In: Proc. 8th Int. Conf. on Data Engineering, Feb. 1992, Tempe, Arizona, pp.348-357

Mapping a Version Model to a Complex-Object Data Model

Wolfgang Käfer
Sonderforschungsbereich 124
University Kaiserslautern
6750 Kaiserslautern, Germany
kaefer@informatik.uni-kl.de

Harald Schöning
Sonderforschungsbereich 124
University Kaiserslautern
6750 Kaiserslautern, Germany
schoenin@informatik.uni-kl.de

Abstract

Support for version management of highly interconnected complex objects is a strong requirement for database systems supporting CAD. In this paper, we present such a version model and exemplify it by means of the VLSI design process. However, we are not only interested in the version model itself, but also in its implementation on top of an existing complex-object data model without version support. In contrast to similar solutions basing on the relational data model, this approach allows for a simple and efficient implementation of the version model, allowing for powerful retrieval operations. Sharing of data, which occurs necessarily among versions, is system controlled. This prohibits redundant storage of data. We conclude that implementing a complex-object database system supporting versions is not more complicated than implementing a complex-object database system without version support.

1. A VLSI design environment

In this paper, we present a version model for computer aided design (CAD) purposes and its implementation on the basis of a complex-object database management system (DBMS). We illustrate the functionality of our model with the help of a VLSI design example, but we claim that it is useful for all kinds of CAD.

The VLSI design approach to be considered here is based on the two strategies *hierarchical decomposition* and *stepwise refinement* [9]. The latter copes with the development of a chip starting from its behavioral description, passing an *structural* and an *topological* description, and ending up with its mask layout. In the structural description, the module (which is a better term than "chip" in this domain) is described as a collection of up to 100 simpler modules and connections among these modules (*hierarchical design method*). Placing these modules together and generating the corresponding geometrical information leads to the topological description of the module (its *floorplan)*. In this domain, each module is represented by a rectangle.

We use the design of a 2-bit adder as a simple example. Figure 1.1 depicts the structural and topological design of a 2-bit adder based on three 1-bit adders and one half-adder. Each 1-bit adder is built up by one AND and one XOR gate.

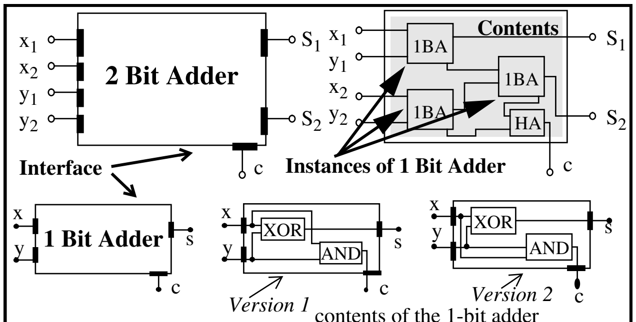


Figure 1.1: Interfaces, instances and contents

Data modelling

Looking at figure 1.1, we can identify three object types, namely *Interface, Instance,* and *Contents* as well as three link types among them, namely *(Instance) part_of (Contents), (Instance) instance_of (Interface),* and *(Contents) implements (Interface)*. This abstract description of the data is basically sufficient from the viewpoint of the structural domain. Concerning the floorplan we need a deeper insight into the objects and how they are connected. Thus, our object types turn to complex object types (i.e. aggregations of simpler or elementary objects) and the links serve only as an abstraction of more specific connections among the elementary objects of different complex objects:

- Interface
  The interface of a module is described by a rectangle, the so-called frame, and several intervals on this frame (i.e. possible positions of pins) to which wires may be connected. Interfaces are subject to versioning, since some of their properties (e.g. size of the frame) may change.
- Instance
  The instance of a module expresses the usage of the interface of the module at a *specific position* within the contents of the surrounding module. Thus, versions of instances are not meaningful.
- Contents
  The implementation of a module consists of several instances of other modules and the wires, nodes, etc. needed to connect the pins of the instances with one another and with the pins of the surrounding module. Of course, contents are subject to versioning.

In this environment, versions of an object behave like the object itself, i.e. they have the same structure and the same behavior as the object. Therefore, the object may be seen as a representative (abstraction) of all of its versions. Versions may be either (improved) revisions of a former state or independent alternatives. In the former case, a derivation graph originates from the relationship “is developed from”.

This short introduction of VLSI design should suffice to identify the three main problems concerning data modelling and data handling not only in this particular application area but also in the whole area of CAD:

- handling of composed objects
  The data describing an complex object may consist of a large number of elementary objects and relationships among them. The data model should allow for a convenient handling of such composed objects as a whole as well as the object’s internal structure.
- (object and) version management
  The data model should provide an abstract view to the data in order to reduce the complexity of the database schema and to facilitate versioning operations and version selection. This includes the versioning of composed objects and the organization of versions in a version derivation graph.
- configuration management
  Versioning leads to the version selection problem, i.e. for each object (e.g. the 2-bit adder) which is part of the final result, the versions participating in the result have to be identified (e.g. the contents versions of the instances of the 1-bit adder). The data model should support this configuration process.

Architectural Issues

Implementing a version model from the scratch which covers the above mentioned problems is a complex task. However, some of the features required for the version model are already provided by complex-object DBMS, which allow for the definition, the retrieval, and the manipulation of complex objects, but none of them provides version facilities as required by complex CAD systems. Complex-object DBMS are characterized by providing

- powerful *retrieval and manipulation operations on complex* (composed) objects,
- various storage structures (including clustering mechanisms) for the *efficient access to these objects*,
- but *no adequate version facilities*.

Thus, we built our object and version management system **on top** of a complex-object DBMS, i.e. we have to **model the versioned data in terms of the complex-object data model** and we have to transform the operations on the versioned data into operations provided by the DBMS.

In section 2, we give an introduction to the object and version model. Section 3 gives a short overview of the complex-object data model which we choose as the target of our transformation process. The mapping of the version model to this complex-object data model is discussed in section 4. We close with a short glance at related work and some conclusions.

2. The object and version model

In this section we give an overview of our object and version model (or OVM, for short). We start with the data modelling facilities including the description of the objects and their constituting elementary objects as well as several kinds of relationships among these components. Then we proceed with the description of versioning facilities, i.e. the version derivation graph and configuration aspects. We end up with the description of the operations concerning the object, version, and configuration handling.

2.1 Schema definition

The definition of (complex) objects

As mentioned in the introduction we have an abstract and a detailed view to complex objects. From the abstract viewpoint an object is described by a number of properties, called attributes. For example, an *Interface* object is identified by its number of pins, the function realized by the corresponding module, etc. A version of an object is also identified by some attributes, one of which is the version number which is unique among all versions of the same object. In order to fit not-versioned objects (e.g. *Instance*) into this schema, we handle them like objects having exactly one version. Other attributes describe the properties which distinguish this version from the others of the same object. The IDENTIFIER attributes introduced serve for a database-wide identification of objects and versions. The versions of one object are organized in a version derivation graph.

Version derivation graph

A version (except the first one) is normally derived from another version. Thus, there is a relationship “derived from” between the versions, which spans a graph among the versions (it is really a graph, because a version may be derived from more than one version, or more than one version may be derived from the same version). Derivation of a version from another version is a model for a creative act of the designer. Hence, from which version(s) a version is derived must be specified by him, because this information cannot be deducted by the system. The shape of the version derivation graph may be restricted in the OVM schema to be a list, a tree or a directed acyclic graph.

The definition of relationships

Furthermore, we have to describe the relationships among the objects and among the object versions (of different objects), respectively. We describe these relationships by the definition of link types. For convenience, we define link types by separate statements, but nevertheless they are part of the object type description. For each object type the link type is named (e.g. *Interface.implements* and

Contents.implements are the names of the link type between *Interface* and *Contents*). The link type definitions are enhanced by cardinality information which describes how many instances of the appropriate type may be linked to one another. For example, each *Interface* may have an arbitrary number of *Contents*, whereas each *Contents* must belong to exactly one *Interface*. We sketch the OVM schema introduced so far for our VLSI design environment in figure 2.1.

```
DEFINE OBJECT_TYPE Interface AS
    Frame, Pin_Interval VERSIONED
    (OBJECT_ATTRIBUTES: if_id:            IDENTIFIER,
                        name:             LIST_OF (CHAR),
                        function:         LIST_OF (CHAR),
                        no_of_pins:       INTEGER,
    VERSION_ATTRIBUTES: scale_factor:     REAL,
                        no_feedthroughs:  INTEGER)
  VERSION DERIVATION GRAPH IS LIST;
DEFINE OBJECT_TYPE Instance AS
    Frame, Pin NOT_VERSIONED
    (OBJECT_ATTRIBUTES: ist_id:           IDENTIFIER,
                        position:         HULL(2));
DEFINE OBJECT_TYPE Contents AS
    Netlist, Wires, Nodes VERSIONED
    (OBJECT_ATTRIBUTES: ct_id:            IDENTIFIER,
                        name:             LIST_OF (CHAR),
    VERSION_ATTRIBUTES: no_of_wires:      INTEGER,
                        length_of_wires:  REAL,
                        no_feedthroughs:  INTEGER);
  VERSION DERIVATION GRAPH IS TREE;
DEFINE CONFIGURATION_TYPE It_Ct_Is
    FOR Interface, Contents, Instance
    (ATTRIBUTES:        conf_id:          IDENTIFIER,
                        name:             LIST_OF (CHAR));
DEFINE LINK_TYPE  Interface.has_Instances (0, VAR),
                  Instance.has_Interface (1,1);
DEFINE LINK_TYPE  Interface.implements (0, VAR),
                  Contents.implements (1,1);
DEFINE LINK_TYPE  Instance.part_of (1, VAR),
                  Contents.consists_of (1,100);
```

Figure 2.1: Object type and link type schema definition

The definition of (elementary) objects and links

Besides object-describing information as shown by the attributes in the schema, there are the object data which constitute the objects. For example, the *Interface* of a module is defined by a rectangle, which we call *Frame*, and the intervals on the frame to which wires can be connected (*Pin_Interval*). A *Frame* is described by its width and height. The *Pin_Interval* is specified by its position on the frame. An *Instance* (of an interface) is characterized by the positions of its *Pins* (which have to be inside the interval specified by the corresponding *Pin_Interval*). *Interfaces* share the *Frame* elementary object with all of its *Instances*, because the positions of *Pins* and *Pin_Intervals* are both relatively to the *Frame*. The *Contents* of an interface consists of a *Netlist* of *Wires* and *Nodes*.

This description shows that our design objects are complex objects consisting of elementary objects and their interconnections which we call elementary links. In our example, there are the elementary object types *Frame* linked to *Pin_Interval* and to *Pin* (which is linked to *Pin_Interval*), *Netlist* linked to *Wire* which again is linked to *Pin* and *Node* (Figure 2.2.)

```
DEFINE ELEMENTARY_TYPE Frame (f_id: IDENTIFIER,
        width: REAL, height: REAL);
DEFINE ELEMENTARY_TYPE Pin_Interval (pi_id: IDENTIFIER,
        name: LIST_OF (CHAR), position: HULL(2));
DEFINE ELEMENTARY_TYPE Pin
        (p_id: IDENTIFIER, position: HULL (2));
DEFINE ELEMENTARY_TYPE Netlist (n_id: IDENTIFIER);
DEFINE ELEMENTARY_TYPE Node (n_id:IDENTIFIER,
        position: HULL (2));
DEFINE ELEMENTARY_TYPE Wire
        (w_id: IDENTIFIER, polygon: LIST_OF (HULL (2)));
DEFINE ELEMENTARY_LINK_TYPE
        Frame.to_Pin_Interval (1,32),Pin_Interval.to_Frame (1,1);
DEFINE ELEMENTARY_LINK_TYPE
        Pin_Interval.to_Pin (0, VAR), Pin.to_Pin_Interval (1,1);
DEFINE ELEMENTARY_LINK_TYPE
        Frame.to_Pin (1,32), Pin.to_Frame (1,1);
DEFINE ELEMENTARY_LINK_TYPE
        Wire.to_Pin (0,2), Pin.to_Wire (0,1);
DEFINE ELEMENTARY_LINK_TYPE
        Wire.to_Node (0,2), Node.to_Wire (3, VAR);
DEFINE ELEMENTARY_LINK_TYPE
        Netlist.to_Wire (1,1), Wire.to_Netlist (1,VAR);
```

Figure 2.2: Database schema of the VLSI database

Configuration

Looking at figure. 2.2 we can recognize elementary link types which belong to elementary object types within one object type (*intra-object link types*, e.g. *Netlist - Wire*) and elementary link types which cross object type boundaries (*inter-object link types*, e.g. *Pin - Wire*). Intra-object links are not affected by versioning, because each object version spans the whole object. Inter-object links span several object versions (of different object types) and hence are affected by versioning, if we generate different numbers of versions for the participating objects. We allow for inter-object links only if the versions of the objects are linked via an object link. Figure 2.3 illustrates the problem using one *Instance* version and two *Contents* versions.

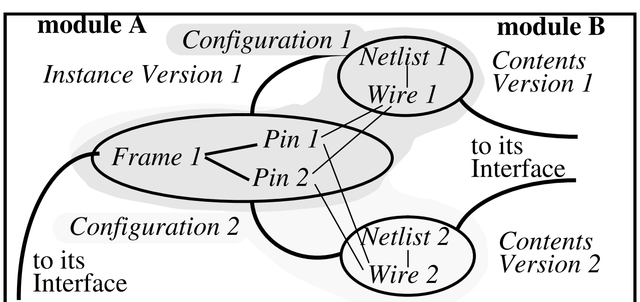


Figure 2.3: Configuration management

Assume, we have an *Instance* of a module called *A* which includes only two *Pins* (Pin 1 and Pin 2). This instance is used to implement module *B*. Thus, the instance of

A is referenced by the first *Contents* version of *B* via the *part_of / consists_of* link. However, on the floorplan level we have to connect the pins of the instance with wires of the contents, i.e. we connect Pin 1 and Pin 2 via *to_Wire / to_Pin* with Wire 1. In a second step, we generate a second version of the *Contents* of the *B* module and we use the same *Instance* of module *A* to implement it. In this *Contents* version we link Pin 1 and Pin 2 with Wire 2. If we do so, we get an inconsistent state of our database, because the schema definition (cf. figure. 2.3) allows at most one wire to be connected to one pin. We could solve this problem by creating a new version of the *Instance* object[1] (with new *Pin* elementary objects which can be connected with *Wire 2*). However, this strategy leads to a snow ball effect if the pins have any inter-object links to other objects (which is not true in our sample database). A second solution for this problem is to define a **configuration** for the *Instance* and *Configuration* objects, i.e. to select one version of object type *Instance* and one version of object type *Contents*[2]. In this case, all inter-object links have to be evaluated within the scope of a particular configuration. Thus, retrieving the versions within such a configuration implies the withdrawal of all inter-object links which do not belong to the versions of the configuration. E.g. creating a configuration for the first *Contents* version and the first (and only) *Instance* version and retrieving the versions of this configuration would withdraw the link to Wire 2 from Pin 1 and Pin 2 (cf. Figure 2.3).

2.2 Operations on objects and versions

Creation of objects and versions

We have to differentiate between the creation of an object itself and the creation of a version of this object. For example, during a hierarchical top down design it is convenient to create objects without knowing any details of the object. Thus, the insert operation creates a new object of a specific type and gives values to all object attributes. At the same time or in a second step the operation allows for the creation of a new (respectively the first) version of this specific object. Thereby, all newly created elementary objects of the version are inserted into the database. The version receives a (system-generated) version number, which identifies the version only in the scope of the corresponding object. Thus, the version with the highest version number is the latest version of the object. After the creation of the version, it has to be connected to the version derivation graph by **CONNECT** (and respectively **DISCONNECT**) operations. For convenience, we can extent the **CREATE** operation by a **AS_SUCCESSOR_OF** clause, which allows for the specification of versions which become predecessors of the version at hand in the version derivation graph.

1. In our example database, we have to generate a new instance object, because instances are not versioned in the database schema.
2. We neglect connections to *Interfaces* in this discussion for simplicity reasons. The whole definition is shown in figure 2.1.

Update of objects and versions

The conventional update operation is not performed by an update in place like in systems without version support, but rather by the creation of a new version. However, there is a need to change organizational data of a version, such as “this version is released by manager x” or “tests α,β,γ have been successfully performed for this version”. Updating these data should be possible without the creation of a new version. Deletion of versions must be provided only due to physical reasons (i.e. overflow of storage space).

Configuring versions

In a first step we have to **CREATE** a configuration giving values to all attributes of the configuration. At the same time or via a **CONFIGURE** operation we can assign a number of versions to this configuration by enumerating them. Due to the hierarchical design method the extension of configurations via the **CONFIGURE** operation is convenient, because we can configure the whole design object (e.g. the whole chip) level by level during the design process.

Retrieval of objects and versions

The **RETRIEVE** operation generates a rather complex result, because we can retrieve information concerning the objects themselves, the object versions, and the elementary objects which belong to the versions. What kind of information is desired has to be specified in the projection clause of the **RETRIEVE** Statement. Analogously, the **WHERE** clause contains conditions which have to be evaluated on the object attributes, the version attributes or on attributes belonging to elementary object types. The result of such a **RETRIEVE** statement is (in general) a set of objects (perhaps of different object types) or a set of configurations, each containing a set of versions, each containing a heterogeneous set of elementary objects (of all elementary objects types belonging to the object type(s)).

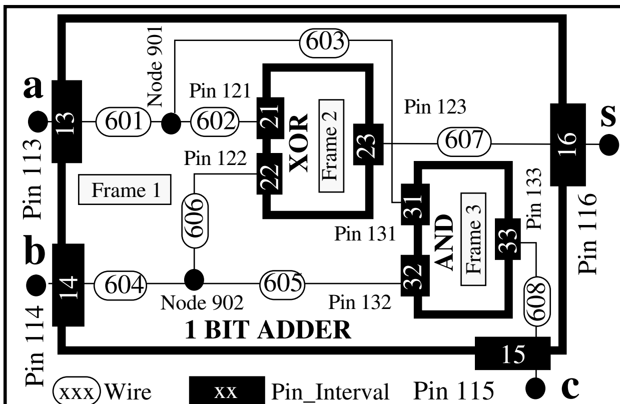


Figure 2.4: Graphical representation of the of the 1-bit adder

We use the well-known SQL-skeleton for the description of our operations:

```
{ RETRIEVE / CONFIGURE / CREATE / UPDATE }
              <projection_list>
FROM          <object_type> / <configuration_type>
WHERE         <qualification_condition>
```

In the following, we illustrate the operations on some examples of our VLSI design environment. Figure 2.4 depicts the detailed data of the first version of the 1-bit adder. In order to insert this data into our design database, we have to execute the following operations:

(1) Creation of the interfaces of the modules 2-bit adder and 1-bit adder (cf. figure 2.5). Since we do not know at this point in time, how the interface should really look like, we generate first empty objects with some basic information.

(2) Generation of the first version of the interface of the adders with the detailed information (i.e. the data concerning the elementary objects and links). Please note that the **CREATE** statement both inserts a version into the specified object and a couple of elementary objects into the elementary object tables. All *Pin_Interval* objects are automatically connected to the *Frame* object during the creation of the *Interface* version (cf. figure 2.5).

```
CREATE Interface
(OBJECT:            name := "1-bit Adder",          ⇒ if_id = 4711
                    function := "s, c := a + b",
                    no_of_pins := 4)
FROM Interface;
⇒ the Interface object gets the IDENTIFIER value "4711".
CREATE Interface
        (VERSION:   scale_factor := 1.0,            ⇒version_no = 1
                    no_feedthroughs := 0)
        Frame       (width := 2.1,                  ⇒ f_id = 1
                     height := 5.8)
        Pin_Interval (name := "x",                  ⇒ pi_id = 13
                     position := [(0,1.4),(0,1.7)])
        …
        Pin_Interval (name := "s",                  ⇒ pi_id = 16
                     position := [(5.8, 1.2),(5.8,1.6)]);
FROM Interface
WHERE if_id = 4711;
```

Figure 2.5: Creation of an *Interface* object

(3) Creation of the three instances of the 1-bit adder (we show only one of them in figure 2.6). The insertion (of the first and only version) of the *Instance* of the 1-bit adder is more complex than the insertion of the *Interface* version. Firstly, we have to use data (i.e. the *Frame* elementary object) which is already inserted into the database, and secondly, we have to connect the new *Pin*s to the appropriate *Pin_Intervals* and we have to link the newly created *Instance* object to the appropriate *Interface* object. In the former case, we reference the stored object by its IDENTIFIER value instead of providing its data. In the latter, we make usage of the **LINKED_TO** clause which allows for the specification of all elementary objects, versions or objects to which the elementary object, version or object at hand has to be linked.

```
CREATE Instance (OBJECT
      LINKED_TO (has_Interface : {4711}           ⇒ ist_id = 711
              position := [(1.0,1.0),(5.8,2.1)]))
      Frame   {1}
      Pin     (position := (0,1.55)               ⇒ p_id = 113
              LINKED_TO (to_Pin_Interval : {13}))
      /* definition of pins with p_id: 114, 115 omitted */
      Pin     (position := (5.8, 1.4)             ⇒ p_id = 116
              LINKED_TO (to_Pin_Interval: {16}))
      FROM Instance;
```

Figure 2.6: Creation of an *Instance* object (1-bit adder)

(4) Generation of the *Contents* for the interfaces (firstly, for the 2-bit adder using instances of the 1-bit adder and secondly, for the 1-bit adder itself). Figure 2.7 shows only the insertion of the first version of the *Contents* of the 1-bit adder, which faces us with a new problem. The netlist is linked to a set of wires and each wire is linked to a pin or node. However, some of these elementary objects are not present in the database, because they are a part of the **CREATE** statement at hand. Therefore, we mark these elementary objects with an temporal IDENTIFIER value (denoted with a leading "#" sign), which we can use in the **LINKED_TO** clause.

```
CREATE Contents
   (OBJECT  LINKED_TO (implements : {4711}),
            name := "XOR, AND")                       ⇒ ct_id = 14711
   (VERSION LINKED_TO (implements : {1}),
            no_of_wires := 8,                         ⇒version_no = 1
            no_feedthroughs := 0)
   Netlist  ()                                        ⇒n_id = 77
   Wire (LINKED_TO (to_Node: {#1}, to_Pin: {113}))   ⇒w_id=601
   Wire (LINKED_TO (to_Node: {#1}, to_Pin: {121}))   ⇒w_id=602
   Wire (LINKED_TO (to_Node: {#2}, to_Pin: {131}),
            polygon := {(0.2,2.0),(2.0,2.0),(2.0,0.9)}) ⇒w_id=603
   /* definition of wires: 604,605,606,607 omitted*/
   Wire (LINKED_TO (to_Pin: {133,115}),    polygon := {(0.3,0.7)})
⇒w_id = 608
   Node     (n_id := #1, position := (1.8, 0.2))      ⇒n_id = 901
   Node     (n_id := #2, position := (0.4,0.3))       ⇒n_id = 902
FROM Contents;
```

Figure 2.7: Insertion of the first version of the *Contents*

If we want to improve our design based on the first *Contents* version, we have to perform the following steps:

(5) Selection of the first version of the Interface and the Contents of the 1-bit adder (cf. figure 2.8). The first statement retrieves only the object information, i.e. no version attributes and no elementary objects.

```
RETRIEVE  OBJECT
FROM      Interface.implements-Contents
WHERE     name = "1-bit Adder";
RETRIEVE  VERSION
FROM      Contents
WHERE     version_no=2 AND Contents.ct_id =14711;
RETRIEVE  VERSION
FROM      Interface.implements-Contents
WHERE     Interface.name = "1-bit Adder" AND
          Interface.version_no=1 AND Contents.version_no=1;
```

Figure 2.8: Selections of objects and versions

(6) Generation of the second (improved) version of the

Contents of the 1-bit adder (cf. figure 2.9) as successor version of version 1. The insertion statement is very similar to that of the first version (cf.Figure 2.7). However, since the second version is only a slight modification of the first version, we do not have to specify the whole data of the version. It suffices to specify only the new elementary objects and to reference the others by their IDENTIFIER values.

```
CREATE Contents AS_SUCCESSOR_OF (1)⇒ version_no = 2
  (VERSION LINKED_TO (implements : {1}),
               no_of_wires := 8,
               length_of_wires := 15.9,
               no_feedthroughs := 0)
  Netlist      {77},
  Wire         {601, 602, 607, 608},
   Wire (LINKED_TO (to_Pin: {132}, to_Node: {901})
               polygon := {(0.2,0.4)})⇒ w_id = 630
  /* definition of wires: 646,660 omitted */
  Wire (LINKED_TO (to_Pin : {#1},to_Node: {#1}) ⇒ w_id = 645
  Node         {901},
  Node         (n_id = #1, position := (0.5,0.3)))⇒ n_id = 920
FROM Contents
WHERE Contents.ct_id = 14711;
```

Figure 2.9: Insertion of another 1-bit adder *Contents* version.

(7) Creation of a configuration, assignment of the versions, and selection of the configured versions and the corresponding elementary objects (cf. figure 2.10).

```
CREATE      It_Ct_Is  ⇒ ct_id = 99
            (ATTRIBUTES: name := "first and only")
FROM        It_Ct_Is;
CONFIGURE It_Ct_Is
  (VERSIONS:    Interface : (4711, version_no = 1),
                Contents : (14711, version_no = 2))
FROM        It_Ct_Is
WHERE       conf_id = 99;
RETRIEVE    VERSION
FROM        It_Ct_Is
WHERE       conf_id = 99;
```

Figure 2.10: Creation and Selections of configurations

Figure 2.11 shows a graphical representation of the objects and the elementary objects which describe the 1-bit adder. The left side of figure 2.11 shows the representation of the first version of the *Contents*, whereas the right side shows the second version. All elementary objects which are shown by dotted lines are identical to the elementary objects of the first version. Furthermore, the *Instance* objects 711, 812, and 913 share a lot of elementary objects (again shown by dotted lines) with the *Instance* objects 712, 813, and 914.Thus, we have to allow for shared elementary objects between versions of the same objects as well as between versions of different objects (of the same object type or of different, but overlapping object types).

Summarizing the modelling facilities and the operations of our version model, we can recognize some important characteristics:

- Objects consist of highly interconnected elementary objects (e.g. object type *Contents*).
- Objects (and therefore object types) are overlapping. For example, *Frame* elementary objects may belong both to *Interface* and to *Instance* objects.
- Additionally to the link types among object types, we have elementary link types which we differentiate in intra-object links (e.g. between *Frame* and *Pin_Interval*) and inter-object links (e.g. between *Pin_Interval* and *Pin*). Thus, we have to handle configurations.

Two consecutive versions of an object often share a lot of data (besides the data which is shared among objects of the same or different types). Thus, it is a vital requirement to share the common elementary objects instead of storing them multiple times.

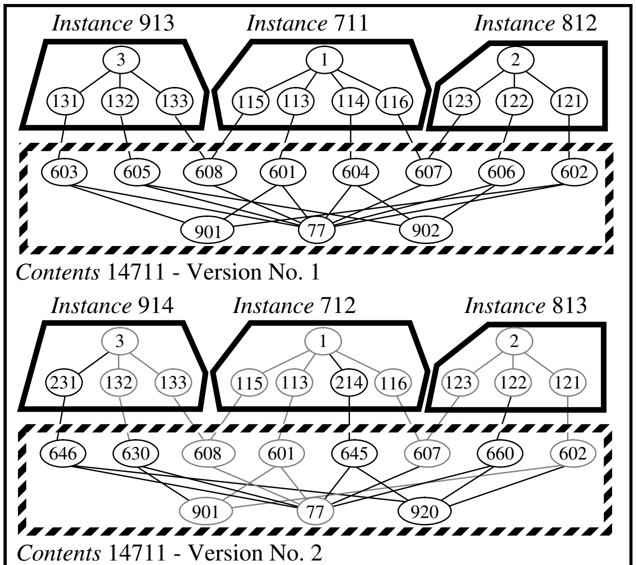


Figure 2.11: Objects and elementary objects of the 1-bit adder

The version model presented here should be seen as a basis for the further considerations rather than a complete description of the versioning facilities needed for VLSI design. We claim that we have incorporated the basic features into our model, but there are lots of "syntactical sugar" which we did not introduce here (and which is not essential for the following considerations), for example the use of user defined keys for the identification of objects rather than the system generated identifiers shown here.

3. The molecule atom data model - overview

In order to implement the OVM by a transformation approach one has to use a complex object data model which allows for a direct mapping of the features identified for the OVM. The well-known NF^2 model [8], for example, does not support the storage of overlapping objects without redundancy. The molecule-atom data model (MAD model) is a complex object data model which seems to be suitable for this purpose. It supports complex objects with shared data and links among basic data elements. In the following, we will introduce the basic properties of the MAD model exemplified by our sample VLSI design environment. However, in this context, there is no notion of objects and versions as in the OVM, because the MAD model does not support versioning. We will first introduce the notions of atoms and molecules, on which the MAD model is based. We will then illustrate the schema definition facilities of the MAD model. Finally, the retrieval and manipulation operations are shortly discussed.The mapping of the OVM to the MAD model is discussed in section 4.

3.1 The MAD model's building blocks:

The basic building blocks of the MAD model objects are *atoms*, which may be compared to tuples in the relational model. Atoms belong to a exactly one *atom type* (analogous to a relation) and consist of a set of attributes of various types. Besides the data types known from many implementations of the relational model, the MAD model allows for some additional data types, e.g. a TIME type and a HULL type, which represents n-dimensional rectangles. For example, HULL(1) represents lines in a 1-dimensional space.

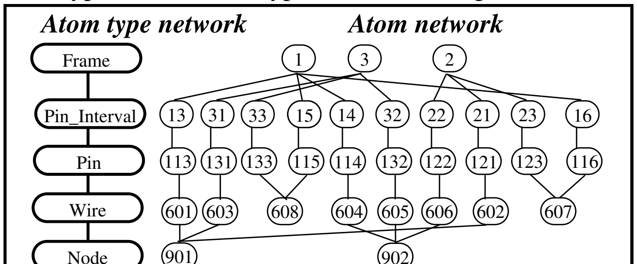
Two sample atom type definitions

```
CREATE ATOM_TYPE Frame
  (f_id              : IDENTIFIER,
   width             : REAL,
   height            : REAL,
   to_Pin            : REF_TO (Pin.to_Frame)(1,32),
   to_Pin_Interval   : REF_TO (Pin_Interval.to_Frame)(1,32));
CREATE ATOM_TYPE Pin_Interval
  pi_id              : IDENTIFIER,
  name               : LIST_OF (CHAR),
  position           : HULL (2),
  to_Frame           : REF_TO (Frame.to_Pin_Interval)(1,1)),
  to_Pin             : REF_TO (Pin.to_Pin_Interval) (0,VAR);
```

this pair of REF_TO attributes realizes the link between Frame and Pin_Interval

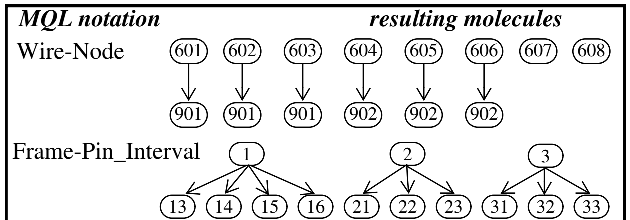
Example 3.1: MAD schema for a part of our environment

Attributes of type REF_TO are used to link atom types to one another. The value of a REF_TO attribute is a duplicate-free list of IDENTIFIER values, all pointing to atoms of the same type. This link is symmetric, i.e., if an atom type *A* contains a REF_TO attribute pointing to atom *B*, then *B* must contain a REF_TO attribute pointing to *A*, too. Analogously, if the REF_TO attribute of an atom *a* of type *A* contains the IDENTIFIER value of atom *b* of type *B*, the corresponding REF_TO attribute of *b* must contain the IDENTIFIER value of *a*. The two REF_TO attribute values together form a link between *a* and *b*. Example 3.1 shows the MAD schema for a part of the VLSI environment presented in section 2. Example 3.2 depicts on the left side five atom types and the link types defined among them, and on the right side a sample atom network of these atom types which represents a part of our VLSI example (a graphical view to the data is shown in Figure 2.4). The links among these atoms as well as the link types are undirected.



Example 3.2: A sample database

Due to the symmetrical link concept, the atom types in a MAD schema form an undirected network, as well as the atoms of a MAD database do. Based on these networks, *molecule types* and *molecules* may be defined by specifying a coherent subgraph of the atom type network and assigning a direction to each link type, such that the resulting directed graph has exactly one root (the so-called *root atom type*). Molecules are formed from the atom network by following the links among the atoms in the direction defined by the molecule type definition. Example 3.3 illustrates molecule type definitions and the resulting molecules. For each atom of the root atom type there is exactly one molecule.



Example 3.3: Sample molecule type definitions:
MQL notation, and resulting molecules

3.2 Retrieval and manipulation operations

The MAD model's query language is called MQL. An MQL retrieval operation has the following shape

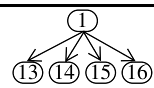
SELECT	projection clause
FROM	molecule type definition
WHERE	restriction clause

The **FROM** clause contains molecule definitions. The **SELECT** clause is used to project only those parts of the molecule which are interesting. One can project all parts of a molecule (**ALL**), atoms of certain types or only some attributes of certain atom types. Furthermore, one can project atoms of a certain type only if they fulfil a qualification (qualified projection). The **WHERE** clause allows for a restriction of the molecule set specified in the **FROM** clause.

Example 3.4 shows two sample queries and their results based on the atom network shown in example 3.2. Query 1 illustrates that the condition used to select a specific molecule is not restricted to the root atom of a molecule. Query 2 shows an example of a qualified projection.

In addition to the **SELECT** statement there are manipulation operations which also work on sets of molecules. Example 3.5 shows an INSERT statement inserting one molecule consisting of one *Pin_Interval* atom, one *Pin* atom and one *Wire* atom, and connects it to the specified *Frame* atom. The *Pin* atom gets an additional reference to a *Frame* atom which is specified by a query, whereas the additional reference of the *Wire* atom is specified directly by the appropriate IDENTIFIER value of a *Node* atom. In general an arbitrary number of molecules may be inserted at a time.

Query 1 and its result



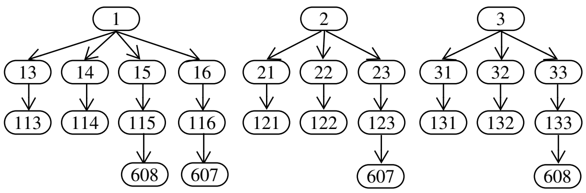

```
SELECT   ALL
FROM     Frame-Pin_Interval
WHERE    EXISTS Pin_Interval: Pin_Interval.name='S'
```

Find the Frames which have a Pin_Interval "S"

Query 2 and its result

```
SELECT   Frame, Pin_Interval, Pin,
         SELECT   Wire
         FROM     RESULT
         WHERE    NOT EXISTS Node
FROM     Frame - Pin_Interval - Pin - Wire - Node
```

Show all Frames, their Pin_Intervals, their Pins, and those of their Wires which are not connected to a Node



Example 3.4: Sample MQL queries and their results based on the atom network of example 3.2

Among the schema definition and manipulation statements is the **CREATE ATOM_TYPE** statement shown in example 3.1. As a special feature of the MAD model, the schema may be modified dynamically, not only by defining or deleting atom type definitions, but also by removing or adding attributes from atom types. The corresponding modification statements are **SHRINK ATOM_TYPE** and **EXPAND ATOM_TYPE**. The representation of atoms in our database allows for an attribute to be added without any modification to the data in the database. More information about the MAD model can be found in [6, 7].

```
INSERT   Pin_Interval (name:= 'Just another pin'),
         Pin (  position:= (0.2,0.4),
                to_Frame:= (  SELECT   Frame(f_id)
                              FROM     Frame
                              WHERE    Frame.position=(0.0))),
         Wire (to_Node:= {905})
FROM     Pin_Interval-Pin-Wire
WHERE    Frame.f_id = 1
```

Example 3.5: Sample **INSERT** statement

4. Mapping the OVM onto the MAD model

In this section we discuss the mapping of the OVM onto the MAD model. Obviously, our view to complex versioned objects was already influenced by the MAD model ideas. This allows for a straightforward mapping of objects and versions to molecules. We first show the transformation of the schema defining OVM operations, then the transformation of OVM operations to operations of the MAD model.

4.1 OVM schema to MAD schema transformation

We must represent the fact that there are multiple versions of an object, each having describing attributes. Thus, we map the components of an object to a so-called object molecule consisting of a data part constructed from the components of the object type (the elementary object types), and a description part containing the object attributes and the version attributes. In order to obtain the data part, the **DEFINE ELEMENTARY_TYPE** statements are simply transformed to **CREATE ATOM_TYPE** statements. The **DEFINE ELEMENTARY_LINK_TYPE** statements are realized by adding appropriate REF_TO attributes to these atom types (cf. example 3.1). The description part is constructed when defining a (versioned) object type. An additional REF_TO attribute is added to each atom type involved in order to be able to link the atoms to a version representative. This version representative is an atom describing a specific version, i.e. it stores the version-describing attributes. All version representatives of the same object are linked to the object representative atom which stores the object-describing information. The version representatives are also used to model the version derivation graph, by simply introducing "forward in the graph" and "backward in the graph" REF_TO attributes. The names of the version representative and of the object representative are derived from the object name by adding the suffix _VSN and _OBJ, respectively.

Hence, a **DEFINE OBJECT_TYPE VERSIONED** statement is mapped to a set of **CREATE ATOM_TYPE** and **EXPAND ATOM_TYPE** statements as shown in example 4.1 (1). The mapping of **DEFINE OBJECT_TYPE NOT_VERSIONED** is done analogously. **DEFINE LINK_TYPE** statements are realized by a pair of **EXPAND ATOM_TYPE** operations as shown in example 4.1 (2). The statement **DEFINE CONFIGURATION_TYPE** is also mapped to a **CREATE ATOM_TYPE** statement including the definition of REF_TO attributes pointing to the corresponding version representative atom types and a set of **EXPAND ATOM_TYPE** statements.

To summarize the transformations discussed above, we can state that we keep the version management information separate from the complex-object data, thus leaving the proper object structure almost unchanged (with exception of the additional REF_TO attributes). This separation is also reflected in the transformation of our version operations to MAD model operations.

4.2 Transformation of OVM operations

When an object is created, object-describing attributes are specified. The creation of a version implies the insertion of all atomic objects belonging to the version[1] and the insertion of the version representative, as it is shown in example 4.2. The version has to be connected to the version graph, as it is specified either in the **CREATE** statement or in a separate **CONNECT** statement. This is done by setting the ap-

1. Atomic objects have to be inserted, if they are not yet known in the database. Otherwise, they are represented only by references in the **CREATE** statement and also in the resulting MAD model operation.

propriate REF_TO attribute of the new version. Updates of the version (object) describing attributes are transformed to updates on the version (object) representative. Obviously, the transformation process is very simple for CREATE statements. The object name of the CREATE statement is replaced by a corresponding molecule definition in the MQL statement. The assignments of attribute values can be included unchanged into the MQL statement. LINKED_TO clauses are transformed to assignments to the REF_TO attributes representing the link. Only for versions there is a little difficulty. Version representatives are linked to versions of the object to which the object representative is linked. Therefore, in the OVM the specification of the version number is sufficient. In the MQL statement this version number has to be transformed into the internal identifier of the version atom. This is done by a SELECT statement within the assignment to the REF_TO attribute.

Statement to be transformed:

```
(1)DEFINE OBJECT_TYPE Interface AS
      Frame, Pin_Interval VERSIONED
      (OBJECT_ATTRIBUTES:
         if_id:               IDENTIFIER,
         name:                LIST_OF (CHAR),
         function:            LIST_OF (CHAR),
         no_of_pins:          INTEGER,
      VERSION_ATTRIBUTES:
         scale_factor:        REAL,
         no_feedthroughs:     INTEGER)
      VERSION DERIVATION GRAPH IS LIST;
(2)DEFINE LINK_TYPE  Interface.has_Instances (0, VAR),
                     Instance.has_Interface (1,1);
```

Resulting MQL-Statements:

```
(1)CREATE ATOM_TYPE Interface_OBJ
         (if_id           :  IDENTIFIER,
         name             :  LIST_OF (CHAR),
         function         :  LIST_OF (CHAR),
         no_of_pins       :  INTEGER,
         versions         :  REF_TO (Interface_VSN.object));
   CREATE ATOM_TYPE Interface_VSN
         if_v_id          :  IDENTIFIER,
         version_no       :  INTEGER,
         scale_factor     :  REAL,
         no_feedthroughs  :  INTEGER,
         forward          :  REF_TO (Interface_VSN.backward) (0,1),
         backward         :  REF_TO (Interface_VSN.forward)(0,1)
         Frame_Ref        :  REF_TO (Frame.Interface_VREF),
         Pin_Interval_Ref :  REF_TO (Pin_Interval.Interface_VREF),
         object           :  REF_TO (Interface_OBJ.versions));
   EXPAND ATOM_TYPE Frame BY
         Interface_VREF: REF_TO (Interface_VSN.Frame_Ref);
   EXPAND ATOM_TYPE Pin_Interval BY
         Interface_VREF: REF_TO (Interface_VSN.Pin_Interval_Ref);
(2)EXPAND ATOM_TYPE Interface_OBJ BY has_Instances :
         REF_TO (Instance_OBJ.has_Interface) (0, VAR);
   EXPAND ATOM_TYPE Interface_VSN BY has_Instances :
         REF_TO (Instance_VSN.has_Interface) (0, VAR);
   EXPAND ATOM_TYPE Instance_OBJ BY has_Interface :
         REF_TO (Interface_OBJ.has_Instances) (1, 1);
   EXPAND ATOM_TYPE Instance_VSN BY has_Interface :
         REF_TO (Interface_VSN.has_Instances) (1, 1);
```

Example 4.1: Transforming **DEFINE OBJECT_TYPE** and **DEFINE LINK_TYPE** statements

Statement to be transformed:

```
CREATE Contents
(OBJECT LINKED_TO (implements : {4711}),
                  name := "XOR, AND")
   (VERSION LINKED_TO (implements : {1}),
                  no_of_wires := 8,
                  no_feedthroughs := 0)
   Netlist        ()
   Wire           (LINKED_TO     to_Node: {#1},
                                 to_Pin: {113}))
   Wire           (polygon := {(0.3,0.7)},
                  LINKED_TO (to_Pin: {133,115})
   Node           (n_id := #1,
                   position := (1.8, 0.2))
FROM Contents;
```

Resulting MQL-Statement:

```
INSERT Contents_OBJ
      (name := "XOR, AND", implements:={4711}),
      Contents_VSN (    no_of_wires := 8,
                        no_feedthroughs := 0,
                        implements :=
                  (SELECT  if_v_id
                   FROM    Interface_VSN - Interface_OBJ
                   WHERE   if_id = 4711 AND version_no=1)),
      Netlist (),
      Wire (to_Node := {#1}, to_Pin := {113}),
      Wire (polygon := {(0.3,0.7)}, to_Pin := {133,115}),
      Node (n_id := #1, position := (1.8, 0.2))
FROM  Contents_OBJ - Contents_VSN - Netlist - Wire - Node;
```

Example 4.2: Transformation of a **CREATE** statement handling versions

When retrieving data, one has to pay special attention to links which span several versions because of data sharing among versions: only those atoms shall be retrieved which really belong to the desired versions. The same problem arises if REF_TO attributes are to be selected: the intersection of its actual value and the version references has to be computed. This does not concern the links between the atoms of the versions which are used to construct the molecule, because they are maintained when the molecule is built up. We use the qualified projection offered by MQL to restrict the atoms to be projected to those atoms, which are also referenced by the version representative. Apart from

Statement to be transformed:

```
RETRIEVE VERSION FROM Contents
WHERE version_no=2 AND Contents.ct_id =14711;
```

Resulting MQL-Statement:

```
SELECT Contents_VSN, Netlist,
      SELECT Wire,
            SELECT  Pin
            FROM    RESULT
            WHERE   Pin.p_id ELMT Contents_VSN.Pin_REF,
            SELECT  Node
            FROM    RESULT
            WHERE   Node.n_id ELMT Contents_VSN.Node_REF
      FROM RESULT
      WHERE Wire.w_id ELMT Contents_VSN.Wire_REF
FROM Contents_VSN - (Contents_OBJ, Netlist - Wire - (Pin, Node)
WHERE Contents_VSN.version_no = 2 AND
                  Contents_OBJ.ct_id = 14711;
```

Example 4.3: Transformation of a retrieval statement this problem, retrieval statements are treated analogously to **CREATE** statements.The root of the molecule is the configuration representative atom in the case a whole configuration is retrieved, the object representative atom or the version representative atom when objects or versions are retrieved. If versions are qualified by means of object attributes, the object representative atom has to be included into the molecule definition. Example 4.3 shows a sample query retrieving version no. 2 of the *Contents* object 14711.

5. Summary and Conclusions

We have presented a general version model and its mapping onto a complex-object data model. The version model is capable of handling versions of complex objects, which may be organized in arbitrary version graphs. The retrieval facilities allow for more powerful operations than most of the other proposals do. One can project version graph information, object data, or an arbitrary mixture of both.

The mapping onto a complex-object data model exhibits several advantages. Firstly, the version model's implementation is as efficient as the implementation of the complex-object data model. Furthermore, redundancy elimination on the atom level can be done automatically by the system, without burdening the user or the version component with this aspect (which is neglected in other proposals, e.g. [2]). Of course, the atoms themselves may vary only in some of their attribute values. This raises the question, whether redundancy elimination also should be employed at the attribute level. The MAD model offers the utilities to do so,; an example of redundancy elimination at this level with the help of the MAD model can be found in [5].

The clustered storage of complex objects, versions, design objects etc. can be delegated to the complex-object database system.

Comparison to Other Work

Our version model covers the proposal of [2], where design objects, versions, and generalized references were the building blocks of the version model. Like in their proposal, our design data remain unchanged in structure compared to the non-versioned approach. The clustering of versions to revisions and alternatives as proposed there is also possible in our model, either by extending the version graph modeling (allowing for an additional hierarchy graph), which leads to arbitrarily complex clusters, or by introducing revision and alternative atoms as additional stages between object and version (which would enforce a clustering of this form). We did not present such extensions in this paper in order to emphasize the principles of our ideas. However, the way how such a feature can be integrated should be obvious to the reader. [2] lacks the description of retrieval facilities, and it is not clear, whether retrieval can be directed by design object data rather than version properties.

[4] emphasizes the need for a version graph, which is also supported by our model. Equivalence relationships as presented in [4] could be integrated in the same way into our system as the revisions and alternatives discussed in the comparison to [2]. [1] introduces a *version generalization*, which groups the versions of one object together. This is equivalent to our link between object and version atoms.

Conclusions

Looking at the transformations necessary to perform the mapping of a quite general object and version model onto a common-purpose complex-object data model, one can conclude that managing versions of complex objects according to our version model is apparently not more complicated than managing complex objects. However, it should be stressed that a prerequisite for the transformation presented in this paper is the capability of the underlying complex-object data model to handle network-like structures.

Our version model can easily be extended by additional relationships among versions and objects without affecting the design object data. Thus, a flexible tailoring to the needs of a specific environment is feasible. The OVM is implemented on top of the PRIMA [3] system (an implementation of the MAD model) and is currently being integrated with a VLSI design environment.

6. References

[1] Batory, D.S., Kim, W.: Modeling Concepts for VLSI CAD Objects, in: ACM TODS, Vol. 10, No. 3, 1985, pp. 322-346.

[2] Dittrich, K.R., Lorie, R.A.: Version Support for Engineering Database Systems, in: IEEE TOSE-14 (1988), No. 4, pp. 429-437.

[3] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications, in: Proc. VLDB'87, Brighton, UK, 1987, pp. 433-442.

[4] Katz, R.H., Chang, E., Bhateja, R.: Version Modelling Concepts for Computer-Aided Design Databases, in: Proc. SIGMOD'86, Washington D.C. USA, 1986, pp. 379-386.

[5] Käfer, W., Ritter, N., Schöning, H.: Support for Temporal Data by Complex Objects, Proc. VLDB90, Brisbane, Australia, 1990, pp. 24-35.

[6] Mitschang, B.: Towards a Unified View of Design Data and Knowledge Representation, in: Proc. 2nd Int. Conf. on Expert Database Systems, 1988, pp. 33-50.

[7] Schöning, H.: Integrating Complex Objects and Recursion, in: Proc. DOOD'89, Kyoto, Japan, 1989, pp. 535-554.

[8] Schek, H.-J., Scholl, M.H.: The Relational Model with Relation-Valued Attributes, in: Information Systems, Vol. 11, No. 2, 1986, pp. 137-148.

[9] Zimmermann, G.: Top-Down Design of Digital Systems, in: Logic Design and Simulation, E. Hörbst (Ed.), Elsevier Science Publ., B. V., 1986.