significant.

The **Number** class is provided to abstract the methods associated with the manipulation of numbers. It inherits the methods defined in the **Magnitude** class, which means that **Number** objects can be compared with each other, and adds arithmetical operations such as $+$ and $-$ as well as functions for mathematics, testing, truncation, conversion, coercion, creating intervals and printing. Concrete subclasses of the **Number** class do not need to re-implement those methods which are defined by their predecessors, but may need to specialize them or add new ones; for example, the **Imaginary** number class would need to redefine the generic version of multiplication. In order to implement a new **Number** class, such as the complex number class **Complex**, it is necessary to specialize only a few functions. The other functions are inherited from the abstractions, and thus less new code needs to be written (Pinson and Wiener, 1988, p. 201).

2.2.6. *Genericity.* Inheritance, polymorphism and data abstraction provide very powerful mechanisms for sharing code and behaviour between objects in an object-oriented system. However, there is yet another mechanism called *genericity*, or parameterization of classes, which can be used with typed object-oriented languages and systems. Consider a standard data structure like a matrix. A programmer could define a class **Matrix** containing **Integer** objects and then use inheritance to produce subclasses for Float and **Complex** objects. However, the functions used to implement the operations for a matrix of complex numbers would be largely the same as those used for a matrix of real numbers. It is therefore sensible in this case to create a *generic* class **Matrix** $\langle T \rangle$ where $\langle T \rangle$ is a *generic parameter* which indicates that the type of object which will actually populate the matrix is to be defined when a matrix object is created.

A generic class provides a static means by which code may be shared since at compilation time, actual class definitions will be generated with the parameter $\langle T \rangle$ replaced by the actual type everywhere it is used. Unfortunately, once a parameter has been expanded into a type, it is no longer possible to refine the class by inheritance, nor is it possible to change the type at run time. Meyer (1988) gives a detailed comparison between genericity and inheritance and examples of its use in the creation of a class library for the Eiffel language are given in Meyer (1987). A discussion of the application of genericity in C++, where it is provided by parameterized types, can be found in Stroustrup (1988a) and Koenig (1991).

2.2.7. *An example.* To illustrate the power of polymorphism, inheritance and genericity, consider the definition of a rational polynomial type **Rational Polynomial**. Assume that **Rational Polynomial** is a subclass of **Rational** which contains two **Polynomial** objects rather than the two **Integer** objects contained by its superclass. Further assume that **Rationals** and **Polynomials** know how to combine themselves with other **Rationals** and **Polynomials** by the application of the usual mathematical operators. A class diagram for **Rational Polynomial** is shown in Fig. 15. The open circle indicates that there may be zero or one denominator elements. Now assume that two instances of **Rational Polynomials**, P and Q, are created, and that they are to be added together to create a new **Rational Polynomial** S. That is

$$S = P + Q.$$

In order to perform this addition, the object-oriented system must first create the new **Rational**

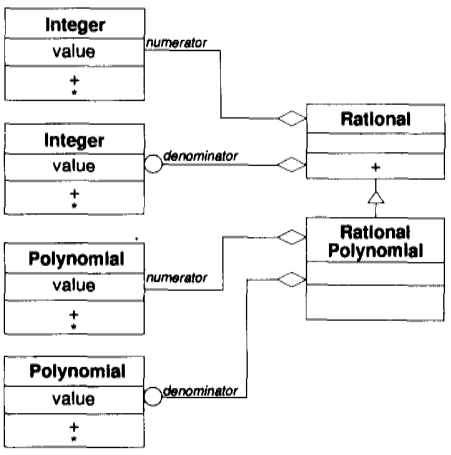

FIG. 15. The class diagram for a rational polynomial class.

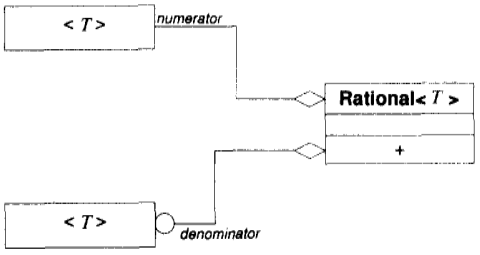

FIG. 16. The class diagram for a generic rational object.

Polynomial S then find a + function in the class definition for P so that Q can be added to P. Since there is no + operator in the rational polynomial class definition, by inheritance, the + operator in the class **Rational** would be used. This would tell P that addition of two **Rationals** x/y and w/z is $(x \times z + w \times y)/(x \times z)$, and polymorphism would be used to ensure that **Polynomial** multiplication and addition was invoked where appropriate.

If instead of inheritance, it was decided to use genericity to implement rational number types, the class diagram could be considerably simplified as shown in Fig. 16. Here there is a generic class **Rational** which contains a numerator and possibly a denominator of some other type $\langle T \rangle$. To create a rational polynomial object the user of this generic class would need to declare an object of class **Rational** ⟨Polynomial⟩. All the operations required to implement the summation of two rational polynomials would be implemented using calls to the corresponding **Polynomial** operations.

2.2.8. *Breaking the encapsulation rules.* In the discussion so far, we have argued that class definitions should be completely encapsulated and need make no reference to any other class except through inheritance. In truth this is rarely the case and this section describes one good reason for breaking the strict rules of encapsulation. Consider again the code fragment given in Section 2.2.4.

The modulus of a complex number $z = x + jy$ is given by $M = \sqrt{x^2 + y^2}$ which is a real number. Hence, the reader may have been surprised that the following code was not given for the example:

```
Complex z = 0.5 + j0.5;
Real M;
M = z.ABS( );
```

the signature of ABS in this example is

$$\text{ABS} : \text{Complex} \rightarrow \text{Real}$$

and therein lies the problem. In order to return a real value for $|z|$, the **Complex Number** object must have some knowledge of the **Real Number** object, but having this knowledge breaks the strict rules of encapsulation.

Another example may clarify the situation. It is quite common in mathematics to combine numbers of different types, for example:

$$z = 1 + (0.5 + j0.5).$$

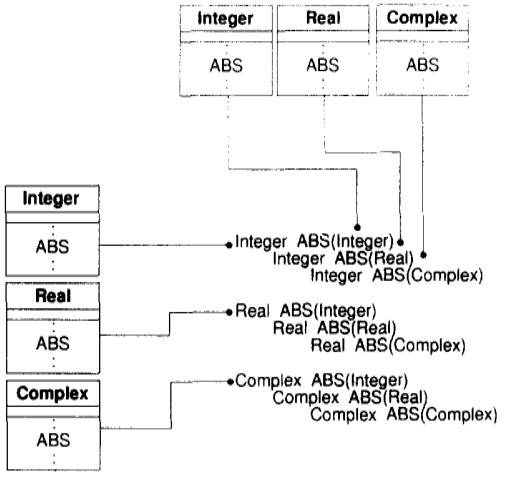

FIG. 17. Binding objects of different types.

In object-oriented languages such combinations are not quite so easy to achieve:

```
Complex c = 0.5 + j0.5;
Real x = 1;
Complex z = c + x;
```

in order to execute the expression z = c + x the object-oriented system must be able to provide an operator function **Complex + (Real)** so that the **Real** object x can be passed as data to the **Complex** object c. In general, in order to cope with all cases, there must be a signature defined for all possible combinations (Fig. 17). For three classes this means that up to nine different bindings for each operator and function must be programmed by the class developer and in general $O(n^2)$ different bindings for each operator and function for n classes.

There is no known way of overcoming this problem without breaking the strict rules of encapsulation, but it is possible to simplify cross-class computations by careful design of so-called "coercion" functions (Abelson *et al.*, 1985, p. 147). This is the method used in the Smalltalk class library, for example.

Coercion works as follows. First of all an ordering must be found that relates objects in the hierarchy. For this simple example this might appear as shown in Fig. 18. This essentially states the mathematical notion of specialization, that is, that integers are specializations of reals, and real numbers are specializations of complex numbers. In other words, an integer can be coerced into a real and a real into a complex without loss of information (but not vice versa). The reason for the ordering, which is not the

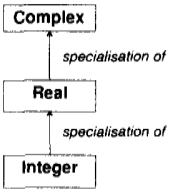

FIG. 18. Ordering of the Number classes for coercion.

TABLE 2. Coercions between simple datatypes needed to support mixed type operations

	Integer	Real	Complex
Integer	—	`toReal`	`toReal` `toComplex`
Real	Undefined	—	`toComplex`
Complex	Undefined	Undefined	—

same as inheritance, is to enable the programmer of the class library to specify how datatypes can be coerced into types suitable for combination. The coercion method typically used is to promote one or more data types in an expression until they may be combined using single class operators. In the example, x would be promoted to Complex ($z' = x + j0$) before being added to c. For the basic datatypes under discussion, the table of coercion functions needed are shown in Table 2. It is clear that some knowledge of the structure of the class-libraries is needed to provide the coercion functions, but at least that knowledge is kept strictly controlled. It should also be clear that operations between types take place once the types are coerced to be the same type so that operations do not have to be specified for each type combination. Some languages provide built-in support for coercion—smalltalk is one example; in other languages it must be provided by the class designer and supported by the class extender.

2.2.9. *Other language features.* Aside from the features already discussed, an object-oriented programming language or system may also offer support for:

- *Garbage collection*—some object-oriented systems provide automatic garbage collection whereby, at intervals, objects that are no longer being used are removed from computer memory. This frees the programmer from worrying about low-level memory management. Other systems require the programmer to create and destroy objects explicitly. In exploratory programming work, automatic garbage collection is preferable. In production code, particularly for time-critical or embedded applications, explicit memory management may be preferable.
- *Object persistence*—the ability to store an object onto permanent media and restore it later. This enables objects to "outlive" the programs that created them and is useful in many applications. The level of persistence provided by object-oriented systems can vary from no provision, through the simple writing and reading of objects to and from disk to full database support.
- *Concurrency*—since OOP encourages the development of decentralized software architectures, it seems a natural step to provide decentralized control too, by executing processes within an object-oriented system in parallel. Most current object-oriented systems do not provide support for parallel execution.
- *Exception handling*—the ability to be able to recover gracefully from run-time errors is a feature supported by only a few object-oriented languages and systems.

2.2.10. *Requirements of an object-oriented system.* In this section several important features that may be supported by an object-oriented programming language or system are considered. Some of these features are additional to those that are required for a language or system to be called object-oriented. These four required features have been detailed by Pascoe (1986) and are listed below:

- information hiding—data are only accessible within the scope of a module; private data can be changed without affecting other modules;
- data abstraction—the internal representation and procedures for data access and manipulation are defined in the class definition. An important by-product of this is that data types defined by classes become "first class" objects within the OOP system. This means that they effectively become part of the language and can be used like the built-in types which enables OOP systems to be extended by the user;
- late or dynamic binding—an object selects the appropriate method, or procedure, to perform a given task either during compilation or at run-time; in this way, objects can respond differently to the same commands and this in turn reduces the complexity of software systems (Booch, 1990); and
- inheritance—classes that specialize their super-class by inheriting all the methods and data from their predecessor (ancestor) and adding, deleting or refining the data and methods as appropriate; this is a means of achieving software re-use.

By making use of a language or system that supports the four required elements, it is possible to achieve significant improvements in productivity in the development of software, and improvements in quality and maintainability of the finished result. The support of some of the additional features may make the software development process even more productive. In the next section we consider the broad nature and supported features of several object-oriented programming languages and systems.

2.3. *Object-oriented languages and systems*

The object-oriented paradigm is delivered to programmers in two ways, either as a complete programming system or as a conventional language with object-oriented features. Both types of implementation have advantages and disadvantages. In this section a brief taxonomy of some of the available systems and languages is presented. Several standard texts on the subject of OOP expand on this material, in particular Pascoe (1986), Cox (1986), Meyer (1988) and Rumbaugh *et al.* (1991).

2.3.1. *Sketchpad* (c. 1963). The first object-oriented system was Sketchpad developed and described by Sutherland (1963). Sketchpad was an interactive graphics user interface, based on constraints, that enabled its user to build up objects in a hierarchical manner but it was not a programming system.

2.3.2. *Simula-67* (c. 1967). Simula, developed in the mid-1960s by Dahl and Nygaard (1966) at the University of Oslo, is a fully fledged object-oriented programming language which provides inheritance, classes, limited information hiding, virtual functions (dynamic binding), polymorphism and support for concurrency. Although originally developed for discrete event simulation, which may account for its lack of adoption for other applications, Simula-67 (Birtwhistle *et al.*, 1973), which is still available, may be used to develop other applications. It nevertheless has to be said that it is chiefly of interest today because of its influence on other object-oriented languages and systems, particularly Smalltalk and C++.

2.3.3. *Smalltalk* (c. 1972). Smalltalk is widely regarded as the parent of modern object-oriented sytems and as such, it is worth examining in a little detail. It was developed at Xerox PARC during the 1970s as part of research into personal computer systems led by Kay (1977), whose vision of personal computing was based on the concept of a "dynabook", a note-book sized personal computer with a graphical user interface that would provide a completely homogeneous computing environment. Many things that are now becoming standard features of computing systems, particularly graphical user interfaces, were first explored as part of this research. The computing environment that emerged to tie these things together was Smalltalk, which came to maturity in 1980 as Smalltalk-80. The language and environment are described by Goldberg and Robson (1983) and Goldberg (1985), good tutorial introductions are Robson (1981) and Xerox (1981), and Ingalls (1981) describes the design principles behind the system.

The Smalltalk developers borrowed the graphical facilities first demonstrated in Sketchpad and the language structures of Simula to create a completely seamless, perfectly object-oriented computing environment. From Smalltalk and Simula comes much of the terminology now used in object-oriented computing: the idea of an object as an encapsulation of data and procedures, the idea of class and class hierarchies, inheritance, messages and methods. In Smalltalk everything, from the primitive data types, through the classes, to the graphical tools that serve the environment, is an Object. Computation is done by the objects sending messages to other objects which either process the messages themselves or pass them on to other objects, or to objects further up the inheritance tree.

The environment is completely homogeneous: the user can browse the functions of any object in the system and modify the environment at will. This makes Smalltalk a superb environment for prototyping applications. However, the computation model and the sheer scale of the environment is one of its greatest weaknesses. It takes an experienced programmer a considerable amount of time to learn and become familiar with the system (O'Shea *et al.*, 1986), although there is evidence that novices with no preconceptions pick up the ideas rather more quickly (Kay, 1977). Love (1987), who used Smalltalk to develop applications for oil exploration, has described the following strengths and weakness of the system:

Strengths:
- productivity is enhanced through access to the large collection of reusable classes and methods, and by inheritance. Programmers write tens of lines of code rather than thousands;
- very good user interface;
- totally integrated environment, more accessible;
- good for graphics applications;
- large quantity of good code is accessible, modifiable and reusable; and
- extensible, applicable to scientists and others interested in modelling.

Weaknesses:
- no facilities for group working such as communications, co-ordination, version control, configuration management; it is a single user system;
- no file system or utilities for file management, file transfer or manipulation;
- low computation speed. Smalltalk is based on an interpreter;
- lack of training materials;
- small user population;
- expensive hardware, although this is perhaps less important given current workstation prices; and
- single programming language, cannot re-use existing code.

Diederich and Milton (1987) also highly recommend the Smalltalk system for what they call "fearless programming" which is important for rapid prototyping. They state that the investment in learning the system is more than repaid by the powerful facilities it provides for experimenting with algorithms, rapid prototyping and graphical interfaces. However, developers of software must be cautious for two reasons: they must invest the large amount of effort required to learn the system, and when this investment has been made, they must accept the fact that users of their software must also learn the Smalltalk system to use it. Unfortunately, the real world inhabited by most users of software for which Smalltalk-80 would be the ideal prototyping environment, is one of stand-alone applications with operating systems and interfaces that may not even support graphics.

2.3.4. *LISP-based object-oriented systems* (c. 1980). In many ways, the facilities offered by the many LISP based implementations of OOP systems mirror those of Smalltalk. Judging by the number of systems available, it is obvious that LISP provides a good basis for object-oriented systems. The implementations range from pure language extensions like Flavors (Cannon, 1980; Moon, 1986), Object LISP (Drescher, 1985), Common LOOPS (Bobrow *et al.*, 1987b) and the Common Lisp Object System (DeMichiel and Gabriel, 1987; Bobrow *et al.*, 1988) to fully fledged graphical programming environments like LOOPS (Bobrow and Stefik, 1982) and KEE (Fykes and Kehler, 1985). A good tutorial introduction to the LISP "flavour" of OOP is provided in

Stefik and Bobrow (1986). The principal difference between these systems and Smalltalk-80 is in the provision of multiple inheritance where an object's properties are controlled by an acyclic inheritance mesh. This can reduce the number of classes needed to model a system at the cost of increased complexity in the semantic interpretation of the inheritance relationships between classes.

2.3.5. *Evolutionary approaches to object-oriented programming* (*c.* 1985). The use of Smalltalk and the LISP based environments must be viewed in the context of current practice. Such environments make a key assumption about the way computation should be done, not only in the programming sense but also from the user's point of view. There is no distinction between the programming environment and the application environment. It is therefore difficult, if not impossible, to develop applications that can run in isolation from the programming environment. Applications may be easier to create in such systems, but they are probably only easy to use if the user is familiar with the environment. There are of course other implications, not least of which is the protection of propriety rights in software. This is rather difficult to guarantee or justify in an environment that is otherwise completely open. In short, the justification for the use of such systems in the development of new CACSD tools can only be made if there is a corresponding revolutionary change in the way that computers, operating systems and application software are viewed and used by users.

In order to overcome this dilemma, Cox (1985, 1986) has advocated an evolutionary approach to the adoption of the OOP method. His approach is to add object-oriented features to conventional languages so that investment in existing code is protected, production code can be developed which is independent of any environment, and programmers can make a gentle transition to the more powerful software-engineering implications of OOP. In addition, he advocates the invention of a new culture in software which has long been a feature of hardware designs, namely the use of "integrated circuits" in the development of systems. Such "Software-IC's" (Ledbetter and Cox, 1985) would be binary code objects for which the manufacturer publishes the class description and protocol and which software developers could buy and use in exactly the same way that electronic engineers use the packaged functionality of hardware ICs. The first object-oriented language actively to support this idea is Objective-C (Cox, 1986).

The Objective-C language implements encapsulation, inheritance and polymorphism by means of Smalltalk-like object-oriented extensions to C (Lozinski, 1991). Objective-C does not have compile-time type checking of classes, requiring instead only that all objects in an application are able to respond to all messages that are to be sent to them. Providing the protocol is correct, any class can communicate with any other and additional flexibility is obtained by allowing run-time message definition. Inheritance can be used to refine any class, even built-in classes, because only the class interface definition is needed. Objective-C is a "hybrid language" in that the object-oriented extensions provided are not part of the base language, which is C, but are "compiled" by a pre-processor into the base language. This provides for evolution since programmers can gradually migrate from mainly C to mainly OOP, although it could be argued that the only reason for using Objective-C would be for its object-oriented extensions. A foundation set of Software-IC's, which include collection classes, are provided with the commercial Objective-C compiler. There are also Software-ICs available for graphics and other applications which are sold separately.

A slightly different evolutionary approach to OOP is provided by C++ developed by Stroustrup (1986) as a successor to the C-language. This language can also be regarded as a hybrid in that it supports conventional programming (in C) as well as OOP, and some "compilers" for the language are preprocessors for C. The earliest version of C++ was an attempt to add the features found in Simula to the C language. However, these additions were not complete and neither, because of the idiosyncrasies of the C language itself, were they particularly elegantly implemented.

Like Objective-C, C++ allows programmers to make a gentle transition to OOP by being completely upwardly compatible with C but at the same time, provides classes and inheritance as well as stronger type checking particularly at function interfaces. However, it can also be regarded as a true language in that it can be compiled directly into machine code.

C++ has no standard class library which is a severe disadvantage. A further disadvantage of C++ is that inheritance does not provide automatic support for subclass refinement. This means that developers of classes that may be refined by others must make sure that methods that may be overridden are declared to be virtual. If this is not done then the superclass must be modified which will not be possible if only a library version of a class is available. Thus re-use may be compromised. Neither is it possible to construct messages or refine classes at run-time which makes it difficult to use C++ as a language for applications that may be extended by the user. If such extension is required, the software developer must invest considerable resources in developing an infrastructure which will support it.

In some senses, C++ remains too low-level. For example, the programmer needs to be aware of which objects are static, which are stack-based and which are created in freestore. As not all extensions to the language are object-oriented, it has been suggested that the language is harder to learn than Objective-C (Lozinski, 1991). However, the extensions such as strong typing, polymorphism, classes, genericity and the data hiding provided by the controlled access to class variables and functions make the language far superior to C.

Another hybrid approach is provided in programming environments like Think-C, which runs on Macintosh computers and Turbo-Pascal which runs on

DOS-based PC systems. Such environments provide object-oriented extensions to a base language and often support graphical user interface development on a single architecture. The productivity of the programmer is increased by the close integration of the software development tools.

A disadvantage with evolutionary languages is that they do not *force* the programmer to use the object-oriented paradigm. The best use can only be made of the object-oriented features of these languages if a completely new way of thinking about program construction is adopted. By building on conventional languages, the designers of these languages may have made their offerings more attractive to programmers yet at the same time failed to push the state-of-the-art of programming forward. Only time will tell if programmers actually use the object-oriented features of these evolutionary languages.

2.3.6. *Pure object-oriented languages* (c. 1988). The best compromise between the environment-based languages such as Smalltalk and Lisp dialects and hybrid extensions or upgrades to conventional languages such as Objective-C and C++ are provided by new languages which have been designed as pure object-oriented languages. Such languages do not have the excess baggage required to preserve compatibility with existing conventional languages but at the same time are not tied to a revolutionary programming system. Two examples of such new languages are Eiffel, developed by Bertrand Meyer (Meyer, 1991) and Trellis (Kilian, 1990), developed by the Digital Equipment Corporation.

Eiffel is probably the best production code generating object-oriented language available to date. It has the following key features:

- it is a strongly typed object-oriented language;
- programs consist of collections of class declarations that include attributes and methods; and
- it supports multiple inheritance, genericity, memory management, exception handling and assertions.

Eiffel uses C as a portable assembly language and comes complete with a modest class library and software development support tools. The key concept in Eiffel is the class declaration which lists the attributes and methods of a class. Support for refinement of classes is provided.

An interesting feature of Eiffel is the support for a contractual model of programming which is provided by pre- and post-conditions, invariants and exceptions. A *pre-condition* is a condition that the caller of an operation agrees to satisfy. A *post-condition* is one that the operation itself agrees to satisfy. An *invariant* is a steady-state condition of a class that must always be achieved. If these conditions are violated an exception is raised. These features are very important for the development of reliable software libraries that are to be re-used.

Trellis is an object-oriented language and development system that supports genericity, multiple inheritance, concurrency and clean interfaces to data-types and procedures defined in conventional languages. A compromise between Smalltalk and a language-based system, Trellis provides an extensive program development environment including browsers, debuggers and an incremental compiler. When programs are complete, the development support tools are removed from the environment to leave a compact workspace which can be used like a conventional executable image.

2.3.7. *Ada.* It is worth mentioning the pretensions of Ada to be an object-oriented language. Although this claim is supported by articles contained in collections such as Peterson (1987a), the case is readily dismissed by consideration of the required elements of Pascoe (1986) which are information hiding, data abstraction, dynamic binding and inheritance. The main structuring facility in Ada is the *package*, which provides information hiding and data abstraction. However, an Ada package differs from an object-oriented class in that a package encapsulates a type but is not the type itself (Korson and McGregor, 1990). This important difference means that the connection between the state and behaviour of an Ada "object" is weak, and imposes the syntactic burden of an additional parameter to most of the package's procedures. More serious, however, is the fact that Ada does not provide dynamic binding and inheritance and therefore cannot be regarded as an OOP system. This is not to say that object-oriented methods cannot be applied to Ada programs in the early stages of software development—indeed Booch (1986) has demonstrated that object-oriented design can usefully be applied to Ada and other similar modular languages (such as Modula-2). It is rather that the incomplete support for all the features of the object-oriented paradigm makes it hard to turn object-oriented designs into Ada software. As Stroustrup (1988b) puts it:

> "A language supports a programming style if it provides facilities that make it convenient (reasonably easy, safe and efficient) to use that style. A language does not support a technique if it takes exceptional effort or skill to write such programs; in that case the language merely enables programmers to use the technique. For example you can write structured programs in FORTRAN...but it is unnecessarily hard to do so because (the language does not) support (that) technique."

It is worth noting that extensions are being considered by the Ada 9X working group that will make Ada more useful for OOP (Ada, 1988).

2.3.8. *Programmer support tools.* An unfortunate feature of the language-based OOP systems is that they are not generally supported by the programming aids that are a feature of systems like Smalltalk. Particularly, interpreters for fast testing of new code, system and class browsers, and integrated editor-compilation and configuration management systems are missing from the basic language. It is perhaps ironic that such tools feature more in programming systems for personal computers, through development environments such as Borland C++ and THINK-C which are available for IBM-PC and Macintosh, respectively, than for high performance workstations. The programming environments that are available for

workstations, such as Object Center, are usually an order of magnitude more expensive than their PC equivalents, and therefore much of the software development is still done using the conventional edit-compile-run cycle. Perhaps the standardization efforts going on in the workstation arena based on operating systems and windowing technology will encourage the wider availability of such tools in the future.

2.3.9. *Summary.* From the point of view of a programmer who wishes to develop applications that will run on conventional computing hardware, the question of whether a language or a system should be used for the development is fairly easy to answer. Smalltalk and LISP based-systems are good for prototyping but not so good for production code development, because it is sometimes difficult to separate an application from the environment. Trellis, by providing prototyping and a run-time separation of application from environment, might be a good choice except that it is proprietary. Therefore, the only real choice is between one of the compiled languages. Technically, Eiffel and Simula would seem to be the best languages to use, except that the only current implementations are proprietary. The choice then comes down to Objective-C versus C++. Given the pedigree of C++, the wide availability of compilers and the wide industrial support is has gained, it is likely to become the dominant language despite its clear technical inferiority.

2.4. *Software reuse*

The effective re-use of software is a problem that has vexed software engineers for some time. It is McIllroy (1976) who is generally credited with raising the question "why isn't software more like hardware?" in that software components are usually re-implemented rather than ordered from a catalogue. There are a number of non-technical reasons why software is not re-used more often. Meyer (1988, p. 28) gives some of them, but before they may be addressed there are even more serious technical hurdles to re-use that must be overcome first. These technical issues are to do with the way programming languages provide support for the five requirements for reusable software components:

(1) a reusable software component must be able to deal with variation in types;
(2) a reusable component must support variations in data structures and algorithms;
(3) a reusable component must provide related routines for manipulating data contained in the component;
(4) a reusable component must be representation independent; and
(5) there must be support for commonality between groups of implementations.

The most common form of software re-use, that of functional re-use, only works for functions that can be regarded as black boxes and that do not deal with complex data types. This is one reason why engineering applications that make extensive use of matrix data structures are so well served by reusable libraries. Other areas of engineering are not so fortunate. Even MATLAB, a package with great scope for functional re-use that has proved successful as a CACSD application, has difficulty providing clean functions for dealing with the more complex data objects that are found in control systems anlysis. Consider multivariable frequency response analysis and design which is one example application area. This is provided for MATLAB by the Multivariable Toolbox (Boyle *et al.*, 1988) in which many of the functions exist merely to construct and de-construct the data types. These functions are not encapsulated within some multivariable frequency response data type and so functional re-use can be seen to violate issue 3.

Another re-use mechanism found in Ada and other modular languages is the package. These languages address issue 3 by keeping all functions that are to do with a particular data type in one place. However, they fail to deal with issues 1, 2, 4 and 5.

Operator overloading and genericity are yet further aids to re-use that are provided by Ada. Operator overloading is a less general form of polymorphism [called *ad hoc* polymorphism (Cardelli and Wegner, 1985)] by which functions implementing a particular operation are bound at compilation time by static binding. This means that the same named function may be used with different data types but the decision about what operation is actually used cannot be made at run-time. Genericity, described in Section 2.2.6, provides only for static, compile-time, sharing of code.

Object-oriented programming, on the other hand, can solve most of the issues listed above by capturing the commonality between groups of related data structure implementations through inheritance and polymorphism, and protecting users from internal details by use of abstract data types and information hiding. Articles on just how well object-oriented methods do actually solve the technical problems of re-use can be found in Meyer (1987, 1990), Lazerev (1991) and Al-Haddad *et al.* (1991). The more general issues of re-use are discussed in Wegner (1984), Ledbetter and Cox (1985) and Biggerstaff and Perlis (1989a, b).

2.4.1. *Re-use through inheritance.* One of the ways that the potential benefits of re-use might be achieved through object-oriented programming is by the use of class-libraries. There are two ways that this can be achieved, either by using useful classes directly or by specializing existing classes by use of inheritance. In either case the savings in code development are potentially considerable.

It should be apparent that providing such a class library greatly increases the power of an object-oriented language. It may readily be extended as demonstrated by Pinson and Wiener (1988), who describe the addition of the Imaginary and Complex classes to the Smalltalk-80 hierarchy. The extension requires less actual code than would be the case with, say, FORTRAN, since much of it actually already exists and is already shared in the class hierarchy.

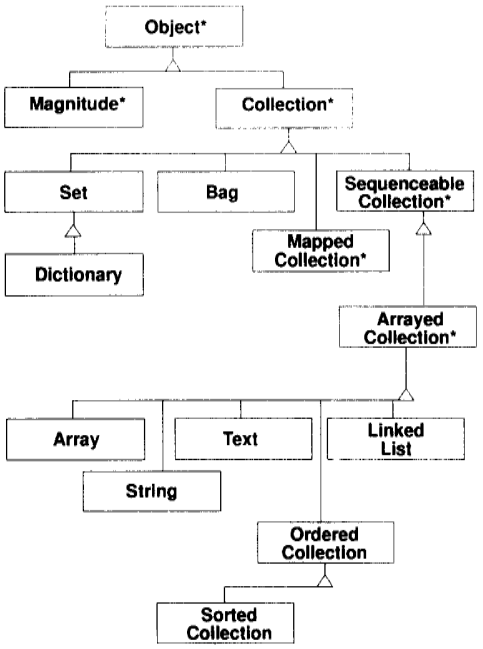

FIG. 19. Part of the Smalltalk collection class.

2.4.2. *Support for associations and aggregations.* An important group of reusable classes that all object-oriented systems should support, but not all do, are the so-called container classes. These are of particular significance to programmers because they provide convenient ways to implement aggregations of objects and associations between objects. In Smalltalk, a wide range of container classes are provided. These are arranged as shown in Fig. 19. Note that this hierarchy contains many of the collections that are regularly required in computing: sets, arrays, indexed arrays, associative arrays (dictionaries), and linked lists. Many of these classes can hold any type of object and are therefore highly reusable. The methods associated with container classes include functions for accessing, adding, deleting, sorting and iterating over objects in the collections.

Container classes in other languages can either be completely general purpose, in that any object may be placed in any container at run-time, or generic in that the type of objects that a container may contain is specified at compile time. Of the object-oriented languages discussed here, only C++ provides no standard set of container classes.

2.4.3. *Class libraries.* By providing a set of container class and a set of basic abstract data types, any object-oriented language or system provides many more opportunities for code re-use. Of the production systems considered in Sections 2.3.5 and 2.3.6, Objective-C, Trellis and Eiffel all provide a basic class library. The suppliers of Objective-C have also tried to market the idea of software components bought "off the shelf" by offering additional sets of "Software-ICs".

Eiffel provides a basic class library but also extends the possibilities for re-use by providing genericity. Genericity, in the form of parameterized types has

```
Object
    Class
    Collection
        Arraychar
        ArrayOb
        Bag
        SeqCltn
            Heap
            LinkedList
            OrderedClt
                KeySort
            Stack
        Set
            Dictionary
                IdentDic
            IdentSet
    Date
    Float
```

FIG. 20. Part of the NIHCL class library.

also been added to C++ from version 3.0. An excellent article by Meyer (1990) gives great insight into the development of reusable data-types through the experiences gained in developing the Eiffel class library.

A particular weakness of C++ is the lack of a standard pre-defined class library. Although the language comes complete with class definitions for the complex number type, streams for input and output and some classes useful for simulating concurrent process, the range of predefined reusable classes comes nowhere near the scope offered by rival languages. It is fortunate therefore that third parties have created freely available class-libraries for use by C++ programmers. Examples of such libraries are National Institutes of Health Class Library (NIHCL), developed by Gorlen (1987) and Gorlen *et al.* (1990), the GNU C++ Library (libg++) (Lea, 1991) and the Texas Instruments COOL library (COOL, 1990). These class libraries provide many of the more useful classes for software development either as a single inheritance tree descending from a single root like that of Smalltalk (NIHCL), or as a forest of classes (COOL and libg++) (see Figs 20–22).

The tree type of class library makes maximum re-use of common features between classes, resulting in smaller individual classes, at the cost of making it more difficult to understand a given class due to its relationship with other classes. The forest type of class library provides loosely coupled, independent classes

```
Pair<KeyType, ValType>
Range
Rational
Complex
Bignum
Generic
    String
    Collection
    Regexp
    Vector
        Vector<Type>
            Association<KeyType, ValType>
    List
```

FIG. 21. Part of the COOL class library.

String
Integer
Rational
Complex
Random
List
LinkedList
  DoublyLinkedList
Vector
Stack
Set
Bag

FIG. 22. Part of the GNU C + + class library.

at the cost of having possibly large individual classes and failing to exploit commonality resulting in similar code being duplicated in separate classes.

5. *Object-oriented software design*

In order to make the best use of the object-oriented paradigm in the development of software it is necessary to revise the ways by which software is specified, implemented and supported througout its lifetime.

There are several semi-formal methods of capturing the requirements for a new software system (analysis) and identifying the major components (design) prior to implementation in a programming language. Davis (1988) surveys the methods used in the analysis and design stages. During the design stage, the design moves from a specification of what the system should do, to how the software will do it. This is a decomposition process which adds progressively more and more detail until it is judged that sufficient detail is available for coding to begin. There are a number of methods available for performing this decomposition but the most traditional is probably top–down functional decomposition (DeMarco, 1987; Constantine and Yourdon, 1979). The approach is essentially to identify the functions and flows of data in the system; the data are then determined from the functions. The disadvantage is that interesting data tend to end up being global and so the system becomes hard to modify later (Booch, 1986). A variation of functional decomposition is provided by Jackson Structured Design (Jackson, 1975, 1983; Cameron, 1986) in which some of the important data structures are identified first and then used to identify the procedures. This is a kind of "middle-out" approach to software design.

With the emergence of the OOP, a new set of design techniques, based on object-oriented analysis and design (Booch, 1990; Coad and Yourdon, 1990; Wirfs-Brock and Wilkerson, 1990), have evolved. Such methods de-emphasize the functions in a system, preferring instead to encapsulate the data and procedures into a set of object classes. The object model supports the complete software development life-cycle. In the analysis phase, objects are identified which often serve as a model of the real-world application. This ensures that a well-organized statement of the problem, written in the terminology of the problem domain, is actually built into the application. During the refinement of the definitions and the implementation of the application entities, other entities, or classes, are identified. These form the major software components and may be built into the system from the bottom up and re-used in future projects. Clearly, object-oriented design methods are ideal for, but not restricted to, the development of systems which will eventually be implemented in an object-oriented language. However, of more significance is the possibility of unifying the stages of the software development life-cycle by having a common language throughout which should increase the productivity of software developers.

Object-oriented analysis and object-oriented design, like the object-oriented paradigm itself, are relatively new concepts and no single methodology has yet emerged. Good introductions to this topic are provided by Korson and McGregor (1990) and Henderson-Sellers and Edwards (1990). Current research is reviewed by Wirfs-Brock and Johnson (1990) and the special problems encountered in designing software classes for re-use as software components are discussed in Meyer (1990) and Gibbs *et al.* (1990). Many of the object-oriented analysis and design techniques which have been proposed include some form of graphical "language" to help to conceptualize the design as it progresses. Examples of diagrammatic aids which are central to the corresponding design methods are described by Booch (1990), Coad and Yourdon (1990) and Ackroyd and Daum (1991) and many of these are synthesized in a graphical language proposed by Edwards and Henderson-Sellars (1993). A recent comparison of methodologies used in the object-oriented analysis of software systems is given in de Chanpeaux and Faure (1992). The authors of this paper have found the Object-Modelling Technique (OMT) notation developed by Rumbaugh *et al.* (1991) to be simple to understand, concise, and elegant.

2.6. *Summary*

This part of the paper has provided a review of the object-oriented paradigm, and presented the terminology and the basic features of OOP systems in a way that will appeal to readers with an engineering background and with the minimum of jargon. Inheritance and the structuring of program data have been described and shown to be the key to the extensible and reusable class libraries which are a principle feature of the paradigm.

Given the current state-of-the-art in operating systems and computing facilities, it is likely that compiled OOP languages will become dominant. This is despite the technical superiority of pure OOP environments like Smalltalk which have suffered in the past by providing poor support for applications that have to be run independently of the development environment. Of the compiled languages, it is also expected that C + + will become dominant (Jacobson, 1993). Recent literature has been reviewed on the use of object-oriented techniques for software system analysis and design, which taken with an OOP language provide a powerful unified framework for software development. It is likely that software

support for object-oriented software analysis, design and development will develop rapidly over the next few years.

In Section 3, it is shown that the object-oriented paradigm is especially suitable for the development of future computer-aided control system design (CACSD) environments. In addition, object-oriented methods for graphics-based applications and database design will be reviewed.

3. COMPUTER-AIDED CONTROL SYSTEM DESIGN

3.1. *Introduction*

In a control system, the plant, sensors, actuators and controllers are physical artefacts. They are objects that can be touched; their behaviour can be observed by use of instrumentation and can be manipulated by adjustments of knobs and switches.

A good engineering model of a control system should mimic the behaviour of the real system as closely as the modelling medium allows. This means that the model should have a similar appearance, behaviour and responsiveness. In order to provide such a model, a computer-aided control system design (CACSD) environment must provide:

- powerful modelling paradigms in order to ensure that the physical behaviour of the system can be correctly identified and represented;
- sophisticated graphics to ensure that the model "looks" and "feels" right;
- excellent data handling which serves as the necessary basis of the instrumentation of the model;
- powerful manipulative software so that the model can be simplified for analysis and design; and
- software support for the design process itself in order to correctly identify versions of the model with the results generated and design decisions made.

Modern control systems are exceedingly complex. In order to achieve maximum efficiency and lessen the environmental impact of a process, many variables must be optimized and trade-offs balanced. The solution of the resulting design problem is often beyond the ability of a single human being. Computer aids are essential weapons in the control engineer's armoury in the battle against complexity and the taming of uncertainty. In a perfect partnership, the engineer would use his inventiveness and intuition to overcome the uncertainty while the machine handles the complexity (MacFarlane *et al.*, 1989).

The promise of interactive CACSD systems, defined by Rimvall (1987) as:

> "the use of digital computers as a primary tool during the modelling, identification, analysis and design phases of control engineering,"

has so far failed to be realized. As MacFarlane *et al.* (1989) point out, a designer should be able to use the computer to help him to build; analyze; browse; search; compare and evaluate; reason and hypothesize; synthesize; design; manipulate and modify; experiment; catalogue, store and retrieve. This is a broad list of requirements which is hard to deliver in a single package. Consequently, control engineers are poorly served by the current generation of CACSD packages, which are overly complex, cumbersome and error prone to use, poorly integrated and yet still only deliver part of the menu. Part of the reason for this is that the abstractions supported in the packages are too close to the procedural operation of the underlying computing hardware. They are not powerful enough to provide the cohesion and level of integration needed for the ideal CACSD environment. The object-oriented paradigm provides the models of computation which are most likely to overcome these problems in the short term.

In the first part of this paper the object-oriented paradigm was presented and the benefits for the software engineer were described. In this part of the paper the case for object-oriented programming (OOP) in CACSD is presented. Firstly, this will be done from the perspective of the control engineer. It will be shown that objects are a much more natural basis for good CACSD than the simple data types and functions currently used. Secondly, the software technologies needed to implement object-oriented CACSD tools will be discussed. In addition to OOP itself, there are architectural issues, graphical user interfaces, model kernel and database management facilities to address. The uses already made of the object-oriented paradigm by workers in the field of CACSD are then reviewed. Finally, some areas for further research in the application of OOP to CACSD are suggested.

3.2. *A user's view of object-oriented CACSD*

The end-user is the most important element of any computing system. All too often, in the rush to build the great new algorithm, design technique or package, the user is forgotten. Certainly, in the development of CACSD, the user interface has often taken second place. Even the state-of-the-art matrix environments that have largely taken over CACSD (Rimvall, 1987) have user interfaces that are difficult to use, particularly for novice or inexperienced users. As the quest to dominate the market drives developers to add more features and further complicate their CACSD packages, the users are in danger of being swamped by "creeping featurism" a term coined by Larry Wall (Wall and Schwartz, 1990) to mean the incessant tendency of software developers to "add just one more feature" to their products.

The requirements of interactive CACSD are undoubtedly wide—in fact they are much wider than the current generation of tools can support. They are so wide that many researchers doubt the ability of a single software package to support them all (Taylor and Frederick, 1984). The authors believe that a change of paradigm is required if the potential of computer-aided design is to be fully realized. The old paradigm is firmly lodged in the functional model of FORTRAN. The data encapsulated, message-passing, object-oriented paradigm is the one most likely to succeed.

Recognizing that the user should be the chief beneficiary of this paradigm shift, the benefits of the

object-oriented view of CACSD is presented in this section. In order to structure this discussion, the term "usability" is first defined. Then the key concepts underlying the object-model are presented in the context of an end-user's experience of CACSD software.

3.2.1. *Usability*. The state-of-the-art in CACSD is the matrix environment that has evolved from MATLAB (Moler, 1980). These are popular because (Rimvall, 1988):

- matrix environments support a simple and yet fast and flexible command language interface;
- the command language is interactively extendible; and
- the basic data structure (the complex matrix) corresponds to the primary data structure used in modern state-space control theory.

Having recognized the potential of matrix environments for CACSD, the trend has been for developers to add algorithms for a large range of design methods (horizontal integration) and to further extend packages by the addition of tools for systems modelling, nonlinear systems simulation and identification (vertical integration). The advantages of this for the control engineer are:

- he can satisfy all his computing needs and yet needs to learn only one package;
- no transfers of data between different, incompatible packages are necessary; and
- results derived from different design methods can be directly compared.

The drawbacks are that there is poor support for data structures, such as transfer functions, nonlinear system models and signals, which are not easily manipulated as matrices. Further, as a result of the continued extension, the packages become very large and unwieldy to use.

To counter these disadvantages, Rimvall (1988) has suggested that the package and, in particular, its user interface, has to be *uniform* and *consistent* throughout the domains of its use. Furthermore, the user has to be given on-line *informative support* throughout the use of package. Rimvall then goes on to identify and describe the following kinds of uniformity and consistency: *operational uniformity*; *functional uniformity*; *uniformity over size*; *representational uniformity*; *data consistency*; *documentary uniformity and uniformity of user proficiency*. Rimvall's criteria are used in the next section to evaluate an object-oriented model of CACSD and to demonstrate that OOCACSD has the potential of satisfying all the criteria.

3.2.1.1. *Objects*. Objects provide the *representational uniformity* advocated by Rimvall (1988). However, they also are responsible for providing *operational uniformity* and *data consistency*. In order to achieve representational uniformity, any CACSD package must provide a set of data structures adequate to describe the "objects" found in control systems design. It is relatively straightforward to decide what these objects ought to be in CACSD (see also Section 3.5.2). In very broad terms, the fundamental objects are systems and results. A system could be:

- an *object-diagram* illustrating the interconnection of subsystems;
- a *block diagram* illustrating the cause and effect relationships between components;
- a *subsystem* such as a plant model, actuator model, sensor model or controller model;
- a *model* in the form of a set of nonlinear differential equations;
- a *model* in the form of a set of linear state equations;
- a *model* in the form of a transfer function; and
- a *model* of an open- or closed-loop control system; etc.

Results could be:

- *time and frequency responses*;
- *poles and zeros*;
- *singular value plots*; and
- *frequency spectra*; etc.

A *result* can also be a *system*—for example, a linear state-space model may be the result of the linearization of a nonlinear model.

Many of these objects are provided in conventional CACSD packages. A small collection of packages, say, for example, a linear analysis package, a nonlinear systems simulation package and a graphical block-diagram editor could provide them all. So it is necessary to answer the question "what is the advantage of object-oriented CACSD (OOCACSD) over conventional CACSD"? The answer is the encapsulation of state with behaviour that an object-oriented system provides.

Most conventional CACSD packages have a FORTRAN heritage. This means that data and functions are strictly separated. Typically, computation is performed by functions operating on data to produce new data. Such software is a very close reflection of what is actually happening at the machine level. In an OOCACSD package, because data and behaviour are not separate, any object "knows" how to perform a given function on itself to generate a result. The result might be a change of state, the creation of a new object, or the execution of another function on itself or another object. Any function that is unknown to the object will fail to execute, reducing the possibility of error later in the computation. In effect, there is a strictly defined protocol, or *contract of behaviour* for every object which ensures *operational uniformity*. The danger of misusing a function by calling it with the wrong kind of data is greatly reduced, and the safety and robustness of the CACSD package is correspondingly enhanced.

3.2.1.2. *Polymorphism*. Polymorphism is a mechanism for reducing the number of commands that the user of an OOCACSD package needs to remember. It is used to create functional uniformity, that is, a consistent set of synonyms for performing functionally equivalent yet algorithmically different operations on different data (Rimvall, 1988).

TABLE 3. Operator overloading: some uses of the + operator in CACSD

Addition of scalars	$a + b$
Addition of vectors	$\mathbf{a} + \mathbf{b}$
Addition of polynomials	$a(s) + b(s)$
Addition of complex numbers	$z + w$
Superposition of time responses	$g(t) + h(t)$
Parallel combination of linear blocks	$G_1(s) + G_2(s)$
Multiplication of log-frequency responses	$\log_e(G_1(j\omega)) + \log_e(G_2(j\omega))$
Concatenation of two strings	`''hello'' + ''world''`

The simplest example of polymorphism is *operator overloading* which is such a natural concept that mathematicians and control engineers use it all the time without noticing.

Consider the addition operator +. In a CACSD package, this operator could be used to indicate a range of operations as shown in Table 3.

When engineers and mathematicians write down such expressions, they give hardly a thought to the special conditions that must hold for each of these operations to be valid or what must be done to perform the addition. The fact that it is well known that the operation is defined between these objects makes these considerations implicit.

In software systems, such rules must be defined explicitly. A function must be defined for each operator supported by each data type and it must be made explicit how operators and functions between different data types are to be interpreted and handled. In many programming languages, such functions have to be given unique names. Even so, it has been possible for the compiler or interpreter to recognize certain combinations of types and to automatically generate the required code. Thus a FORTRAN compiler recognizes scalar multiplication between the primitive data types INTEGER, REAL and COMPLEX and requires only one operator "*" for all cases. Similarly, the matrix environment MATLAB automatically recognizes and executes various combinations of multiplication over scalar, vector and matrix types. Even full operator overloading for built-in types has been adopted in at least one commercial CACSD package, Xmath (Floyd *et al.*, 1991).

The benefits to the user of this should not be underestimated. Not only is the number of operators reduced but the notation used in interacting with the CACSD package looks more like conventional mathematics. This makes the package easier to use, easier to remember and, by creating a form of *documentary uniformity*, self-documenting.

Polymorphism is not restricted to operator overloading. It is also possible to overload function names too. For example, `tf2ss` and `zp2ss` are two MATLAB functions which are used to generate state-space models from other objects [in this case transfer functions in expanded polynomial or factored (pole-zero-pole) form, respectively]. In an OOCACSD system, these functions could both be accessed using a single overloaded function, say, `toSS`. In addition, many object-oriented systems enable polymorphic functions and operators to be defined for user-defined data types.

3.2.1.3. *Object identity*. This is an important idea in object-oriented software. It simply means that any object created in an object-oriented system is unique and can be named. This is a much stronger concept than is at first realized, and is at the root of the idea of *data consistency*.

Consider an "object" in a conventional CACSD package like MATLAB. Imagine that the user creates a vector **a**. Over the course of a single MATLAB session, **a** can serve in the role of a polynomial, the numerator of a transfer function, a list of time or frequency values, a row of a state-space matrix, and a list of parameter values for a root-locus calculation. At no time can **a** be considered consistent. Its value and meaning depends purely on the context. On the other hand, a vector object in an OOCACSD system suffers from no such ambiguity. It remains a vector throughout its lifetime. If **a** is used to initialize the coefficients of a polynomial, what results is a new polynomial object different from **a**, not a new role for **a**. If the polynomial was also named **a**, then the vector **a** would have to be destroyed. Given such strong object-identity, it is relatively easy to add *data persistence*, and so to provide additional support for *data consistency* through a database management system.

3.2.1.4. *Class and inheritance*. These are principally grouping concepts for the attributes and behaviour of objects. Class helps to strengthen the idea of *operational uniformity*, whilst inheritance provides a useful extra dimension to the concept of class that is useful in systems modelling and for the end-user extension of CACSD software. These ideas are illustrated below.

- Operational uniformity. This simply means that similar objects respond in predictable ways to given operations. For example, if a property of the class of systems is that they can be made to produce a step response, then, in an OOCACSD package, the method **stepResponse** should be common to all members of the class **System**. Similarly, a method **plot**, for producing graphical output, should be common to all members of the class **Result**. Such predictable uniformity of behaviour greatly simplifies the use of a package.
- Modelling. Class inheritance is useful in systems modelling where it is used to capture and encapsulate the generic aspects of models by means of type hierarchies. For example, consider the **Vessel** hierarchy shown in Fig. 23 (Andersson, 1990). At the highest level (root) of this hierarchy, the general concept that vessels have an inlet and an outlet is modelled. Further down, more specialized models are "sub-classed" from vessels. Thus, it is seen that gas vessels have an additional attribute which is their volume while tanks have a (cross-sectional) area. At the bottom (leaf) of the tree, a **TankWithOverflow** is defined to have a maximum level and presumably a model of what happens when this level is exceeded. Such a

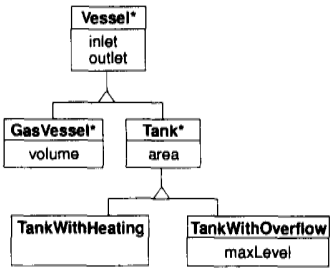

FIG. 23. Vessels: an example of the use of inheritance in modelling.

hierarchy allows the modeller the freedom to gather common attributes and behaviour at a high level while specialized attributes and behaviour precipitate down to the lower levels. In this way, the modeller is free to modify specific behaviour without affecting the rest of the model hierarchy while at the same time ensuring that generic behaviour, when modified, will be reflected in all subclasses. An actual instance of a TankWithOverflow is illustrated in Fig. 24 where it can be seen that the attributes of all the super-classes in the model hierarchy are inherited.

- Extendability. It is also possible to provide class and inheritance within the context of an interactive CACSD package where they would be used to create new data types by sub-classing built-in data types. However, if this is to be provided uniformly, then considerably more openness is required for both the data and behaviour definition than is normally provided.

FIG. 24. Instance of TankWithOverflow.

3.2.1.5. *Associations and aggregations*. These provide an additional layer of modelling capabilities. The need to model associations between objects and to group collections of parts into aggregate objects is poorly served in the current generation of CACSD packages. Both are extremely important concepts that help to serve *data consistency* as well as *representational uniformity*.

Aggregation is an important adjunct to identity which strengthens *representational uniformity*. For example, in MATLAB a state-space model is represented by four separate objects, the matrices **A, B, C, D**. These objects only respresent a state-space model by convention; there is no internal mechanism to enforce this relationship. To separate the state-space model of a plant from that of the controller, cumbersome notational "tricks" need to be used:

```
plant_A = [. .];
plant_B = [. .];
plant_C = [. .];
plant_D = [. .];
control_A = [. .];
control_B = [. .];
control_C = [. .];
control_D = [. .];
```

In an OOCACSD system, the initialization of plant and controller is probably equally cumbersome:

```
Matrix A = [. .];
B = [. .]; . .
C = [. .]; D = [. .];
StateSpace plant = [A, B, C, D];
A = [. .]; B = [. .]; . .
C = [. .]; D = [. .];
StateSpace controller = [A, B, C, D];
```

However, once defined, the StateSpace object, which is an aggregate of four state matrices, can be used as a single object. So instead of

```
y = stepResponse(plant_A, plant_B, . .
    plant_C, plant_D, t);
```

the step response of the plant can be obtained simply by execution of

```
TimeResponse y
  = stepResponse(plant, t);
```

There is no danger of misusing the `stepResponse` function because (a) none of the state-matrices can be accidentally redefined and (b) `stepResponse` is a built-in function of the StateSpace object.

Association is an important concept that can be useful for building systems by recording the connections between subsystem. When combined with aggregation, it can be used to create hierarchical systems. Other uses for associations and aggregations can be found in data modelling. For example, systems and the analysis, simulation or design results that they generate can be related to each other; also versions of systems as they evolve through a design can be recorded and managed. Additionally, associations and aggregations are important in maintaining data consistency in a database for CACSD.

3.2.1.6. *Uniformity over user proficiency*. This ensures that the spectrum of end-users of a CACSD package get the support they need from the software. Users come in at least three different varieties. There are expert users who are very familiar with the packages they are using. At the other extreme are novice users who are new to the package and need a lot of help whilst they are learning how the package works. In-between are infrequent users who use the package from time to time but not frequently enough to become experts. These users all require different levels of support from a package:

- Expert users want responsiveness and flexibility. Commands must be quick to enter; all commands must be available at all times; it must be possible to combine commands to create new ones.
- Novice users want the system to be supportive. There should be plenty of help and guidance, possibly by the package taking the user step by step through various scenarios.

- Infrequent users need plenty of help, but only when they ask for it.

Very early packages tended to be aimed at novice users. As the developers and frequent users of these packages became experts, they began to demand less supportive environments that gave them the power and flexibility they required. This resulted in complex command languages, often with hundreds of terse and easily forgotten command names. Support for the novice and infrequent users correspondingly decreased. This trend must be reversed and extra care should be exercised in developing CACSD packages in order to ensure that the transition from basic to advanced use is gradual.

A positive advantage of the *contract of behaviour* or *protocol* that an object-oriented system provides is that the number of possible commands that can be issued in a given context is greatly reduced. This makes it easier to understand the software system. It also has the potential of making user help much easier to provide and more meaningful. For example, if the user is dealing with a transfer function object, there is no need to provide help on any functions that do not take transfer functions as arguments. Help messages and tutorial operations can be built in to the object itself so that they can be invoked when needed without effecting the use of the object by experts. Polymorphism ensures that help or tutorial assistance can always be invoked in a consistent way.

3.2.1.7. *Documentary uniformity*. This is a means of facilitating the use of a CACSD package. In the sense originally implied by Rimvall (1988), this term refers to the organization of both off-line manuals and on-line help systems. Clearly, the documentation of the OOCACSD system must also be well thought-out. However, by emphasizing the data, of which there are fewer kinds than operations in most software systems, object-orientation provides a natural basis for arranging documentation. Superclass and subclass hierarchies may suggest useful ways of structuring the documentation. Association, inheritance and aggregation relationships can be used to guide cross-referencing. An additional bonus is the uniformity of documentation provided by polymorphism which can help to clarify records of the user's own work.

3.2.1.8. *Uniformity over size.* This is important in order to ensure that the numerical algorithms used in computation degrade gracefully with problem size. In terms of CACSD, this simply means that the best possible algorithms are used at all times and that adequate exception handling is provided to enable the user to be warned when loss of precision is likely to occur. Object-oriented languages have some advantage over conventional languages in these respects. It is possible, for example, to tune the methods to suit the data types, say, to exploit the structure of certain kinds of matrices (Floyd *et al.*, 1991). Methods should also be based on the best numerical techniques available. At the moment, the level three basic linear algebra subprograms (Dongarra *et al.*, 1990) and LAPACK (Demmel, 1989) provide a stronger foundation than the older packages LINPACK (Bunch *et al.*, 1979) and EISPACK (Smith *et al.*, 1977; Garbow *et al.*, 1976) on which MATLAB (Moler, 1980; Little and Moler, 1990) is based. The availability of a consistent exception handling feature may govern the choice of implementation language.

3.3. *Architectural issues*

In the previous section the object-oriented paradigm was discussed from the perspective of the user's experience of a CACSD package. From the developer's point of view, there must also be benefits if an OOCACSD environment is to emerge. The benefits will come about in two ways. Firstly, the positive benefits of developing software using object-oriented methods discussed in Section 2 will have an effect on the quality, re-usability and ease of maintenance of a CACSD package. This aspect will not be covered further here. The second benefit will come about through the emergence of a new object-based architecture for CACSD. This architecture is introduced and its benefits discussed in this section.

3.3.1. *Package integration.* A major drawback in computer-aided control engineering is due to difficulties in package integration. Horizontal integration (by algorithmic extension of the package) or vertical integration (by addition of specialized tools) (Rimvall, 1988) of CACSD packages is only satisfactory if the package has those features necessary to promote usability that were discussed in the previous section. Otherwise the base package simply becomes overloaded, unwieldy and slow.

The control systems design process includes such diverse activities as model building, system identification, nonlinear systems simulation, linearization, linear analysis, controller synthesis, controller implementation, project and model management, and documentation. As users come to demand computer support for all these activities, then the current generation of CACSD packages, even an OOCACSD, will no longer be able to cope.

Taylor and Frederick (1984) have given the name computer-aided control engineering (CACE) to the computer-based support of the whole control system design process. In the course of research into systems for CACE it has become clear that a single package cannot provide all the facilities that are required. Instead, the integration of many packages is required.

All CACSD packages have an architecture that roughly corresponds to that shown in Fig. 25. In broad

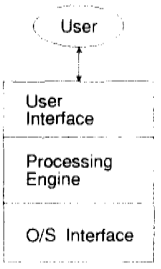

FIG. 25. Architecture of typical CACSD packages.

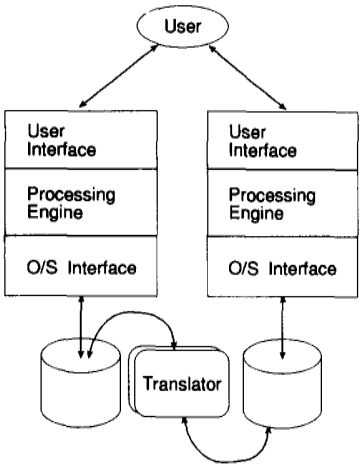

FIG. 26. Possible means of package integration.

terms there is a processing engine that does the computational work, a user interface which provides the means to control the package and observe the results and an operating system interface which, usually by means of files, allows the package to store models and data.

The problems in integrating CACSD packages are:

- The user interfaces are incompatible, the user needs to be familiar with several different styles of command entry, several different command languages and several different sets of commands.
- The operating system interfaces (file-formats) are incompatible.

Package developers, or frustrated users, have overcome these difficulties by providing data exchange facilities based on data translators as illustrated in Fig. 26. It is clear that such an *ad hoc* approach to integration is far from acceptable since there would be $n(n-1)$ data translators required for n packages and the problem of incompatible user interfaces is not addressed at all.

To overcome these difficulties, a different architecture is required. The best that can be done with the current state-of-the-art is to provide a common user interface and common data store (or repository). The earliest integrated CACSD environment of this type was the Federated Computer-Aided Control Design System described by Spang (1984, 1985). This has been followed more recently by an **E**nvironment for **C**ontrol **S**ystems **T**heory, **A**nalysis and **SY**nthesis (ECSTASY) (Munro, 1988) and the **M**ulti-disciplinary **E**xpert-aided **A**nalysis and **D**esign (MEAD) environment (Taylor *et al.*, 1989). The architecture of all these systems is essentially that shown in Fig. 27. Note that the normal external interfaces of the constituent packages are connected to the environment through command language and data translators. The user communicates with the packages through a single user interface via a supervisor program. The supervisor is responsible for converting the user's commands into package commands. The packages pass data, through data converters, to a common database where it may

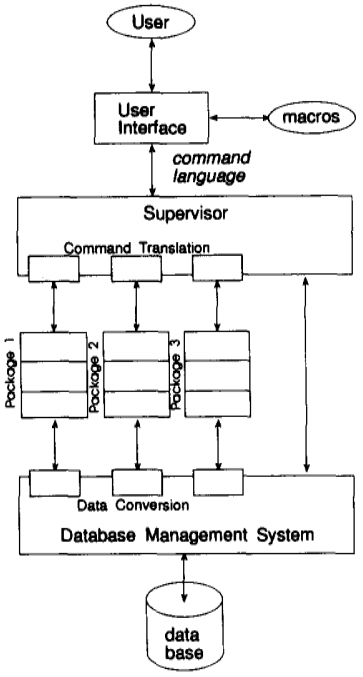

FIG. 27. External package integration.

be shared with other packages. Added value is gained by giving the user direct access to the database where models and results may be manipulated.

The disadvantage of this method of integration is due to the continual development and enhancement of the constituent packages which makes the effective maintenance of such an environment impossible. However, the common user interface and database remain useful ideas with which to proceed to develop a better architecture.

3.3.2. *A tools-based approach.* A fundamental change in the architecture of the constituent CACE tools needs to occur if further progress to sustainable integration is to be made. It needs to be recognized that a large amount of duplication of function occurs when trying to integrate packages of the sort illustrated in Fig. 25. Great progress could be made if the CACSD environment provided the user interface and data storage services and the processing was provided by simple stand-alone tools. This so-called *tools-based architecture*, illustrated in Fig. 28, was first advocated by Andersson *et al.* (1991) and was further refined by Barker *et al.* (1992, 1993).

The fundamental data object in CACSD is a dynamic system model. Such models are used to represent nonlinear plant, actuators, controllers, sensors as well as the linear abstractions that are used in design. Recognition of this is a pre-requisite of a tools-based architecture, and a means of representing these models systematically and flexibly is a necessary part of the CACSD environment. Several workers, in particular Mattsson (1988), Cellier *et al.* (1991) and Cellier and Elmqvist (1992), have made the case for an object-oriented language approach to modelling which provides this necessary flexibility.

Two such modelling languages are DYMOLA (Elmqvist, 1978; Cellier and Elmqvist, 1992) and

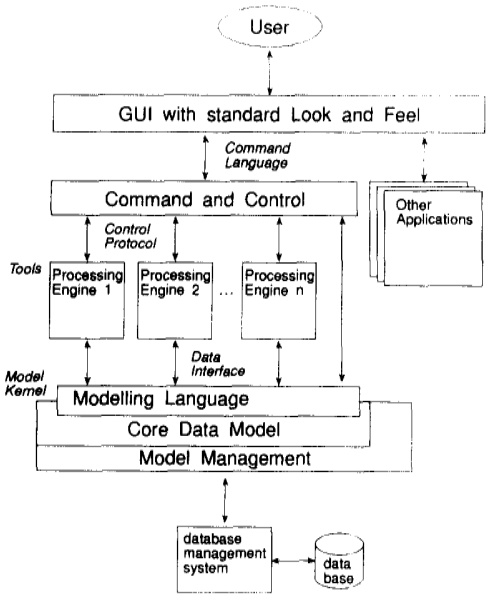

FIG. 28. A tools-based architecture.

Omola (Andersson, 1989, 1990). The latter language is based on object-oriented principles and supports encapsulation of attributes and behaviour, separation of class and instance, structuring facilities and inheritance.

The tools in the tools-based architecture can be regarded as objects that contain models as attributes and act on messages received from the user interface. These objects process the models to produce new models and generate results. They are supported in this task by a *core data model* which provides basic object-oriented data-types used in computations and a model management layer which records the structure of objects, associations between models and results and version hierarchies.

The command and control layer acts as the text-based user interface to the tools. Its purpose is to convert the commands issued by the user into messages to which the tools or the model kernel can respond. In contrast to the packages that reside in systems like ECSTASY or MEAD, the tools in the tool-based architecture are much simpler, performing typically only one or two simple operations. As such, they only require a very simple control interface. This interface is provided by a well-defined messaging service based on simple tool-based communication protocols. Because the tools are simple, to perform a high-level task might require the co-operation of more than one tool, the creation of many primitive data types in the model kernel and the issuing of many messages.

A very similar architecture, for computer-aided software engineering (CASE), has already been proposed by standards influencing bodies such as the European Computer Manufacturer's Association (ECMA) (ECM, 1990) and the Object-Management Group (Stone, 1992). The technologies that emerge from such initiatives should be seriously considered and adapted for CACSD as appropriate.

In the next three sections, the three key components of the tools-based architecture, the user interface, the model kernel and the database management system (DBMS), are discussed in detail.

3.4. *User interfaces*

The command-line interface of packages like MATLAB is well liked by users of CACSD tools. In this paper, it would be unwise to give the impression that the adoption of object-oriented idea will necessarily change this. Hence, when OOCACSD is discussed, it is not from the viewpoint that the user interface will necessarily have to be object-oriented too.

In a language like Smalltalk-80 (Goldberg and Robson, 1983), objects are manipulated by sending them messages. This is reflected in the Smalltalk interpreter thus:

```
t := TimeInterval new: (0 to: 10
                        step: 0.1)
model := StateSpace new: (A, B, C, D)
y := StepResponse new: (model time: t)
```

This appears rather foreign to anyone used to more traditional languages. In C++, methods are explicitly invoked by use of member functions. But even this syntax appears rather odd:

```
TimeInterval t(0, 0.1, 10);
StateSpace model(A, B, C, D);
TimeResponse y=model.step(t);
```

However, such syntactical niceties can easily be hidden by a well-designed command interpreter. Therefore, the more comfortable MATLAB compatible notation can be retained:

```
TimeInterval t=[0:0.1:10];
StateSpace model=[A, B, C, D];
TimeResponse y=step(model, t);
```

By adopting the object-oriented paradigm, complexity is reduced by the support and use of abstract data types and polymorphism. However, there are still more benefits to be gained by the more extensive use of graphics within an CACSD system.

3.4.1. *Graphical user interfaces.* There is no doubt that graphics is an essential part of the CACSD environment. Yet it is curious how little of the potential of graphics has been realized. Part of the reason for this is the difficulty of programming interactive graphics applications using conventional languages like FORTRAN and C. Once again, the object-oriented paradigm has much to offer, and indeed has indirectly influenced many of the developments. In this section, two simple examples of the potential for interactive graphics in CACSD are presented, then requirements for graphics in the context of CACSD are examined in more detail and the object-oriented software that is available for user interface composition is reviewed.

3.4.2. *Interactive graphics.* There are two important classes of graphics in use in the current state-of-the-art of CACSD. These are the input of system models and the output of results. The latter requires little in terms of interaction and indeed little has so far been offered.

In MATLAB (version 4.0) and Xmath (Floyd *et al.*, 1991) fairly sophisticated post-processing of plots is available. However, much more could be done to aid the visualization of results. For example, the possibilities of animation are only just beginning to be explored.

The most important examples of interactive graphical user interfaces for CACSD are those which are used for the input of block-diagram models. Of these, there are many examples including SYSTEM-BUILD (ISI, 1989), Model-C (SCT, 1987), and SIMULINK (Checkoway *et al.*, 1992) which provide nonlinear systems simulation to MATRIX$_x$ (Walker *et al.*, 1984), CTRL-C (Little *et al.*, 1985) and MATLAB, respectively. Third-party user interfaces for ACSL (MGA, 1991) are provided by Protoblock (Grumman, 1989) and EASE$^+$ (Expert Ease, 1989). All these systems provide hierarchical block-diagram system input to complement the analysis and simulation facilities provided by the host packages. In addition there are package independent prototype interfaces like CES (Barker *et al.*, 1988b), Hibliz (Elmqvist and Mattsson, 1989), ARGOS (King, 1986) and ECSTASY (Munro, 1988). Barker (1988) gives a recent review of these and other graphical user interfaces used in CACSD.

There are, however, other possibilities for interactive graphics that have yet to be exploited in CACSD and for which the object-oriented paradigm serves both as inspiration and implementation method. These are interactive desk-top systems for CACSD, browsers for models and results and controller parameter-design systems.

3.4.2.1. *Desk-top systems*. The desk-top metaphor has become very popular in graphical operating systems such as the Macintosh (Gregg, 1984), Windows 3 (Udell, 1990) and Open Windows (Sun, 1991). These all provide the ability to create interfaces to several independent applications, or different views of the same application, and have them appear at the same time on the same screen as overlapping windows. Pictures which represent data objects can be manipulated on the screen like pieces of paper on the engineer's desk. Such interfaces with their scrolling windows, pop-up menus, dialogue boxes are the "icons" of the modern windows, icons, mouse, pointer (WIMP) graphical user interface (GUI). The example in Fig. 29 shows how such facilities might be provided in a desk-top system for CACSD. The figure illustrates an "iconified" system object. The black boxes around the periphery of the system icon indicate that the system has been selected by the user. A menu, associated with analysis options, has been activated. The contents of the menu are governed by the object's protocol; hence only those analysis options which are valid are displayed. Such an interface is easy for both the novice user, who is only presented with valid options, and the expert user, who can get to the anlysis option he requires quickly.

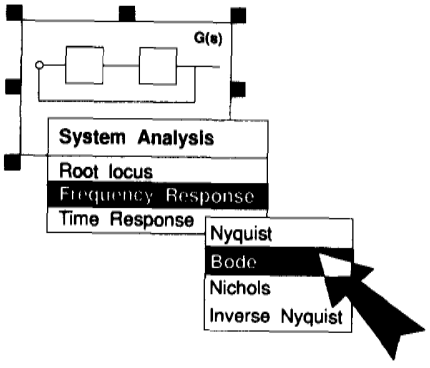

FIG. 29. An object-based graphical user interface for CACSD.

3.4.2.2. *Browsers*. Part of the aim of using object-oriented techniques is to simplify the building of plant models from models of its components. To support this aim it must be easy to be able to find an adequate model from a library or a model which can easily be modified. When one begins to re-use and modify components, version control and configuration management become critical issues. The modeller would want to record why changes in a model were made, what configuration of components was used to construct a composite model, and what models were used to generate a given set of results. Book-keeping is also important in order to keep a record of what has been done by the engineer, possibly during the whole lifetime of a given project. Such a record would enable any experiment to be performed again or any set of results to be regenerated, since the detailed set-ups used would be recorded.

These requirements imply the need for very good browsing tools and documentation facilities. Such browsers are a key component of the Smalltalk environment (Goldberg, 1985), for example, where multi-windowed techniques are used to provide access to source code. They are also to be found in the G2 system, a tool for developing real-time expert systems for complex applications that require continuous and intelligent monitoring, diagnosis, and control (Moore *et al.*, 1990). By adapting the graphical techniques used in these applications it should be relatively easy to support CACSD. A mock-up of a possible CACSD model and results browser, based on the Smalltalk browser, and influence by the MEAD browsers described in Rimvall *et al.* (1989), is illustrated in Fig. 30.

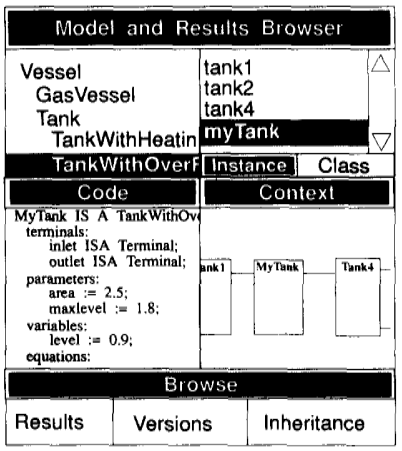

FIG. 30. A browser for CACSD models and results.

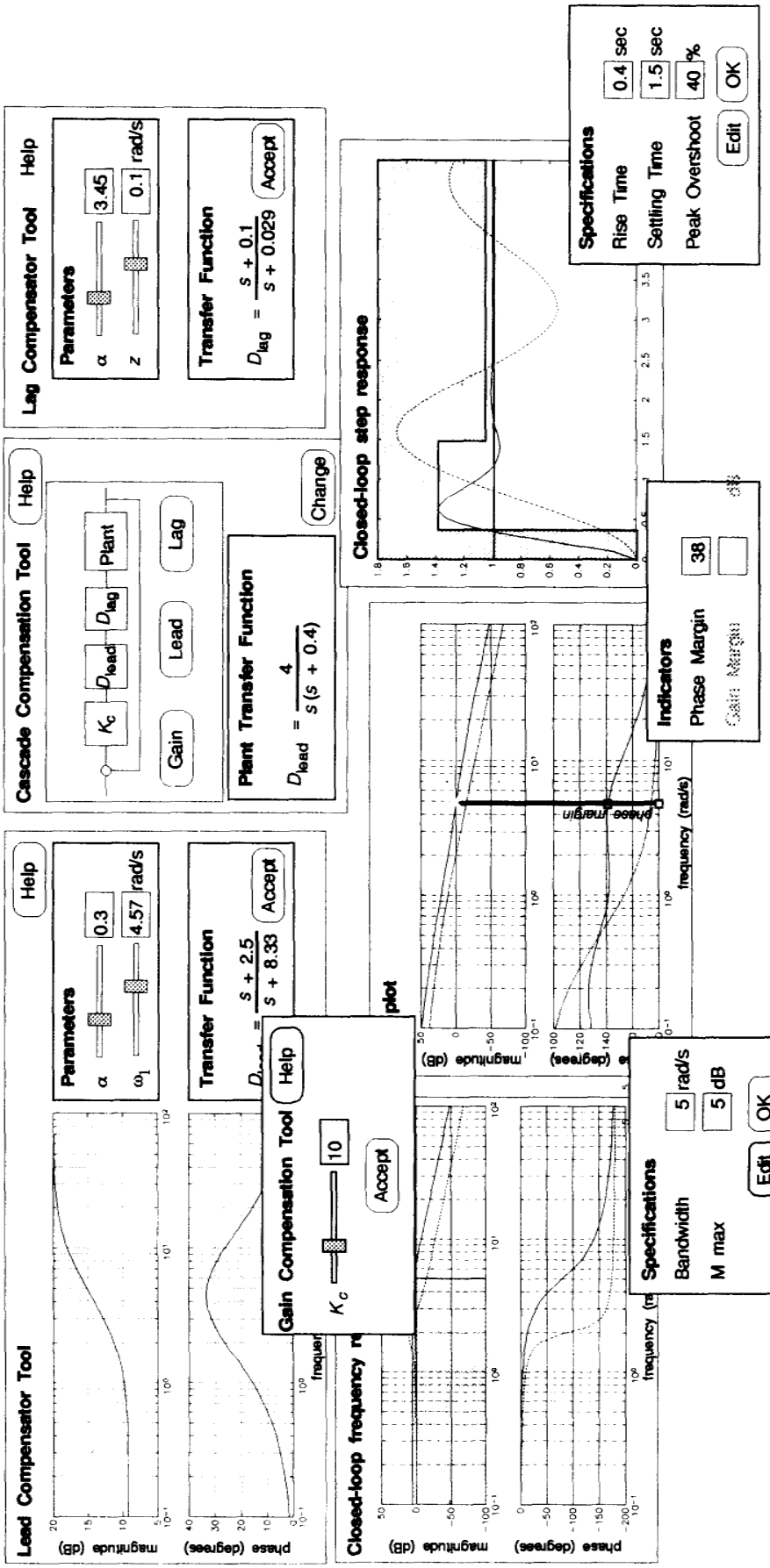

FIG. 31. An interactive frequency response design system.

3.4.2.3. *Interactive design tools.* The statement that "design is an interactive process" is one that cannot be over-emphasized when considering CACSD. One needs to be able to "try things out to see what happens" quickly and safely. To illustrate this, an example of an "ideal" frequency response design facility is illustrated in Fig. 31. The designer has several windows, each of which provides feedback on some aspect of the current status of the design (in this case that of a lag-lead compensator). In the main window the uncompensated open-loop frequency response curve is shown. The open-loop gain can easily be changed by moving the slider. The display automatically adjusts to indicate this change but so do all the other displays: the time response, closed-loop frequency response and compensator time response. All the performance measures also change instantaneously. As the phase lead compensator is adjusted by modifying the centre frequency or the value of the lag fraction the effect is instantly observed. When all the design constraints have been satisfied, the "OK" button is pressed and the software for the digital implementation is automatically generated. It should be clear that it would be easy to satisfy a given set of constraints with this system and there is no need for any command language.

This design facility is achievable with current object-oriented graphics technologies by making use of an interactive graphical user interface GUI model called the Model-View-Controller (MVC) paradigm (Kranser and Pope, 1988). The MVC interaction model, which was originally developed for the Smalltalk-80 system, is illustrated in Fig. 32. A *model* represents the application objects. In the example above, the models would represent the plant and compensator transfer functions. Associated with each model are one or more views which are used to indicate the instantaneous state of their respective models. In the example, the parameter values, Bode and time response plots and the shaded areas which represent the design specifications, are all views associated with the plant and compensator models. All views associated with a given model will automatically change when the state of the model changes. The user can change the state of the model either by the use of controllers or directly under model control. In this case, all the sliders and buttons are controllers. Only one implementation of an MVC architecture for CACSD has so far been developed and this is described in Hara *et al.* (1988, 1992).

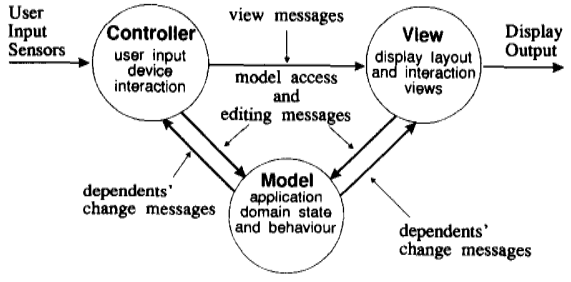

FIG. 32. The model-view-controller model of an interactive graphical user interface.

3.4.3. *Principal requirements.* The principal components that make up a GUI are:

- application graphics;
- interface components or controls; and
- "Look and Feel".

These components are discussed below.

3.4.3.1. *Application graphics.* Graphics is key to any engineering application. Typically, application graphics is concerned with:

- the provision of views of the objects contained in the application;
- the provision of a means by which the objects inside the application can be modified; and
- the provision of a means for visualizing the results of the user's manipulations.

For example, in block diagram editors such as SIMULINK, SYSTEM-BUILD or eXCeS's BlockEdit, shown in Fig. 33, the application graphics are provided to allow the block diagram to be viewed, for blocks and connections to be moved around, added or deleted and for operations to be defined.

It is for the support of such application graphics that the standards GKS and PHIGS have the greatest potential. It is also the area in which the modern GUI toolkits have their greatest weakness since the programmer typically has to deal with graphics at the "bitmap" level rather than at the level of objects—lines, boxes, circles and text. In the absence of reasonably efficient combinations of application graphics standards and bitmap graphics, it falls to object-oriented programming to provide the necessary high-level structures with which bit-mapped views of high-level graphical objects can be manipulated.

3.4.3.2. *Interface components.* These provide the control features of a GUI for CACSD. These may be for manipulating the view of the application—pan and zoom controls, scroll bars, dialogue boxes for controlling the entry of data and menus for the control of the application's execution. In most toolkits for the development of interactive graphics, adequate provision is provided for constructing these components and it must be noted that it is the lack of such facilities that is a particular weakness of traditional standards which have developed from primarily batch oriented applications.

3.4.3.3. *"Look and Feel".* This is also a critical feature of good interactive software. "Look and Feel", as the term implies, is concerned with how an application appears to the user and how it behaves. For example, one of the earliest CACSD packages, the UMIST Control Design System's "look" (Rosenbrock, 1972), is based on a very simple and consistent user interface style. However, its strict execution model makes experienced users "feel" as if the software is being condescending. On the other hand, MATLAB is also based on simple concepts which makes its basic operation and data manipulation methods easy to grasp. It feels powerful to experienced users who appreciate its flexibility and extendibility, but by the same token it feels forbidding

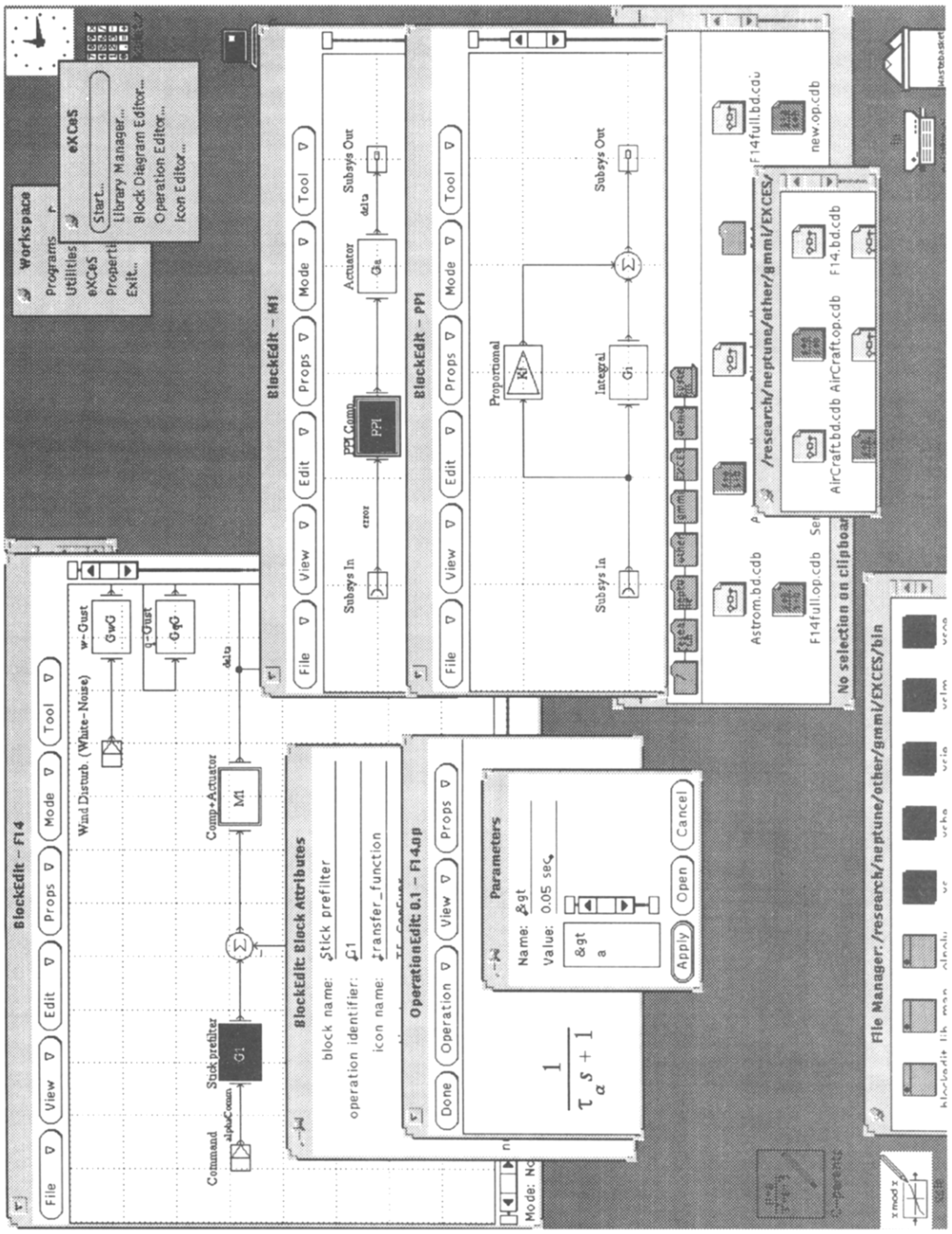

FIG. 33. The eXCeS GUI.

to the novice user by virtue of the vast array of commands and their lack of structure.

Look and feel is particularly important in interactive graphics. It is important to the user simply because by learning the controls and behaviour of one application he knows a large number of the commands for another built to the same standards.

Look and feel standards are generally associated with particular hardware and operating system platforms, and are usually specified in "style-guides". For high performance graphics workstations two "styles" have come to dominate. These are Motif (OSF, 1989) and OPEN LOOK (OPEN LOOK, 1989). It is too early to say which of these, if either, will come to dominate. From a purely aesthetic point of view, OPEN LOOK, which was designed from first principles, has the edge over Motif which is essentially an implementation of the Presentation Manager style of Microsoft Windows and OS/2. From the user's point of view the differences between these two styles are minor but confusing where they occur. It may be expected that the two styles will converge in functionality as users begin to demand applications that behave the same way on either OPEN LOOK or Motif supporting platforms.

3.4.4. *Object-oriented programming and graphics.* Modern, multi-windowed, graphical user interfaces have always been associated with the object-oriented paradigm. The earliest example of a workstation with such an interface was the Star, developed at Xerox PARC (Smith *et al.*, 1982; Johnson *et al.*, 1989a), part of the same development that resulted in the Smalltalk languages. This work was adopted as a model for the Apple Lisa (Williams, 1983) and then the Apple Macintosh (Gregg, 1984).

The event-driven architecture which is a feature of windowed graphical user interfaces is the key to highly interactive graphics. A window manager collects events from various input devices and broadcasts them to the applications which are sharing the screen. The applications act on their events and in doing so, may in turn send messages back to the window manager which modifies the display. Unlike conventional applications, which generally have a single thread of control and process input and output in a sequential manner, graphical based applications are highly asynchronous. The advantage of this is that the user, rather than the application, has full control over program execution and this can make the application easier and more intuitive to use. The disadvantage is, of course, that such applications are much more difficult to program in conventional languages. They also require greater computing resources.

3.4.4.1. *Graphics toolkits.* In Smalltalk, the consistency of the object-oriented paradigm extends through to the graphics tools which define the interface and environment. Once the initial learning cycle is completed, it is as easy to program graphics in Smalltalk as it is to program any other application.

The naturalness of the object-oriented paradigm for expressing and coding graphical user interface has been recognized, as is evidenced by the hybrid object-oriented "languages" that support such interfaces on the Macintosh (Wilson *et al.*, 1987; Schmucker, 1986; Borenstein and Mattson, 1989), Sun Microsystems' XView (Heller, 1989) and the X-Toolkit for the X Window System (Sheifler and Gettys, 1986; Young, 1989; Miller, 1990). In the case of the Macintosh, OOP is provided by language extensions and links to the graphics toolbox; in the latter two cases, both based on the C language, support is provided by use of variable length parameter lists, a programming style and object-code libraries. Such facilities—though better than nothing—are still rather crude.

A better approach is that taken by InterViews (Linton *et al.*, 1989) and ET++ (Weinand *et al.*, 1988a) which are two C++ class libraries developed truly to support object-oriented "user-interface composition" on top of the X Window System (XWS) (Sheifler and Gettys, 1986). There are also X graphics classes provided by Objective-C and Eiffel. The main drawback of these systems is that they do not yet follow either of the recognized "look and feel" specifications for open-system user interfaces, although C++ class definitions are beginning to emerge for these. There are also interface graphics frameworks under development that can potentially support many user interface styles; one such is ITS being developed by IBM (Wiecha *et al.*, 1990).

Another important feature of graphics applications is the ease with which the objects that model the real world, such as text, pictures, block diagrams and simulation results, can be developed. Here too, the object-oriented paradigm has much to offer. Pictures and documents may usefully be represented as hierarchies of connected parts. The earliest computer-aided drawing system, Sketchpad (Sutherland, 1963), was very definitely object-oriented. Since that time, graphics standards have not tended to emphasize this feature, although there has been some success in merging the object-oriented paradigm with the GKS and PHIGS standards (Wisskirchen, 1990). In the future it is likely that class libraries, like InterViews, which do provide basic classes for graphics and text structuring and so provide the potential for powerful application production.

3.4.5. *A case study: eXCeS.* The user interface Control Engineering workStation (CES) (Barker *et al.*, 1987, 1989) was developed within the authors' research group and serves as a good example of the problems that are faced when updating CACSD graphics for modern expectations.

To assure portability and machine independence, CES was developed deliberately using the graphics standard GKS. However, continued lack of full support for the standard on the hardware platform adopted, and the continued development of graphics itself, particularly the emergence of XWS as a *de facto* standard for workstation graphics, led to the redesign of CES. The result, called the extended X-based CES (eXCeS) (Barker *et al.*, 1991), was specified during 1988 and development has continued since then. A screen-shot of the interface is shown in Fig. 33.

From the experience of developing eXCeS the following major principles, which are in agreement

with the findings of other workers, have been learned "the hard way".

- Interface development requires rapid prototyping which enables models of interfaces to be quickly, and *graphically*, constructed and interaction objects to be quickly tried out and subjected to user trials.
- To achieve consistency, look and feel guidelines should be enforced by the development tools used. At the same time, the ability to over-ride such enforcement where necessary must be possible.
- Application graphics requires a powerful imaging model, a rich set of basic graphical objects and the support for structured hierarchies of displayed objects.

The eXCeS interface was developed using a combination of the OPEN LOOK intrinsics toolkit (OLIT) (Miller, 1990) and the X-library (Xlib). This combination provides poor support for all of the above requirements. Application graphics must be coded in C using an object-oriented style that is neither natural nor particularly concise. The coding was done by hand, and its complexity prevented any modification, following user trials, that may have been practical had rapid prototyping methods been used. The OLIT interface objects are designed to conform to the OPEN LOOK "look", but the toolkit provides very few constraints on how they can be used in an application, so the enforcement of the "feel" must be done manually. Application graphics development using Xlib has to be done at a very low level.

For the development of research prototypes, the authors now recommend the use of InterViews which has been made considerably more powerful in recent releases by the availability of Unidraw (Vlissides and Linton, 1990), a set of classes intended to simplify the creation of interactive editors; glyphs (Calder and Linton, 1991), so-called "light-weight" objects which can be used to create different styles of interface; and IBUILD (Vlissides and Tang, 1991), a graphical tool for constructing InterViews applications. For commercial products, developers are still constrained to the toolkits available for the development language and look and feel required.

3.5. *The model kernel*

The starting point for all control systems design is a set of performance specifications based on an idealized model of a control system. Indeed, models are central to the whole control system design process. In CACSD packages, models can take on many different forms. The representation that is closest to a standard is the form used in CSSL packages and its derivatives. This is a combination of modelling and a simulation control language that was developed 25 years ago when digital simulation was still very close to its roots in analogue computation. CSSL consequently still has the flavour of a *functional block* approach to modelling.

The power of computers and the sophistication of the symbolic processing capabilities of the software they can run has moved on considerably. There is really no justification for continuing to use old technology when far better, more supportive modelling languages are available. This is particularly true when such advances have the potential for solving the integration problem described in Section 3.3.1.

This section discusses the *model kernel* box shown in Fig. 28. The term model kernel was coined by Andersson *et al.* (1991), but in this paper its definition is broadened to take into account aspects of the CACSD environment other than modelling. In particular, the model kernel is viewed as a layered infrastructure. At the highest level is the modelling language in which models are defined. At the next level is a core data-model which provides basic control engineering data types. Finally at the lowest level, and in some senses also around the other two layers, is a model management layer which provides the basic housekeeping facilities for the environment and interacts with the underlying database system.

3.5.1. *Modelling language layer.* The need to provide suitable "languages" for the specification of complex hierarchical control systems has long exercised workers in the field of CACSD. The computer-aided control engineering group at the Lund Institute of Technology in Sweden has been particularly active in this area. As long ago as 1978, Elmqvist (1978) invented a structured modelling language called DYMOLA. This work was continued by Åström and Kreutzer, (1986); Mattsson (1988) and Andersson (1990). Other workers, notably Maciejowski (1984), Cellier (1991), Cellier *et al.* (1991) and Cellier and Elmqvist (1992) have contributed to the debate on how best to model systems formally. The outcome of this work is the belief that some form of object-based modelling technique is preferable to the cause-and-effect modelling currently used in most CSSL based languages.

The culmination of the research at Lund is the object-oriented modelling language Omola (Andersson, 1989, 1990) which provides support for the important object-oriented principles of aggregation and inheritance within the context of dynamic systems modelling. The support for these principles is designed to provide the maximum potential for model and model component re-use. The basic object hierarchy of Omola is illustrated in Fig. 34. The fundamental objects are models, terminals, parameters, variables and realizations.

- **Models** are the basic structuring components and may contain zero or more **Terminals**, **Parameters**, **Variables** or **Realizations**.
- **Terminals** provide interconnections of components and are sub-classified into **SimpleTerminals**, **RecordTerminals** and **VectorTerminals**. The properties of **Terminals** provide a means of checking the consistency of models, and also automatically modifying the meaning of an associated model object according to the connection semantics (Mattsson, 1988).
- **Parameters** and **Variables** provide the mechanisms by which constants and time variable quantities, respectively, may be defined.
- **Realizations** provide the means by which the

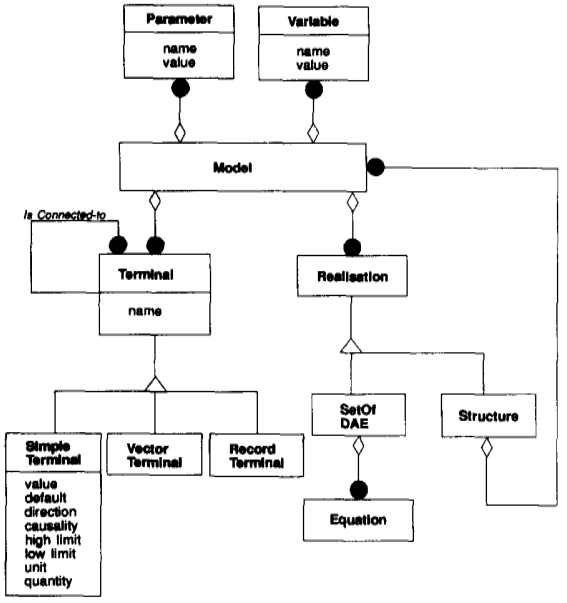

FIG. 34. The object-hierarchy of Omola.

dynamic behaviour of models may be defined. Differential algebraic equations (DAE) are used to provide maximum flexibility and to provide the mechanism by which models can have differing behaviour depending on their interconnection with the rest of the system. A subclass of **Realization** is **Structure** which provides the means by which systems may be hierarchically structured.

Omola may be used to define a model of a system in terms of the basic system components. It is also possible to use inheritance in order to provide a powerful means of model library creation. For example, it is easy to specialize or generalize basic components and then to use the specializations as components in further models or to further specialize them. This provides a much safer means of creating and re-using large component models within the basic activities of CACSD.

Having defined models in Omola, it is possible to provide different instantiations of the model according to the user's requirements. For example:

- for simulation an Omola model may be collapsed into suitable simulation code;
- for analysis the linear dynamics of a model may be extracted; and
- for documentation, a textual or a diagrammatic description of the model structure may be obtained.

The tools required to manipulate Omola in these ways would be provided by the CACSD environment. They would be transparent to the user, who would merely request that the environment provide particular views of the models. The use of a common definition is clearly a solution to the integration problem described in Section 3.3.1.

Omola is flexible enough to provide support for different model representations. For example, it has already been used to model and simulate chemical process plant (Nilson, 1992), electrical distribution networks (Mattsson, 1992) and discrete event dynamic systems (Andersson, 1992). It should be relatively straightforward to extend this range of applications to, for example, block diagrams, electrical circuits, mechanical mobility diagrams and bond graphs.

One area not addressed particularly well by Omola is how graphical information concerned with the creation of models using graphical tools might be handled. A simple prototype graphical editor for Omola models is provided in OmSim, a combined model editing and simulation environment developed at Lund (Andersson, 1991). However, it is unlikely that this will be a suitable basis for more general purpose graphical data entry. A partial data model for the graphical objects required in such an editor are presented in Fig. 35. This figure is based on the data structures found in eXCeS. The reconciliation of this object model with that of Omola is an area of future research.

The issue of how different types of model might be input, displayed and manipulated within a consistent framework is a more difficult problem. Initial ideas include the universal graph editor proposed by Chen *et al.* (1992). Work on the transformation of diagrams using rule-based methods has been described by

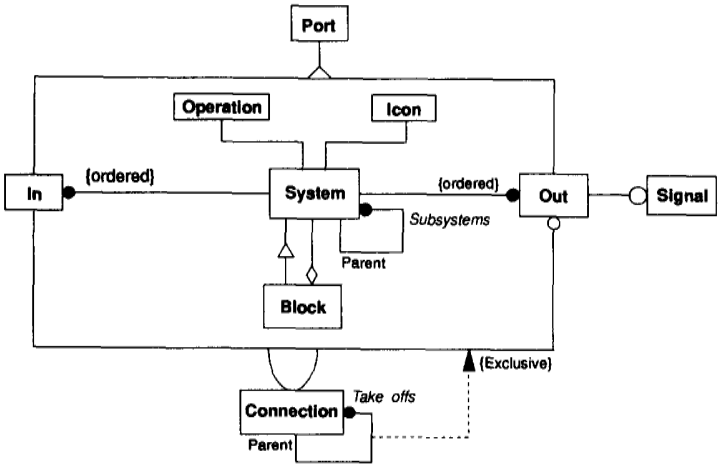

FIG. 35. Graphical object hierarchy.

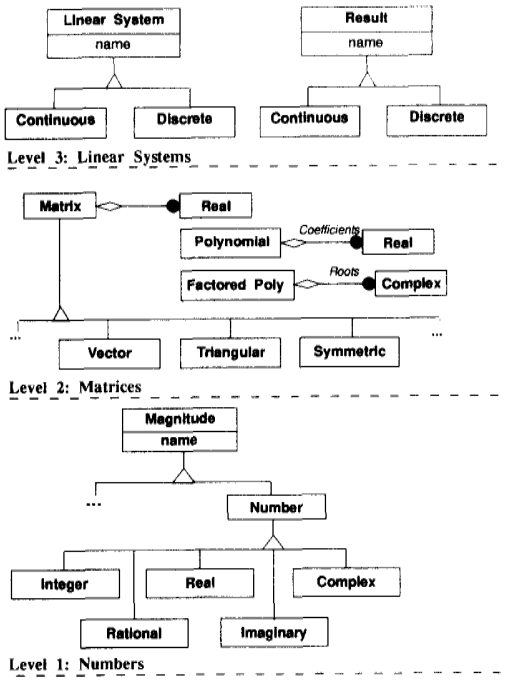

FIG. 36. Partial object hierarchy for CACSD.

Barker *et al.* (1988a) and Grant *et al.* (1992), and the automatic layout of diagrams needed to display diagrams after transformation is described in Barker *et al.* (1988a) and May *et al.* (1991).

3.5.2. *Core data model.* The term "core data model" refers to the basic components from which CACSD tools are built. Examples of such objects range from numbers, transfer functions and matrices, through time and frequency responses, to systems. Relational database systems can provide support for some of these, and at least one CACSD environment, ECSTASY (Munro, 1988), is built on such a database. However, an object-oriented approach could also be used to define these objects and in addition would provide better facilities for the storage of aggregate structures like state-space models. The object-oriented approach has the further benefit of modelling the operations required to be performed by or on the objects within the object-description. The use of an object-oriented data model would also provide a more complete description from which the standardization of CACSD facilities could proceed.

A layered partial class hierarchy for basic CACSD components is shown in Fig. 36. It is based on the Magnitude and Collection class hierarchies described in Section 2 (Goldberg and Robson, 1983; Pinson and Wiener, 1988) and is influenced by Joos and Otter (1991).

The level one classes provide the basic number objects Integer, Rational, Real, Imaginary and Complex plus their associated methods. At level two, simple aggregation objects Matrix, Polynomial and FactoredPoly are defined. At level three, LinearSystem objects for control system definition, and Result objects for analysis and simulation are specified. Only the root classes are shown, but level three objects would include StateSpace and TransferFunction models, TimeResponse and FrequencyResponse results, and their corresponding discrete time equivalents. Each high level object is implemented using the objects and methods defined in the layers below.

The advantages of the object-oriented approach are:

- The use of structured data types is safer than providing similar structures by the combination of simple data types. A state-space model is, after all, a single control data object which is implemented with four matrices. The need to manipulate this as four separate named matrices as must be done in MATLAB's Control Systems Toolbox is prone to error.
- Built-in methods ensure that the correct function is called when an operation is required. For example, if toStateSpace is a method attached to transfer-function models, then there is no need to provide differently named functions for the different forms of transfer function model. There is less chance of user error because it becomes the object's responsibility to apply the correct function and not the user's.
- Polymorphism can be used to make the manipulation of objects simpler for the user by reducing the number of operators needed.
- Provision of inheritance and aggregation enables the user to extend the system in an easy but safe way by the specialization of existing objects or the creation of new objects from the combination of existing ones.

3.5.3. *Model management.* This level of model kernel support is analogous to computer-aided software development. At this level, which can be thought of as a kind of project management, models and the data they generate must be controlled so as to provide some means of quality assurance and design auditing. Provision needs also to be made for viewing results, browsing the models and model libraries and controlling the database.

During the design process for control systems, numerous models of the plants and controllers involved are developed, refined and verified (Taylor *et al.*, 1988). The many versions of the models must be recorded as they evolve, and the results of simulation, analysis and verification, and documents supporting the design must all be logged and related to the correct versions of the models used. The resulting system is analogous to a software configuration control and version management system, and object-oriented database management systems have and are being developed to support such requirements (Björnerstedt and Hulté, 1989; Bobrow *et al.*, 1987a). Research is also underway to solve the more subtle problem of recording changes to the database schema itself (Skarra and Zdonik, 1987). Some initial work on the development of object-oriented database management systems for CACSD has been reported by Hope *et al.* (1990, 1991). Version control in engineering databases is reviewed by Katz (1990) and a practical

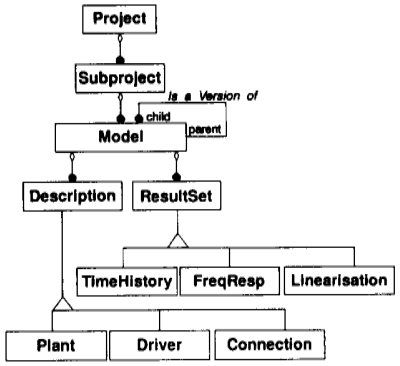

FIG. 37. Project class hierarchy.

implementation based on the automatic indexing of results and models is discussed by Joos (1991).

3.5.4. *Project support.* The project hierarchy described by Taylor *et al.* (1988) is illustrated as a class hierarchy in Fig. 37. It can be seen that **Projects** are sub-classified into **Subprojects** and then into **Models**. In the **Project** database there may be a number of **Subprojects** stored within any single **Project** and similarly a large number of **Models** within each **Subproject**. It is also necessary to encapsulate the notion of version within the **Model** class so that the descriptions and results can be associated with the version of the **Model** that generated them. All of these requirements would seem to be attainable by use of object-oriented methods.

3.6. *Object-oriented database management systems*

The third important component in the next generation CACSD environment is the database management system (DBMS). The object-oriented paradigm is making a significant impact on research into database techniques and technologies. It has been recognized that relational database technologies (Codd, 1970) which are superbly adapted to data processing applications, are less than ideal for the support of applications such as computer-aided software development, office information systems and computer-aided design (Kent, 1979).

Relational databases consist of data arranged in tables of fixed-length records. They are ideal for applications where (Brown, 1991):

- most information is static and defined in advance;
- changes to the data structures (schema) are infrequent and controlled by a group of database administrators;
- data stored is atomic and of fixed length;
- there are few types of entity to be stored with a large number of instances of each type and only simple; fixed relationships exist between each type;
- the database is initially populated with a large amount of data and thereafter grows only slowly;
- data is updated in place; and
- transactions are short, atomic and may be concurrent.

Conversely, applications like CACSD (Maciejowski, 1988) and related applications such as CASE have almost the opposite requirements (Maier, 1989; Brown, 1991):

- information evolves continuously;
- change to the schema is expected and frequent; many users will want to change the schema;
- data stored is atomic but also structured; data items may be large and complex and of variable length;
- there are many data types with fewer instances of each type; complex relationships may exist between types and new relationships may be created;
- little data are initially loaded but the database grows rapidly during the design phase as new objects and their relationships are added;
- versions of data items may need to be maintained; and
- transactions may be long-lived and complex.

Relational database management systems simply lack modelling power and performance; record-based models are not adequate for handling the complex data structures that are manipulated by computer-aided design software. To make use of a relational database for the persistent storage of data, applications must encode their internal data structures into the form acceptable to the database. The encoding involved greatly complicates the application programs and so most computer-aided design systems perform their own data management on top of the operating system's file structure. This is far from ideal, since the file system has no knowledge of the nature of the data stored in its files and is powerless to provide the data integrity checks and data management facilities which are a feature of database systems.

Computer-aided design systems which are built on top of relational databases use them at start-up to load design data into virtual memory from where the application builds the data structures it requires. At the end of a session, the internal records are translated back into tabular form and copied back into the database. This can give a considerable performance overhead, particularly when the data are loaded to make a small change and then dumped back to disk, or when transferring data between applications. Such design systems are merely making use of a database as an index for associative access to files containing design data (Taylor *et al.*, 1988), and are foregoing the other data management features such as recovery, concurrency control, integrity checking and buffer management.

A better solution appears to be offered by object-oriented database management systems (Maier and Stein, 1986; Dittrich, 1988; Stone and Hentchel, 1990). Good recent reviews of this technology are given by Kim (1990) and Brown (1991), who also provides a comprehensive bibliography of the subject. Essentially, object-oriented databases provide storage of data as objects, complete with methods, and overcome many of the modelling and performance problems associated with the relational model. The differences have been explored in Smith and Zdonik (1987). One key feature is that computation can take place within the database by execution of the methods defined and stored with the objects. This can significantly ease the computational load of the

application program and almost eliminates the need to move data to and from the database. Other core concepts are (Kim, 1990):

- the provision of unique identifiers for each object in the database which makes it easy to refer to the components of an object;
- the support of complex attributes: the instance variables of an object may be any class, user defined datatype or primitive datatype; and
- the provision of class hierarchies and inheritance.

Unfortunately, no single object-oriented data model has yet emerged as there are a number of issues still to be resolved, but articles such as Beech (1987) and Atkinson *et al.* (1989) are appearing which set an agenda for this discussion. There is also the exciting possibility of a merger between programming language and database to anticipate (Andrews and Harris, 1987; Kim, 1990). Such a merger, when combined with the object-oriented design methods discussed in Section 2, has the potential to unify completely software development within a single paradigm.

There are at least two commercial object-oriented database management systems available and several research prototypes in the industrial and academic sectors. These are reviewed in Stonebreaker (1990), Brown (1991) and Joseph *et al.* (1991).

3.6.1. *Databases for control system design.* Database technology has been proposed for CACSD by at least three authors. Maciejowski (1988) discusses data structures for CACSD and reviews work on the development of data-models which encapsulate an hierarchical system model. A simple data-model is used in ECSTASY (Munro, 1988) to provide persistent storage of linear system models and some nonlinear functions. The database built from this simple model enables the components of the ECSTASY environment (MATLAB with ACSL or TSIM) to communicate with each other and also provides support for linear system transformation and simulation code generation. Taylor *et al.* (1988) describe the use of a database as a linkage between simulation and analysis tools. They also argue for the use of database technology to support general project management, which in CACSD includes requirements for the maintenance of associations between model components, descriptions, results and version management. A package which provides these features, known as MEAD, has been developed and is described in Taylor *et al.* (1989) and Rimvall *et al.* (1989). As far as is known, only Hope *et al.* (1990, 1991) and Joos (1991) have reported on the application of object-oriented database systems to CACSD although experiences in the related areas of the computer-aided design of VLSI (Gupta *et al.*, 1989) and computer-aided software engineering (Brown, 1991) show promise for the technology.

In terms of choosing an object-oriented database from these currently available, the choice probably depends on the language that can be used to merge the application to the database, the level of support for features such as version control and schema modification, and the commitment of the developer to "standards" such as the "object-oriented database manifesto" (Atkinson *et al.*, 1989).

3.7. *Other applications of OOP in CACSD*

This section completes the review of object-oriented methods and their application to CACSD with an historical perspective of other applications of OOP to CACSD.

Workers in CACSD have been somewhat slow to recognize the potential of OOP in CACSD and this paper seeks to redress this. However, there have been isolated examples and these are described below.

Phaal (1986, 1987) was one of the first to realize the potential of OOP systems. He used Smalltalk-80 to implement Maciejowski's core data model as well as a simple block diagram editor and other graphics-based tools. This work has recently been extended by Tan and Maciejowski (1989) who used Prolog to provide a data language for CACSD.

Experimental work with OOP systems has been carried out at the Lund Institute of Technology, using the KEE system (Fykes and Kehler, 1985). This has resulted in a proposal for the object-oriented system modelling language, Omola, already described above. Zygmont (1987) has also made use of the KEE system to develop a simple model class hierarchy and a design-rule database. It has been recognized, at least by the Lund group, that systems like KEE and Smalltalk, though good for prototyping, cannot be used to develop tools which will be widely used by the rest of the CACSD community. They are therefore using C++ and InterViews to re-implement Omola. Similar developments have taken place at the Tokyo Institute of Technology. An interactive design environment, similar to that described in Section 10.2.3, has been developed in both Smalltalk and C++ using the MVC paradigm (Hara *et al.*, 1988, 1992). This work has also resulted in the development of a class library for CACSD called La++.

Object-oriented programming has been shown to be useful for simulation (Kreutzer, 1986), particularly for discrete event systems (Unger *et al.*, 1986; Eldredge *et al.*, 1990) where the communicating processes that are a feature of such simulations map well onto the message-passing execution model of OOP systems. Discrete event architectures can also be used for continuous system simulations (Zeigler, 1984) but these may be too inefficient to be used directly (Cellier *et al.*, 1991). None the less, one approach to implementing this facility is described in Vangheluwe *et al.* (1989).

4. CONCLUSIONS

Research in the object-oriented programming community is seeking to merge languages, class libraries and graphics development tools, to provide database support and to develop new design techniques in order to create powerful new programming facilities for the development of production code. Such tools are slowly emerging and workers in CACSD development must realize the potential of

these developments and grasp the ideas early in order to generate the CACSD environments of the 1990s.

In this paper the state-of-the-art in CACSD has been examined and it has been found to be lagging behind what could be possible given the advances made in computing technology. A model for a next-generation interactive CACSD environment based on tools has been presented. The use of the object-oriented paradigm in the development of three major components of this environment—the graphical user interface, model kernel and the database—has been discussed. The implementation of such an environment will enable the evolution of a standard open design environment for the control community.

The homogeneous way of thinking about, developing and using software that the object-oriented paradigm seems to offer provides enormous benefits. The software revolution may well come about in this decade and with it, hopefully, a revolution in CACSD.

5. FURTHER READING

5.1. *Background*

A fascinating collection of papers charting the historical development of structured programming, including reprints of Dijkstra's original expositions and several other key papers cited in this survey, may be found in Yourdon (1979, 1982). Some of the problems with the structured programming paradigm, and other "silver bullets" intended to slay the software-crisis "werewolf", are discussed by Brooks (1987). A rejoinder to Brooks' article discussing the silver bullet that is OOP is given in Cox (1990).

5.2. *Tutorial introductions to OOP*

For tutorial introductions to the object-oriented paradigm, articles by Robson (1981), Pascoe (1986), Thomas (1989), Wegner (1989), Duff and Howard (1990) and Gibson (1990) may be useful. A recent paper by Snyder (1993) provides a different view of the concepts and terms used in the object-oriented paradigm. More in-depth introductions are provided by Rentsch (1982), Stefik and Bobrow (1986), Korson and McGregor (1990) and Wegner (1990). Many key articles charting the evolution of OOP are collected in the IEEE tutorial collections (Peterson, 1987a, b). Meyer (1988) and Cox (1986) provide ample justification for the object-oriented approach.

5.3. *OOP languages and systems*

For particular languages, the standard texts are cited in Section 3. Useful summaries of language features and comparisons of languages are provided in Cox (1986), Stefik and Bobrow (1986), Meyer (1988), Booch (1990), Rumbaugh *et al.* (1991), Al-Haddad *et al.* (1991) and Lozinski (1991).

5.4. *Software re-use*

The topic of re-use is covered in detail in Biggerstaff and Perlis (1989a, b). Advantages of OOP for the development of reusable software are discussed by Meyer (1989), Johnson and Foote (1988), Al-Haddad *et al.* (1991) and Lazerev (1991).

5.5. *Design and analysis of software*

The best text-book introductions to object-oriented design and analysis methods are by Booch (1990) and Rumbaugh *et al.* (1991). de Chanpeaux and Faure (1992) provide a useful comparison of analysis methods. Other articles on this topic are cited in Section 5.

5.6. *Graphics*

The role of object-oriented programming in interactive graphics is hinted at in Johnson *et al.* (1989b). The graphics support of Smalltalk is described in Goldberg (1985). Interviews and ET++ are described in Linton *et al.* (1989) and Weinand *et al.* (1988b), respectively, although, because both are under development, more detailed descriptions are provided in the documentation that accompanies the libraries. Wisskirchen (1990) describes the merger of object-oriented programming with traditional graphical standards.

5.7. *Database management systems*

The relational model of database management was first described by Codd (1970). Kent (1979) gives the disadvantages of the relational model for applications like office information systems and computer-aided design. Good recent reviews of the object-oriented database technology are given by Kim (1990) and Brown (1991), who also provides a comprehensive bibliography of the subject and Kim *et al.* (1989) is a collection of important articles on the emerging technology. For details of the many commercial and research databases see Stonebreaker (1990), Brown (1991) and Joseph *et al.* (1991). The *IEEE Transactions on Knowledge and Data Engineering*, *ACM Transaction on Databases*, *Office Information Systems*, and *Information Systems*, as well as the *SIGMOD Record*, are useful for keeping abreast of new developments.

5.8. *New developments*

Object-oriented programming and paradigm is a rapidly developing field and it is difficult to keep abreast. For up-to-date material, good sources are the *Journal of Object-Oriented Programming*, *OOPS Messenger*, the Proceedings of the annual OOPSLA conferences—which are published in *ACM SIGPLAN Notices*—and proceedings of the European Conferences on Object-Oriented Programming—which are published by Springer in their *Lecture Notes on Computer Science* series. OOP is a regular topic in *Byte* (see the August 1981, June 1985, August 1986, March 1989 and October 1990 issues in particular), and articles on the topic also regularly crop up in the *Communications of the ACM* (see, for example, the September 1990 issue), *Computer* and *IEEE Software*.

Acknowledgements—The authors are pleased to acknowledge the assistance of their colleagues M. Chen, R. Antonov and I. T. Harvey in the development of data models for CACSD on which the examples are based. This is supported by the U.K. Science and Engineering Research Council under grant No. GR/H16360. The pioneering work and ideas of Jan Maciejowski, John Edmunds, Siân Hope, Jim Taylor,

Magnus Rimvall, Mats Andersson and Hans-Dieter Joos in this area have been particularly useful.

We also acknowledge our colleagues on the eXCeS development project who have been highly influential in forming our ideas on GUIs. Finally, we gratefully acknowledge the anonymous reviewers whose careful comments have enabled us to substantially raise the quality of this paper.

REFERENCES

Abelson, H., G. J. Sussman and J. Sussman (1985). *Structure and Interpretation of Computer Programs*. MIT Elect. Comput. Sci. MIT Press, McGraw-Hill, NY.

Ackroyd, M. and D. Daum (1991). Graphical notation for object-oriented design and programming. *J Object-Oriented Program*, **3**, 18–28.

Ada (1988). Ada 9x project report: Ada 9x requirements documents. Technical Report (AD-A226 495). Ada Information Clearing House.

Al-Haddad, H. M., K. M. George and M. H. Samadzadeh (1991). Approaches to reusability in C++ and Eiffel. *J. Object-Oriented Program*, **4**, 34–45.

Andersson, M. (1989). An object-oriented language for model representation. In *Proc. IEEE Control Sys. Soc. Workshop Comput. Aided Control Sys. Des.*, Tampa, FL, pp. 8–15.

Andersson, M. (1990). Omola—an object-oriented language for model representation. Technical Report CODEN:LUFTD2/(TFRT-3208)/1-102/(1990). Department of Automatic Control, Lund Institute of Technology, Sweden.

Andersson, M. (1991). *OmSim and Omola User's Guide*. Version 1.0. Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

Andersson, M. (1992). Discrete event modelling and simulation in Omola. In *Proc. IEEE Control Syst. Soc. Symp. CACSD*, Napa, CA, pp. 238–241.

Andersson, M., S. E. Mattsson and B. Nilsson (1991). On the architecture of CACE environments. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 41–46.

Andrews, T. and C. Harris (1987). Combining language and database advances in an object-oriented development environment. *SIGPLAN Not.*, **22**, 430–440.

Åstrom, K. J. and W. Kreutzer (1986). System representations. In *Proc. IEEE Control Syst. Soc. Third Symp. CACSD*. Arlington, VA.

Atkinson, M., F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier and S. Zdonik (1989). The object-oriented database systems manifesto. Technical Report Altaïr 30–89. GIP ALTAIR. Domaine de Voluceau, BP 105-Rocquencourt, 78153 Le Chesnay Cedex, France.

Barker, H. A. (1988). Graphical environments for computer-aided control system design. In *Prepr. Fourth IFAC Symp. Comput. Aided Des. Control Syst.*, Beijing, pp. 60–66.

Barker, H. A., M. Chen and P. Townsend (1988a). Algorithms for transformations between block diagrams and signal flow graphs. In *Prepr. 4th IFAC Symp. Comput. Aided Control Sys.—CADCS '88*, Beijing, pp. 231–236.

Barker, H. A., M. Chen, P. Townsend and I. T. Harvey (1988b). CES—a workstation environment for computer-aided design in control systems. In *Prepr. 4th IFAC Symp. Comput. Aided Control Sys.—CADCS '88*, Beijing, pp. 248–251.

Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling and P. Townsend (1987). The development of an intelligent man-machine interface for computer-aided design of control systems. In *Prepr. 10th IFAC World Congress Autom. Control*, Munich, pp. 255–260.

Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling and P. Townsend (1989). Development of an intelligent man-machine interface for computer-aided control system design and simulation. *Automatica*, **25**, 311–316.

Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling and P. Townsend (1992). Modern environments for dynamic system modelling. In *Proc. IASTED Int. Conf. Control, Modelling and Simulation*. Innsbruck, Austria.

Barker, H. A., M. Chen, P. W. Grant, C. P. Jobling and P. Townsend (1993). Open architecture for computer-aided control engineering. *IEEE Control Systems*, **12**, 17–27.

Barker, H. A., M. Chen, P. W. Grant, I. T. Harvey, C. P. Jobling, A. P. Parkman and P. Townsend (1991). The making of eXCeS—a software engineering perspective. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 27–33.

Beech, D. (1987). Groundwork for an object database model. In Shriver, B. and P. Wegner (Eds), *Research Directions in Object-oriented Programming*, pp. 317–354. The MIT Press, Cambridge, MA.

Bézivin, J., J.-M. Hullot, P. Cointe and H. Lieberman (Eds) (1987). *Proc. Eur. Conf. Object-Oriented Programming*. Vol. 276 of *Lecture Notes in Computer Science*. Springer, Berlin.

Biggerstaff, T. J. and A. J. Perlis (1989a). *Software Reusability*. Vol. 1. ACM Press, Addison-Wesley, Reading, MA.

Biggerstaff, T. J. and A. J. Perlis (1989b). *Software Reusability*. Vol. 2. ACM Press, Addison-Wesley, Reading, MA.

Birtwhistle, G., O.-J. Dahl, B. Myrhaug and K. Nygaard (1973). *Simula Begin*. Studentliteratur (Lund) and Auerbach (NY).

Björnerstedt, A. and C. Hulté (1989). Version control in an object-oriented architecture. In Kim, W. and F. H. Lochovsky (Eds), *Object-oriented Databases, Concepts and Applications*. ACM Press, Addison-Wesley, Reading, MA, pp. 451–485.

Bobrow, D., D. Fogelsong and M. Miller (1987a). Definition groups: making sources into first-class objects. In Shriver, B. and P. Wegner (Eds), *Research Directions in Object-oriented Programming*, pp. 129–146. The MIT Press, Cambridge, MA.

Bobrow, D. G. and M. J. Stefik (1982). LOOPS: an object oriented programming system for Interlisp. Technical Report, Xerox PARC.

Bobrow, D. G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel (1987b). CommonLoops: merging LISP and object-oriented programming. In Peterson, G. E. (Ed.), *Object-oriented Computing*, Vol. 2, Computer Society Tutorials. IEEE Computer Society Press, Los Alomitos, CA, pp. 169–181.

Bobrow, D. L., G. de Michiel, R. P. Gabriel, S. E. Keene, G. Kiczales and D. A. Moon (1988). Common lisp object system specification. *SIGPLAN Not.* **23**.

Booch, G. (1986). Object-oriented software development. *IEEE Trans. Software Engineering* **SE-12**, 211–221.

Booch, G. (1990). *Object-oriented Design with Examples*. Benjamin-Cummings, Redwood City, CA.

Borenstein, P. and J. Mattson (1989). *Think C User's Manual*. Symantec Corporation, 10201 Torre Avenue, Cupertino, CA.

Boyle, J., M. P. Form and J. M. Maciejowski (1988). A multivariable toolbox for use with MATLAB. In *Proc. 1988 Am. Control Conf.*, Atlanta, GA, pp. 707–718.

Brooks, Jr, F. P. (1987). No silver bullet. *Computer*, **20**, 10–19.

Brown, A. (1991). *Object-oriented Databases: Applications in Software Engineering*. Int. Series Softw. Eng, McGraw-Hill, NY.

Bunch, J., J. Dongarra, C. Molder and G. W. Stewart (1979). *LINPACK User's Guide*. SIAM, Philadelphia, PA.

Calder, P. A. and M. A. Linton (1991). Glyphs: flyweight objects for user interfaces. Technical Report, Center for Integrated Systems, Stanford University, CA.

Cameron, J. R. (1986). An overview of JSD. *IEEE Trans. Software Engineering*, **SE-12**, 222–240.

Cannon, H. I. (1980). Flavors. Technical Report, MIT Artificial Intelligence Laboratory. Cambridge, MA.

Cardelli, L. and P. Wegner (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, **17**, 471–522.

Cellier, F. and H. Elmqvist (1992). The need for formula manipulation in object-oriented continuous system modelling. In *Proc. IEEE Control Syst. Soc. Symp. CACSD*, Napa, CA, pp. 1–8.

Cellier, F., B. P. Zeigler and A. D. Cutler (1991). Object-oriented modelling: tools and techniques for capturing properties of physical systems in computer code. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 1–10.

Cellier, F. E. (1991). *Continuous System Modelling*. Springer, NY.

Checkoway, C., K. Kirk, D. Sullivan and M. Townsend (1992). *SIMULINK–User's Guide*. MathWorks, Inc., Natick, MA.

Chen, M., P. Townsend and C. Y. Wang (1992). A development environment for constructing graph-based editing tools. In *Proc. 1992 Eurographics Conf.*, Cambridge, U.K., pp. 345–355.

Coad, P. and E. Yourdon (1990). *Object-oriented Analysis*. Yourdon Press, NY.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, **13**, 377–387.

Constantine, L. L. and E. Yourdon (1979). *Structured Design*. Prentice-Hall. Englewood Cliffs, NJ.

COOL (1990). *C++ Object-Oriented Library User's Manual*. Texas Instruments, Information Technology Group, Austin, TX.

Courtois, P. (1985). On time and space decomposition of complex systems. *Communications of the ACM*, **28**, 596.

Cox, B. (1986). *Object Oriented Programming—an Evolutionary Approach*. Addison Wesley. Reading, MA.

Cox, B. J. (1984). Message/object programming: an evolutionary change. *IEEE Softw.*, 50–61.

Cox, B. J. (1990). There is a silver bullet. *Byte*, **15**, 209–218.

Dahl, O. and K. Nygaard (1966). SIMULA—an Algol-based simulation language. *Communications of the ACM*, **9**, 671–678.

Davis, A. M. (1988). A taxonomy for the early stages of the software development life-cycle. *J. Syst. Softw.*, **8**, 297–311.

de Chanpeaux, D. and P. Faure (1992). A comparitive study of o–o analysis. *J. Object-Oriented Program.*, **5**, 21–24.

DeMarco, T. (1978). *Structured Analysis and System Design*. Yourdon Press, NY.

De Michiel, L. G. and R. P. Gabriel (1987). The common lisp object system: an overview. In Bèzevin, J., J.-M. Hullot, P. Cointe and H. Lieberman (Eds), *Proc. Eur. Conf. Object-oriented Programming*, Vol. 276 of *Lecture Notes in Computer Science*. Springer, Berlin.

Demmel, J. (1989). LAPACK: a portable linear algebra package for supercomputers. In *Proc. IEEE Symp. CACSD*, Tampa, FL, pp. 1–7.

Diederich, J. and J. Milton (1987). Experimental prototyping in Smalltalk. *IEEE Softw.*, 50–64.

Dijkstra, E. (1965). Programming considered as a human activity. In *Proc. 1965 IFIP Congr.*, Amsterdam, The Netherlands, pp. 213–217.

Dijkstra, E. (1969). Structured programming. In *Rep. NATO Sci. Comm. Conf.*, Rome, Italy; reprinted in Yourdon (1979).

Dittrich, K. R. (Ed.) (1988). *Advances in Object-Oriented Database Systems*. Vol 334 of *Lecture Notes in Computer Science*. Springer, Berlin.

Dongarra, J. J., J. D. Croz, I. Duff and S. Hammarling (1990). A set of level 3 basic linear algebra subrograms. *ACM Trans. Mathematical Software*, **16**, 1–17.

Drescher, G. L. (1985). *The Object LISP User Manual (Preliminary)*. LMI Corporation. Cambridge, MA.

Duff, C. and B. Howard (1990). Migration patterns. *Byte*, **15**, 223–232.

ECM (1990). A reference model for computer-assisted software engineering environments. Technical Report ECMA/TR55, European Computer Manufacturer's Association, Geneva, Switzerland.

Edwards, J. M. and B. Henderson-Sellars (1993). A graphical notation for object-oriented analysis and design. *J. Object-Oriented Program*, **5**, 53–74.

Eldredge, D. L., J. D. McGregor and M. K. Summers (1990). Applying the object-oriented paradigm to discrete event simulations using the C++ language. *Simulation*, **54**, 83–91.

Elmqvist, H. (1978). A structured model language for large continuous systems. Ph.D. thesis, Department of Automatic Control, Lund Institute of Technology, Sweden. Tech. Report Number: CODEN:LUFTD2/(TFRT-1015).

Elmqvist, H. and S. E. Mattsson (1989). Simulation for dynamic systems using graphics and equations for modeling. *IEEE Control Syst., Mag.*, **9**, 53–58.

Expert Ease (1989). *EASE$^+$—User's Guide*. Expert Ease Systems, Inc., Belmont, CA.

Floyd, M. A., P. J. Dawes and U. Milletti (1991). Xmath: a new generation of object-oriented CACSD tools. In *Proc. Eur. Control. Conf.*, Vol. 3, Grenoble, France, pp. 2232–2237.

Fykes, R. and T. Kehler (1985). The role of frame-based representation in reasoning. *Communications of the ACM*, **28**, 904–920.

Garbow, B. S., J. M. Boyle, J. J. Dongarra and C. B. Moler (1976). *Matrix Eigensystem Routines—EISPACK Guide*. Vol. 6 of *Lecture Notes in Computer Science*. Springer, Berlin.

Gibbs, S., D. Tsichritzis, E. Casais, O. Nierstrasz and X. Pintado (1990). Class management for software communities. *Communications of the ACM*, **33**, 90–103.

Gibson, E. (1990). Objects—born and bred. *Byte*, **15**, 245–254.

Goldberg, A. (1985). *Smalltalk-80: the Interactive Programming Environment*. Addison-Wesley, Reading, MA.

Goldberg, A. and D. Robson (1983). *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, Reading, MA.

Gorlen, K. E. (1987). An object-oriented class library for C++. *Softw.—Prac. Exp.*, **17**, 899–922.

Gorlen, K. E., S. M. Orlow and P. S. Plexico (1990). *Data Abstraction and Object-Oriented Programming in C++*. Wiley, NY.

Grant, P. W., C. P. Jobling and C. Rezvani (1992). Some control engineering applications of Prolog. In D. R. Brough (Ed.), *Logic Programming: New Frontiers*. Intellect. Oxford, U.K.

Gregg, W. (1984). The Apple Macintosh computer. *Byte*, **9**, 30–54.

Grumman (1989). *Protoblock—User's Guide*. Grumman Aerospace Corporation, Bethpage, NY.

Gupta, R. W., H. Cheng, R. Gupta, I. Hardonag and M. A. Breuer (1989). An object-oriented VLSI framework. *Computer*, **22**, 28–38.

Guttag, J. V. (1977). Abstract data types and the development of data structures. *Communications of the ACM*, **20**, 396–404.

Hara, S., G. Yamamoto, M. Oshima and R. Kondo (1992). A distributed computing environment for control system analysis and design. In *Proc. IEEE Control Syst. Soc. Symp. CACSD*, Napa, CA, pp. 47–54.

Hara, S., I. Akahori and R. Kondo (1988). Computer aided control system analysis and design based on the concept of object-orientation. In *Prepr. 4th IFAC Symp. Comput. Aided Control Sys.—CADCS '88*, Beijing, pp. 220–225.

Heller, D. (1989). *XView Programming Manual*. Vol. 7 of *The Definitive Guides to the X Window System*. O'Reilly and Associates, Sebastopol, CA.

Henderson-Sellers, B. and J. M. Edwards (1990). The object-oriented systems life-cycle. *Communications of the ACM*, **33**, 142–159.

Hope, S., P. J. Fleming and J. G. Wolff (1990). Designing an object-oriented database and DBMS for a CAD system. In *Proc. Tech. Object-oriented Lang. Syst. (TOOLS2)*, Paris.

Hope, S., P. J. Fleming and J. G. Wolff (1991). Object-oriented database support for computer-aided control system design. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 200–204.

Ingalls, D. H. H. (1981). Design principles behind Smalltalk. *Byte*, **6**, 286–298.
ISI (1989). *SYSTEM-BUILD—User's Guide*. Integrated Systems, Inc., Santa Clara, CA.
Jackson, M. A. (1975). *Principles of Program Design*. Academic Press, London.
Jackson, M. A. (1983). *System Development*. Prentice-Hall, London.
Jacobson, I. (1993). Is object technology software's industrial platform? *IEEE Softw.*, **10**, 24–30.
Johnson, J., T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard and K. Mackey (1989a). The Xerox Star: a retrospective. *Computer*, **22**, 11–30.
Johnson, J., T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard and K. Mackey (1989b). The Xerox Star: a retrospective. *Computer*, **22**, 11–30.
Johnson, R. E. and B. Foote (1988). Designing reusable classes. *J. Object-Oriented Program.*
Joos, H.-D. (1991). Automatic evolution of a decision-supporting design project database in concurrent control engineering. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 113–118.
Joos, H.-D. and M. Otter (1991). Control engineering data structures for concurrent engineering. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 107–112.
Joseph, J. V., S. M. Thatte, C. W. Thompson and D. L. Wells (1991). Object oriented databases: design and implementation. *Proc. IEEE*, **79**, 42–64.
Katz, R. H. (1990). Toward a unified framework for version modelling in engineering databases. *ACM Computing Surveys*, **22**, 375–408.
Kay, A. C. (1977). Microelectronics and the personal computer. *Sci. Am.*
Kent, W. (1979). Limitations of record based information models. *ACM Trans. Database Systems*, **4**, 107–131.
Kilian, M. (1990). Trellis: turning design into programs. *Communications of the ACM*, **33**, 65–68.
Kim, W. (1990). Trellis: turning design into programs. *Communications of the ACM*, **33**, 65–68.
Kim, W. (1990). Object-oriented database systems survey. *IEEE Trans. Knowl. Data. Eng.*, **1**.
Kim, W. and Lochovsky, F. H. (Eds) (1989). *Object-oriented Concepts, Databases, and Applications*. ACM Press.
Kim, W., E. Bertino and J. F. Garza (1989). Composite objects revisited. In *Proc. SIGMOD 1989*; *ACM SIGMOD*, pp. 337–347. Also published in *ACM SIGMOD Record*, Vol. 18, No. 2.
King, R. A. (1986). Database management system based on an object-oriented model. In *Expert Database Conf.*, Charleston, SC.
Koenig, A. (1991). Templates as interfaces. *J. Object-Oriented Program*, **4**, 51–55.
Korson, T. and J. D. McGregor (1990). Understanding object-oriented: a unifying paradigm. *Communications of the ACM*, **33**, 40–60.
Kranser, G. E. and S. T. Pope (1988). A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *J. Object-Oriented Program*, **1**, 26–49.
Kreutzer, W. (1986). *System Simulation: Programming Styles and Languages*. Int. Comput. Sci., Addison-Wesley, Reading, MA.
Lazerev, G. L. (1991). Reusability in Smalltalk: a case study. *J. Object-Oriented Program.*, **3**, 11–20.
Lea, D. (1991). *User's Guide to GNU C + + Library*. 1.39.0 Edn. Free Software Foundation, Cambridge, MA.
Ledbetter, L. and B. Cox (1985). Software Ics. *Byte*, **10**.
Linton, M. A., J. M. Vlissides and P. A. Calder (1989). Composing user interfaces with InterViews. *Computer*, **22**, 8–22.
Liskov, B. H. and S. N. Zilles (1974). Programming with abstract data types. *SIGPLAN Not.*, **9**, 50–59.
Little, J. N., A. Emami-Naeini and S. N. Bangert (1985). CTRL-C and matrix environments for the computer-aided design of control systems. In Jamshidi, M. and C. J. Herget (Eds), *Computer Aided Control Systems Engineering*. North Holland, Amsterdam, pp. 111–124.
Little, J. N. and C. B. Moler (1990). *MATLAB—User's Guide*. MathWorks, Inc., Natick, MA.
Love, T. (1987). Experiences with Smalltalk-80 for application development. In Peterson, G. E. (Ed.), *Object-oriented Computing*, Vol. 1, Computer Society Tutorials. IEEE Computer Society Press, Los Alomitos, Ca, pp. 52–56.
Lozinski, C. (1991). Why I need Objective-C. *J. Object-Oriented Program.*, **4**, 21–28.
MacFarlane, A. G. J., G. Grübel and J. Ackermann (1989). Future design environments for control engineering. *Automatica*, **25**, 165–176.
Maciejowski, J. M. (1984). A core data model for computer-aided control engineering. Technical Report CUED/F-CAMS/TR-257, Cambridge University Engineering Department, Cambridge, U.K.
Maciejowski, J. M. (1988). Data structures and software tools for the computer-aided design of control systems. In *Prepr. 4th IFAC Symp. Comput. Aided Des. Control Syst.—CADCS '88*, Pergamon Press, Beijing, P. R. China, pp. 27–38.
Maier, D. (1989). Making database systems fast enough for CAD applications. In Kim, W. and F. H. Lochovsky (Eds), *Object-oriented Databases, Concepts and Applications*. ACM Press, Addison-Wesley, Reading, MA, pp. 573–582.
Maier, D. and J. Stein (1986). Development of an object oriented DBMS. In *Proc. ACM OOPSLA 1986* (also published in *SIGPLAN Not.* **21**). pp. 472–482.
Mattsson, S. E. (1988). On model structuring concepts. In *Prepr. 4th IFAC Symp. Comput. Aided Control Sys.—CADCS '88*, Beijing, pp. 269–274.
Mattsson, S. E. (1992). Modelling of power systems in Omola for transient stability studies. In *Proc. IEEE Control Syst. Soc. Symp. CACSD*, Napa, CA, pp. 30–36.
May, M., S. Kluge, M. Plessow, J. Sieck and W. Vigerske (1991). A review on automatic block diagram layout. In *Prepr. 5th IFAC Symp. Comput. Aided Control Sys.—CADCS '91*, Swansea, U.K., pp. 158–163.
McIllroy, M. D. (1976). Mass produced software components. In J. M. Buxton, P. Naur and B. Randell (Eds), *Software Engineering Concepts and Techniques, NATO*, pp. 88–98 (reprint Proc. 1968 NATO Conf. Softw. Eng.).
Meyer, B. (1987). Reusability: the case for object-oriented design. *IEEE Softw.*, **4**, 50–64.
Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, NJ.
Meyer, B. (1990). Lessons from the design of the Eiffel libraries. *Communications of the ACM*, **33**, 69–88.
Meyer, B. (1991). *Eiffel: the Language*. Prentice-Hall, Englewood Cliffs, NJ.
MGA (1991). *Advanced Continuous Simulation Language (ACSL): Reference Manual*. Mitchell & Gauthier Associates, Concord, MA.
Miller, J. D. (1990). *An OPEN LOOK at UNIX—a Developer's Guide to X*. M & T Books, Redwood City, CA.
Moler, C. (1980). *MATLAB—User's Guide*. Department of Computer Science, University of New Mexico. Albuquerque, NM.
Moon, D. A. (1986). Object oriented programming with Flavors. *SIGPLAN Not.*, **21**, 1–8.
Moore, R. L., H. Rosenhof and G. Stanley (1990). Process control using a real-time expert system. In *Prepr. 11th IFAC World Congress Autom. Control*, Tallinn, Estonia, pp. 234–239.
Munro, N. (1988). ECSTASY—a control system CAD environment. In *Proc. IEE Int. Conf. Control '88*, Oxford, U.K., pp. 76–80. (Conf. Publication No. 285.)
Nilsson, B. (1992). Object-oriented chemical process modelling in Omola. In *Proc. IEEE Control Syst. Soc. Symp. CACSD*, Napa, California, pp. 165–172.
OPEN LOOK (1989). *OPEN LOOK—Graphical User Interface Application Style Guidelines*. Addison-Wesley, Reading, MA.
OSF (1989). *OSF/Motif—User's and Style Guides*. Revision 1.0. Open Software Foundation, Cambridge, MA.

O'Shea, T., K. Beck, D. Halbert and K. J. Schmucker (1986). The learnability of object-oriented programming systems. In OOPSLA, *SIGPLAN Not.*, pp. 502–504.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, **15**, 1053–1058.

Pascoe, G. A. (1986). Elements of object-oriented programming. *Byte*, **11**.

Peterson, G. E. (Ed.) (1987a). *Object-Oriented Computing*, Vol. 1, Computer Society Tutorials. IEEE Computer Society Press, Los Alomitos, CA.

Peterson, G. E. (Ed.) (1987b). *Object-Oriented Computing*, Vol. 2, Computer Society Tutorials. IEEE Computer Society Press, Los Alomitos, CA.

Phaal, P. (1986). A Smalltalk environment for computer-aided control system design. In *Proc. IEEE Control Syst. Soc. 3rd Symp. Computer-Aided Control Syst. Des.*, Arlington, VA, pp. 19–24.

Phaal, P. (1987). An object oriented environment for control system design. Ph.D. thesis, University of Cambridge, Cambridge, U.K.

Pinson, L. J. and R. S. Wiener (1988). *An Introduction to Object-Oriented Programming and Smalltalk*. Addison-Wesley, Reading, MA.

Rentsch, T. (1982). Object-oriented programming. *SIGPLAN Not.*, **17**, 51–57.

Rimvall, M. (1987). CACSD software and man–machine interfaces of modern control environments. *Trans. Inst. Meas. Control*, **9**.

Rimvall, M. (1988). Interactive environments for CACSD software. In *Prepr. 4th IFAC Symp. Computer Aided Design in Control Systems—CADCS '88*, Beijing, pp. 17–26.

Rimvall, M., H. Sutherland, H. H. Taylor and P. J. Lohr (1989). GE's MEAD user interface—a flexible menu- and forms-driven interface for engineering applications. In *Proc. IEEE Control Syst. Soc. Workshop Comput.-Aided Control Syst. Des.*, Tampa, FL, pp. 24–34.

Robson, D. (1981). Object-oriented software systems. *Byte*, **6**, 74–86.

Rosenbrock, H. H. (1972). The use of computers for designing control systems. *Meas. Control*, **5**, 409–412.

Rumbaugh, J. (1993). Disinherited! Examples of misuse of inheritance. *J. Object-Oriented Program* **5**, 22–24.

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

Schmucker, K. J. (1986). *Object-Oriented Programming for the Macintosh*. Hayden, Hasbrouck Heights, NJ.

SCT (1987). *Model-C—User's Guide*. Systems Control Technology, Inc., Palo Alto, CA.

Shearer, J. L., A. T. Murphy and H. H. Richardson (1971). *Introduction to System Dynamics*. Addison-Wesley, Reading, MA.

Sheifler, R. and J. Gettys (1986). The X window system. *ACM Transactions on Graphics*, **5**, 79–109.

Simon, H. (1982). *The Science of the Artificial*. MIT Press, Cambridge, MA.

Skarra, A. and S. Zdonik (1987). Type evolution in an object-oriented database. In Shriver, B. and P. Wegner (Eds), *Research Directions in Object-oriented Programming*, The MIT Press, Cambridge, MA. pp. 393–415.

Smith, B. T., J. M. Boyle, J. J. Dongarra, B. S. Garbow and Y. Ikebe (1977). *Matrix Eigensystem Routines—EISPACK Guide Extension*. Vol. 51 of *Lecture Notes in Computer Science*, Springer, Berlin.

Smith, Irby, Kimball, Verplank and Harslem (1982). Designing the star user interface. *Byte*, **7**, 242–282.

Smith, K. E. and S. B. Zdonik (1987). Intermedia: a case study of the differences between relational and object-oriented database systems. In *Proc. ACM OOPSLA 1987*, Los Angeles, CA, pp. 452–465, *SIGPLAN Not.*, **22**.

Snyder, A. (1993). The essence of objects: concepts and terms. *IEEE Softw.*, **10**, 31–42.

Spang III, H. A. (1984). The federated computer-aided control design system. *Proc. IEEE*, **72**, 1724–1731. Special Section on Computer-Aided Design of Control Systems: Systems and Algorithms.

Spang III, H. A. (1985). The federated computer-aided control design system. In Jamshidi, M. and C. J. Herget (Eds), *Computer Aided Control Systems Engineering*. North Holland, Amsterdam, pp. 209–228.

Stefik, M. and D. G. Bobrow (1986). Object-oriented programming—themes and variations. *AI Mag.*, **6**, 40–62.

Stone, C. M. (1992). The object management group. *Byte*, **17**, 125.

Stone, C. M. and D. Hentchel (1990). Database wars revisited. *Byte*, **15**, 233–242.

Stonebreaker, M. (1990). Special issue on database prototype systems. *IEEE Trans. Knowl. Data. Eng.*, **2**, 1–3.

Stroustrup, B. (1986). *The C++ Programming Language*. Addison-Wesley, Reading, MA.

Stroustrup, B. (1988a). Parameterized types for C++. In *Proc. USENIX C++ Conf.*, USENIX Association, Berkley, CA.

Stroustrup, B. (1988b). What is object-oriented programming? *IEEE Softw.*, **5**, 10–20.

Sun (1991). *Open Windows—User's Guide*. Version 3.0. Sun Microsystems, Inc., Mountain View, CA.

Sutherland, I. (1963). Sketchpad: a man–machine graphical communication system. *AFIPS*, **23**, 329–346.

Tan, C. Y. and J. M. Maciejowski (1989). DB-Prolog: a database programming environment for CACSD. *Proc. IEEE Control Syst. Soc. Workshop Comput.-Aided Control Syst. Des.*, Tampa, FL, pp. 72–77.

Taylor, J. H. and D. K. Frederick (1984). An expert system architecture for computer-aided control engineering. *Proc. IEEE*, **72**, 1795–1805. Special Section on Computer-Aided Design of Control Systems: Systems and Algorithms.

Taylor, J. H., D. K. Frederick, C. M. Rimvall and H. Sutherland (1989). The GE MEAD computer-aided control engineering environment. In *Proc. IEEE Control Syst. Soc. Workshop Comput.-Aided Control Syst. Des.*, Tampa, FL, pp. 16–23.

Taylor, J. H., K.-H. Neh and P. A. Mroz (1988). A data-base management scheme for computer-aided control engineering. In *Proc. 1988 Am. Control Conf.*, Atlanta, GA, pp. 719–724.

Thomas, I. (1989). PCTE interfaces: supporting tools in software engineering. *IEEE Softw.*, **6**, 15–23.

Udell, J. (1990). Three's the one. *Byte*, **15**, 122–128.

Unger, B., A. Dewar, J. Cleary and G. Birtwistle (1986). The jade approach to distributed software development. In E. J. H. Kerckhoffs, G. C. Vansteenkiste and B. Zeigler (Eds), *A. I. Applied to Simulation*, Vol. 18 of *Simulation Series*.

Vangheluwe, H. L. M., G. C. Vansteenkiste, J. D. Verweij and E. J. H. Kerckhoffs (1989). The object oriented paradigm applied to continuous system simulation. In *Proc. 3rd Eur. Simulation Conf.*, Edinburgh, pp. 163–169.

Vlissides, J. M. and M. A. Linton (1990). Unidraw: a framework for building domain-specific graphical editors. *ACM Trans. Info. Syst.*, **8**, 237–268.

Vlissides, J. M. and S. Tang (1991). A Unidraw-based user interface builder. Technical Report, Center for Integrated Systems, Stanford University, CA.

Walker, R. A., S. C. Shah and N. K. Gupta (1984). Computer-aided engineering for system analysis. *Proc. IEEE*, **72**, 1731–1745.

Wall, L. and L. Schwartz (1990). *Programming Perl*. Nutshell Guides. O'Reilly & Associates, Sebastopol, CA.

Wegner, P. (1984). Capital intensive software technology. *IEEE Softw.*, **1**.

Wegner, P. (1989). Learning the language. *Byte*, **14**, 245–253.

Wegner, P. (1990). Concepts and paradigms of object-oriented programming. *OOPS Messenger*, **1**, 7–61.

Weinand, A., E. Gamma and R. Marty (1988a). ET++—an object-oriented framework in C++. In *Proc. ACM OOPSLA 1988*, pp. 46–57. *SIGPLAN Not.*, 239.

Wiecha, S. C., W. Bennett, S. Boies, J. Gould and S. Greene (1990). ITS: a tool for rapidly developing interactive applications. *ACM Trans. Info. Syst.*, **8**, 204–236.
Williams, G. (1983). The Lisa computer system. *Byte*, **8**, 33–50.
Wilson, D. A., L. S. Rosenstein and D. Shafer (1987). *Programming with MacAPP*. Macintosh Inside Out. Addison-Wesley. Reading, MA.
Wirfs-Brock, R. and B. Wilkerson (1990). *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs, NJ.
Wirfs-Brock, R.J. and R. E. Johnson (1990). Surveying current research in object-oriented design. *Communications of the ACM*, **33**, 105–124.
Wirth, N. (1971). The programming language Pascal. *Acta Informatica*, **1**, 35–63.
Wisskirchen, P. (1990). *Object-oriented Graphics: from GKS and PHIGS to Object-oriented Systems*. Symb. Comput.: Comput. Graph.-syst. and Applic., Springer-Verlag, Berlin.
Xerox (1981). The Smalltalk-80 system. *Byte*, **6**, 36–48.
Young, D. A. (1989). *X Window Systems Programming and Applications with Xt*. Prentice-Hall, Englewood Cliffs, NJ.
Yourdon, E. N. (Ed.) (1979). *Classics in Software Engineering*. Yourdon Press, New York, NY.
Yourdon, E. N. (Ed.) (1982). *Writings of the Revolution: Selected Readings on Software Engineering*. Yourdon Press, New York, NY.
Zeigler, B. (1984). *Multi-facetted Modelling and Discrete Event Simulation*. Academic Press, New York, NY.
Zygmont, A. (1987). Object-oriented programming and CACSD. In *Proc. ASEE Annual Conf.*, ASEE, pp. 648–652.

APPENDIX A: THE OBJECT MODELLING TECHNIQUE

As control engineers, the readers of *Automatica* will be familiar with the modelling of engineering entities by diagrams. A particular appeal of the object-modelling technique (OMT) is that it supports the modelling of object-oriented software systems with a graphical notation that is concise, elegant and simple to understand. The authors have found other notations, such as those of Booch (1990), Meyer (1988) and Ackroyd and Daum (1991) less intuitive. In this appendix, those parts of the OMT notation would be mapped into a programming language; the reader is directed to the textbook (Rumbaugh *et al.*, 1991).

The OMT models a software system from three related but different viewpoints. These are:

- the object model which describes the structure of objects in a system;
- the dynamic model which describes the timed behaviour of the system and is modelled by state diagrams; and
- the functional model which is concerned with the transformation of values and is based on a data-flow diagram.

Only the object-model is covered here.

A.1. *The object model*

"An object model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize each class of objects." The object model closely corresponds to the real world and as such is more resilient to change. The object model is also an intuitive graphical description of a system that is valuable for communicating with customers and for documenting the structure of a system.

The two principle components of the object model are *objects* and *classes*. An object is a concept, abstraction or "thing" with well-defined boundaries and a meaning within the context of the application in which it exists. A close analogy is with a component, say a compensator or actuator, of a control system. Every object has its own identity: *the* PID Compensator in the tank level control loop, and is distinct from similar objects. A *class* describes a group of objects with similar properties (attributes), common behaviour (operations), common relationships to other objects and common semantics. Thus a class is the analogue of a component catalogue, describing the properties and behaviour of a PID Compensator, of which the PID *Compensator in the tank level control loop* is an actual example (instance).

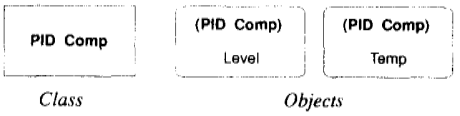

FIG. A.1. Class and objects.

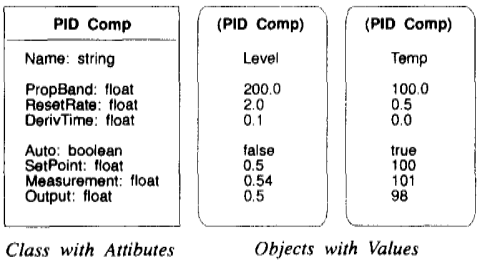

FIG. A.2. Attributes and values.

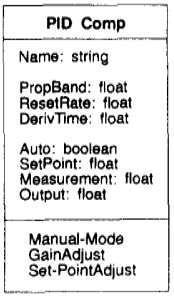

FIG. A.3. Operations.

A.1.1. *Objects and classes.* The OMT uses diagrams to illustrate classes and objects. For example, Fig. A.1 shows the class PID Comp of PID compensators as a rectangle, and two instances of the class, the objects Level and Temp, as rounded boxes. A boldface label is used to indicate the name of a class; the label is enclosed in parentheses in the picture of an object.

A.1.2. *Attributes.* These are data values held by the objects in a class; *name, proportional band, reset rate* are all attributes of the class PID Comp. Each attribute has a value for each object instance. For example, in Fig. A.2, *proportional band* has value 200% in object Level. Different objects may have the same or different values for a given attribute in a model diagram.

In the OMT notation, attributes are listed in the second part of a class box, while in an object box, attribute values are given.

A.1.3. *Operations.* These are functions that may be applied to or by objects in a class. Operations are listed in the lower third of a class box (see Fig. A.3). Operations, which apply to all objects of a given class, are not shown in an object diagram.

A.2. *Links and associations*

A *link* is a physical or conceptual connection between object instances; it is an instance of an *association*. An association describes a group of links with common structure and common semantics. For example, a compensator in a control system is *connected to* an actuator, the notion

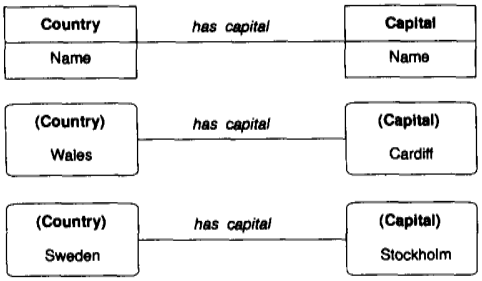

FIG. A.4. Links and associations.

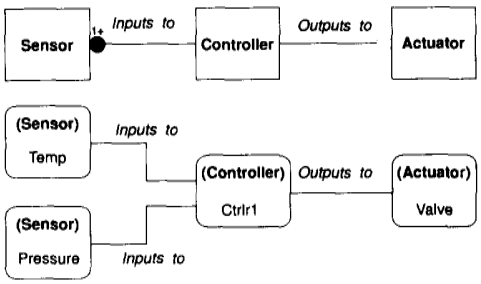

FIG. A.5. Associations with multiplicity.

"connected to" is an association between compensator and actuator classes.

A simple one-to-one association between countries and their capital cities is shown in Fig. A.4. The association is a line drawn between two classes, a link is the corresponding line drawn between two classes. The name of the association or its corresponding links is indicated by a label in italics shown above the line.

A.2.1. *Multiplicity*. This specifies how many instances of a class may relate to a single instance of a related class. In the OMT notation, one-to-one associations are indicated by single lines, an open circle at the end of an association represents zero-or-one, and a filled circle zero-or-more. Thus a control system may be modelled with the associations shown in Fig. A.5. The diagram may be interpreted to mean that a *controller* is *connected to* zero-or-more sensors and zero-or-one actuator. Sometimes a number (such as 1+) is appended to the circle to make the multiplicity explicit. Thus in the figure, the notation actually states that one-or-more (at least one) sensors are connected to a controller in a control system.

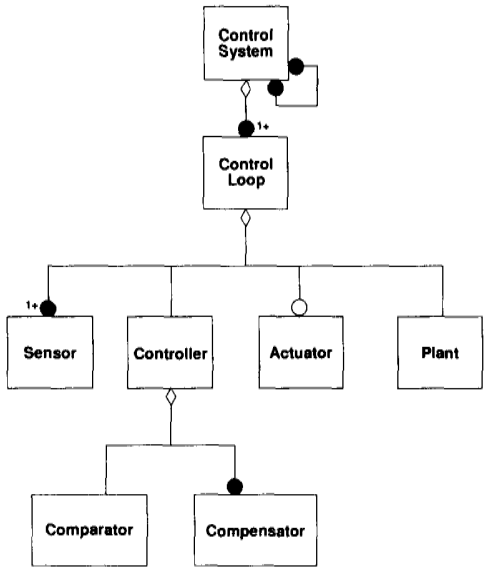

FIG. A.6. Aggregation.

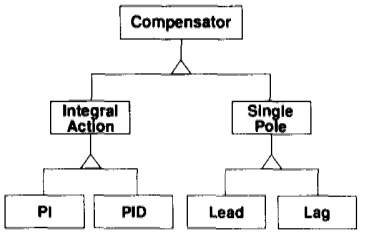

FIG. A.7. Generalization and inheritance.

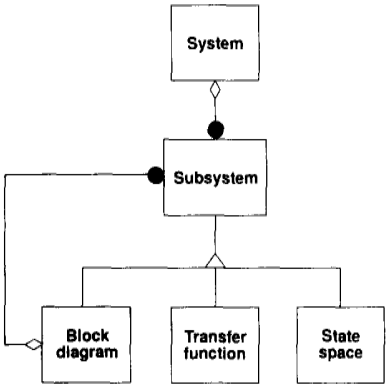

FIG. A.8. Recursive aggregation.

A.2.2. *Aggregation*. This is a special form of association used to show the "is part of" relationship between a component and its subcomponents. It is drawn like an association except that a small diamond indicates the assembly end of a relationship. Fig. A.6 shows a control system which consists of one-or-more *sensors*, a *controller*, zero-or-more *actuators* and a *plant*. The controller itself consists of a *comparator* and zero-or-more *compensators*. Use of the object model makes it easy to visualize the levels in a parts hierarchy.

A.3. *Generalization and inheritance*

Generalization is the relationship between a class and one or more refined versions of the class. For example, a PID *compensator* is a refinement of a *dynamic compensator* which itself is a refinement of a compensator. The class being refined is called the *superclass*, the refined class is called the *subclass*. The subclass *inherits* all the attributes of its superclass, but may redefine or over-ride some. The indicator for generalization or inheritance is a triangle connecting a superclass to its subclasses as shown in Fig. A.7. Words written next to the triangle indicate which properties of an object are being abstracted by a particular generalization.

A.4. *Recursive aggregation*

Recursive aggregation is used when a subsystem may itself be part of a subsystem. For example, Fig. A.8 illustrates a possible description of a system that might form the basis of a CACSD environment. It indicates that a *system* may contain zero-or-more *subsystems*. A *subsystem* may be specialized through inheritance into *transfer function*, *state-space*, and *block diagram* subsystems. A block diagram may itself contain zero-or-more subsystems.

APPENDIX B: GLOSSARY OF TERMS

This is a glossary of terms used in the literature of object-oriented systems. It is derived mainly from terms used in the Smalltalk and Loops languages but these terms have come into widespread use. Sources: Robson (1981) and Stefik and Bobrow (1986).

Abstract class—a class which has no instances, see metaclass.

Abstract data type—a formal computer science term for a type definition which describes both the data elements contained in a data type and also a complete description of the functions which enable the data type to be used. A computer language or system that provides abstract data types should enable new data types to be created and used in the same way as the built-in data types provided by the language or system.

Abstraction—see *data abstraction.*

Aggregate association—in Rumbaugh and colleague's OMT, an aggregate association is a mechanism by which a class or object becomes part of another class. An aggregation hierarchy indicates how subsystems in a software application are combined to create systems.

Association—an association is a relationship between classes on the same level of an aggregation hierarchy. It indicates inter-component communications.

Attribute—a synonym for *instance variable.*

Binding—a term used in describing the means by which a given object-oriented language or system associates methods to objects. See *static binding, dynamic binding, early binding, late binding.*

Class—for most purposes, the terms *class* and *type* are synonymous, but the term *class* is most often used in the object-oriented literature. A class is a description of one or more similar objects. For example, HumanBeing is a description of the structure and behaviour of all human beings. Tony and Chris are concrete instances of the class and have their own particular characteristics. In most object-oriented programming languages, classes describe the instance variables and methods of their instances as well as the position of the class in the inheritance lattice.

Class inheritance—when a class is placed in the class lattice, it inherits variables and methods from its superclasses. This means that any variable that is defined higher in the class lattice will also appear in instances of this class. If a variable is defined in more than one place, the over-riding value is determined by the inheritance order. In many systems, the inheritance order is depth first up to joins, and left-to-right in the list of superclasses.

Class variable—a class variable is a variable stored in the class whose value is shared by all instances of the class. Compare with *instance variable.*

Concrete class—a class which has instances.

Concurrency—the ability for parts of a computational process to be executed in parallel.

Constructor function—in C++ terminology, a constructor function is a function that creates an instance of a class.

Container class—a class whose instances contain other objects.

Contains—a relationship which is orthogonal to the inheritance relationship. It is used in parts hierarchies.

Controller function—an interface function (method) that causes a change of the internal state of an object.

Controller-observer function—an interface function that causes a change of the internal state of an object and also causes the state to be reflected on the output stream of an object.

Data abstraction—the principle that programs should not make assumptions about implementations and internal representations. A *data type* is characterized by operations on its values. In object-oriented programming the operations are methods of a class. The class represents the data type and the values are its instances.

Data hiding—see *information hiding.*

Delegation—a technique for forwarding a message to be handled by another object.

Deferred class—an Eiffel term for a class where at least one method must be deferred in a subclass.

Dynamic binding—the ability to associate a method to some data at run-time.

Early binding—a description of the form of *static binding* that is typical in strongly typed compiled languages. Functions are bound to data at compilation time and the links so created cannot be modified at run-time. Compare with *late-binding* and *dynamic binding.*

Encapsulation—a technique of grouping state and behaviour within a well-defined boundary.

Exception handling—the ability of a software system to respond sensibly to errors, often by exiting gracefully or by calling user defined exception handlers.

Functional decomposition—decomposition of software according to the functions it contains.

Garbage collection—the automatic removal of data objects from computer memory once they become unwanted.

Generic class— a class who's instance variables are defined by a type parameter.

Genericity—the ability to provide a type as a parameter in a definition of an operation. The definition then serves as a template which is replaced with actual code at compile time when the actual type is specified.

Information hiding—the principle by which the implementation of a software component is hidden from users of the component, such that the implementation may be changed without effecting the object's interface, or any other object that depends on it.

Instance—the phrase *instance* of describes the relationship between an object and its class. The methods and structure of an instance are determined by its class. In some programming languages, particularly LISP extensions and Smalltalk, all objects (including classes) are instances of some class. The noun *instance* refers to objects that are not classes.

Instantiate—to make a new instance of a class.

Instance variable—instance variables are variables for which local storage is available in instances. This is in contrast to class variables, which have storage only in the class. Instance variables are sometimes called *slots.*

Interface function—a function that provides control access to an object. A synonym for method.

Late binding—the means by which polymorphism is implemented in some strongly typed object-oriented systems. Compilation merely ensures that a function call is valid and functions are bound to data by table look-up at run-time. Contrast with *early-binding* and *dynamic binding.*

Lattice—a lattice is a directed acyclic graph that describes the class inheritance hierarchy in an object-oriented system. In single inheritance systems, the lattice reduces to a tree. By not allowing cycles, the possibility that a class may have itself as a superclass (even indirectly) is eliminated.

Message—the specification of an operation to be performed on an object. Similar to a procedure call, except that the operation is named indirectly through a *selector* whose interpretation is determined by the class of an object, rather than a procedure name with a single interpretation.

Message receiver—the object to be manipulated, according to a message.

Message sender—the object requesting a manipulation.

Metaclass—this term is used in two ways: as a relationship applied to an instance, it refers to the class of the instance's class; as a noun it refers to a class all of whose instances are classes. A metaclass is sometimes taken to mean a machine readable description of a class definition which can be accessed at run-time.

Method—the function that implements the response when a message is sent to an object.

Object—the primitive element in object-oriented programming. Objects combine the attributes of procedures and data. Objects store data in variables, and respond to messages by carrying out procedures (or methods).

Object persistence—the ability for an object to survive the application that created it. Usually provided by writing an object to disk or by use of a database.

Observer function—an interface function whose effect is to cause the state of an object to be indicated on the object's output without effecting the state.

Operation—a synonym for method.

Parameterized types—C++ term for genericity.

Polymorphism—the capability for different classes of objects to respond differently to exactly the same protocols. Protocols enable a programmer to treat uniformly objects

that arise from different classes. A critical feature is that even when the same message is sent from the same place in code, it can invoke different messages.

Protocol—a standardized set of messages for implementing something. Two classes which implement the same set of messages are said to follow the same protocol.

Selector—the selector in a message specifies the kind of operation the receiver of a message should perform. See also *message*.

Signature—the pattern used to select the particular variation of a function that is needed to perform a polymorphic operation on a particular data type or collection of datatypes.

Slot—see *instance variable*.

Specialization—the process of modifying a generic thing for a specific use.

Static binding—the indication that a computer language or system will bind function calls to data at compilation time.

Structured programming—the programming paradigm first described by Dijkstra (1965) by which programs are written using only certain well-defined programming structures, goto's are avoided, and functions become the key to program decomposition.

Subclass—a class that is lower in the inheritance lattices than a given class.

Superclass—a class that is higher in the inheritance lattices than a given class.

Virtual function—C++ terminology for a method which may be refined in a subclass.