An Extensible Object-Oriented Approach to Databases for VLSI/CAD

Technical Report CRI-85-09
October 7, 1985

(DISC/85-3 - April 23, 1985)

Hamideh Afsarmanesh
Dennis McLeod

Department of Computer Science
University of Southern California

David Knapp
Alice Parker

Department of Electrical Engineering
University of Southern California

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0871

23 April 1985

This research was supported by the National Science Foundation, #ECS 8310774, and the Joint Services Electronics Program through the Air Force Office of Scientific Research under contract # F49620-81-C-0070.

Abstract

This paper describes an approach to the specification and modeling of information associated with the design and evolution of VLSI components. The approach is characterized by combined structural and behavioral descriptions of a component. Database modelling requirements specific to the VLSI design domain are considered and techniques to address them are described. An extensible object-oriented information management framework, the 3DIS (3 Dimensional Information Space), is presented. The framework has been adapted to capture the underlying semantics of the application environment by the addition of new abstraction primitives. An experimental prototype implementation of the database and its browsing-oriented interface is described.

Table of Contents

List of Figures

1. Introduction

The Very Large Scale Integrated circuit (VLSI) design environment is characterized by a large volume of data, with diverse modalities and complex data descriptions [Bushnell 83], [Davis 82], [McFarland 83], [Nestor 82], and [Knapp 85]. Both data and descriptions of data are dynamic, as is the underlying collection of design techniques and procedures. Design engineers, who are not usually database experts, nevertheless become the designers, manipulators, and evolvers of their databases. A final distinctive property of VLSI design environments is a requirement to model both the dynamic behavior of a circuit and its static structure.

While database modeling concepts and tools are useful for managing VLSI design environments, record-oriented database models are inadequate for structured and dynamic VLSI design data. Semantic database models offer concepts and techniques that are better suited to the capture of the underlying semantics of the VLSI domain.

In this paper, we characterize a class of digital VLSI design environments, describe a unified system for VLSI design, and present an object-oriented information framework appropriate to model these environments. The remainder of this section concerns digital VLSI design application domains and their specific database modeling requirements. Issues in conceptual data modeling for VLSI/CAD (Computer Aided Design) are presented in Section 2. Section 2 briefly describes an extensible object-oriented framework suitable for modeling VLSI design environments, the 3DIS (3 Dimensional Information Space), which has been adapted to capture some of the underlying semantics of circuit structure and behavior. In particular, the 3DIS model has been extended by the addition of new abstraction primitives to support recursive definition of environment entities and concepts. Section 3 describes the modeling of VLSI

circuits in the ADAM (Advanced Design AutoMation) system. An example 3DIS database for the ADAM VLSI design system is presented in Section 4 of this paper. Finally, Appendix I gives a more detailed view of the database.

1.1. The VLSI Circuit Design Domain

The VLSI circuit design process typically begins with a descriptive high-level specification of the design, consisting primarily of dataflow and timing graphs, which together describe the data-transformation and timing behavior of the desired hardware. Less detailed structural (i.e. schematic) and physical specifications are given, describing static properties of the target circuit. The descriptive graphs are hierarchical in that their components can be recursively decomposed into simpler components. For example, a dataflow node **multiply** can be decomposed into simpler **shift** and **add** constructs.

Several relationships might be specified among the components of these graphs; e.g. between specific time intervals and data operations. Also, various kinds of constraints can be attached to the graphs; for example, the duration of a time interval can be limited, a schematic wire can be specified to be a bidirectional bus connection, and the area of a physical bounding box can be limited. The descriptive graphical representations contain both numeric and symbolic attributes on their arcs and vertices.

Building a well-defined and complete high-level design specification is itself a nontrivial task. The descriptive specification of the design is usually large and complex. Many kinds of data are involved and it is in a large part recursively defined. Furthermore, the specification must be checked for completeness and consistency before the design process actually begins. In order to do so, VLSI design environments must support some 'checking routines', e.g. [Carter 79], [Parker 84a], [Pitchumani 84], that inspect the specification data without involving irrelevant details.

VLSI circuit design usually involves using a design library. This library contains components to be used in the construction of new components. It can also contain designs that are themselves under construction; these may be subparts of a larger design (e.g. the control unit for a CPU), or they may simply represent independent projects. Finding the appropriate library component to use in a given situation may be difficult [Leive 81]. For example, if an adder is desired, there might be several components named 'adder', a few named 'ALU', and a few 'complex standard' (i.e. microprocessors). In other situations, the behavior desired may not match the stated behavior of any component in the library without some transformation being applied. For example, A microprocessor is capable of implementing a floating-point multiplication under software control.

The output of the design system usually includes a set of graphs, relationships, and constraints similar to those of the descriptive specification, but with a much more detailed physical description.

1.2. ADAM: Advanced Design AutoMation

The ADAM system [Knapp 83a], [Granacki 85], [Park 85] is intended to provide a unified system for VLSI design, starting with a functional and timing specification and proceeding to circuit layout. The ADAM system describes VLSI circuits by means of four recursively defined and explicitly interrelated hierarchies. In ADAM, the representational formalisms of the input descriptive specification, the library components, and the output design are identical. This in turn facilitates the task of design verification and validation, e.g. testing the equivalence of specified and implemented dataflow graphs.

1.2.1. The ADAM VLSI Application Domain

ADAM supports several major circuit design activities. These activities comprise the main part of the process by which the primarily dataflow and timing descriptive specifications are mapped into the primarily physical output components [Parker 84b]. An appropriate information modeling environment for ADAM must support these tasks:

- **Algorithm Synthesis**: The dataflow graph is transformed and its operators decomposed if necessary in order to optimize speed, area, power, and other tradeoffs. For example, an operation may be specified as a loop traversed a fixed number of times; but speed requirements may lead to its being expanded into an in-line sequence in order to perform serial/parallel transformations [McFarland 81].

- **Partitioning**: Some part of the specification is partitioned so that the parts can be dealt with separately. For example, the dataflow graph might be partitioned into primarily control-oriented and data-oriented parts [Thomas 77]. Such operations may involve rebuilding the representation hierarchies.

- **Floor Planning**: Given partitions and constraints, high-level chip plans can be constructed that aid in the prediction and optimization of critical properties [Otten 82]. For example, a section of the physical chip die might be dedicated to data operations and another to control functions, before their detailed structure is known. This creates constraints on both the structure and low-level physical layout that must be monitored and flagged if they are violated. Such constraints are not quite the same as semantic integrity constraints; they may be estimates, guidance, or absolute constraints, they can be modified and retracted, and they are generated during design time.

- **Data Path and Control Synthesis**: The major data paths are allocated hardware resources and the order of operations is fixed [Granacki 82], [Hafer 83], [Hitchcock 83]. Controllers are specified and synthesized [Evangelisti 79], [Leiserson 83], [Nagle 82], and [Park 85]. Interconnect between data operators and between data operators and control hardware is synthesized. Microprograms are constructed. In each of these steps, information about the reusability

of operators, their control signals, and their costs in speed and area are crucial. For example, a floating-point multiplier may have already been included but a fixed-point multiplier is also needed. Can the floating-point multiplier be used directly, or must it be modified? Is it available during the interval when the fixed-point operation is needed? Are the data buses suitably connected or would new ones have to be added? These questions can be difficult to answer if the floating-point multiplier is a library component for which little is known about internal operations, if it has been built up from smaller components, which might also shared by other operations, or if it has been optimized for floating-point arithmetic only.

- **Built-In Test Synthesis**: Hardware is added to make the end product testable [Breuer 85]. This is increasingly important as the ratio of pin count to chip area decreases and chips become less controllable and observable. To make it possible to test circuits, information must be attached to components and designed circuits describing how they can be tested and with what time and space penalties Some combinations of circuit types and test methodologies, support testing without extra hardware; others require expensive test control and generation hardware, while still others require replication of units for error detection and correction.

- **Module Selection**: Design library elements are brought in to the physical and schematic subspaces to implement operators, memories, and random logic [Leive 81]. As the library elements become more complex the search for suitable modules becomes more difficult and the cost/speed/power tradeoffs become more difficult to keep track of.

- **Placement and Routing**: Modules are allocated physical positions on the layout, and interconnect wires are mapped out [Soukup 81]. Physical positions of features may not conflict, and design rules must be observed. The volume of information at this step is enormous.

- **Validation and Verification**: At any step of the design process, performance and function may be validated using an appropriate simulator or formal verification tool [Carter 79], [Parker 84a], and [Pitchumani 82]. The design, or parts of the design, are shown to

be correct with respect to some part of the specification. For example, the physical layout may be processed in order to extract an equivalent circuit, which is compared to the desired circuit in order to prove logical equivalence.

- **Backtracking**: Part of the design may be 'undone' when it is discovered not to be adequate or correct. This creates a host of consistency and concurrency problems. For example, a component used in a variety of places may not be suitable for one application. A new version can be constructed, but the new and old versions must coexist because of the other uses of the component [McLeod 83].

1.3. Information Management Requirements of VLSI Design Environments

This section describes the fundamental characteristics of digital VLSI design environments. The major goal here is to set forth the distinctive information management requirements of VLSI/CAD design applications in terms of the general characteristics of the VLSI design process:

- The design data is introduced and accessed by many kinds of users including analog simulators, dataflow optimizers, human users, etc. Thus various representations of design data must coexist.

- The number of application packages is large. A central representation can be used to minimize difficulties with incompatible data formats.

- The design data is of large volume, and of various modalities and complexities, e.g. graphical, symbolic, numerical, textual and formatted data.

- Structural information (e.g. data-description, data-interrelation, and data-classification) is complex and of large quantity. Structures must support programs, documents, messages, constraints, graphs, etc.

- The structure of the stored information changes over the lifetime of the database. Thus the structural framework must support dynamic use.

- The end-users, design engineers and CAD application programmers, are familiar with their application environment, but are not likely to have expertise in databases or programming. Yet the same end-users often become the ultimate designers, manipulators, maintainers, and evolvers of the database.

Since the structural information is large, complex, and dynamic, it must be as conveniently accessible to end-users as the database information contents.

2. Conceptual Data Modeling for VLSI/CAD

Historically, the reported work in the VLSI database design literature describes management of design information as collections of raw data in files. Automated filing systems have no structures to model the semantics of design environments. Interpretation of the stored design data is completely hidden in the application programs and the users' minds. Data manipulation is performed only by application programs, while the data itself does not express any of the application semantics. These database systems are costly to maintain and evolve as the design information is modified in the course of the database life cycle.

2.1. Record-Oriented Database Models

Record-oriented database models are usually exemplified by the hierarchical, network, and relational data models. Capabilities of these database models in supporting VLSI/CAD design environments have been studied both in the literature and in practice. Modeling constructs of these database models are greatly influenced and limited by the record structure of physical database organizations [Kent 79], [Hammer 81]. For instance, record structures are not suitable to model loosely-formatted information such as documents, graphs, messages, or programs, that arise in many design application environments.

Hierarchical database models represent data as records that are organized as nodes of tree-like hierarchies. Network database models also use record structures to represent data, but they organize records in a network, using only one-to-many relationships. The structures defined on the modeling constructs of the network and hierarchical database models are mainly implementation dependent and may not capture the semantics of the application.

Relational data models have a well-defined set of structuring primitives based on the mathematical notion of a relation, where a relation is a set of n-tuples. Tuples are used to represent many-to-many relationships among data. Although relational data models perform much better in modeling design environments than hierarchical and network models, they still lack the semantic expressiveness required to model complex and evolving design data [Kent 79].

Furthermore, these data models provide limited guidance to users. Design, maintenance, and use of record-oriented databases require high level database expertise. These models are not suitable for non-database-expert VLSI designers who intend to build, use, and maintain their own databases. Examples of VLSI/CAD design databases that have used record-oriented database models include [Wong 79], [Eastman 80], and [Stonebraker 82].

2.2. Semantic Database Models

Recently, the suitability of semantic database models as tools to help in the construction and use of design databases has begun to be examined [Katz 82], [McLeod 83], [Batory 84], and [Dittrich 85]. Semantic database models have succeeded in expanding data modeling beyond the capabilities of record-oriented data models. They provide a rich set of semantically expressive constructs and mechanisms that reflect more of the meaning of both data and its logical structure, and make it easier for users to understand and use

databases [Afsarmanesh 84]. The high level semantic structure of the design data is modeled by several fundamental abstraction primitives that organize and interrelate the modeling constructs of databases. Specific modeling concepts are introduced in some semantic database models to capture the database dynamics.

2.2.1. Object-Oriented Database Models

Some semantic database models are object-oriented in the sense that the modeling constructs and the construct manipulators of these models are defined as objects. Objects correspond to the concepts, entities, and activities of application environments 'naturally'. A more recent feature in object-oriented database models, incorporated in the 3DIS data model, is the uniform view and treatment of the structural and non-structural (data) database contents that simplify database manipulation and modification tasks.

2.3. A Brief Summary of the 3DIS Data Model

The 3 Dimensional Information Space (3DIS) [Afsarmanesh 85a] is a simple but extensible information management framework. This data model is mainly intended for applications that have dynamic and complex structures, and whose designers, manipulators, and evolvers are non-database-experts. As a step towards addressing the modeling needs of such application environments, the 3DIS unifies the view and treatment of all kinds of information including the description and classification of data (meta-data).

3DIS databases are collections of interrelated objects, where an object represents any identifiable piece of information, of arbitrary kind and level of abstraction. For example, a component **H42padder**, an attribute **Designer-Names**, a string of characters **Low-Order-Bit**, a structural component (meta-data) **OEM-Component**, and a procedure

Insert-an-OEM-Component are all modeled uniformly as objects in a homogeneous framework. Therefore, what distinguishes different kinds of objects is not how they are modeled, but rather the set of structural and non-structural (data) relationships defined on them.

Each object has a globally unique *object-id* that is an identifier either defined by users, or generated by the system. An object can also have several user-specified surrogate *object-names* which also uniquely identify it. Objects may be referred to via their unique identifier-names, object-names, or via their relationships with other objects.

The 3DIS model supports *atomic, composite,* and *type objects*:

- **Atomic** objects serve as the *symbolic identifiers* for atomic constants in databases. These objects carry their own information content in their object-ids. Therefore the object-ids of atomic objects are to be specified by users. Atomic objects cannot be decomposed into other objects. The contents of atomic objects are uninterpreted by 3DIS databases, in the sense that they are either displayable or executable without any further interpretation of their information content. Strings of characters, numbers, Booleans, text, audio, and video objects, as well as behavioral (procedural) objects, are examples of atomic objects. Text objects represent long character strings, while audio and video objects represent digitized images and voices. Behavioral objects represent the routines that embody database activities, modeling an object that is executable. Behavioral objects accomplish modeling of data definition, manipulation, and retrieval primitives, e.g. **Insert-an-OEM-Component.**

- **Composite** objects describe (non-atomic) entities and concepts of application environments. The information content of these objects can be interpreted meaningfully by the 3DIS database through decomposition into further objects. Mapping objects are a kind of composite objects. Each may be decomposed into a domain type object, a range type object, an inverse mapping object, and the minimum and maximum number of the values it may return. Mappings represent both the descriptive characteristics of an object,

e.g. **Model-Name**, and the associations defined among objects, e.g. **Has-Model-Constituents**. Mappings model both single and multi-valued relationships. An example composite object is a node **FA1.Df**, where **FA1.Df** in Figure 4-3 is the *symbolic name* (colloquially, logical reference name) of an object. These objects are not displayable, except in terms of their relationships with atomic objects. For example, FA1.Df's **Intended-Function is Low-Order-Bit**, FA1.Df's **Structural-Dimension is 1**, etc. If a composite object is related to certain other composite objects then it may be displayed recursively in terms of the atomic objects related to those composite objects.

- **Type** objects contain the descriptive and classification information of a database. Every type object is a structural specification of a group of atomic or composite objects. It denotes a collection of database objects, called its *members*, together with the shared common information about these members. A type object can be a *subtype* of another type object (*supertype*). Subtypes inherit all of their supertypes' properties. A type object can be the subtype of more than one type object. Therefore, the subtype/supertype relationships among type objects can be represented by a directed acyclic graph (DAG). Examples of type objects are **In-House-Component** and **Dataflow-Model**.

Basic relationships among objects are defined through the three fundamental abstraction mechanisms of *classification*, *aggregation*, and *generalization*:

- **Classification** represents member/type relationships by relating an atomic/non-atomic object, e.g. **H42padder**, to its generic type object(s), e.g. **In-House-Component**.

- **Aggregation** represents member-mapping/type relationships by relating a type object, e.g. **In-House-Component**, to the mappings that define its members, e.g. **Designer-Names, Realization-Bindings**, etc.

- **Generalization** represents subtype/supertype relationships by relating a type object, e.g. **In-House-Component**, to a more general type object, e.g. **Component**.

The 3DIS model has also been extended to accommodate other kinds of abstractions that are useful in VLSI design applications. For example, abstraction primitives to support the definition of recursively defined entities and concepts such as sets, lists, and binary trees are included in the model. In particular, for the ADAM design database, the 3DIS supports the recursive definition of VLSI cells, as will be described in the next section.

An integral part of the 3DIS model is its simple and multi-purpose geometric representation. This geometric framework graphically organizes both structural and non-structural database information in a 3-D representation space. It reflects a mathematically founded definition for 3DIS modeling constructs in terms of the geometric components that represent them. The three axes in the space represent the domain (**D**), the mapping (**M**), and the range (**R**) axes. Relationships among objects are modeled by (domain-object, mapping-object, range-object) triples that represent specific points in the geometric space.

Figure 2-1 illustrates a perspective view of the geometric representation of the 3DIS database example of Section 4. In this figure, **FA-1** and **FA-2** are members of the type object **Single-Node**, while they are also the **Model-Constituents** of **H42padder-Dataflow**. Figure 2-2 illustrates the right view of the simplified geometric representation for the **H42padder-Dataflow**. The figures have been simplified to represent only a part of the information in the database.

Several geometric components such as points, lines, and planes encapsulate database information at several levels of abstraction, which has an advantage of enhancing the readability of the display. For instance, in Figure 2-1, the vertical line emanating from the object **H42padder-Dataflow**

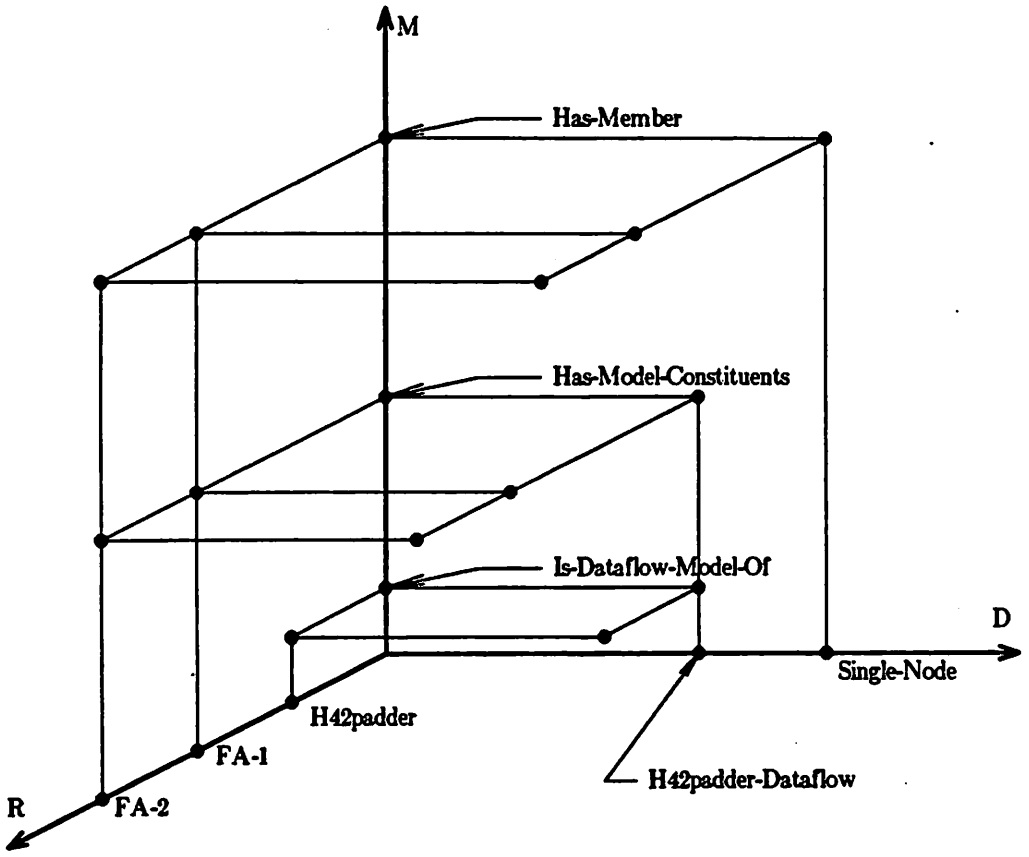

Figure 2-1: Perspective view of a part of information in a 3DIS database

corresponds to all mappings defined on that object. Similarly, an orthogonal plane passing through the same object, which is also represented as the right view in Figure 2-2, contains the information about all objects directly related to **H42padder-Dataflow**. The variety and level of information encapsulation supported by the geometric representation is a unique feature of the 3DIS data model.

The geometric representation also provides an appropriate foundation for information browsing and serves as an environment for a simple graphics-based user interface. Database browsing is supported by a 'display window' to the geometric framework, through which users may focus on and investigate an object or an 'information neighborhood' of their interest. A further description of the 3DIS user interface is given in [Afsarmanesh 85b].

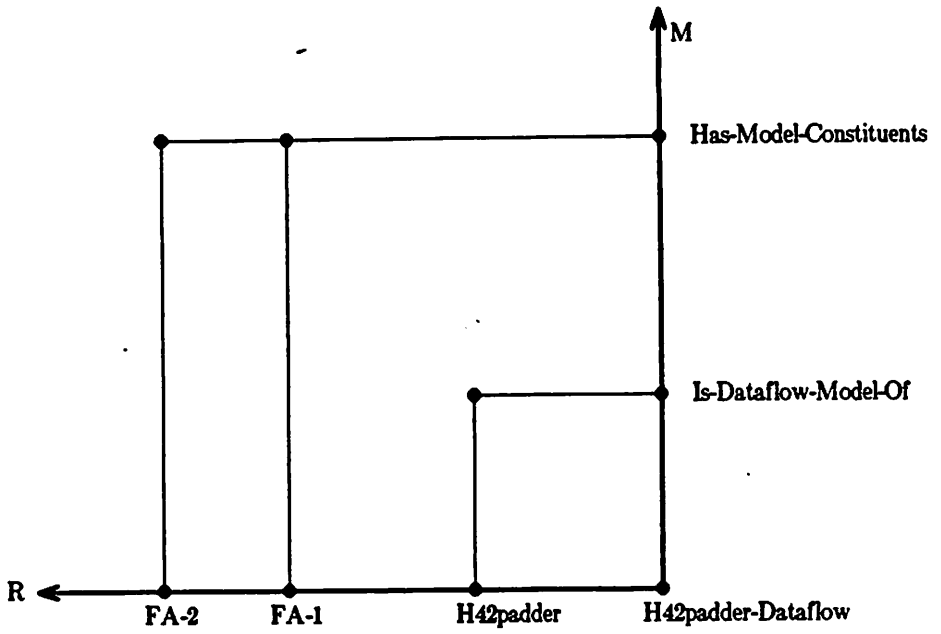

Figure 2-2: Right view of H42padder-Dataflow

3. A Conceptual Schema for Modeling Digital Circuits

The digital circuit design process can be regarded as a search of a multidimensional 'design space' [Director 81] for a particular solution that meets constraints on functionality, timing, power, cost, and so on. The entire design space can be broken down into subspaces which are near-orthogonal in the sense that decisions taken in one subspace affect decisions taken in another subspace weakly across some region of interest. For example, a single functional specification can be mapped into several different implementations with varying speeds, power dissipations, and costs.

The conceptual model described below is based on four subspaces chosen for their near-orthogonality. These subspaces are each hierarchies in their own right. For example, one of the subspaces is used to represent schematic (structural) information; this subspace is a hierarchy with block diagrams at the

top, registers and ALUs at the middle, gates a little lower, and transistors at the bottom. Design entities are described in terms of these subspaces and a set of relations across them.

3.1. The Component

The fundamental structure of the ADAM conceptual schema is the **component**. A component can represent either a specification, a design in progress, or a member of the design library.

A specification is represented as an incomplete component; that is, it contains information about the target design represented in the same way as the completed design. The information that is present in the specification usually represents the operations the target must perform and the constraints it must meet.

A design in progress is an incomplete component. As the design process progresses, it should gradually become more and more complete, until finally it can be manufactured. In the initial stages of design, the target component contains primarily dataflow and timing information; in the later stages it will contain more schematic information, and finally most of the information will be physical layout. The original dataflow, timing, and schematic information are preserved for documentation and verification/validation purposes.

The design library is used to store both procured components (OEM components) and 'In-house' components. An in-house component may be either complete or incomplete, while incomplete OEM-components are not allowed.

3.2. Models of a Component

A component is described in terms of four **models** and a set of relationships (**bindings**) across the constituents of the models. The models correspond to the four subspaces of the design space. Each model represents the component in a different way. For a given design task, any or all of the models may have to be referenced. The models are:

1. The dataflow model. This model describes the data transformation operations performed by the component. Its primitives are **nodes** and **values**. Nodes represent data transformations; values represent data passed between nodes.

2. The timing and sequencing model. This model describes time-domain and branching behavior of the component. Its primitives are **ranges** and **points**. A range represents a time interval during which an operation can take place; points represent infinitesimal 'events', which are partially ordered because the ranges have signs as well as durations.

3. The structural model. This describes the schematic diagram of the component. Its primitives are **modules** and **carriers**. A module represents a schematic block, gate, transistor etc.; a carrier represents a schematic wire.

4. The physical model. This describes the layout, position, size, packaging and power dissipation of the component. The primitive elements are **blocks** and **nets**, which represent layout cells and interconnect respectively.

For example, the OEM-component '74181', which is a 4-bit TTL ALU slice, has a dataflow model with add, subtract, AND, and OR nodes, which represent its data transformations; it has a timing-and-sequencing model that describes its propagation delays for various combinations of inputs; it has a schematic diagram that either consists of a box with connection points or a gate diagram; and it has a physical description that signifies its being packaged in a 14-pin DIP.

3.2.1. Hierarchy within the Subspaces

The four models are each hierarchically structured. For example, a dataflow node is either primitive or it is defined recursively in terms of other nodes and values. Similarly, a value is either primitive or defined recursively in terms of other values. Similar recursive definitions are used in all four hierarchies.

3.2.2. Models and Links

The generic name **Model** is used for nodes, ranges, modules, and blocks. The dataflow model of a component is therefore a **Node**, which can be recursively decomposed as described above. The generic name **Link** is used for values, points, carriers, and nets. These too can be decomposed, with the exception of points, which represent atomic events of infinitesimal duration and enumerated kind.

3.2.3. Relationships across Subspaces

All relationships between models and links of different subspaces are explicitly represented by means of **bindings**. These bindings express the interrelationships between the constituents of the four subspaces. There are two basic types of bindings:

1. **Operation** bindings, which relate dataflow elements to structural elements and time ranges.

2. **Realization** bindings, which relate structural elements to physical elements.

For example, an operation binding is used to express the relationship between an operation (dataflow), the ALU in which it is performed (structure), and the time interval during which it happens, while a realization binding is used to represent the correspondence between a particular layout region and the ALU.

3.3. The Target, the Specification, and the Library

The design being constructed is called the *target*. The target should be functionally equivalent to the *specification*; it is composed of primitive elements and members of the *design library*. Near the top of the hierarchies, the dataflow of the target might be syntactically identical to the dataflow of the specification, but at the low levels this is unlikely. For example, suppose the specification contains a multiplication node. The definition of multiplication can be regarded as a series of doublings and conditional additions. But under a given set of timing, power, and area constraints, the dataflow actually implemented might be radically different. For reasons like this, the specification and the target are considered to be two completely different components. The relationships between constituents of the target and the specification can be very complex, and their exact description is still under investigation.

Furthermore, there may be substantial differences between the way in which a library component is used and its actual capabilities. In the target, an addition might be implemented in a general-purpose ALU. If no other operations are performed in that ALU, then the only node bound to it in the context of the target is the addition. But the physical ALU that is present in the design is capable of far more than that: it can also subtract and perform logical operations. This 'unused behavior' can be found by examining the dataflow and timing models of the library component that represents the generic ALU.

Various kinds of verification and validation methods can be implemented using the three different representations. For example, the method of inductive assertion can be used to verify that a multiplication in the specification is really decomposed properly in the target. Moreover, the unused behavior can be useful both in determining proper control signals and in verifying that the proper control signals are used.

4. An Example

A small example, illustrative of the digital circuit design conceptual schema, will now be discussed. The example is that of a particular component, a two-bit binary adder, which can be represented as in Figure 4-1. First the schema of the component will be discussed; then the dataflow model of the component will be examined in detail. The timing, structural, and physical models of the component will not be examined in this section. The full conceptual schema is described in Appendix I. Finally, the way in which bindings are used to unify the four subspaces will be discussed.

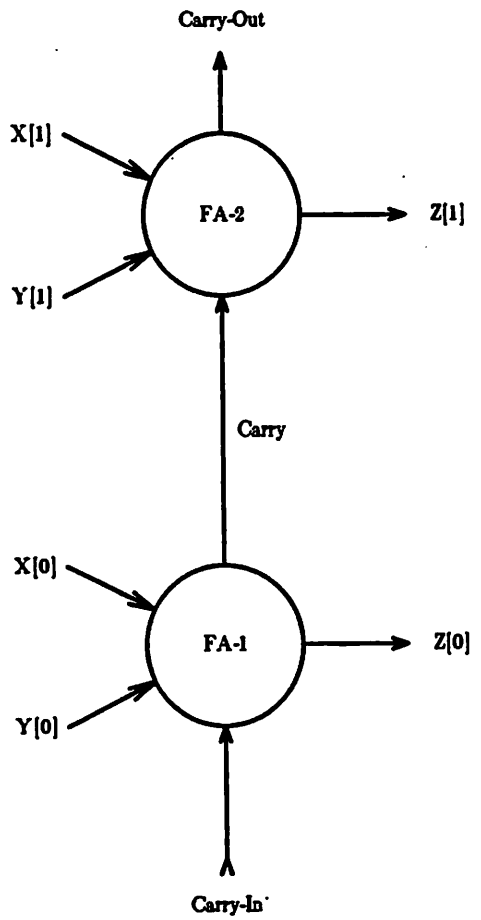

Figure 4-1: Two-bit adder example

4.1. The Component Schema

The subtype/supertype (generalization) hierarchy of component definitions is shown in Figure 4-2, where boxes represent type objects, the arrows represent subtype/supertype relationships, and the undirected lines that come out of the boxes lead to mappings (properties) that describe members of the types. The type **Component** has properties that denote its name, four **Models** of the component, and two sets of **Bindings**.

There are two subtypes of **Component**. The **OEM-Component** represents a component supplied by an OEM (Outside Equipment Manufacturer). As such it is characterized by the name of the manufacturer, the manufacturer's designation (**Kind**), and a list of **Suppliers**. Other properties, such as **Price**, have been omitted in the interest of simplicity.

The other component subtype, **In-House-Component**, represents a component that is manufactured in-house. It may not even be a complete design; that information is captured by the **Complete-Bit**[1]. Target designs and specifications are both examples of incomplete components. A design library component, on the other hand, could be complete or incomplete as design policy dictated. By way of contrast, no OEM component is allowed to be incomplete. The in-house component also has a set of **Designer-Names**, denoting the people responsible for its construction, a **Process**, which identifies a particular fabrication process, and a **Guru**, i.e. someone who knows how to use the component and is willing to answer questions.

The member of the **Component** type used for this example is shown in

---

[1]Presumably more complex historical information could be attached, e.g. a **Verification-History**. Such properties have not been considered in this context.

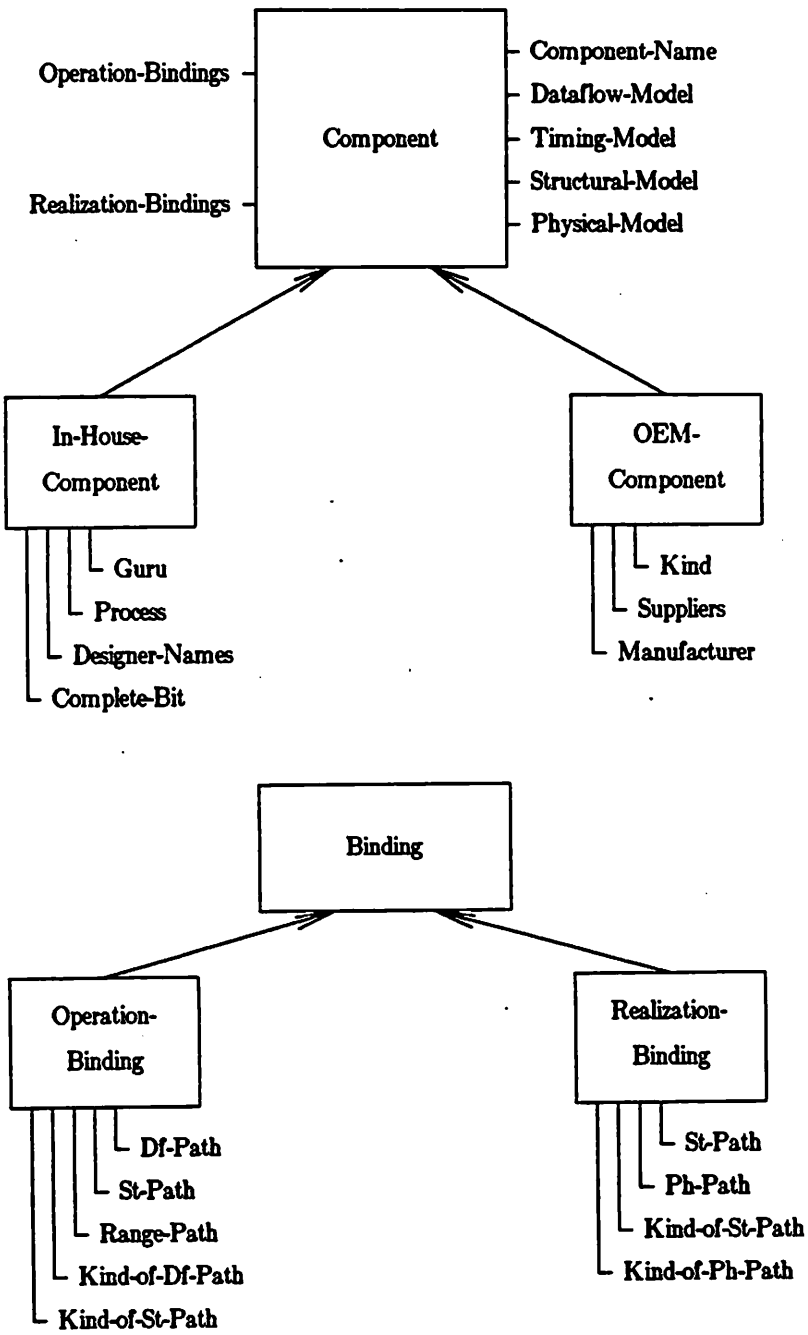

Figure 4-2: The generalization hierarchy of Components and Bindings

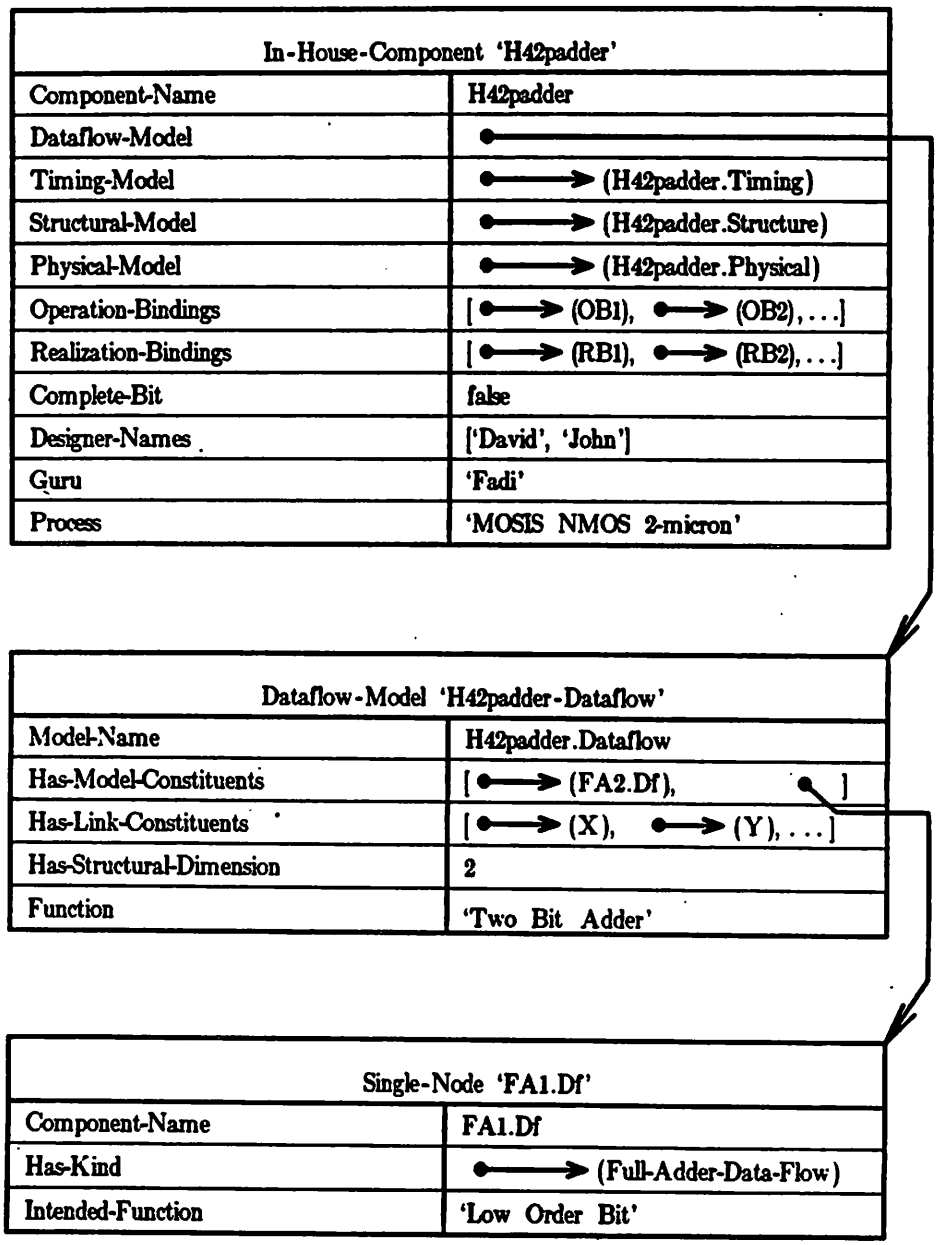

Figure 4-3: A Component member and its partial dataflow model

Figure 4-3. This **Component** is an **In-House-Component**. Its name, a property inherited from the supertype, is 'H42padder'. The four **Models** are similarly named; Figure 4-3 shows only the 'H42padder.Dataflow' **Model** in detail, where again some mappings such as **Complete-Bit** and **Designer** have been omitted in the interest of simplicity[2]. **Operation-Bindings** and **Realization-Bindings** are also shown schematically as lists of logical references to the actual binding objects, which will be discussed in Section 4.3. The other properties of 'H42padder' are self-explanatory. The dataflow graph of 'H42padder' is given in Figure 4-1.

4.1.1. Models and Subspaces

Each of the four models of the component represents a different aspect of the component. The models can be thought of as projections of the component onto four design subspaces; hence the **Dataflow-Model** of a component is its projection onto the dataflow subspace. In this example, we will examine only the dataflow subspace.

The reason that only the dataflow model need be considered in detail is that the other models are syntactically much the same as the dataflow model. The differences mostly consist of the addition and deletion of minor attributes as the underlying phenomena being modeled dictate. For example, the structural counterpart of a dataflow **value** is the **carrier**. Naturally the carrier attribute **driver**, which describes hardware implementation attributes like *tri-state*, *open-drain* and so on, has no counterpart in the dataflow model, which is used to represent abstract functions. Many such slight differences exist but are not discussed in the body of the paper. The interested reader is referred to the Appendix and to [Knapp 83b] and [Knapp 85].

---

[2] In all of the following figures, the use of parentheses ( ) denote objects whose details have been omitted in the interest of simplicity. Square brackets [ ] represent list delimiters.

4.2. The Dataflow Subspace and Dataflow Models

The schema for **Dataflow-Models** is shown in Figure 4-4. Objects of type **Model** each have a name, a **Complete-Bit** similar to that of **Components**, and a **Designer**. There are four subtypes of **Model**, one for each subspace. Shown in Figure 4-4 is only one, **Dataflow-Model**, also called **Node** for short. The other three subtypes of **Model** are **Structural-Model**, **Timing-Model**, and **Physical-Model**. Their generalization hierarchies are shown in Figures I-1 and I-2 of the Appendix.

Dataflow-Model has the following properties:

- **Function**: this property indicates the overall function performed by the Node. For example, in Figure 4-3 the function of 'H42padder-Dataflow' is that of 'Two Bit Adder'.
- **Dimension**: this property indicates the bit width of the Node.
- **Has-Link-Constituents**: this property tells what links (in this case, i.e. for dataflow models, links are **Values**) are contained within the model.
- **Has-Model-Constituents**: this property tells what models (**Nodes**) are contained within the model.

The model and link constituents of a model together express the application domain semantics of that model, thereby supporting its recursive definition. In the example of Figure 4-3, which corresponds to the two-bit adder dataflow graph of Figure 4-1, the link-constituents are the input, output and carry **Values**; and the model-constituents are the **Nodes** 'FA-1' and 'FA-2'. The constituents of a model are represented as lists of logical references.

The objects that are logically referred to in **Has-Model-Constituents** are of type **Node-Component**, which also designates that they are either of

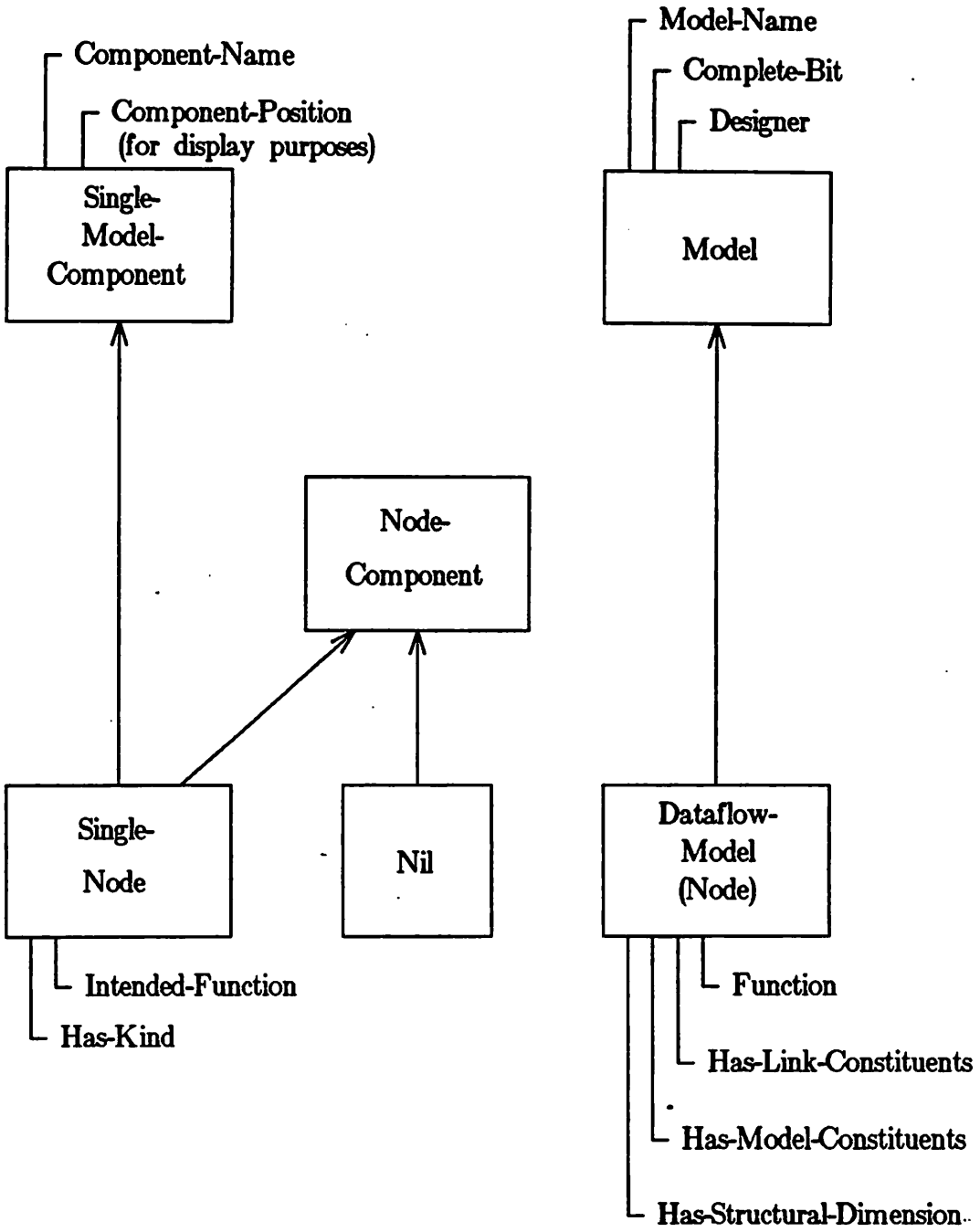

Figure 4-4: The generalization hierarchy of Dataflow Models

type **Single-Node** or **Nil**. If the reference is to **Nil**, then the constituent is not further defined, i.e. the **Node** is either a primitive or its definition does not exist at present. In either case the recursive definition of the model ends at this point. If the reference is to a **Single-Node**, as is the case in the example, the recursive definition of the model continues through it. In the example, the **Single-Nodes** are called 'FA-1' and 'FA-2'. 'FA-1' has the **Intended-Function** 'Low Order Bit'; presumably 'FA-2' is the high order bit of the adder. Both 'FA-1' and 'FA-2' could have the value 'Full-Adder-Data-Flow' in their **Has-Kind** properties; that means they are both one-bit full adder nodes.

But the **Has-Kind** property means more than that. 'Full-Adder-Data-Flow' is itself a **Node**, and is represented by a **Dataflow-Model**; hence it is further defined in terms of its model and link constituents. This is the recursion abstraction at work: **Models** are defined in terms of other **Models**.

4.2.1. Dataflow Links

Figure 4-5 shows the type-subtype hierarchy of **Links** for the dataflow subspace. **Links** are a little more complicated than **Models**, because they bear the burden of representing connections between **Models**. The type-subtype hierarchies of **Links** in all four subspaces are shown in Figures I-3 and I-4 of Appendix

A Dataflow **Link** is called a **Value**. A **Value** has a **Name**, such as 'Carry', which is inherited from the supertype **Link**. It also inherits a **Complete-Bit** and a **Designer**, with meanings similar to those of the **Component**'s corresponding properties.

The reason a **Value** should have an explicitly mentioned **Designer** is that a **Value** is potentially a structured entity, for example a complex floating-point

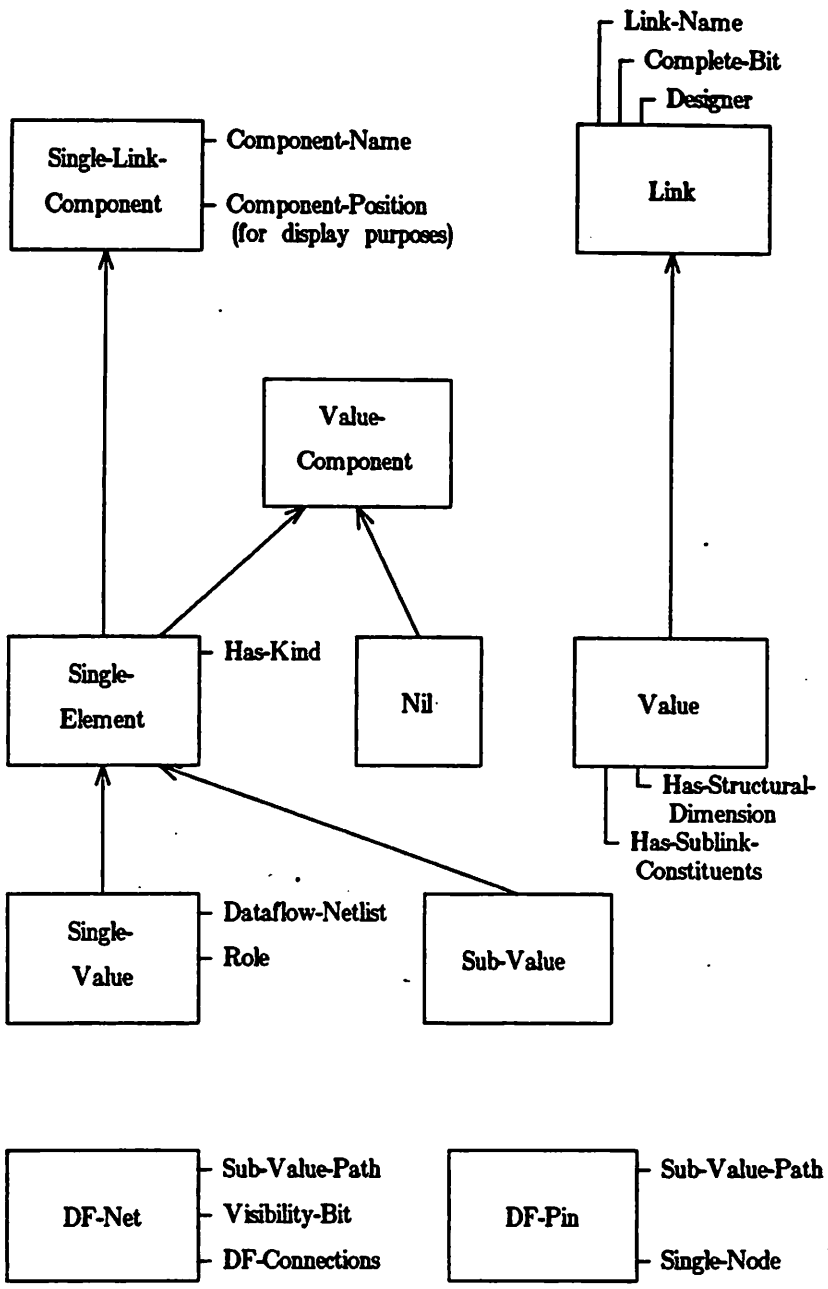

Figure 4-5: The generalization hierarchy of Dataflow Links

number. If the **Value** is a simple array, then the **Has-Structural-Dimension** property tells the dimension of the array. If the **Value** is structured, then its **Has-Sublink-Constituents** property defines the structure. **Sublink-Constituents** are of type **Value-Component**, which also indicates that they are either of type **Nil**, or if they are of type **Single-Element** it signifies that they are again either of type **Single-Value** or **Sub-Value** (Figure 4-5).

For example, a floating-point number 'Flonum' is a structured value consisting of two fields 'Mantissa' and 'Exponent'. These are **Sub-Values**, which have **Has-Kind** properties of their own. The **Has-Kind** property of 'Mantissa' might refer to a **Value** named 'Long-Signed-Integer'. On the other hand, the **Has-Kind** property of 'Exponent' might refer to 'Excess-64-Integer'.

The input 'X' of Figure 4-1 is a **Single-Value**. Figure 4-6 shows 'X' in more detail. The **Has-Kind** property of 'X' points to the **Value** 'Two-Bit-Integer'. The **Value** 'Two-Bit-Integer' has the **Structural-Dimension** 2. 'Two-Bit-Integer' also has **Sublink-Constituents** consisting of two **Sub-Values**, named 'High-Order-Bit' and 'Low-Order-Bit' respectively. The **Has-Kind** properties of these bits have logical references to the primitive **Value** 'Bit'. The **Has-Sublink-Constituents** of the **Value** 'Bit' is **nil**, so the recursive definition of 'Two-Bit-Integer' ends at this point.

But 'X' represents something more than its constituents. It has a **Role** which is 'second vector input'. Furthermore, it has connections, represented by a **Dataflow-Netlist**. The **Dataflow-Netlist** is a list of logical references to **DF-Nets**. In Figure 4-6, the two bits of the 'Two-Bit-Integer' 'X' are connected separately, only the connections of the 'Low-Order-Bit' being shown.

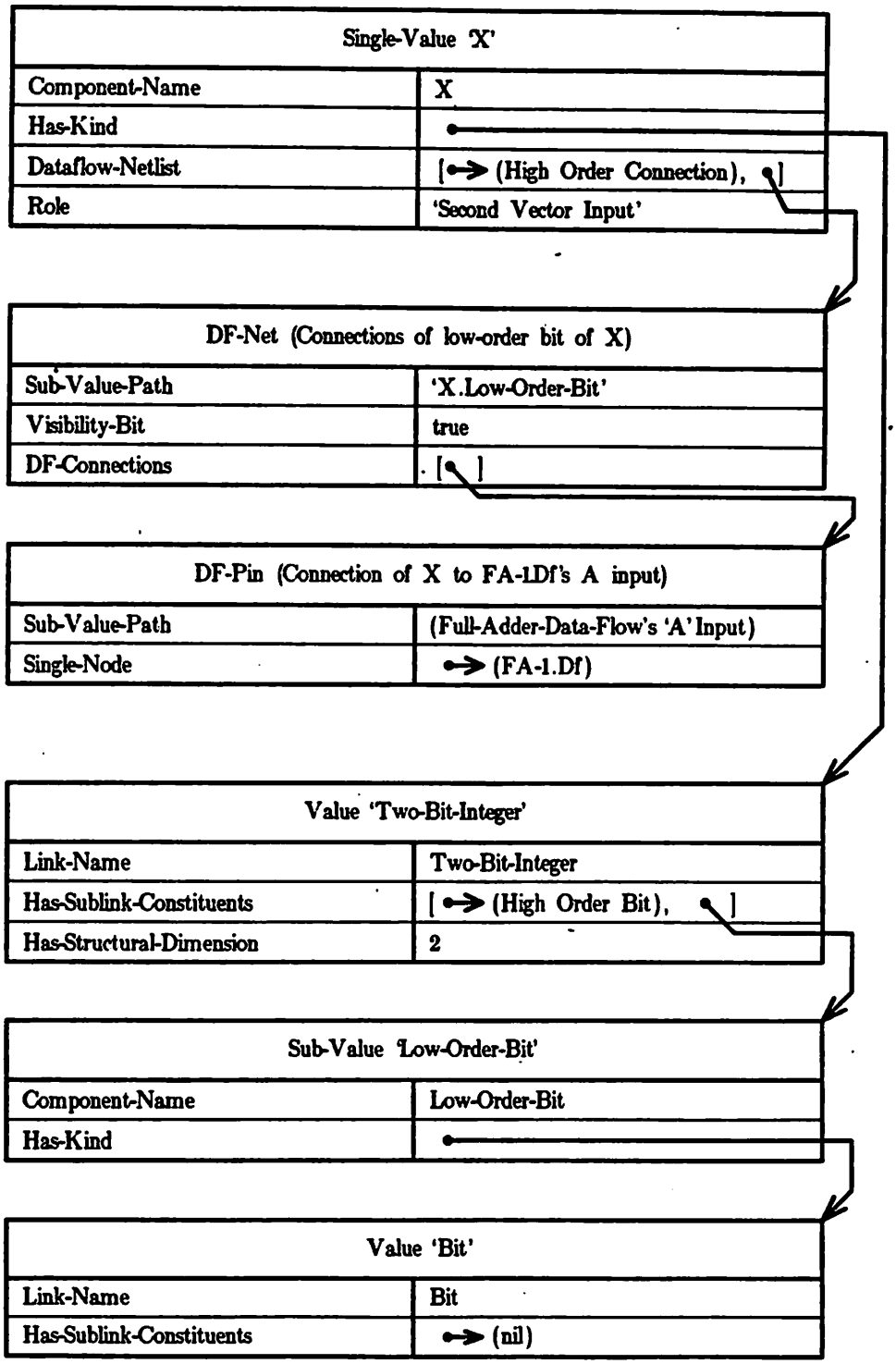

Figure 4-6: The definition and connections of the Value 'X'.

The **DF-Net** has a **Sub-Value-Path**. This is a path into the structure of the value being connected. For example, if the high-order bit of the mantissa of a complex floating-point number 'A' was being connected as an individual, the path would look like 'A.Real.Mantissa.Bit63'. In Figure 4-6, the path simply points to the low-order bit of 'X'.

The **DF-Net** also has a **Visibility-Bit**; this determines whether the bit can be 'seen' from outside 'H42padder-Dataflow'. Since 'X' is an input, this bit is **true** for all its **DF-Nets**. Other structured **Links** may have their visibilities determined on a field-by-field basis, which is why the visibility information is attached to the individual connections rather than to the **Single-Link** itself.

DF-Connections are used to describe connections in the dataflow subspace. The **DF-Connections** of a **DF-Net** are references to **DF-Pins**. A **DF-Pin** refers to a **Single-Node**, e.g. 'FA-1.Df', and a **Sub-Value-Path**, which represents a connection point on that **Single-Node**. In the example of Figure 4-6, the **Sub-Value-Path** of the 'X' connection point is a path to the 'A' input bit of the 'Full-Adder-Data-Flow' model, which is given in parentheses in Figure 4-6 (recall that 'Full-Adder-Data-Flow' is the **Has-Kind** property of 'FA-1.Df').

Using both the **Sub-Value-Path** of a link, as expressed in the **DF-Net**, and the **Sub-Value-Path** of a Single-Node connection point, as expressed in the **DF-Pin**, very general kinds of connections can be constructed.

For example, using both paths in their full generality would allow us to make arbitrary permutations of structured array values at connection points. If a two-bit **Value** 'P' was to be connected to the 'X' input of 'H42padder.Dataflow', it would be possible to connect 'P[1]' to 'X[0]' and 'P[0]' to 'X[1]', thus achieving a bitwise reversal at the point of connection.

4.3. Bindings

The two binding sets represent the interrelationships between the elements of the models. **Operation-Bindings** show the relationship between an operation (or value), a structure, and a time interval; (for example, an addition, an adder, and a microcycle). Similarly, a different Operation-Binding might represent the relationship between a value, a bus or register, and a microcycle.

Realization-Bindings are used to represent the relationships between structural elements and physical realizations (for example, between an adder's schematic (structure) and its layout).

Both kinds of **Bindings** have properties that represent paths into the four hierarchies, e.g. 'St-Path' as shown in Figure 4-2. The reason paths must be used is that **Bindings** refer to unique **Single-Model-Components**. Such a **Single-Model-Component** may be deep down in the recursion hierarchy, and the only way to uniquely specify it is by giving a complete path down into the hierarchy, starting at the root **Component**.

The **Kind-of-Df-Path** property of **Operation-Binding** simply indicates whether the binding is to a **Node** or a **Value**; similarly the **Kind-of-St-Path** property indicates whether the binding is to a **Carrier** or a **Module**. These are examples of the use of a 'generic interrelation abstraction'[3]. All combinations are permitted in the schema. Similar considerations apply to **Realization-Bindings**.

The reason that there is no **Kind-of-Range-Path** property for **Operation-Bindings** is that the only valid timing element for a binding is a

---

[3]This abstraction primitive and the recursion abstraction mentioned earlier were specifically defined for the ADAM VLSI design database, and are supported by the 3DIS data model.

Range. **Points** have infinitesimal duration, and hence are never suitable for binding either operations or values to structural elements.

5. Conclusions

This paper has described a representation and a schema that form the fundamental data structure of the ADAM (Advanced Design AutoMation) project. We briefly described the 3DIS, an extensible information modeling framework that captures the underlying semantics of the ADAM VLSI application environments, and supports the requirements specific to this domain. We have applied the database modeling framework to the ADAM system and have presented an example 3DIS database design.

The 3DIS database is object-oriented in that all data entities, relationships defined on entities, events and operations, as well as the description of data (meta-data) are modeled as objects. It provides a well structured unified view of the application information that reduces the requested level of expertise for database manipulation, and database development. The extension of the 3DIS model to support the specific modeling requirements of engineering design environments, such as modeling recursively defined entities and concepts, simplifies the task of database design.

We expect significant benefits from the presented approach in construction of the overall ADAM system. Since the design data is unified by the database, adding application packages is greatly simplified. Since non-experts can access the underlying schema easily, the designers of CAD packages need not be database experts to use the system flexibly.

The representation schema is based on the idea of unifying the design data in three major structures called the specification, the target, and the

library, respectively. Each of these consists of a single component or a collection of components, where all components are modeled in the same way. A component is represented in terms of four nonisomorphic hierarchies: dataflow, timing, structural, and physical. The four hierarchies are linked by explicit relationships called bindings.

This schema has several advantages. It cleanly represents the data of interest. None of the important relationships between specification and the target is obscured. Designer freedom is limited to the degree permitted by the specification. The same concepts and techniques can be used to analyze and construct target designs, specifications, and library components. Finally, the design details are hidden until they are needed.

The unification of the database with the synthesis and analysis tools results in an automated process from algorithm specification to circuit layout. This in itself is expected to simplify the design process and enhance design correctness.

A Pascal-based graphical interface to the 3DIS, implemented on an IBM PC/XT [Afsarmanesh 85b] is currently under construction. A Pascal-based graphical editor for an older, file-oriented version of the design data structure (DDS) has been implemented [Knapp 84].

There are some other design automation software packages using the old DDS and other specialized file formats. These packages include: a clocking scheme synthesis package, a pipeline allocation package, an operation and value collision detector [Park 84], [Park 85], and [Parker 84a], a knowledge-based system for the insertion of testability-enhancing components into otherwise finished register-transfer designs [Abadir 85], a knowledge-based system for

making PLAs (Programmed Logic Arrays) testable [Breuer 85], and a general-purpose register-transfer level allocator [Parker 84c]. Other packages currently under development for ADAM and building on the database described in this paper include a silicon compiler, an area estimator, a restricted natural-language interface for design specification, a design activity planner, and a semantic-net representation for designer knowledge and high-level design description information.

Future work on this project includes integrating the system into a coherent whole. In particular the data structures of the synthesis packages must be changed from the DDS file format to the new 3DIS-oriented database support system in order for ADAM to function as a single unified system. The implementation of the 3DIS database system and its user interface must be completed. The definition and implementation of database activities, e.g. the invocation of semantic checking routines, must also be added.

5.1. Acknowledgements

The authors would like to thank all of the people at USC and elsewhere whose comments, suggestions, and participation helped to pose the questions and refine the solutions presented here. In particular, the authors would like to thank Mel Breuer, Bernie Cohen, John Granacki, Lou Hafer, Sally Hayati, Fadi Kurdahi, Sany Leinwand, Mike McFarland, Nohbyung Park and K. V. Bapa Rao.

I. Appendix A: Generalization Hierarchies for Four Subspaces

This appendix gives a more global view of the type-subtype hierarchies. Parts of these figures have been shown in the body of the report; the full figures are included as a comprehensive reference. There are a few minor differences between the dataflow hierarchy, which was used as an example, and the other hierarchies. Some of these differences are discussed here. In the figures that follow, the following two conventions apply. First, where property names are given and the range types of the properties are not obvious, colons (:) are used to indicate the type object over which the property is defined. Second, where the type objects are starred (*), the property defines a one-to-many relationship.

I.1. Notes

In the example given in the body of this report we discussed the dataflow subspace in detail. The other subspaces are not structured in exactly the same way. Here we will discuss some major differences.

1. The **Static-Storage-Property** of a **Module** represents the possibility that it might have static storage elements in it, i.e. memory. Hence a register would have this property **true**, where a gate would have **false**.

2. **Timing-Models** have a set of **Durations**. These represent either constrained or achieved time intervals. Hence it would be possible to have both constraints and achievements listed for the same range.

3. **Timing-Models** also have a **Causality** property, which distinguishes constraining, measuring, and causative arcs from one another.

4. **Physical-Models** have a number of unique attributes.

    a. **Shape**: this expresses the bounding polygon of a piece of layout.

    b. **Boxes**: A block may not consist entirely of sub-blocks. In that case it has some primitive (layer, rectangle) boxes.

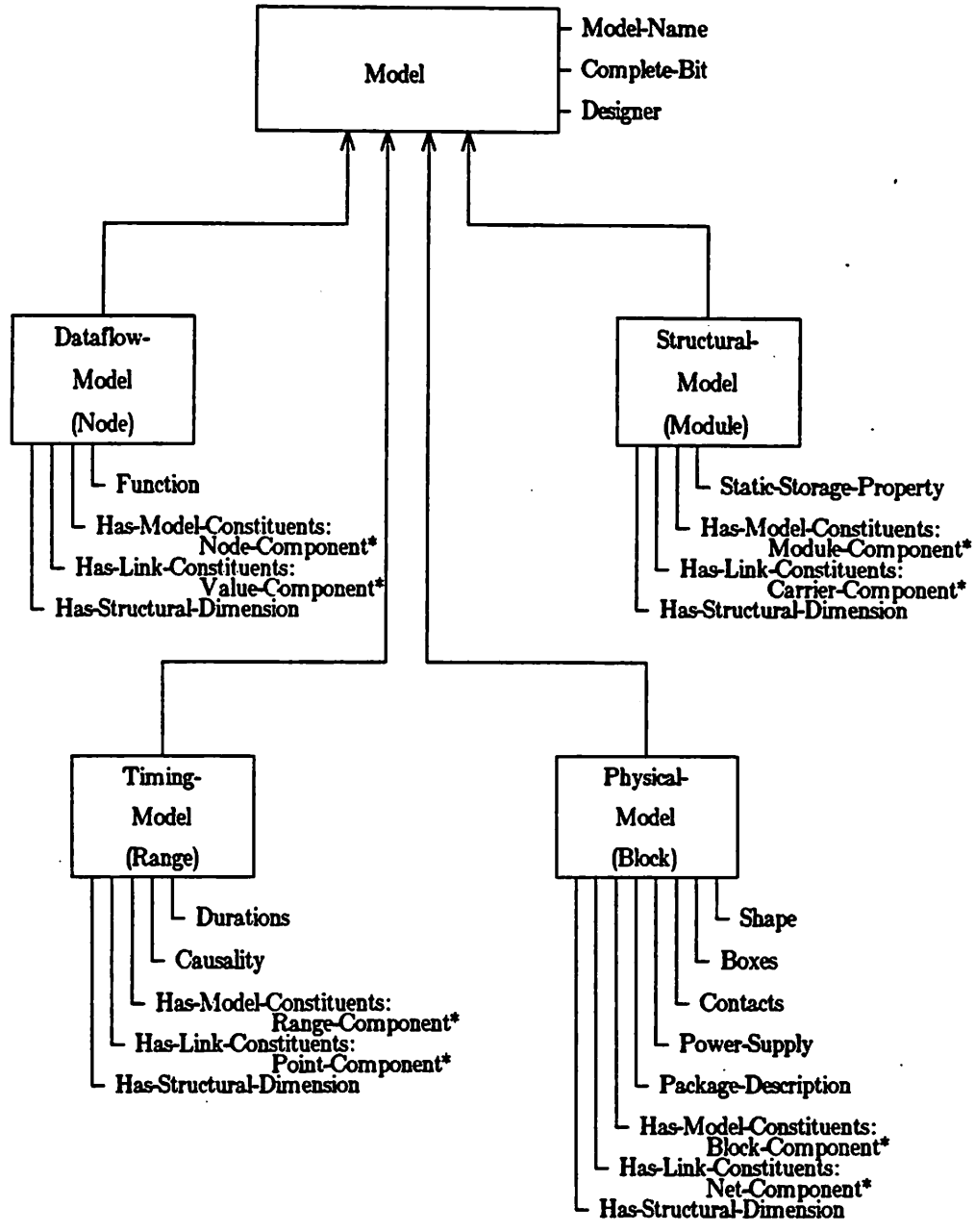

Figure I-1: The generalization hierarchy of Models

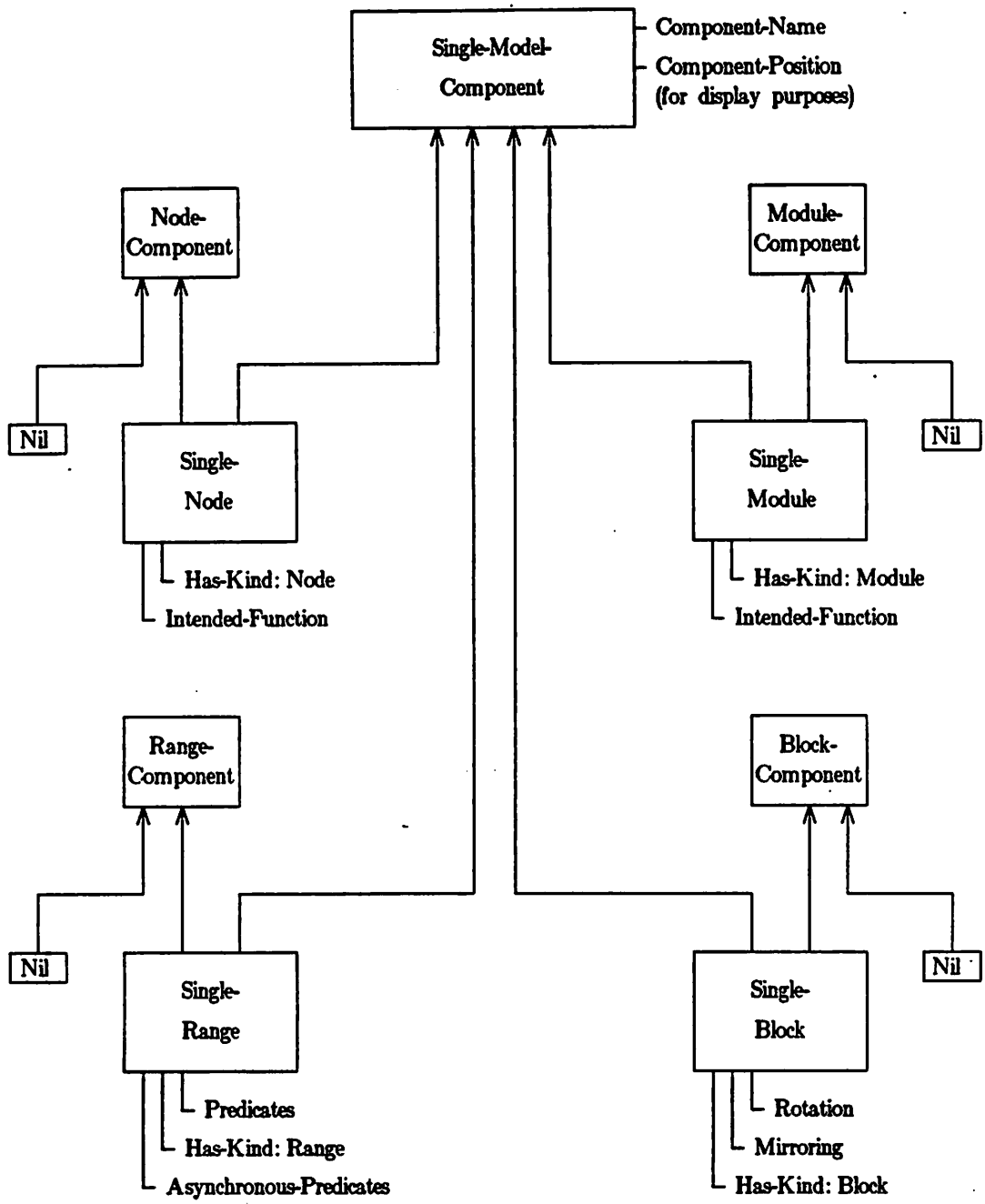

Figure I-2: The generalization hierarchy of Single-Models

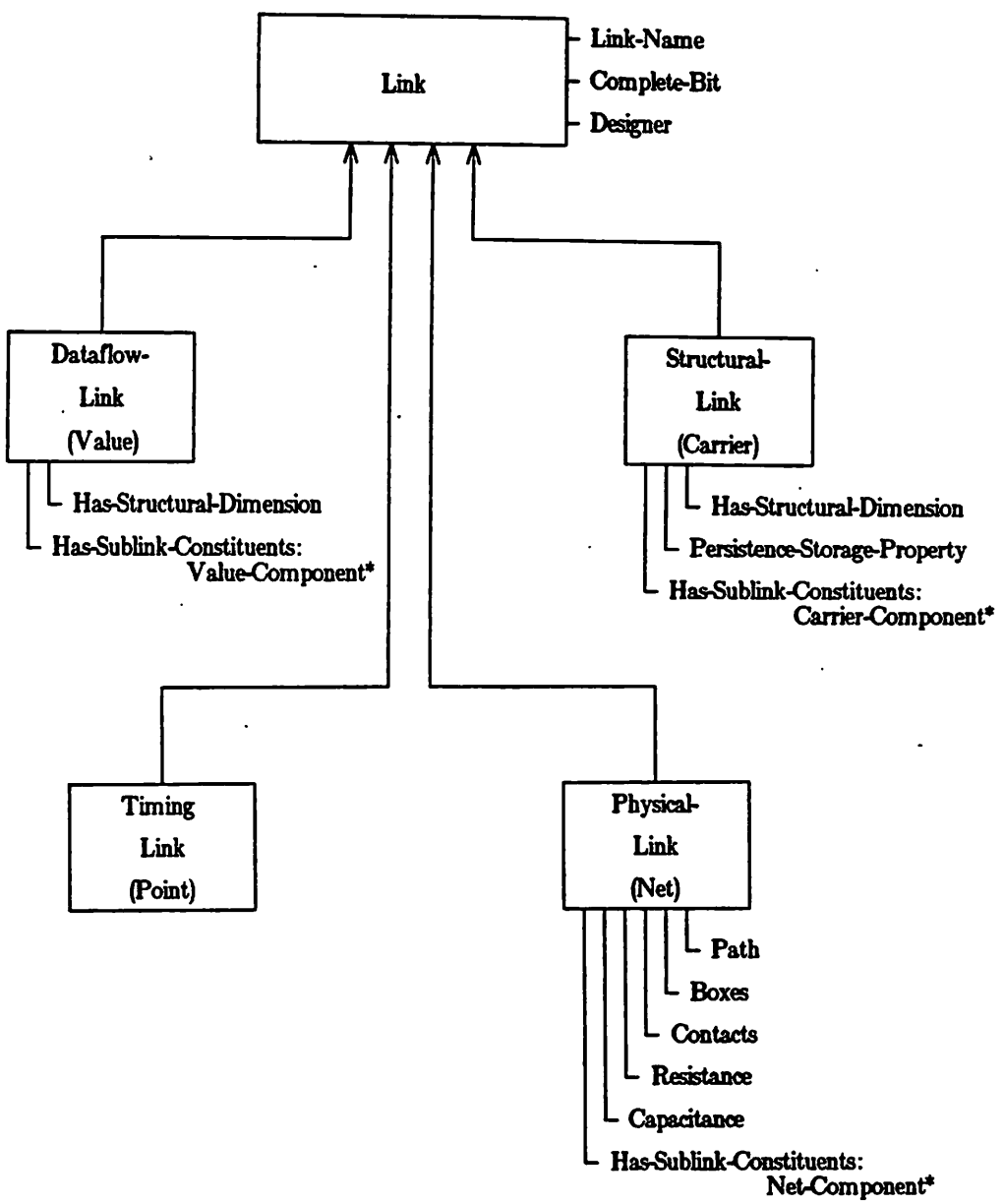

Figure I-3: The generalization hierarchy of Links

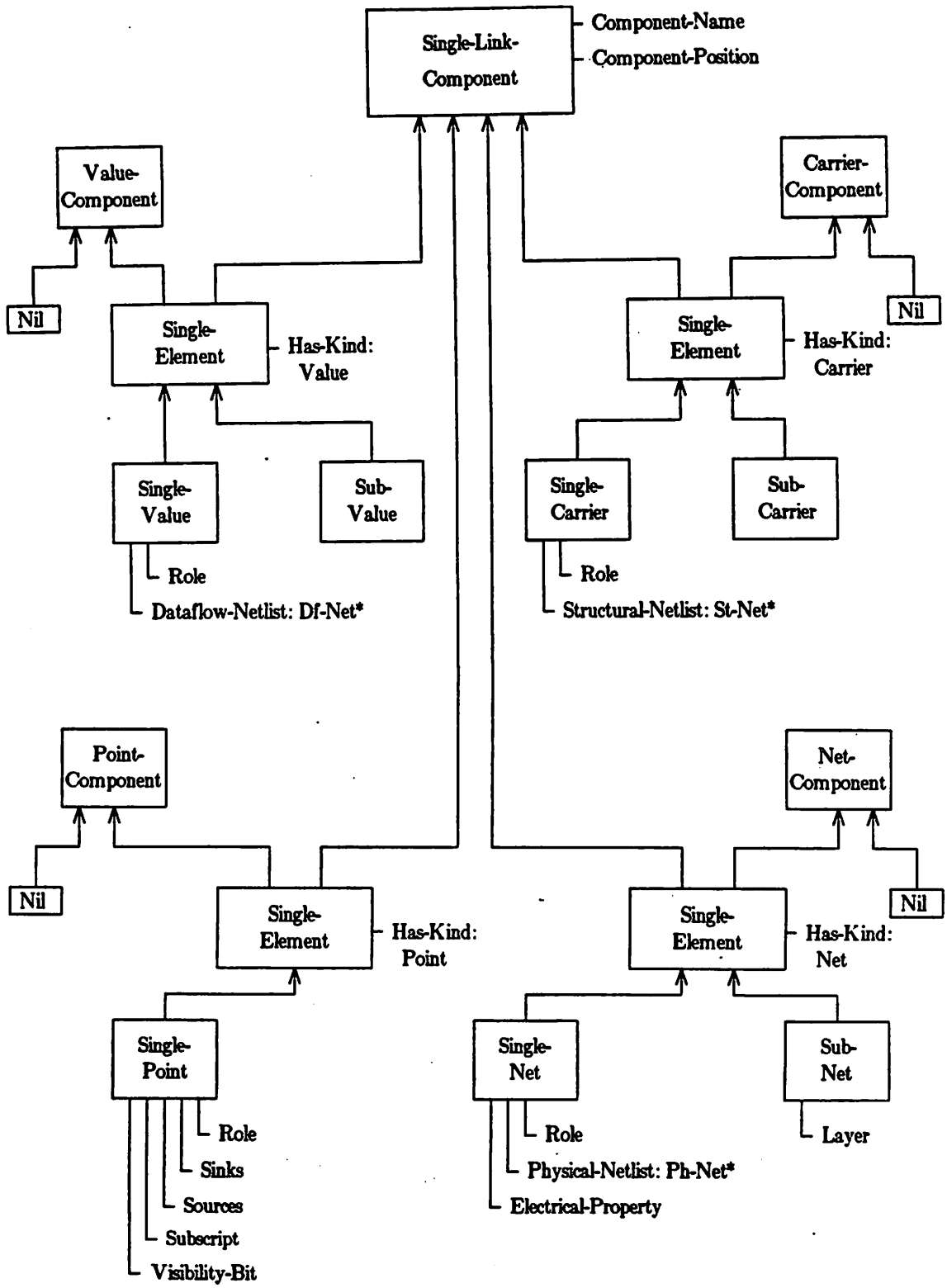

Figure I-4: The generalization hierarchy of Single-Links

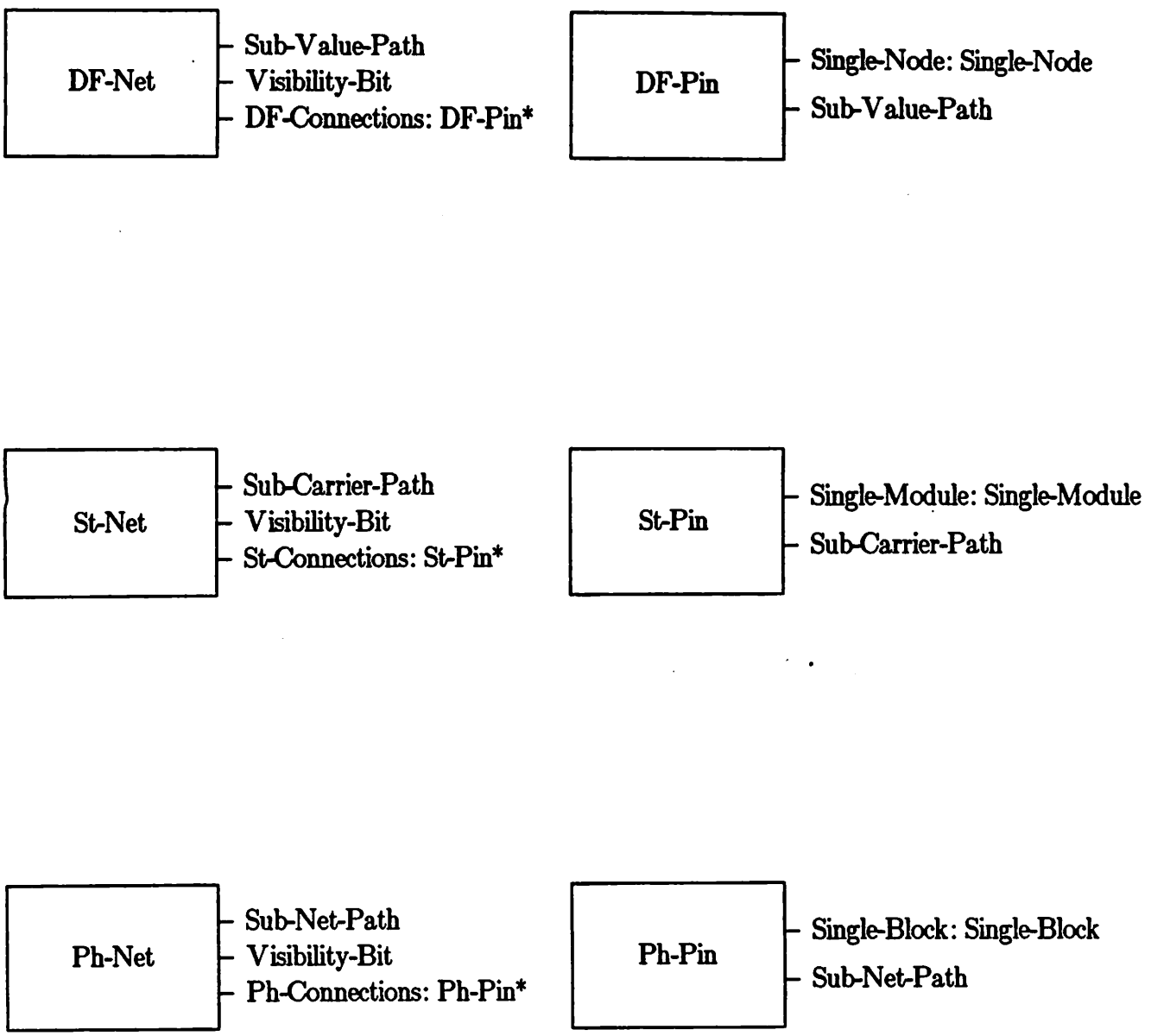

Figure I-5: Nets and Pins

c. **Contacts**: Similar to boxes, but representing particular interlayer interconnection points.

d. **Power-Supply**: Describes the power requirements (not the connections) of the block

e. **Package-Description**: Describes the package of an OEM or packaged component, as opposed to a layout.

5. A **Single-Block** has two attributes, **Rotation** and **Mirroring**, that describe coordinate transformations applied to the block in determining its position and orientation.

6. A **Single-Range** has the following predicate attributes:

   a. **Predicates**: describe the conditions under which normal branching will occur.

   b. **Asynchronous-Predicates**: describe the conditions under which branching is not synchronized to a particular point in the timing graph, e.g. resets. The semantics of both kinds of predicates is described in detail in [Knapp 83b].

7. **Single-Modules** and **Single-Nodes** have **Intended-Function** attributes, which signify the functions they perform in the target design, as opposed to the function(s) they may have as isolated entities. Hence, for example, a **Single-Module Add4** might have **Intended-Function** 'Address Indexing Adder', whereas its intrinsic function (that of its **Has-Kind**) is just **Adder**.

8. A **Net** has the following properties that do not have direct counterparts in the other subspaces:

   a. **Boxes**: These are the constituent rectangles of the physical layout of the **NET**. They are colored rectangles, i.e. they have layer information attached.

   b. **Contacts**: These are the interlayer connections that form parts of the **Net**.

   c. **Path**: This is the abstract path along which the **Net**'s boxes are laid out.

d. **Capacitance**: The summed parasitic capacitance of the **Net**.

e. **Resistance**: The series parasitic resistance of the **Net**.

9. A **Sub-Net** has an additional property **Layer**. This can be derived in some cases from the **Boxes** of its **Kind**; in other cases this information will be difficult to express in a single attribute.

10. A **Carrier** has a **Persistence-Storage-Property**, which describes its ability to store charge. Under some circumstances charge storage can be used as a memory mechanism.

11. **Points** do not have any attributes. This is because they are of enumerated type, and their attributes are implicit in the type. Those types are:

    a. **Simple** points have one in-arc and one out-arc. These points represent events.

    b. **Alpha** points have one out-arc and no in-arcs. These points represent loop reentry points. The out-arc must have an indexing subscript, as the loop is considered to be a (possibly infinite) set of instantiations of the arc(s) between the alpha and the omega points.

    c. **Omega** points have one in-arc no out-arcs. These points represent loop backjump points.

    d. **Or-fork** points have one in-arc and a number of out-arcs. They represent branch points. Each out-arc must have a predicate attached to it, describing the conditions under which the arc is taken.

    e. **And-fork** points have one in-arc and a number of out-arcs. These represent **cobegin** constructs, i.e. points at which concurrent streams of events flow apart.

    f. **Or-join** points have a number of in-arcs and a single out-arc. They represent points at which several disjoint execution paths merge.

g. **And-join** points have a number of in-arcs and a single out-arc. They represent coend points.

12. **Single-Points** have sink and source sets; these refer to **Single-Ranges** and express the connectivity of the timing graph.

References

[Abadir 85] Abadir, M.S. and M.A. Breuer.
A Knowledge Based System for Designing Testable VLSI Chips.
IEEE Design and Test of Computers , August, 1985.
to appear.

[Afsarmanesh 84] Afsarmanesh, H., and Mcleod, D.
A Framework for Semantic Database Models.
In *Proceedings of NYU Symposium on New Directions for Database Systems.* New York, NY, May, 1984.

[Afsarmanesh 85a]
Hamideh Afsarmanesh.
3DIS: An Extensible Object-Oriented Information Model.
Technical Report, Department of Computer Science, University of Southern California, April, 1985.

[Afsarmanesh 85b]
Hamideh Afsarmanesh.
The 3 Dimensional Information Space (3DIS), An Extensible Browsing-Oriented Framework for Database Systems.
PhD thesis, Department of Computer Science, University of Southern California, 1985.
Thesis forthcoming.

[Batory 84] D. S. Batory, Won Kim.
Modelling Concepts for VLSI CAD Objects.
technical report TR-84-35, Department of Computer Science, University of Texas at Austin, December, 1984.

[Breuer 85] Breuer, M. and Zhu, X.
A Knowledge Based System for Selecting a Test Methodology for a PLA.
In *Proceedings of the 22nd Design Automation Conference.* ACM and IEEE, 1985.

[Bushnell 83] M. Bushnell, D. Geiger, J. Kim, D. LaPotin, S. Nassif, J.Nestor, J. Rajan, A. Strojwas, H. Walker.
DIF: The CMU-DA Intermediate Form.
Technical Report CMUCAD-83-11, CMU Center for Computer-Aided Design, 1983.

[Carter 79] Carter, W. C., Joyner, W. H. and Brand, D.
Symbolic Simulation for Correct Machine Design.
In *Proceedings of the 16th Design Automation Conference*, pages 280-286. ACM SIGDA and IEEE Computer Society DATC, June, 1979.

[Davis 82] R. Davis, H. Shrobe, W. Hamscher, K. Wieckert, M. Shirley, S. Polit.
Diagnosis Based on Description of Structure and Function.
In *Proceedings of the National Conference on AI*, pages 137-142. AAAI, 1982.

[Director 81] S. W. Director, A. C. Parker, D. P. Siewiorek, D. E. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
IEEE Transactions on Circuits and Systems CAS-28:634-645, July, 1981.

[Dittrich 85] Dittrich, K.R., Kotz, A.M., Mulle, J.M.
An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases.
Technical Report, Institut fuer Informatik II, Universitaet Karlsruhe, West Germany, 1985.

[Eastman 80] C.M. Eastman.
System Facilities for CAD Databases.
In *Proceedings of the 17th Design Automation Conference.* 1980.

[Evangelisti 79] Evangelisti, C. J., Goertzel, G., and Ofek, H.
Using the Dataflow Analyzer on LCD Descriptions of Machines to Generate Control.
In *Proceedings of the 4th International Symposium on Computer Hardware Description Languages.* ACM and IEEE Computer Society, October, 1979.

[Granacki 82] Granacki, J.J. and Parker, A.C.
The Effect of Register-Transfer Design Tradeoffs on Chip Area and Performances.
In *Proceedings of the 20th Design Automation Conference.* December, 1982.

[Granacki 85] John Granacki, David Knapp, and Alice Parker.
The ADAM Advanced Design AutoMation System: Overview, Planner, and Natural Language Interface.
In *Proceedings of the 22nd Design Automation Conference.* 1985.

[Hafer 83] Hafer, L., and Parker, A.
A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic.
IEEE Transactions on Computer-Aided Design CAD-2(1), January, 1983.

[Hammer 81] Hammer, M., and McLeod, D.
Database Description with SDM: A Semantic Database Model.
ACM Transactions on Database Systems 6(3):351-386, September, 1981.

[Hitchcock 83] C. Y. Hitchcock III and D. E. Thomas:
A Method of Automatic Datapath Synthesis.
In *ACM/IEEE 20th Design Automation Conference Proceedings*, pages 484-489. ACM/IEEE, 1983.

[Katz 82] Katz, R. H.
A Database Approach for Managing VLSI Design Data.
In *Proceedings of the 19th Design Automation Conference.* 1982.

[Kent 79] Kent, W.
Limitations of record-oriented information models.
ACM Transactions on Database Systems 4:107-131, March, 1979.

[Knapp 83a] David Knapp, John Granacki, and Alice Parker.
An Expert Synthesis System.
In *Proceedings of the International Conference on Computer Aided Design (ICCAD)*, pages 419-424. ACM-IEEE, September, 1983.

[Knapp 83b] Knapp, D. and Parker, A.
A Data Structure for VLSI Synthesis and Verification.
Technical Report DISC 83-6a, Digital Integrated Systems Center, Dept. of EE-Systems, University of Southern California, October, 1983.

[Knapp 84] David W. Knapp.
The Agis Data Structure Editor
USC DA Group, 1984.

[Knapp 85] David W. Knapp and Alice C. Parker.
A Unified Representation for Design Information.
In *Proceedings of the 1895 Conference on Hardware Description Languages*. IFIP, 1985.

[Leiserson 83] Leiserson, C. E., Rose, F. M. and Saxe, J. B.
Optimizing synchronous circuitry by retiming.
In *Proceedings of Third Caltech Conference on VLSI*, pages 23-36. Computer Science Press, 1983.

[Leive 81] Leive, G.
The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.
PhD thesis, Dept. Of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., January, 1981.

[McFarland 81] McFarland, M.
On Proving the Correctness of Optimizing Transformations in a Digital Design Automation System.
In *Design Automation Conference Proceedings no. 18*, pages 90-97. ACM SIGDA, IEEE Computer Society-DATC, June, 1981.

[McFarland 83] McFarland, M. and Parker, A.
An Abstract Model of Behavior for Hardware Description.
IEEE Transactions on Computers C-32(7):621-637, July, 1983.

[McLeod 83] McLeod, D., Bapa Rao, K. V., and Narayanaswamy, K.
Information Modelling for CAD/VLSI.
In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. San Jose, California, May, 1983.

[Nagle 82] Nagle, A., Cloutier, R., and Parker, A.
Synthesis of Hardware for the Control of Digital Systems.
IEEE Transactions on Computer-Aided Design CAD-1(4):201-212, 1982.

[Nestor 82] J.A. Nestor and D.E.Thomas.
Defining and Implementing a Multilevel Design Representation With Simulation Applications.
In *ACM/IEEE Nineteenth Design Automation Conference Proceedings*, pages 740-746. ACM/IEEE, 1982.

[Otten 82] Otten, R.H.
Automatic floorplan design.
In *Proceedings of the 19th design automation conference.* IEEE and ACM, 1982.

[Park 84] Park, N. and Parker, A.
Synthesis of Optimal Clocking Schemes for Digital Systems.
Technical Report DISC/84-1, Dept. of EE-Systems, University of Southern California, May, 1984.

[Park 85] Park, N. and Parker, A.
Synthesis of Optimal Pipeline Clocking Schemes.
Technical Report DISC/85-1, Dept. of EE-Systems, University of Southern California, January, 1985.

[Parker 84a] Parker, A., Park, N. and Knapp, D.
Simulation Effectiveness and Design Verification.
Technical Report DISC/84-2, Department of EE-Systems, University of Southern California, October, 1984.

[Parker 84b] Alice C. Parker.
Automated Synthesis of Digital Systems.
IEEE Design and Test , November, 1984.

[Parker 84c] Parker, A., Kurdahi, F. and Mlinar, M.
A General Methodology for Synthesis and Verification of Register Transfer designs.
In *Proceedings of the 21st Design Automation Conference.* ACM SIGDA, IEEE Computer Society, June, 1984.

[Pitchumani 82] Pitchumani, V., and Stabler, E.P.
A Formal Method for Computer Design Verification.
In *Proceedings of the 19th Design Automation Conference,* pages 809-814. ACM SIGDA and IEEE Computer Society DATC, June, 1982.

[Pitchumani 84] Pitchumani, V.
An Assertion-oriented Method for Verification of Parallelism in Hardware.
In *IEEE Proceedings of the ICCD*, pages 462-467. October, 1984.

[Soukup 81] Jiri Soukup.
Circuit Layout.
Proceedings of the IEEE 69(10), October, 1981.

[Stonebraker 82] Stonebraker, M., and Kalash, J.
TIMBER: A Sophisticated Relation Browser.
In *Proc. 8th Int. Conf. Very Large Data Bases*, pages 1-10. Mexico City, Mexico, Sept. 8-10, 1982.

[Thomas 77] Thomas, D.
The Design and Analysis of an Automated Design Style Selector.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April, 1977.

[Wong 79] S. Wong, W.A. Bristol.
A CAD Database.
In *Proceedings of the 16th Design Automation Conference.* 1979.

An Extensible Object-Oriented Approach to Databases for VLSI/CAD

Hamideh Afsarmanesh
Dennis McLeod

Department of Computer Science
University of Southern California

David Knapp
Alice Parker

Department of Electrical Engineering
University of Southern California

Digital Integrated Systems Center Report
DISC/85-3

Department of Electrical Engineering-Systems
University of Southern California
Los Angeles, California 90089-0871

23 April 1985

This research was supported by the National Science Foundation, #ECS 8310774, and the Joint Services Electronics Program through the Air Force Office of Scientific Research under contract # F49620-81-C-0070.

Abstract

This paper describes an approach to the specification and modeling of information associated with the design and evolution of VLSI components. The approach is characterized by combined structural and behavioral descriptions of a component. Database modelling requirements specific to the VLSI design domain are considered and techniques to address them are described. An extensible object-oriented information management framework, the 3DIS (3 Dimensional Information Space), is presented. The framework has been adapted to capture the underlying semantics of the application environment by the addition of new abstraction primitives. An experimental prototype implementation of the database and its browsing-oriented interface is described.