SEMANTICS OF CAD OBJECTS FOR GENERALIZED DATABASES

RIEU D. , NGUYEN G.T

IMAG
Universite de Grenoble
Laboratoire de Genie Informatique
BP 68
38402 St-MARTIN-D'HERES
France

Abstract : This paper presents some fundamental issues related to the design and implementation of CAD oriented database systems. Integrating databases in sophisticated CAD environments requires functionalities usually not provided by existing systems, e.g relational or entity-relationship systems. For instance, consistency controls have to provide richer and more flexible features than the former "take it or leave it" paradigm.
The basic aspects are examined here with respect to the **nature** of CAD objects, to their consistency **states** and to their **dynamicity**. The nature of CAD objects concerns the complexity of their structure and of their relationships. The state of CAD objects concerns their completeness with considerations to their consistency. Finally, the dynamicity of CAD objects concerns the evolving nature of their structure and of their property values, i.e their behavior, with connections to versions and representations.
These concepts are defined and detailed with examples drawn from VLSI circuit design. The functionalities of a prototype CAD database system offering new solutions in these areas are presented, calling upon object-oriented concepts and logic-programming. It is currently being tested for VLSI circuit design in cooperation with the CAD Research Dept at CNET.
Mail : !mcvax!inria!imag!rieu or !mcvax!inria!imag!nguyen
This work is supported in part by the French Department of Telecommunications (CNET contract 843B061) and the French Department of Research (contract PRC/BD3 850222112111).

1. INTRODUCTION

This paper presents some fundamental issues concerning the design and implementation of CAD oriented database systems. Integrating databases in sophisticated CAD environments requires functionalities usually not provided by existing systems, e.g relational or entity-relationship databases [7, 9, 10]. For instance, consistency controls have to provide richer and more flexible features than the former "take it or leave it" approach [3, 6].
The basic features required for integrated CAD applications are examined here with respect to the **nature** of CAD objects, to their consistency **states** and to their **dynamicity**.
The nature of CAD objects concerns the complexity of their structure and of their semantic relationships.
The state of CAD objects concerns their completeness with considerations to their consistency.
Finally, the dynamicity of CAD objects concerns the evolving nature of their structure and of the property values, i.e their behavior, with connections to versions and representations.
These concepts are defined and detailed with examples drawn from VLSI circuit design [1, 5]. The functionalities of a prototype CAD database system that provides new solutions in these areas are presented, calling upon object-oriented concepts and logic-programming [11, 13]. It has been implemented and is currently being tested for VLSI circuit design in cooperation with the CAD Research Dept at CNET.

2. NATURE OF CAD OBJECTS

The **nature** of CAD objects concerns their **static** description. It includes the structure of their intended properties and their relationships to other objects, i.e their position with respect to a given **environment**.
An object for short is a basic item or a structured collection of related objects, i.e its components. Fundamentally, these components may be defined independently, or intimately tied to the object.
Basically, CAD objects have a complex structure, and may further include a large number of components. They also bear properties which are subject to manipulations by the users.
The potential actions are performed by specific engineering tools. They can :
- derive calculated objects from argument objects,
- control the consistency of the object with respect to particular specifications or with respect to other objects characteristics.
Such specifications entail specific tools which require the definition of :
- **functional relationships** between the objects involved for the computation of the derived objects,
- **integrity constraints** for the definition and control of consistency.

2.1 Object definition

The main characteristic of CAD oriented database systems is to integrate :
- the modelling of complex structures,
- specific engineering tools,
- consistency control and enforcement rules,
- semantic relationships between objects. In our approach, an object is modelled as a **structured set occurence**. This set defines objects having similar properties and compliant with identical constraints.

An object is defined by :
- **properties** concerning its structure,
- **constraints** defining the relationships among its properties,
- **functional links** defining the computation rules to derive an object from others.

The functional links call upon specific **functions** defined with specific engineering tools. They are implemented with ad-hoc programs.
For example, VLSI circuits are designed in several steps, including : functional specifications, logic specifications, electric specifications, layout design, which yield different **representations** at increasing levels of details.
Our approach is to integrate these representations within the same formalism. It uses a specific object-oriented approach. It is implemented in Prolog [13].
In the following, a **cell** is a particular representation of a given circuit.
A cell definition includes a **type** (functional, logic, electric ...), an **interface** and a **structure**.
- the interface defines the external view of the cell. It includes an **envelope** and **io-ports** (Figure 1) :

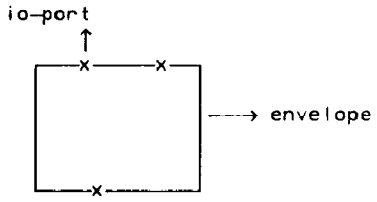

Figure 1. Interface of a cell.

- the structure defines the internal view of the cell. It includes **components** and **nets** that connect the components together and with the cell io-ports (Figure 2).

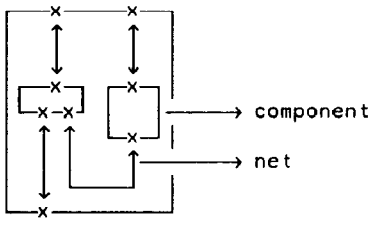

Figure 2. Structure of a cell.

- the following is a simplified definition of a cell in our system (Figure 3) :

```
DEF_E Cell
| type :     Typerep  ⟶ functional or logic...

| envelope: Polyline ⟶ interface
| io-ports: LIST (Port)

| comp :     LIST (Component)
| equi :     LIST (Net)

| connect (entsort, ensort.realize.comp, equi)
| NOT intersect (comp) ⟶ C2   C1 ↑
| ...
FDEF_E
```

Figure 3. Definition of a cell object.

It is assumed in this example that Typerep, Component, Net and Polyline are predefined types. The functions connect and intersect are also supposed predefined.
C1 and C2 are integrity constraints relevant to electric and geometric design rules.
Constraint C1 states that the nets connecting the io-ports of a cell and the implementations of the io-ports of its components must be complete and consistent. The connect function is implemented by a specific program that controls that every input port of a cell is the starting point of some net.
C2 is a geometric constraint stating that no components may overlap.

2.2 Constraints and relationships

It is assumed that links and constraints defined for an object as above hold for every occurence in the object set. They define **general knowledge** about generic objects.
Besides these basic facilities for defining objects and constraints, **specific** knowledge about particular object instances can also be defined, e.g how to design some particular cell from specific library components, what engineering tools must then be used.
For instance, if some object "A" is defined by a function f(B,C), each occurence of "A" is specifically related to the occurences of objects B and C in the database, i.e a component C2 may be designed as a sub-component of cell C1. They are related by an existential relationship.
In our system, there exist **functional or existential relationships**. Specific relationships hold as long as the argument objects are not modified. In the example above, object "A" is designed according to f(B,C). The relationship holds as long as they are not modified.
The properties of structured object sets are defined by their name (e.g in the example above Figure 3 : comp) and the associated domain (example : Components). The latter must be previously defined in the database. It follows that :
- only one notion is needed to define **object sets** and **object property domains**. A domain references its set definition and its occurences altogether.
- this puts an emphasis on **upward** design methodologies, since object definitions involve only existing definitions.
- it partially solves the probleme of cyclic object definitions. Particular cases are **recursive** object definitions and **cross-referenced** definitions.

2.3 Cyclic and recursive object definitions

A **recursive** object definition references the object itself. A **cross-referenced** object definition references an object definition referencing in turn the original object.
Following the assumptions made so far, an object "A" cannot reference itself since it is not already known in the database. Similarly, an object A cannot reference object B if it has not yet been defined.
The following relationship specifies that object A_{i+1} has a property A_i: $A_{i+1} \rightarrow A_i$. The cyclic definition below is prohibited (Figure 4) :

```
Ai ---> Ai-1 ---> Ai-2
 ^                  |
 |                  v
A1 <--- A2 .... Ai-n
   (1)
```

Figure 4. Cyclic definition.

However, usual engineering design applications do need cyclic definitions. Statements such as :

- a cell is composed of other cells,
- cell components implement particular cell definitions,

are commonly used in VLSI design applications [2].

Our system allows such definitions and provides for their control through **specialized set** definitions. They use the usual concept of **isa** relationship, augmented with specific constraints and links attached to the specialized set properties. These mechanisms are detailed in section 3.

The semantics associated with recursive and cross-referenced definitions is different.

For recursive definitions, the existence of primitive objects is assumed, e.g to allow a super-object implementation from the sub-objects. For cross-referenced definitions, the existence of primitive and completed cells is assumed.

Here, the solution to these definitions is to break the cycle and to define a lower level object. The primitive objects and the component cells of the cell definition all belong to the lower objects occurences. A recursive cell definition, say "Cell", thus references a lower level definition, say "Cellcons", which inherits its properties through an **isa** statement, noted "=>".

```
- 1  Definition of cells                Cell => Cellcons
      DEF_E Cell ... FDEF_E                 ^      |
                                            |      |
- 2  Definition of completed cells          |      |
      DEF_E Cellcons ISA Cell               |      |
      | comp : LIST (Cell)                  |      |
      | ....                                |      |
      FDEF_E
```

Similarly, cross references between two object definitions, say "Cell", and "Component", are solved by the introduction of a third set definition, say "Cellcons", that inherits from "Cell" all its properties through an **isa** statement. It is introduced between the "Cell" and its "Component".

```
- 1 Definition of cells              A        ISA
    DEF_E Cell ...ft QDEF_E        Cell => Cellcons
                                     ^        |
- 2 Definition of components         |        |
    DEF_E Component                  |        |
    | implements : Cell              |        |
    | ...                            |        |
    FDEF_E                           |  Component <-|
                                            B

- 3 Definition of "isa" cells
    DEF_E Cellcons ISA Cell
    | comp: LIST (Component)
    | ...
    FDEF_E
```

The "Component" references the "Cell". But now the "Cell" is related to the "Cellcons" by an **isa** relationship. Further, only the "Component" now references the "Cell" (Figure 5).

```
Ai => AIi ---> Ai-1 models the      Ai ---> Ai-1 ---> Ai
 ^                |  cyclic          ^                |
 |                v  definition :    |                v
A1 <--- A2 ... Ai-2                 A1 <--- A2 ... Ai-n
```

Figure 5. Modelling a cyclic definition.

Generally speaking, the rule is to break a cyclic definition in two parts by introducing a new **isa** set definition, that inherits from one of the objects in the cycle.

3. CONSISTENCY AND COMPLETENESS OF CAD OBJECTS

The **state** of a CAD object reflects the evolving design process for that object. It includes the notion of **completeness** and the notion of **consistency**.

During the design process, the object is most of the time modified and incomplete. The designers have to cope with this **imperfect knowledge**. Further, the trial and correct framework of usual design methodologies imply the existence of **temporary inconsistencies** that must be taken into account.

Warning the user of such inconsistencies is worth only if they result from a constraint violation, and not from an **undecidable** control. Indeed, logic applied to databases tells us that incompleteness implies inconsistency [11]. This is ineffective in CAD environments because it is of first importance to detect the following situations :

- incomplete and consistent, which means that no decidable constraint control has resulted so far in a constraint violation, whatever the completeness of the object,
- incomplete and inconsistent, meaning that some decidable control has resulted in a constraint violation.

In our approach, object properties instantiation can be **delayed**. The identification of the occurence is only needed. Further, as long as no constraint is violated, the system presumes that no inconsistency appears for the object. It is therefore assumed consistent as long as no constraint is explicitly violated, otherwise contradicted by the instantiation or modification of properties.

Consistency and completeness are dynamically examined to reflect the exact state of the object after every modification or update. This gives a presumably exact picture of the most recent state of the design, which can be one of the following :

- incomplete and consistent, meaning that the design is correctly in progress,
- incomplete and inconsistent, meaning that the design does not improve,
- complete and inconsistent, meaning that the design is wrong,
- complete and consistent, meaning that the design is correct.

Consider the following example (Figure 6). A net is defined by input and output ports. The layout is considered as a list of segments.

```
DEF_E Net
| /* define net set */
| volt   :  Real                  --> voltage
| in     :  Port                  --> in port
| out    :  Port                  --> out port
| layout :  LIST (Segment)
| compatible (in, out)            --> C1
| graphics:= linkport(in,out)     --> L1
FDEF_E
```

Figure 6. Definition of a net object.

Constraint C1 defines the connection between input and output ports. Function L1 defines the graphic layout of a segment connecting two ports.
Figure 7 corresponds to the definition of a cell with two components. Net e1 connects port p1 to port p2. Constraint C1 is decidable and reflects the consistency of net e1. Further, e1 is complete because function L1 produces automatically the associated segment.

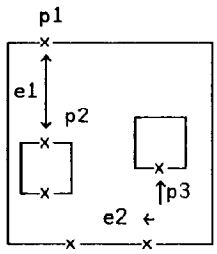

Figure 7. An incomplete cell object.

Net e2 includes only one port, i.e p3. Its occurence is therefore incomplete, because one port is unknown and its associated segment is thus undefined. In this case the combination "incomplete-consistent" is produced. This will eventually be modified upon instantiation of the second port p4 for net e2, yielding the combination "complete-consistent" if C1 still holds, yielding "complete-inconsistent" if not.

4. DYNAMICITY OF CAD OBJECTS

CAD objects are subject to modification that originate from two different sources :
- explicit requests issued by the designers,
- implicit updates derived from the modification of some related objects.

We first detail the implicit dynamicity of CAD objects and further detail issues concerning the explicit dynamicity (Section 5).
CAD oriented databases include objects linked through intricate semantic relationships. Functional and existential relationships have been defined above. **Generic** relationships are also relevant to sets of objects, e.g a net is positionned with respect to the coordinates of its io_ports, the symbolic representation of a circuit is derived from its logic representation. **Specific** relationships can also be associated with particular instances of an object, e.g the layout L1 of a circuit results from geometric transformations applied on some other particular layout L2.
Consistency controls for such relationships require powerful mechanisms for **propagating updates** and **deriving** information. Indeed, CAD objects must evolve with respect to the modifications performed on the related objects.
In our system, updates on object occurences are propagated to all objects that use them. This is computed by recalculating all the relationships involved. Therefore a modification performed on a circuit C1 will be immediatly propagated to its layout L1 and further to its derived layout L2.
Basically, propagating updates is relative to the **context** in which the object involved has been created. Subsequently, they are propagated into the contexts in which they are used. This prohibits illegal updates and provides for consistent modifications.
Therefore, implicit dynamicity results from the maintenance of consistent **generic** or **specific** knowledge on the objects. In contrast, **explicit** dynamicity results from the non deterministic nature of the design process.

5. EXPLICIT DYNAMICITY

The evolving nature of CAD objects results from several considerations. Among these stands fundamentally the methodologies usually of concern in the design process. In our aproach, they call for several successive transformation phases involving three aspects :
- the structural level, concerned with object description,
- the morphologic level, concerned with alternate representations,
- the resolution level, concerned with details of implementation.

These three aspects are tightly connected. A particular design follows a specific path in the three dimensional space described in Figure 8. We describe in the following sections the functionalities implemented in our system to handle the design process dynamicity in the three directions : structural (Section 5.1), resolution (Section 5.2) and morphologic (Section 5.3).

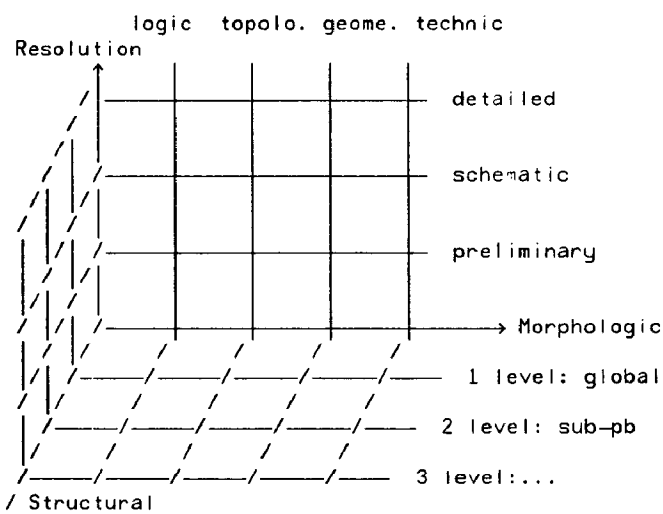

Figure 8. Three dimensional design space.

5.1 Structural dynamicity

The stuctural design is related with the decomposition of the design goal into sub-goals. In section 2, focus has been directed towards **generalized aggregates**, enhanced with **consistency constraints** and **functional links**. Altogether, they provide a good mean for modelling objects and further decompose it into sub-objects.
Structural design consists in modifying and enhancing a given object model, in order to provide a more detailed model.
CAD database systems must therefore provide **extensible** data structures on which the designers may define new properties and new constraints. These data structures must also be able to **evolve** to accept the modifications of existing properties and constraints.
Our system supports structural design in that its **integrated** description and manipulation language allows for object structures enhancements and modifications.

The designer may update an object schema at any time by adding or deleting properties, constraints or links.

5.2 Resolution dynamicity

The resolution dynamicity is related to the top-down design methodology. It can be decomposed in preliminary, schematic and detailed designs. A model for CAD objects must therefore support successive improvements of the object models.

Improving an object model usually corresponds to a particular design attempt. Therefore a specialization mechanism for CAD objects must be the least compelling possible. For instance, two specialized subsets of some object are not necessarily disjoined. If object "B" is a specialization of some object "A", and "B" is not satisfactory, the designer must be able to start again from object "A" without worrying about "B". He must also be able to eventually backtrack to "B" if he wishes. Object inheritance in such an environment becomes mostly complex.

In our system, **isa** objects having the same ancestor may be designed independently. From **segments**, we can thus design horizontal segments **seghori**, and unit segments **segunit** (Figure 9).

```
                Segment
                /    \
            Seghori    Segunit
            /
   Seghoricolored
          |
     Seghorired
```

Figure 9. Isa segment hierarchy.

Such specializations inherit all of the properties of the ancestor object. Improving their definitions is made by providing new properties and new constraints for the specialization sets. For instance, horizontal segments are defined by specializing **segment** into **seghori** and adding the constraint stating that the y coordinate of both the origin and the end of the segment must be equal. Similarly, coloured horizontal segments are defined by specializing **seghori** into **seghoricolored** and adding a new property "color".

The specialized object **inherits** all the ancestor's occurences that match the specialization constraints. Further, an occurence created down in the specialization hierarchy is propagated up to the root to be **acknowledged**.

These inheritance and acknowledgment mechanisms require powerful consistency controls and deductive facilities.

```
                          E1(C1,L1)
                          /    \
creation ⟶        E2(C2,L2)   E3(C3,L3)
of occurrence        /     \          \
              E4(C4,L4)   E5(C5,L5)   E6(C6,L6)
```

If E(C,L) stands for the definition of an object set E with the associated constraint set C and link set L. The object set E1 has two specialized sub-sets : E2 and E3. The set E2 has in turn two specialized subsets E4 and E5. Finally, E3 has a specialized subset E6.

Suppose we want to create an object occurrence in E2. It is first inserted in E2, then acknowledged in E1, and enventually inherited by the sets E3, E4, E5 and E6.

Insertion in E2 and acknowledgement in E1 follows from :

- the links L1 and L2 are used as **derivation rules**. Their evaluation yield the proper functional characteristics.
- the constraints C1 and C2 are used as **weak** consistency rules. They provide the consistency state of the new occurence within the sets E1 and E2. If it does not conform with C1, it is inconsistent in both E1 and E2. If it conforms with C1, it is acknowledged by E1. Its state in E2 is given by evaluation of C2.

Inheritance in E3, E4, E5 and E6 follows from the fact that :

- an object occurence is inherited by a specialization subset if it is consistent in the ancestor set, i.e acknowledged, and if it complies with the constraints and links of the specialized sub-set. Their constraints act therefore as **strong** consistency rules.

Suppose that the new occurence is consistent in both E1 and E2. The following figure describes the case where it does comply with the constraints in C4 but does not comply with the constraints in C3 and C5.

```
     E1      acknowledgment ⟶ consistent
    / \        ↑                /   \
  E2   E3    insertion ⟶  consist. inconsistent ←┐
 / \    \                  /   \      *          ·
E4 E5   E6          consist.  incons.  *          |
                       ↑        ↑        not tested |
            inheritance ┘        └ no inheritance ┘
```

5.3 Morphologic dynamicity

The morphologic dynamicity concerns the management of object **versions** and **representations**. During a particular design, the successive processes handle different representations, e.g functional, logic and electric for a VLSI circuit. Further, these representations may be implemented in different ways, e.g two half adders or multiple logic gates for a bit-slice adder.

We first detail the requirements of the design process in terms of object versions (section 5.3.1), of project management (5.3.2) and next detail the implementation of object representations (section 5.3.3).

5.3.1 Object versions

In a purely linear design process, the design of an object evolves from some representation to some more detailed ones. For a VLSI circuit, the successive representations are : functional, logic, electric, and layout representations.

functional ⟶ logic ⟶ electric ⟶ layout

The design process is however seldom linear. This follows from the facts that :

- each morphologic step yield new informations,
- each design decision may entail some information losses, due to the modifications issued by the users.

Each design decision may indeed provide several alternatives for the next morphologic level.

Since the design process is a trial and error paradigm, backtracking to previous object versions must be available. It follows that no previous object versions must be deleted or modified.

In our approach, an object **version** is the set of all informations relevant to an object on a particular design path in the three dimensional design space of Figure 8.

Each representation of an object is defined with specific tools corresponding to the particular morphologic level under consideration. For instance, VLSI circuits may be represented by functional languages at the functional level, by logic schemas at the logic level.

An object **version** will therefore be defined as the set of all the object **representations** on a particular design path (Figure 10).

Usually, several representations of an object at the same level are **equivalent**. However, representations at different levels are not always equivalent, because then can yield some losses of relevant information which, previously true for a particular level, does not hold for another. The transformation of a representation level into another is therefore only partially automatic. Specific programs must ensure that they are **compatible**.

```
                        PROJECT "Circuit"
                           /   |   \
Representations           /    |    \
    "Cell"      <-------x-----x-----x  rep_i
                         /     |     \
    representation_h x-------x-------x  rep_j
                    /         |       \
                 Version1  Version2 .. Versionk
```

Figure 10. Versions, representations and proj.

5.3.2 Projects

In our system, the object versions are modelled as ocurrences of a particular **project** object (Figure 10).

The representations of an object are characteristics of this project. Constraints between representations are modelled as constraints relating these characteristics. Functions referenced in these constraints are specific programs controlling the equivalence or compatibility between representations. Functional links between representations are implemented by derivation rules. Functions referenced in these links are implemented by specific programs that derive a particular object representation from other representations of the object.

Suppose that a VLSI circuit is modelled with four different representations, each of which being an occurence of a "Cell" object. Constraints and derivation of representations are modelled as constraints and links that call for specific programs. For instance, the "compfonclog" and "complogelec" programs control the compatibility between the different functional, logic and electric representations. The "passlogsym" program derives a symbolic representation of a circuit from its logic representation.

```
DEF_F compfonclog (compfonclog) ---> algorithm
| /* define function */
| FROM Cell x Cell INTO Bool
FDEF_F

DEF_F passlogsym (passlogsym_program)
| /* define function */
| FROM Cell INTO Cell
FDEF_F
```

```
DEF_E Circuit  PROJECT
| /* define project set for Cell */
| repf : Cell          ---> functional rep.
| repl : Cell               logic
| repe : Cell               electric
| reps : Cell               symbolic
| compfonclog(repf,repl) ---> compatibility
| complogelec(repl,repe)      between reps
| reps:= passlogsym(repl) ---> link for automat
FDEF_E                         symbolic rep.
```

Basically, the system provides for :

- clustering the different versions of the same object within the same set definition, e.g "Circuit" in the example above.
- tying together the multiple representations of an object by instantiation of the characteristics of the set,
- control of the consistency of the representations, i.e if an occurence violates the consistency between representations, it is appropriatly marked,
- derivation of a representation from others. It is specified as a rule that apply on the whole project.

Functionalities specific to the projects are :

- **version identification**, i.e each occurence has a unique name in the set it belongs to. This name is provided by the designer of the object. Versions are specified by a system identifier, which is read-only.
- **version management**, i.e a version is created, or data is added to a version, but it is never modified for itself.
- **version completion**, which concerns the evolution of object versions with respect to modifications and update propagation.

Designers are also allowed to **freeze** a particular version of an object. From a updatable version belonging to a specific working environment, the user can create a frozen version which has no more links with its creation environment. Freezing an object version thus implies breaking all functional and existential links that tie the version to other objects from which it derives.

5.3.3 Representations

Defining a version occurence in a particular project allows for tying and controlling the consistency of different representations of an object. Implemention of these representations is the subject of this section.

In the example above, every representation is an occurence of the "cell" set. Different implementations can be designed for a particular representation, each providing the same interface, but having their own structure or internal characteristics (nets and components arrangements). Other proposals define implementations as versions of the same object type, i.e the interface [3]. In contrast, other proposals allow different interfaces and structure for the same object [5].

In our system, representations are occurences of a **representation set**. It is partitionned into sub-sets that cluster the different implementations of a particular representation.

The designer defines for each representation the set of **common characteristics** and the set of characteristics **specific** to each particular object instance.

If the representations of a cell have all the same interface, the definitions of the representations are :

```
DEF_E Cell REPRESENTATION
| /* define representation set */
| /* common characteristics */
| type:      Typerep  ---> same morphologic level
| envelope: Polyline ---┐
| io-ports: LIST (Port)┘     same interface
| /* specific characteristics */
| .                   ---> no specific caracteristic
FDEF_E

DEF_E Cellcons ISA Cell
| /* common characteristics */
| .                   ---> no new caract. in common
| /* specific characteristics */
| comp: LIST (Component)---┐  implementations differ
| net : LIST (Net)         ┘  in structure
FDEF_E
```

When a "Cell" is designed, the representation set must be given. If it does not exist, the designer must instantiate its common characteristics. The implementation name and specific characteristics of the occurence must also be given.

Each representation set is a unique object. It is uniquely identified and is complete if all its characteristics are instantiated. If common characteristics are to be updated, a new object has to be created together with its representations.

If a representation set is used as a characteristic for an object, reference to its sole name means that the implementation is not chosen yet. Reference to its name together with the name of an occurence means that the implementation is that referenced by the occurence.

6. CONCLUSION

Taking into account the nature, consistency and dynamicity of CAD objects in database systems requires dramatic changes in their ability to model complex objects as well as to control their correctness with respect to high level specifications. This implies powerful mechanisms to derive information, propagate updates and handle incompleteness.

Implementing such features calls for improvements and enhancements to usual database functionalities, for instance through an integration with logic-based capabilities.

Several approaches are currently being tested for our system. A prototype that includes the functionalities described in this presentation has been implemented in Prolog on a VAX 11/780 running Unix 4.2 BSD. All the information is created, updated and stored as a set of Prolog clauses.

Another approach consists in integrating a relational database system with a Prolog inference engine [11, 13]. This approach is straightforward because our information structures are very similar to the relational data structures.

- A first integration off-loads to Prolog everything the relational system cannot handle, i.e lists and deductive information, as well as semantic information.
- A second approach consists in the enhancement of the relational database system with functionalities specific to CAD applications. The characteristics of both the database and Prolog system are taken for what they are. This implies the management of deductive informations and consistency controls by Prolog, and the management of the structured data by the database system.

Experiments are currently undertaken using a relational database system simulated in Prolog, a full relational database system called MICROBE that we have implemented, and an extended version implemented at CNET for VLSI circuit design [5].

Acknowledgements.
The authors are greatly indebted to MM. Jacques Lecourvoisier. Christian Jullien and Pascale Winninger from the CAD Research Dept at CNET for many expert comments concerning CAD/VLSI applications.

REFERENCES

[1] ADIBA M., NGUYEN G.T
Information processing for CAD/VLSI on a generalized data management system.
Proc. 10th International Conf. on Very Large Data Bases. Singapore. August 1984.

[2] ADIBA M., NGUYEN G.T
Knowledge engineering for CAD/VLSI on a generalized data management system.
Knowledge Engineering in Computer-Aided Design.
J.S Gero Ed. North-Holland Publ. Co. 1985.

[3] BATORY D.S , KIM W.
Modelling Concepts for VLSI CAD Objects.
ACM Transactions on Database Systems.
Vol 10, num 3. September 1985.

[4] BEYLS et al.
A Design Methodology based upon symbolic layout and integrated CAD tools.
Proc. 19th. Design Automation Conference. Las Vegas. June 1982.

[5] JULLIEN Ch., LEBLOND A.
A database interface for an integrated CAD system.
These Proceedings.

[6] KATZ R.H, LEHMAN T.J
Database support for versions and alternatives of large scale.
University of California Berkeley. Research report. 1983.

[7] KATZ R.H
Managing the chip design database.
University of Wisconsin-Madison.
Research Report 506. May 1983.

[8] LECOURVOISIER J.
CASSIOPEE: un systeme integre pour la CAO de VLSI.
Echo des Recherches. No. 118. November 1984.

[9] LORIE R., PLOUFFE W.
Complex objects and their use in design transaction.
Proc. ACM SIGMOD Conf. San Jose. May 1983.

[10] NGUYEN G.T , OLIVARES J.
Semantic data organization on a generalized data management system.
Proc. International Conf. on Foundations of Data Organization. Kyoto (Japan). May 1985.

[11] NGUYEN G.T
Semantic Data Engineering for Generalized Databases.
Proc. 2nd International Conf. on Data Engineering.
Los Angeles (California). February 1986.

[12] NGUYEN G.T
Object prototypes and database samples for expert database systems.
Proc. 1st International Conf. on Expert Database Systems.
Charleston (South Carolina). April 1986.

[13] RIEU D.
Model and functionalities of a CAD-oriented database system.
PhD Thesis. University of Grenoble. July 1985.