SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 22(4), 287–304 (APRIL 1992)

Object-oriented Implementation Issues in an Experimental CAD System

WAYNE WOLF
Department of Electrical Engineering, Princeton University, Princeton, NJ 08544-5263, U.S.A.

SUMMARY

This case study of object-oriented program design illustrates two limitations of object-oriented programming languages. Existing object-oriented languages do not have good facilities to support two key program design problems: the definition of composite objects, or data structures that include sets of related subobjects; and the specification and run-time management of temporary data structures required to implement efficient algorithms. Both composite objects and temporary data structures are important to the construction of a wide variety of programs. We use the design and implementation of an interactive computer-aided design system to describe how the limitations of present object-oriented languages complicate the design of composite objects and temporary data structures.

INTRODUCTION

Object-oriented programming languages provide powerful programming constructs. However, the major object-oriented mechanisms—data abstraction, inheritance and run-time method determination—do not solve all program design problems. In using C++ to build Archtool, an interactive computer-aided design system, we encountered important program design problems that were not handled by the language. The problems we encountered are not peculiar to C++—they are inherent in the limitations of object-oriented programming mechanisms. This paper describes two key design problems—specification and management of temporary data structures and definition of composite objects—and how we solved those problems using the available object-oriented language mechanisms.

Archtool is an interactive optimization system for hardware designs described as finite-state machines (FSMs) and networks of FSMs. In addition to reading and writing descriptions of FSM networks, it allows the user interactively to apply a number of standard automata theoretic algorithms[1]—minimization, decomposition, etc.—and new, experimental algorithms. It was designed to support research into algorithms for hardware synthesis using an FSM network model.[2] Archtool was implemented in C++ using object-oriented techniques.

Archtool exposed two implementation problems that are particularly hard to solve using existing object-oriented languages. One is how to represent composite objects—complex data structures that include sets of subobjects. Composite objects

0038–0644/92/040287–18$05.00
Received 6 July 1990
Revised 3 May 1991

occur whenever a data structure includes sets of varying size; the complete data structure cannot be described by a single class, but only by a collection of related classes. The design of composite objects classes is hard because the definitions of the component classes are closely related, and the program designer would like help in keeping track of the relationships between classes which form the composite class. The other is how to implement temporary data structures used by algorithms that operate on the objects. Many programs, particularly interactive programs and compilers, build data structures that are used during only one phase of program execution. These temporary data structures are critical to efficient execution. The temporary data structures often have complex relationships to the permanent data structures—for example, an intermediate compiler phase may need links to the global symbol table. The program designer needs help managing the links between the temporary and permanent data structures, especially in allocating and deallocating heap data.

Object-oriented languages offer little support for either of these problems. Both problems have been the subject of sporadic attack by researchers who need to solve them for their own software design projects, but they have not received concentrated attention in the literature. This paper describes why composite objects and temporary data structures arise in practice, why they are hard to implement in existing object-oriented languages, and the best solutions we could find to these problems using the available language mechanisms.

BACKGROUND

Programs that manipulate automata

Archtool was designed to support research into hardware synthesis/compilation by providing a framework for the development and testing of automata-theoretic algorithms. As with programming language compilation, research includes both the division of compilation into manageable tasks and the development of algorithms to solve those compilation tasks. As a result, Archtool had to be flexible in its internal code design and user interface:

1. The program architecture was designed to support the easy introduction of new algorithms; object-oriented methods were used to build up a library of utility functions for finite-state machines and networks.
2. The user interface was designed for easy experimentation with the effects of applying different sequences of optimization algorithms to example designs.

Papert apparently built an interactive system for experimentation with finite-state machine algorithms in the 1960s.[3] Archtool's user interface is patterned after the misII logic optimization system:[4] the user reads in a design using input utilities; the user can interactively apply a variety of optimization algorithms and commands to study the design's current state; the design can be written to a file using output utilities if desired.

Object-oriented programming and C++

Since descriptions of object-oriented languages use varying terms for similar concepts, we need to establish a vocabulary for some basic terms, as well as review C++ syntax. Figure 1 gives a simple example of C++ class definition and use. C++ allows a class definition to be divided into *public, protected* and *private* parts. (This example does not include a protected part; the declarations before the public: token are private.) A private data slot or function is accessible only to functions which are members of the class; a public data slot or function is globally visible. We interchangeably call a function which belongs to a class a *member function* or *method.* The syntax color_circle::color indicates the color method belonging to the class color_circle.

The class definition for color_circle shows that it is derived from the class circle.

```
class circle {
   int r;
   point cen;
 public:
   circle(int radius, point center) { r = radius; cen = center; }

   int area() { return pi * r * r; }
   virtual String msg() { return "circle"; }
};

class color_circle : public circle {
   enum colors colr;
 public:
   y(int radius, point center, enum colors mycolor) :
      (radius,center) { colr = mycolor; }

   int color() { return colr; }
   String msg() { return "colored circle"; }
};

// declarations
point ccenter(5,6);
circle circle1(3,ccenter);
color_circle circle2(2,ccenter,red);
// print statements illustrating virtual function
cout << "circle1 msg = " << obj1.msg() << "\n";
cout << "circle2 msg = " << obj2.msg() << "\n";
```

Figure 1. An example of C++ code

The declaration public circle indicates that all public members of circle are also public in color_circle. We call circle the *base* class and color_circle the *derived* class.

A C++ class also supplies *constructors* and *destructors* for objects. A constructor is applied to an object when it is created on the stack or heap; similarly, a destructor is applied when the object disappears. The function circle () is the constructor for the color class; these simple classes do not need non-standard destructors, but the destructor for the circle class is named ~circle (). The argument (radius, center) in the color_circle() constructor definition passes an argument to the base class's (circle in this case) constructor.

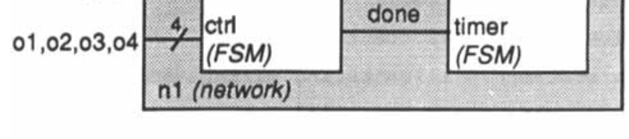
By default, the function body to be invoked on an object is determined at compile time by the object's class. A *virtual function* chooses the body to be invoked at run time. In the example, the msg() function is defined by circle to be a virtual function; circle supplies one function body to be executed for circle-class objects, whereas color_circle supplies a different function body. The calls to msg() in the print statements at the end of the code fragment invoke two different function bodies: the first call prints circle whereas the second prints colored circle.

ARCHTOOL'S FUNCTIONS

Archtool supports three types of components: *finite-state machines* (FSMs), *networks* of FSMs and *black boxes*. An FSM is specified by a state transition table; a network is specified by a list of the components it contains, which may be FSMs, black boxes or other networks, and a list of the nets which specify connections between pins on the components. Figure 2 shows a network built from two FSMs, along with part of the state transition table which specifies one of the FSMs. A black box has no internal structure—it is used to represent a special-purpose component whose structure will not be manipulated by the program. Figure 3 shows a network built from several

input	present	next	output
00	s1	s1	11-
01	s1	s3	010
1-	s1	s4	000
00	s2	s3	10-
...			

finite-state machine



network

Figure 2. Archtool's FSM and network component types

components, including a black box, and shows a network used as a component in a larger network.

Archtool's user interface repeatedly reads a command from the user, parses it, then determines which functions to call to execute the command. Some commands are implemented by the interface itself; others are passed to the *current component.* The interface keeps a list of all the components that have been defined and a pointer to the current component. Commands executed directly by the user interface include

(a) reading a new top-level component from a file
(b) setting the current component
(c) displaying the current list of top-level components
(d) exiting the program.

Most commands are executed by the current component, using functions supplied by the FSM, network and black box classes. Some commands are executed by all component types:

(a) important statistics (which statistics are important vary by the component type)
(b) writing to a file
(c) writing to the interactive output device.

Beyond these, each component type defines its own special commands. The user need not know where the commands are implemented, but an understanding of the logical model of the system helps one understand the command set.

Figure 4 shows a session in which the user works on a single FSM. The first two commands are executed by the user interface: the first reads the component from a file and gives the component the internal name x; the second sets the current component to x (with the leading slash indicating that the component name is absolute, not relative to the current component). The remaining commands, except for quit, are executed by the current component: statistics and display are defined to give relevant information about the state transition graph and to print it to the interactive output, respectively; minimize performs a standard automata-theoretic

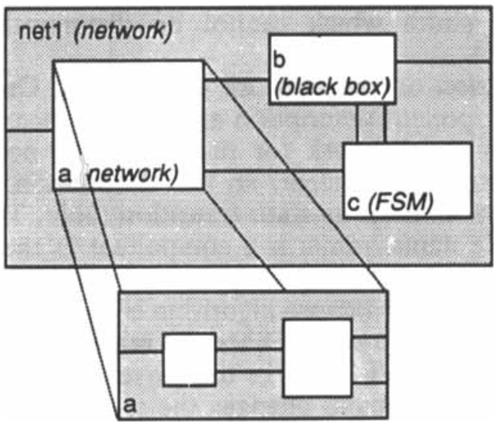


Figure 3. A network and subnetwork

```
% archtool
archtool> read fsm x elclp.fsm
archtool> current /x
archtool> statistics
# states = 21
# transitions = 30
weighted transition cost = 5.71429
archtool> display
.i 3
.o 4
.s 21
.ilb i1 i2 i3
.ob o1 o2 o3 o4
--- {s1.t1} {s2.t1} 1000
--- {s1.t2} {s2.t1} 1000
0-- {s3.t1} {s4.t2} 001-
...
archtool> minimize
archtool> statistics
# states = 17
# transitions = 26
weighted transition cost = 5.14286
archtool> quit
Happy trails
To you,
Until we meet again...
```

Figure 4. Using Archtool on a single FSM

algorithm to combine states which cannot be distinguished by the machine's input/output behaviour.[1]

Figure 5 shows a session on the network of Figure 2. Once again, the first two commands read the component description and set the current component, and the display command prints the network for the user. The network is made of two components of type fsm, ctrl and timer, an fsm declaration gives the component's name and the file which contains its state transition table. The current command in this case takes a relative name—timer is a component in the current network. The up command sets the new current component to be the component which contains the old current component. The mincollapse algorithm is an algorithm for simultaneously generating the cross product of a set of FSMs and minimizing the resulting machine;[5] the * argument specifies that all FSMs in the current network are to be collapsed together. The mincollapse command changes the type of the current component—display now shows the current machine to be an FSM.

```
% archtool
archtool> read network nl loop.elsl
archtool> current /nl
archtool> display
(sub=nl) (i=il,i2,i3) (o=ol,o2,o3,o4) {
(fsm=ctrl,nlctrl.fsm) (i=done,il,i2,i3) (o=ol,o2,o3,o4,reset);
(fsm=timer,nltimer.fsm) (i=reset) (o=done);
}
archtool> current timer
archtool> statistics
# states = 3
# transitions = 6
weighted transition cost = 2
archtool> up
archtool> mincollapse *
archtool> display
.i 3
.o 4
.s 15
.ilb il i2 i3
.ob ol o2 o3 o4
--- {sl}-X-{tl,t2,t3} {s2}-X-{tl} 1000
--- {s2}-X-{tl} {s2}-X-{t2} 0000
0-- {s3}-X-{tl} {s4}-X-{t2} 001-
...
```

Figure 5. Using Archtool on an FSM network

ARCHTOOL'S IMPLEMENTATION

This section reviews the basics of Archtool's implementation before proceeding to the major problems encountered during Archtool's design.

Each component type is represented by a class, and each component instance is represented by an object. Types have a hierarchy represented by the class hierarchy—the fsm, network and black_box classes are all derived from the component class. This class hierarchy is only of interest to the program designer, since the user cannot define new hardware types.

Components have an *ownership hierarchy* that is manipulated by the user. For example, in the example of Figure 2, the network component nl contains two components of class fsm: ctrl and timer. These two FSM components share methods but have distinct data. The user must be able to traverse the ownership hierarchy to examine and apply algorithms to individual components. Archtool maintains the ownership hierarchy itself through *owner* links—pointers from to the component which uses the given component.

Many electronic design capture tools, such as schematic and layout editors,[6,7] require dynamic definition of component types—the user can use existing components to build a new component type. Design capture systems such as Fred[8] and Droid,[9] which are built on top of Lisp-based object-oriented languages, can implement component types directly as language classes. Most design editors do not base component types on the programming language's typing system, but instead keep data structures which record type definitions. Such an implementation is in effect a small language interpreter for component types; the difficulty of implementing a full-blown dynamic type definition system usually restricts these type definition interpreters to relatively simple features. If C++ classes are to be used to define component types, and if new component types can be defined at run time, then the run-time system must include a C++ interpreter which allows new classes to be defined. An interpreter which supports run-time class definition is complex—the arbitrary addition of classes may change the run-time-dispatched functions of other classes.

Archtool was designed for the analysis and algorithmic manipulation of designs, not design capture. As a result, it works on standard component types and does not allow the user to define new component types. Component types are defined directly as classes, taking advantage of C++'s powerful facilities for class definition, inheritance and run-time method determination. Since we did not have a C++ interpreter available to us, restricting Archtool to predefined classes saved a great deal of programming effort without restricting the program's intended function.

A component is actually composed of many objects, representing the component body, the input and output pins, etc.; the structure of components is the topic of the next section. The class hierarchy for components is shown in Figure 6. All component types are derived from the component class, which provides common functions: component name, traversal of the ownership hierarchy and definition of external pins. The component class also defines some virtual functions whose bodies are defined in derived classes: reading/writing files and summary statistics. Component's virtual functions define features that must be supplied by all component types.

Each derived class represents a specific type of component: it supplies implementations for the virtual functions defined by component and defines new functions specific to that class. The fsm class implements functions for minimization,[1] decomposition of an FSM into a network of FSMs with equivalent behaviour,[10] state scheduling,[11] and for adding and deleting states and transitions. The network class defines algorithms for finding the product of a network of FSMs, including both the

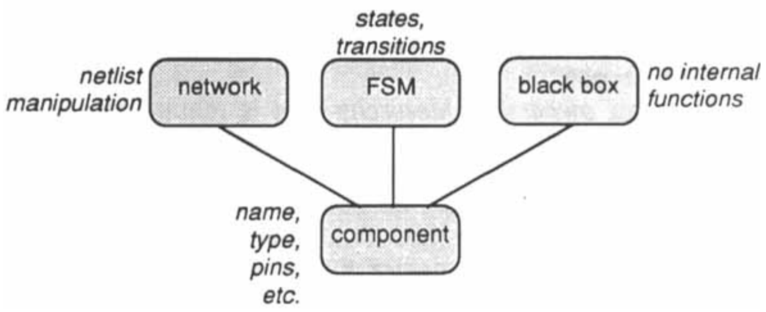


Figure 6. Classes that define component types

standard Cartesian product method and an experimental algorithm.[5]

Execution of most user commands is delegated to the derived component classes—even the procedure for printing a component depends on the properties of the derived class. The user interface receives and parses a command, then decides whether the command pertains to the system (such as a request to set the current component) or to the current component. If the command is for the component, its execute function is called with the command tokens as arguments. Execute is defined as a virtual function in the component class, with each component type supplying its own function body. Since some commands, such as read and write, are nearly identical across different component types, the execute function is one of the few instances we have found where *before* and *after* methods, such as found in Flavors[12] and CLOS,[13] would significantly improve the code structure.

Commands may cause the current component to change type. For example, in Figure 5, the mincollapse command changes the type of the current component from network to fsm. C++ allows type coercion, but does not provide type migration—changing an object of class network, whose memory had already been allocated, into a new object of class fsm would require reallocating and reinitializing the object's memory. The type of the current object is changed by allocating a new object, then replacing the old current component with the new one. In the example, the mincollapse command returns a Boolean flag which tells the interface to swap the current component with a new component returned by the mincollapse method.

COMPOSITE OBJECTS AND CLASSES

As was alluded to in the last section, a component is represented by a collection of objects called a *composite object*. The need for multiple objects to represent a component is clear when we examine the structure of a hardware component we want to model. Figure 7 shows the structure of a network: the component body holds information such as the component name and position in the ownership hierarchy; the pins make electrical connections to other components at the same level of the ownership hierarchy; and nets define electrically connected pins in the

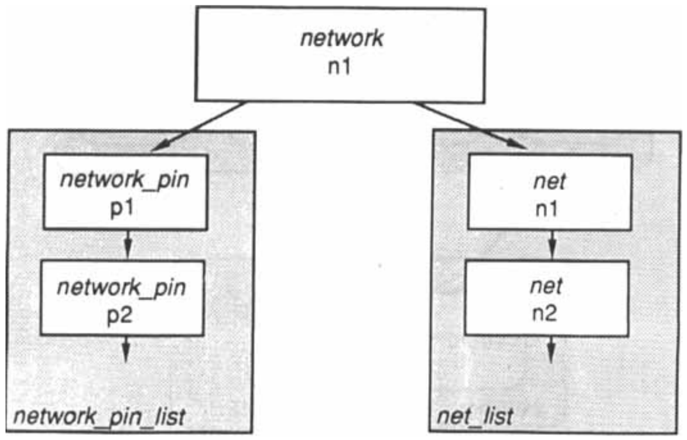


Figure 7. Structure of a network component

network. Some of these objects have their own internal structure—a net, for example, has an object for the net's name, etc., and a list of component/pin pairs that are connected by the net. FSMs have their own internal structure, which includes states and transitions as well as pins. All these data structures are composite objects, not single objects, because they have fields which may have differing numbers of elements—for example, each component may have an arbitrary number of pins.

Composite objects have been studied in the object-oriented database literature: both the Iris[14] and Orion[15] object-oriented databases support composite objects and the Orion team has separately described their approach to composite object management.[16] However, work on composite objects in databases has concentrated on storage management, versioning and other problems unique to databases. This work has not considered the problems of defining sets of related classes and managing the relationships between those classes during inheritance. One of the hard programming tasks in building Archtool was defining a set of classes to describe one type of composite object—the component—and maintaining all the proper relationships between those classes as we derived new composite classes.

We call the collection of classes which define all the objects in an composite object a *composite class*. Building components from composite classes was introduced in Fred, a database for VLSI components implemented in Flavors.[8,17] (Fred was designed primarily for component type description, whereas Archtool concentrates less on hardware description but provides more powerful facilities for algorithm development.) The relationship between object and composite classes for Archtool's component types is shown in Figure 8. The composite classes are shown in grey; each elementary class in the composite is shown in white. The lines between classes identify class inheritance, and the dotted line represents inheritance of composite classes. C++ provides language mechanisms for describing individual classes, but not composite classes, so all the definitions of and relationships between composite classes exist solely in the programmer's mind.

The COMPONENT composite class consists of two C++ classes: component, which defines the component body, and pin, which defines an external pin. An instance of the COMPONENT composite class, if it were to be instantiated, would include one

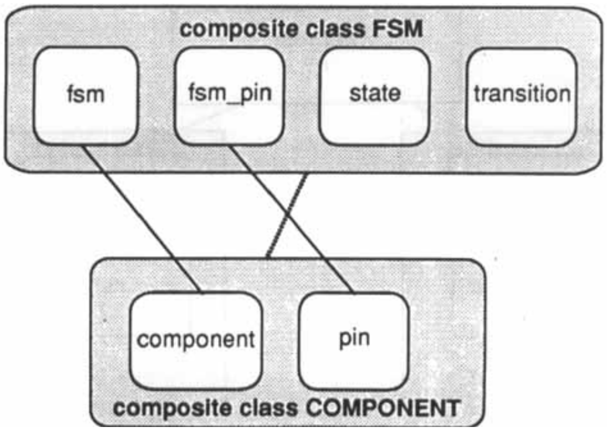


Figure 8. The composite classes for components and FSMs

object of class component; the component object in turn would include a list of objects of class pin, each of which defines one pin on the component.

In a standard C++ class, defining a new derived class requires first declaring the base class, then adding new data slots and member functions to the derived class. Similarly, deriving a new composite class involves both deriving new classes from the COMPONENT base composite class and adding new data and functions to the new composite class. The composite class FSM derives the classes for its component body and pin classes from COMPONENT: the component body class maintains the FSM's states and transitions, among other tasks; the pin class adds labelling information unique to FSM algorithms. It also adds classes to define states and transitions, which are unique to FSMs. These classes need information about other members of the composite class; for example, an FSM state includes a pointer to the component body which defines the state. Part of the FSM body class is shown in Figure 9.

We did not add any macros or other psuedo-language features to represent composite classes in the Archtool code—the relationships between C++ classes used to represent composite classes must be kept track of by the programmer. The lack of an explicit representation for composite classes created programming problems detailed below, but these problems were inevitable. Fred took advantage Lisp's macro definition facilities to provide some facilities for the definition of composite classes in the code. Fred's meta-language defined a component-class declaration and declarations for methods on those component-classes. A component-class was macro-expanded to a series of Flavors class declarations for bodies, pins, etc., along with methods that implemented the interactions between constituents of the composite class. Component-class methods were attached to the proper Flavors classes. Archtool does not use this approach for two reasons: the macro facilities of C++ are less powerful than those of Flavors, making the component-class syntax even more cumbersome; and, more important, syntactic aids do not fully resolve the problems associated with implementing composite classes.

The lack of language support for composite classes creates important problems. The first, which is more directly amenable to syntactic solution, is that the programmer may make mistakes in declaring the constituents of a composite class. For example, defining the FSM composite class requires two class derivations: fsm from

```
class fsm : public component {
   stateP_list states;
   /* all the states */
   transitionP_list transitions;
   /* all the transitions */
   boolean ns_output;
   /* TRUE if next state is a primary output */
   transitions_from_closed_table *outward;
   /* hash table for transitions from given state */
   ...
};
```

Figure 9. The fsm *component body class*

component and fsm_pin from pin. Beyond defining the new classes, the programmer must also coerce the component functions which return objects of class pin to return fsm_pin objects. Managing the relationships between classes can be tedious, but could be automated with proper syntactic aids.

But the problems associated with the fsm_pin class point up a subtler problem that is not so easily solved by syntactic modifications—the modification of the structure of a composite object. A number of program modules in Archtool may change the internal structure of a composite object:

(a) file input/output functions create new pins, states, etc.
(b) algorithms add or delete pins, states etc.

The objects which make up a composite object must be of the proper class: a pin belonging to an fsm object must be of class fsm_pin, not just of class pin. The lack of an explicit relationship between the elements of a composite class make maintaining the proper types of constituent objects difficult.

Since there is a many-to-one relationship between the component body object and other constituent objects, the constituent may be allocated on stack or heap memory, whereas the other constituents are all allocated on the heap. The new operator is used to allocate heap memory, so normally a pin to be added to a component would be allocated with the code

```
pin pl = new pin("pl", input_pin);
```

This implementation, however, requires the function allocating the pin to know the exact class of the pin to be created for this component object. If the pin is being allocated in a function belonging to the component class, rather than a derived class such as fsm, C++ provides no mechanism for component's member function to determine whether the component object belongs to the derived class fsm, requiring a pin of class fsm_pin, or to network, requiring a network_pin pin. Therefore, the component class cannot define any function which allocates structures that may be specialized by derivations of the composite class.

Our solution was to define allocation functions for all the constituent objects as part of the component body classes. The component class defines a function to allocate pins. Derived classes define allocation functions for the constituent object types they add and redefine the allocation functions for constituent objects whose definitions they modify. So in the code fragment

```
network net1;
fsm fsm1;
pin *p1 = net1.new_pin("p1",input_pin);
pin *p2 = fsm1.new_pin("p2",output_pin);
```

the call to net1.new_pin() creates a pin of class pin, since the pin object is not specialized for network objects, whereas the fsm1.new_pin() call returns a pointer to an object of class fsm_pin, which is coerced to a pointer of type pin * by C++. In general, we would also require composite class destruction functions to perform class-specific destruction; however, since the Archtool data structures required no special clean-up operations during destruction we did not implement those functions.

This solution works reasonably well for Archtool's requirements, but might break down in more general cases. Archtool's component model has one object for a component body with a number of subordinate objects to model the rest of the component. As a result, we can define the new operator to be relative to a component body object and embed the knowledge about the types of constituent objects in the body's member functions. A more general composite object with more complex relationships between its constituents, especially one in which several objects were required to be allocated on the stack, would not work well in this implementation. Implementation of general composite objects will require the compiler to know about the relationships between classes in the composite class and objects in the composite object.

TEMPORARY DATA STRUCTURES

The other major problem we encountered while designing Archtool was the management of temporary data structures. The algorithms supported by Archtool, such as FSM minimization, require complex data structures, and different algorithms require different data structures. We would like to keep the data structures for an algorithm local to the code that implements that algorithm and we would like the creation and destruction of those data structures as simple as possible.

The fsm class, for example, implements several data-structure intensive algorithms; good temporary data structures for those algorithms should be designed in accordance with three principles:

1. The code should distinguish between data in the class that is used by most or all member functions from data that is used only by a few related functions.
2. The code should make explicit the links between specialized functions and the data structures on which they rely.
3. The language and run-time support system should provide clean facilities for the automatic creation and destruction of temporary data structures.

Of the three, run-time support for creation and destruction is most important because it is the hardest for the programmer to manually approximate.

Encapsulating the data structures used by different algorithms rquires changing the behaviour of objects as they enter different parts of the program, which is not supported by C++ or other object-oriented languages. The Droid VLSI CAD system supports type mutation for objects, though it is used primarily to drastically change the class of an object for memory efficiency, not to make temporary, minor changes to the object.[9] Type mutation has been studied in object-oriented database systems: Skarra and Zdonik[18] provide a more theoretical discussion; Penny and Stein[19] discuss class modification in the GemStone database; and both Iris[14] and Orion[15] support class modification. Once again, however, attention is concentrated on the database problems introduced by class modification—how changing types affects the storage of persistent objects and interactions between users and those objects. The key programming problem is *efficiency*—how to introduce temporary data structures in a way that is both memory and computationally efficient.

As a basis for understanding the importance of temporarily modifying an object definition, let us review the algorithm for FSM minimization described by Hopcroft and Ullman[20] and the data structures required to support it. Minimization builds equivalence classes of states which cannot be distinguished by applying any input sequence and examining the output values. The algorithm compares pairs of states, dividing the pairs into three categories: definitely not equivalent; definitely equivalent; or equivalent only if some other pair are equivalent.

The basic data structure for minimization is the equivalence class, which is a set of states. Note, however, that efficient implementation requires not only the creation of new data structures which refer to the existing constituents of a composite object, but also the modification of the component's constituent objects. Efficient updating of the equivalence classes requires knowing which equivalence class currently owns a state. Although that information could be kept in an external hash table, it is much simpler to ask a state for a pointer to the equivalence class which owns it. Providing access directly from a state object can be simulated by adding a method which hides an access to an external data structure; a more efficient implementation requires the addition of data to the state object itself.

We tried two methods of implementing temporary data structures in Archtool, one of which was more satisfactory than the other. The first, and less satisfactory method was to create a superstructure of object types. When a program module that implemented an algorithm received a component composite object, it would make a new composite object which formed a superset of the original component. As shown in Figure 10, each object in the superstructure included a pointer to the component object to which it refers, along with additional data definitions required by the algorithm. The intent was for the added functions to be implemented by the superstructure class and for the original component functions to be implemented by indirection to the original class.

The first limitation of this method becomes quickly apparent: the methods of the original object's class are not directly implemented in the superstructure class because

```
class min_state {
   state *mystate;
   equiv_class *equiv_owner;
 public:
   String name() { return mystate->name(); }
   ...
   equiv_class *myclass { return equiv_owner; }
};

class min_fsm {
   fsm *myfsm;
   equiv_class_set classes;
 public:
   String name() { return myfsm->name(); }
   ...
};
```

Figure 10. An algorithm-specific class superstructure

it is not derived from the original object's class. Any function of the original object, such as name in the Figure, must be explicitly implemented by a function call. Defining these functions is tedious and error-prone, even if the definitions require only calling a previously-defined function. They violate the principles of object-oriented programming by causing the original object's functions to be declared twice, once in the object's class and again in the superstructure.

The method also requires duplicating a lot of code from the original classes. For example, separate functions must be provided by the min_fsm class to traverse the list of min_state state objects. Those functions conceptually duplicate the fsm functions for traversing state objects, though separate functions are in fact required. There is also no provision for finding a superstructure object given an original object, which may limit coding. Keeping track of the relationships between the original objects and their superstructure was hard work for the programmer.

A more satisfactory method was to provide data storage in objects that could be shared among algorithm packages. As shown in Figure 11, each constituent of a component composite object provides a pointer for an information object. Upon entry to the algorithm package, the package would fill all the appropriate information slots with pointers to the required objects. Since the information objects are closely connected to the component objects, they can refer easily both to other information objects and to the original component objects. Accessing the information object requires a single additional indirection, such as

```
class fsm {
   ...
   void * my_info;
 public:
   void *info() { return my_info; }
   void set_info(void *newval) {
      if (my_info != NULL)
         error("tried to reuse info field");
      my_info = newval;
      }
   void reset_info() { my_info = NULL; }
   ...
};

class fsm_min_info {
   equiv_class_list myclasses;
 public:
   ...
};
```

Figure 11. Data add-ons for classes

```
((fsm_min_info *)fsm1->info())->classes();
```

where the coercion can be hidden by an access function. The information objects are destroyed when the program leaves the algorithm package; ensuring that the information pointers are NULL upon entry to and exit from an algorithm helps catch inadvertent multiple use of a single information field.

Although we found this method to be easier to use and less error-prone than a superstructure, it does have significant problems of its own. The first is that the use of the void * type and coercion of the type of the information object is always subject to error. We occasionally forgot to properly reference data held in the information object, and type coercion made such problems more difficult to pinpoint.

The more important problem was that the information fields did not follow the constructor/destructor rules. C++ calls a constructor function when an object is created and a destructor function when it is destroyed. In this case, however, we want to execute a constructor function to build the information objects when the algorithm module's code is entered and to execute a destructor function to dispose of the information objects when the program leaves the algorithm module. One could look at this behaviour as changing the objects' types over time—the objects are specialized to derived classes when the program enters the algorithm module, then are returned to their original classes when the program exits the algorithm code. C++ and other object-oriented languages do not directly support such behaviour.

We built and deleted the information objects by writing and manually calling our own constructor and destructor functions for them, but it is not always possible for generic code to support behaviour of the information constructors and destructors fully equivalent to C++'s constructor/destructor semantics. Each algorithm package uses constructor and destructor calls for their information objects which mimic C++'s automatic constructors and destructors:

```
void fsm::minimize() {
  construct_min_info(this);
  minimize_me(this);
  destroy_min_info(this);
}
```

The constructor function initializes the information objects for the entire data structure, whereas the destructor returns the data structure to its original state.

However, not all construction and destruction occurs at the interface to the algorithm package. The package will, in general, delete parts of the existing data structure and create new elements. Since the information object constructors and destructors are not related to the original data structure, the programmer must manage the information objects manually. For example, code that replaces one state with another must call both the standard allocation facilities and take care of the state information:

```
destroy_min_state_info(oldstate);
m1.delete_state(oldstate);
newstate = m1.new_state();
construct_min_state_info(newstate);
```

We occasionally introduced bugs into Archtool by improperly constructing information fields for newly created objects.

Cleaning up information fields is not always possible—some objects are destroyed by functions outside the algorithm package. If the function makes some computation to decide which objects to destroy, it is not possible to destroy the information objects in advance of the call to the external function. The external function cannot easily be made to destroy the information object, since code outside the package knows the information object's type only as `void`. A pointer to an information destruction function could be passed to the external function, but parametrizing all parts of the program with code to destroy all information fields would greatly obscure the control flow of those functions.

CONCLUSIONS

Object-oriented programming is especially well-suited to electronic computer-aided design, because of the close correspondence between hardware components and objects. The well-known benefits of data abstraction and inheritance also quickened the pace at which we could develop new CAD algorithms: data abstraction minimized interactions between program modules; inheritance let us share common functions.

But object-oriented programming languages did not solve all the problems in the design of Archtool. Inheritance relationships between simple pairs of classes did not describe all the important relationships between Archtool's classes. Similarly, our data structures could not vary in time to accommodate different algorithms as easily as we would have liked. These problems are not unique to C++ but are limitations on all object-oriented languages with which we are familiar. Object-oriented programming languages provide valuable new language features, but we should not expect that data abstraction, inheritance and run-time method determination alone would solve all programming problems. The question of where best to solve the problems introduced by composite objects and temporary data structures—by changing language design or by using new libraries along with special programming styles—is still subject to debate. But example systems such as Archtool help identify new problems for programming language designers to tackle.

ACKNOWLEDGEMENTS

This work was performed while the author was with AT&T Bell Laboratories, Murray Hill, New Jersey. Thanks to Bjarne Stroustrup, Jon Shopiro and Kathy Stark for discussions on C++, and to Carl Seaquist and Wayne Bower for discussions on object-oriented implementation problems in CAD systems.

REFERENCES

1. Zvi Kohavi, *Switching and Finite Automata Theory*, second edition, McGraw-Hill, New York, 1978.
2. Wayne Wolf, 'The FSM network model for behavioral synthesis of control-dominated machines', *Proceedings, 27th Design Automation Conference*, ACM Press, June 1990, pp. 692–697.
3. Michael J. Foster, Private communication, 'Discussion on automata theoretic algorithms', October 1988.
4. R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. Wang, 'MIS: a multiple-level logic optimization system', *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, **CAD-6**(6), 1062–1081 (1987).

5. Wayne Wolf, 'An algorithm for nearly-minimal collapsing of finite-state machine networks', *Proceedings, ICCAD-90*, November 1990, pp. 80–83.
6. Dwight D. Hill, Kurt Keutzer and Wayne Wolf, 'Overview of the IDA system: a toolset for VLSI layout synthesis', in Wolfgang Fichtner and Martin Morf (eds), *VLSI CAD Tools and Applications*, Kluwer Academic Publishers, 1987, pp. 233–263.
7. John K. Osterhout, Gorton T. Hamachi, Robert N. Mayo, Walter S. Scott and George S. Taylor, 'Magic: a VLSI layout system', *Proceedings, 21st Design Automation Conference*, ACM/IEEE, June 1984, pp. 152–159.
8. Wayne Wolf, 'How to build a hardware description and measurement system on an object-oriented programming language', *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, **8**, (3) 288–301 (1989).
9. Doug Matzke, 'Discussion on object-oriented methods in Droid', *Private communication*, June 1990.
10. J. Hartmanis and R. E. Stearns, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1966.
11. Wayne Wolf, Andrés Takach and Tien-Chien Lee, 'Architectural optimization methods for control-dominated machines', in Raul Camposano and Wayne Wolf (eds), *High-Level VLSI Synthesis*, Kluwer Academic Publishers, 1991.
12. David A. Moon, 'Object-oriented programming with flavors', *Proceedings, OOPSLA-86*, ACM, 1986, pp. 1–8.
13. D. G. Bobrow, L. G. DeMichel, R. P. Gabriel, S. E. Kleene, G. Kiczales and D. A. Moon, 'Common Lisp object system specification X3J13', *SIGPLAN Notices*, 23 September 1988. Document 88-002R.
14. D. H. Fishman, D. Beech, H. P. Caate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan and M. C. Shan, 'Iris: an object-oriented database management system', *ACM Trans. Office Information Systems*, **5**, (1), 3–26 (1987).
15. Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darel Woelk, Nat Ballou and Hyoung-Joo Kim. 'Data model issues for object-oriented applications', *ACM Trans. Office Information Systems*, **5**, (1), 3–26 (1987).
16. Won Kim, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza and Darrel Woelk, 'Composite object support in an object-oriented database system', *Proceedings, OOPSLA'87*, ACM Press, 1987, pp. 118–125.
17. Wayne Wolf, 'A practical comparison of two object-oriented programming languages', *IEEE Software*, September 1989, pp. 61–68.
18. Andrea H. Skarra and Stanley B. Zdonik, 'The management of changing types in an object-oriented database', *Proceedings, OOPSLA'86*, ACM Press, 1986, pp. 483–495.
19. D. Jason Penny and Jacob Stein, 'Class modification in the GemStone object-oriented DBMS', *Proceedings, OOPSLA'87*, ACM Press, 1987, pp. 111–117.
20. John E. Hopcroft and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.