Complex and Composite Objects in CAD/CAM Databases

Wolfgang Wilkes, Peter Klahold, Gunter Schlageter
FernUniversität Hagen, Praktische Informatik I
Postfach 940, D-5800 Hagen
West Germany

Abstract
An essential modelling concept for CAD/CAM databases is the component relationship which relates a superior composite object to its components. In order to import necessary information into the composite object, often an interface is defined which is copied into the composite object as an abstraction of the component. In this paper we will show that both the concept of interfaces and the concept of composite objects are based on the same mechanism, an instance inheritance relationship. This relationship is integrated into an object oriented datamodel which represents interfaces and composite objects in an easy way. Furthermore, the concept of interfaces is generalized to an abstraction mechanism. This allows to eliminate some severe drawbacks of a rigid interface concept and to model generic component relationships.

1. Introduction
Data modelling in new fields of database applications requires the representation of more complex objects than in conventional applications. Thus, it is not sufficient to have only flat tuples (as in relational databases). Considerably more complex data structures are required to support application oriented data modelling. This has led to the development of more powerful data models. Some extend the relational model for managing hierarchical tuples (e.g. XSQL [HaLo82], NF2-relations [SeSc86], [Dada86]), others build objects out of networks of subobjects (e.g. molecules [HMM87] and molecular objects [Babu84], [BaKi85]). By means of these models it is possible to represent complex structured objects as one unit known to the system.

However, it is not sufficient to assemble complex objects only by local objects. With CAD/CAM in particular, an independent development of individual objects is important to make sure that highly complicated products (aeroplanes, VLSI-chips, software-systems) can be manufactured by means of work specialization. The reusability of already designed parts is of special importance here [IEEE87]. Hence, the data model has to offer a component relationship to use already existing objects as components of higher level objects which are then called composite objects.

The notion of interface has been introduced in software engineering and VLSI design to specify which data of a component are visible for the composite object. The interface object describes the external phenotype of a design object, and it can be regarded as an abstraction of the implementations (versions) of the design object. Usually, only the interface data of a component are imported into the composite object.

In this paper we will show that both relationships - the relationship between an interface and its versions and the relationship between a component and the composite objects which use it - are based on the same mechanism, an instance inheritance relationship. The inheritor object in this relationship does not only inherit the existence of attributes (inheritance at the type level, generalization) but the values of the attributes as well (inheritance at the object level, object generalization).

After the motivation and definition of the instance inheritance relationship in section 2, we will integrate it into an object-oriented data model. The basic concepts of this data model are objects and relationships (section 3). Objects can become rather complex, they may have local subobjects which are related by local relationships. This basic model can be compared to the extended NF2-data model [PiAn86] with the extension of explicit definitions of relationships. By means of the instance inheritance relationship, interfaces and composite objects are modelled (section 4). Moreover, a generalization of the notion of interface results in more flexible mechanisms for specifying which data of a component is visible in a composite object. To illustrate the concepts of the object model we use one general example, the modelling of logical gates, and gates will be redefined several times to emphasize the different concepts of the object model.

2. The Instance Inheritance Relationship
The main idea of this paper is to represent the relationships between an interface and its implementations and between composite objects and components by instance inheritance relationships. In this section, this uniform view will be motivated and the notion of instance inheritance will be introduced (as opposed to class or type inheritance known from object oriented systems).
The component has to provide some information for the composite object, for instance to enable the composite object to connect its components. The subgates of a gate, for example, can only be connected when the pins of the subgates are known. Usually the external properties of an object are separated from the implementation details by means of the interface concept ([Baki85], [MLNB83]). If an object is used as a component, its interface part is visible in the composite object.

Two questions have to be answered:
(1) What is an interface, how is it separated from and related to the implementation details?
(2) How does the interface become visible within the composite object?

At the first glance, views seem to model the concept of interfaces precisely: A view makes part of the underlying object visible. But there is an important aspect of the interface concept which does not fit the concept of views: Design objects often exist in several versions. Usually they all have the same external interface, and they only differ in their implementation. To guarantee that all versions can be used interchangeably, the interface data for one implementation must not be changed. Thus, the interface sets conditions all implementations have to obey. This cannot be modelled by representing the interface as a view of its implementations because of the different directions of update propagation (see figure 1): Updates of an object are propagated to its view, but updates of the interface are propagated to its implementations, i.e. a view depends on its underlying object, but the implementation depends on its interface.

![Diagram of views and interfaces](image)

Figure 1: Views and Interfaces

To model the relationship between an interface and its implementations correctly we introduce the concept of an instance inheritance relationship. The instance inheritance relationship relates an object t (called transmitter) to an object i (called inheritor), and the following holds:

**Rule 1:** If A is an attribute of t, then A is also an attribute of i.

**Rule 2:** The value of the attribute A in i is the same as the value of attribute A in t.

The first rule states that the inheritor's type is a subtype of the transmitter's type, i.e. the inheritor object i has at least the attributes of the transmitter object t. Furthermore, the second rule states that the inheritor object i also inherits the values of all attributes of the transmitter object t. Thus, the transmitter object t can be seen as an abstraction of the inheritor object i, and i inherits all attributes of t together with their values.

The instance inheritance relationship, which relates objects and allows inheritance of attribute values, corresponds to the IS-A relationship between types and subtypes which allows inheritance of attribute definitions (see e.g. [MyBW80]). T1 IS-A T2 expresses that type T2 is an abstraction of type T1. Correspondingly, i is-a t expresses that the transmitter object t is an object which represents object i at a more abstract level.

In the following we shall use is-a (small letters) to denote the instance inheritance relationship as opposed to the IS-A relationship (capital letters) between types. A more general framework for defining inheritance relationships can be found in [Wil88a], where the IS-A relationship defined above is only a special case. In this paper we only need the is-a relationship and we will use the terms instance inheritance relationship and is-a relationship synonymously here.

By is-a relating the implementation object to the interface object we are able to model interfaces appropriately. The is-a relationship meets the requirements for the relationship between an interface and its implementations:

- The interface is an abstraction of all its implementations.
- Every implementation inherits the interface data without being allowed to update them (consequence of rule 2).
- If an update of the interface occurs (which, of course, should happen very rarely), it is propagated to the implementations (which probably have to be adapted to the new situation afterwards).
- Implementations may be updated as far as only their local (i.e. not inherited) data are involved.

The is-a relationship captures the idea of both the concepts of version generalization [BaK85] and of molecular objects [BaBu84]. The interface is related to all its versions and the versions inherit data from the interface as with the version generalization. The correspondence between subobjects of the interface object and subobjects of the implementation object is also modelled by means of the instance inheritance.

The second question was, how do the data of the component’s interface become visible within the composite object. The first approach is to copy the interface data into the composite object. However, this involves the following problem: If we only copy the interface of a component C into a composite object O and do not maintain the relationship between O and C, O is not informed of updates of its component C, and C does not know that other objects use it as a component. This may lead to inconsistencies between original and copied data.
A better approach is to model also the component relationship as an inheritance relationship: The composite object inherits data from its components. This can be regarded as an "upward inheritance". Thereby the requirements for the component relationship are met:

- The composite object inherits data of its component and is not allowed to update them.
- If an update of the component occurs, the update has to be propagated to the composite object which possibly has to be adapted afterwards.
- Updates of the composite object do not change data of its components, only relationships to components or between components are touched.

Thus, our approach is to model the import of information into the composite object as an instance inheritance mechanism. In the following sections we will describe how these relationships can be expressed by means of an object-oriented model.

3. Basic modelling elements of the object model

The remainder of this paper will describe how the concept of instance inheritance relationships and the modelling of interfaces and composite objects can be transposed into a data model. We introduce an object model, which comprises the is-a relationship as an integral part. By means of several examples we show how it can be used in modelling CAD/CAM objects. In this section we start with sketching the basic modelling concepts of the object model before we illustrate the application of the instance inheritance relationship in the next section.

Objects and Relationships

The basic units are objects which belong to specific object types. The object type describes the attributes an object possesses. Integrity constraints may be defined with the definition of an object type. They are local to the object type, i.e. they define conditions the attributes of the objects have to obey.

Objects can be related to each other. A relationship is represented by a relationship object. A relationship object belongs to a specific relationship type which can define several attributes and integrity constraints for the relationship objects.

Example:

```plaintext
obj-type PinType :
    attributes:
        InOut: I/O;
        PinLocation: Point;
end PinType;
```

```plaintext
rel-type WireType :
    relates:
        Pin1, Pin2: PinType;
    attributes:
        Corners: list-of Point;
end WireType;
```

In the example, WireType relationships relate objects of type PinType, and with the relationship the geometry of the wire is stored in the attribute Corners.

Complex objects

A complex object contains local objects, in the following synonymously called subobjects. Subobjects depend on their complex object, they are deleted with their complex object and they are only accessible in the context of their complex object. Subobjects are grouped according to their type, and the set of all subobjects of a complex object belonging to one type is called a subclass. Furthermore, a complex object may contain local relationships among its subobjects, and again the relationship objects representing these relationships are grouped in relationship subclasses. Subobjects may be complex as well which results in several levels of nesting.

The type of a complex object is described by the definition of

- domains of attributes,
- types of subobjects (i.e. the definition of subclasses),
- types of relationships among the subobjects (i.e. the definition of relationship subclasses), and
- local integrity constraints.

Example: (logical circuits built of AND-, OR-, NAND- and NOR-gates)

Circuits shall be built by relating pins of AND-, OR-, NAND-, and NOR-gates. These elementary gates are described by objects of type ElementaryGate which represent pins as local objects of type PinType. Thus, objects of type ElementaryGate are complex objects:

```plaintext
obj-type ElementaryGate :
    attributes:
        Function: GateFunction;
        GatePosition: Point;
    subclasses:
        Pins: PinType;
    constraints:
        count (Pins) = 2 where Pins.InOut = IN;
        count (Pins) = 1 where Pins.InOut = OUT;
end ElementaryGate;
```

Now we can define the object type Gate (see figure 2) which includes the definition of two subclasses for objects and one for relationships:
represents gates assembled by AND, OR, NAND and NOR-gates. The attributes:

Function: GateFunction;
Length, Width: integer;

subclasses:
Pins: PinType;
Subgates: ElementaryGate;

Wires: Wiretype

subrels:

constraints:

end Gate;

end Gate;

The elements of Pins belong to the type PinType and represent the external pins of the gate to be described.
- The elements of SubGates belong to the type ElementaryGate and represent the basic gates which are used to assemble the circuit.
- Pins are related by relationship objects of type WireType.

4. Using the Instance Inheritance Relationship

As any other relationship, the instance inheritance relationship has to be defined at the type level. Corresponding to the definition of IS-A relationships between types and supertypes the instance inheritance relationship is-a is defined with the definition of the inheritor type:

obj-type <inheritor-type> is-a <transmitter-type>

At the type level this definition has the same effect as the definition of a type which is bound to its supertype by means of the IS-A relationship: The inheritor-type inherits the (existence of the) attributes of the transmitter-type. Furthermore, in our model the inheritor-type inherits the definition of subclasses, i.e. the definition of the types of subobjects. But moreover, this definition enforces that every object of the inheritor-type is related to an object of the transmitter-type: With the creation of an inheritor object the related transmitter object has to be specified, and the newly created object inherits all attribute values and subobjects from it.

In the following we use the is-a relationship to model (1) the relationship of an interface to its implementations (which thereby will be generalized to an abstraction hierarchy) and to model (2) the component relationship between a composite object and its components.

4.1 Interfaces and interface hierarchies

We assume that the interface of a gate is described by its function, its dimension, and its external pins. Therefore we divide the description of gates into two parts: GateInterface and GateImplementation. The object type GateInterface is as follows:

obj-type GateInterface:

attributes:
Length, Width: integer;
Function: GateFunction;

subclasses:
Pins: PinType;

end GateInterface;

All implementations of a specific gate are restricted to having the same interface. So we have to relate every object of the object-type GateImplementation to an object of the object-type GateInterface by the is-a relationship. This is expressed at the type level by is-a relating the type GateImplementation to the type GateInterface (see figure 3):

Figure 3: is-a relationship between the types GateInterface and GateImplementation
obi-tvDe Gatelmplementation is-a Gate Interface:
  attributes:
  - Versionld: integer
  subclasses:
  - Subgates: ElementaryGate
  subrels:
  - Wires: Wiretype
  constraints:
  for all Wire in Wires:
    - (Wire.Pin1 in Pins and Wire.Pin1.InOut = IN)
    or (Wire.Pin1 in SubGates.Pins and Wire.Pin1.InOut = OUT)
  and
    - (Wire.Pin2 in Pins and Wire.Pin2.InOut = OUT)
    or (Wire.Pin2 in SubGates.Pins and Wire.Pin2.InOut = IN)
end Gatelmplementation;

Note: The constraint determines that a wire may relate pins of subgates either to pins of other subgates or to pins of the gate itself. Furthermore, the constraint specifies that the semantics of external pins is different at the interface and at the implementation level: External pins of a gate are internally connected to other pins which yields to a conversion of the InOut property (called "shift of semantics" in [BaBu84]).

Due to this definition all attribute- and subclass-definitions of the transmitter-type Gate Interface are inherited by the inheritor-type Gatelmplementation (as with the generalization relationship). But, moreover, with the creation of a Gate Implementation object it has to be specified to which Gate Interface object it is related. As an example, figure 4 shows the Gate Interface object of the gate 'Flip-Flop' and one of its Gatelmplementation objects.

Figure 4: Gate Interface and one Gatelmplementation object of gate 'Flip-Flop'

While implementations of a design object may be arbitrarily updated, most object models define interfaces as being unchangeable to avoid inconsistencies. But actually, interfaces do change and we have to manage changing interfaces (see e.g. [Estu88]). The following observations reflect this requirement:

- Not all changes of an interface do concern all objects using this interface: If a new function is added, this does not affect superior modules which do not need this function.
- Often it is not clear which information actually belongs to the interface. The interface of a chip may be built by its pins only, but some applications may require more information to use the chip as a component of a composite object (for instance, time information for time simulations).
- Not all CAD/CAM areas allow the definition of a clear interface. For instance, in the area of mechanical engineering, it is difficult to define exactly the interface of an object. Moreover, the assembly of components can be accomplished in many more ways than, for instance, in software engineering (call of procedures).

The instance inheritance relationship offers a means to overcome the rigid definition of interfaces: If interfaces are changed, we have to deal with versions of interfaces. Common properties of the versions of an interface may be represented by a "super interface" which is linked to the interface versions by an is-a relationship. We assume, for instance, that a gate is defined by a functional interface which describes its function and its external pins. But several versions of a gate interface with different dimensions may exist. Thus, we further divide the description of gate interfaces:

obi-type FunctionalGateInterface:
  attributes:
  - Function: GateFunctions;
  subclasses:
  - Pins: PinType;
end FunctionalGateInterface;

obi-type SpatialGateInterface is-a FunctionalGateInterface:
  attributes:
  - Length, Width: integer;
end Gate Interface;

The example shows that by means of the instance inheritance relationship we can build more complex interface hierarchies than the conventional two-level interface-implementation-scheme. Objects at higher levels of the hierarchy represent more abstract views of the design object. Thus, the hierarchy can be regarded as an abstraction hierarchy which allows a classification of design objects and their versions according to their common properties as subtle as desired.

4.2 Composite objects

The next step is to assemble gates not only by elementary gates but also by previously designed complex gates. To achieve this goal, we have to incorporate independent component gates into the data and relationship structure of the superior gate object,
i.e. we have to import some data of the components into the composite object.

A component relationship between a composite object and its components can be modelled as follows (see figure 5): The component is represented by its interface object. The data of this interface describe the external properties of the component. Within the composite object, the component is represented by a subobject which inherits the data of the component’s interface. Thus, the subobject is the inheritor in an is-a relationship whose transmitter object is the component’s interface. The composite object sees the interface data of the component as part of the subobject which, of course, may possess its own attributes and subobjects besides the inherited data.

The assembly of lower level gates takes place in *GateImplementation* objects. So we enhance the definition of the object type *GateImplementation* as follows:

```plaintext
@type GateImplementation is-a GateInterface

subclasses:
  SubGates is-a GateInterface:
    attributes:
      GateLocation: Point;
      Wires: Wiretype

and GateImplementation;
```

Note: The type of subclass *SubGates* has been declared implicitly in this definition. Of course, it could have been also defined as an explicit type.

![Diagram](image)

Figure 5: Application of the instance inheritance relationship for modelling component relationship and interface-implementation relationship simultaneously.

The interesting point of this example is that the is-a relationship to *GateInterface* appears twice. This reflects the fact that an object of type *GateInterface* may transfer its data as well to its versions as to the composite objects which use it as a component (see figure 5). In the above definition, the first occurrence of the clause "is-a *GateInterface*" connects an *GateImplementation* object to its interface object, and the

*GateImplementation* object inherits the data of its interface object via this relationship. The second is-a relationship models the component relationship between the composite *GateImplementation* object and its components: The objects of subclass *SubGates* represent the components, and these subobjects are linked to the interface object of the respective component by the is-a relationship. Via this relationship the objects of subclass *SubGates* inherit the component’s interface data. Thus, this example emphasizes the applicability of the is-a relationship for modelling both the interface and the component relationship.

### 4.3 Composite Objects and Abstraction Hierarchies

If we combine the two modelling concepts of abstraction hierarchies and composite objects, we arrive at the following situation (see figure 6):

1. Every design object may be represented by an abstraction hierarchy built by is-a relationships between objects describing the design object at several levels of abstraction. All objects which are is-a related to the same higher level object O can be understood as versions of O. As shown in figure 6, they are embedded into a horizontal "version plane" (comparable to the version planes of [KaCB86]). By means of the abstraction hierarchy, we thus introduce the notion of "versioned versions" with the versioned interfaces described in section 4.1 being a special case.

2. These abstraction hierarchies are interconnected by component relationships. A component relationship is built by is-a relating a subobject of the composite object with an element of the component’s abstraction hierarchy. The higher this element is located in the hierarchy, the less data flows to the composite object, and the composite object is more independent of component updates.

If an abstract representation of the component is selected which is not placed at the bottom of the component’s hierarchy (e.g. objects O2, O21, or O3 in figure 6), we get a "generic component relationship" (in [BaKi85] called a "parameterized version"), i.e. the actual selection of a component’s version is deferred to assembly-time. In this case we need mechanisms to control the deferred selection process. On principle, there are three alternatives [Wilk88b]:

1. A component is selected by queries associated with the composite object expressing the required properties of the component (top-down selection).
2. Design objects supply a specific version by default. If the design object is used as a component via a generic relationship, this default version becomes the actual component (bottom-up selection).
3. The selection mechanism is neither associated with the composite object nor with the component. The selection is guided by information not included in the object definition, instead (e.g. environments in [DiLo88]).

Figure 6: Abstraction hierarchies and composite objects

Objects which are placed at higher levels of the hierarchy are not necessarily restricted to describe only external properties of the design object. Up to now, higher level objects were understood as interfaces. But if a higher level object is defined as being composite, then it also has subobjects which are is-a related to components. According to the instance inheritance property of the is-a relationship, all its versions inherit these subobjects and the is-a relationship to the components, too. Thus, some part of the internal structure of the design object is fixed already at abstract levels of the hierarchy. This illustrates that the abstraction hierarchy is more than a pure interface hierarchy. Any data which are common to the versions, not only interface data, may be put into the abstract object.

This is an important difference to the concept of parameterized versions in [BaKi85] where the component hierarchy is described only as part of the versions. Thus, if a generic reference is created (instantiation of an "object type"), we neither know whether this generic component has components itself nor the possible type or other properties of these lower level components. This information is not available before the generic reference is fixed to a version reference, or using the terminology of [BaKi85], before the socket (generic reference) is plugged with a version. Thus, the model of [BaKi85] is not able to represent component hierarchies at higher levels of abstraction.

Design is a process of developing concrete descriptions for artefacts starting from more abstract descriptions and requirement specifications. Figure 6 shows how this process can be supported in our model by weakening the strong update restriction of inherited data (rule 2 in section 2): If an inheritor object inherits a subobject which is is-a related to a component, it possesses the same components as its transmitter object (as for instance object O12 in figure 6 which inherits O1’s subobject related to component O2). Object O11, on the other hand, is not related to the same component objects as its transmitter object O1, but its components are versions of the components of O1. Thus, it should be possible, to specialize inherited data. This means that the inherited subobject may be replaced by another one which is is-a related to it. Using this mechanism, composite objects may use high abstraction levels of their components in early phases of the design process, and the component will be refined by walking down the hierarchy to lower levels during the design process [BKSW88].

5. Conclusion

The instance inheritance relationship (is-a relationship) is a very powerful mechanism to model several aspects of composition relationships between objects. The inheritor object of a is-a relationship does not only inherit the existence of attributes from a transmitter object but also their values. Using the instance inheritance we are able to model interfaces as well as composite objects very naturally. Furthermore, we succeed in generalizing the notion of interface by constructing abstraction hierarchies of design objects. An object-oriented data model comprising the instance inheritance relationship as an integral part has been introduced, and the features of it have been illustrated by means of several examples.

This paper could not discuss all aspects of composite objects and of the instance inheritance relationship. Some topics have been or have to be investigated more closely, and finally, two of them shall be outlined briefly:

Types of instance inheritance relationships

In this paper we used an instance inheritance relationship which is defined very strictly: The attribute values of an inheritor object have to be equal to the attribute values of its transmitter object. On the other hand, in section 4.3 we discussed a more general form of instance inheritance. There the attributes and subobjects were not necessarily equal to their counterparts in the transmitter object, but they were at least is-a related to them. Other forms of instance inheritance can be considered, for instance selective inheritance and default inheritance as in [Wilk88a]. A more general question is the following*: How closely are abstraction and inheritance mechanisms related? Does one concept imply the other, or are they actually different? It seems to be worthwhile investigating this topic more deeply to get a better understanding and a clear definition of the various concepts.

* This question arose at the ooDBS-II Workshop [Di88] during a discussion initiated by Antonio Albanese.
Version management

The *is-a* relationship reflects precisely the relationship between an object and its versions: The object comprises the data common to all versions. But, besides this connection of versions and objects, there are other aspects which have to be supported by the version management and which primarily concern the organization of the version set. Important are ordering relationships among versions [KSUW85] and grouping of versions to support the process of design in

- keeping track of the design history,
- supporting the parallel development of alternatives,
- providing means for the classification of versions, e.g. in accordance with their degree of correctness.

Which mechanisms are needed and how they can be integrated into an object model is described in [KSWi86] and [Wilk88b]. A first prototype of a version manager which provides an *is-a* relationship between objects and their versions is under implementation. The version manager runs on top of the database AIMP developed at the IBM Scientific Center Heidelberg [Dada86].

An interesting application of the instance inheritance relationship has been pointed out in [NaBR88]: They use the instance inheritance relationship for modeling the derivation of new versions from old ones. This is connected with a prototype concept which supports the integration of versions into the generalization scheme.

References


[Wilk88a] W.Wilkes: Instance Inheritance Mechanisms for Object-Oriented Databases. in [Ditt88]