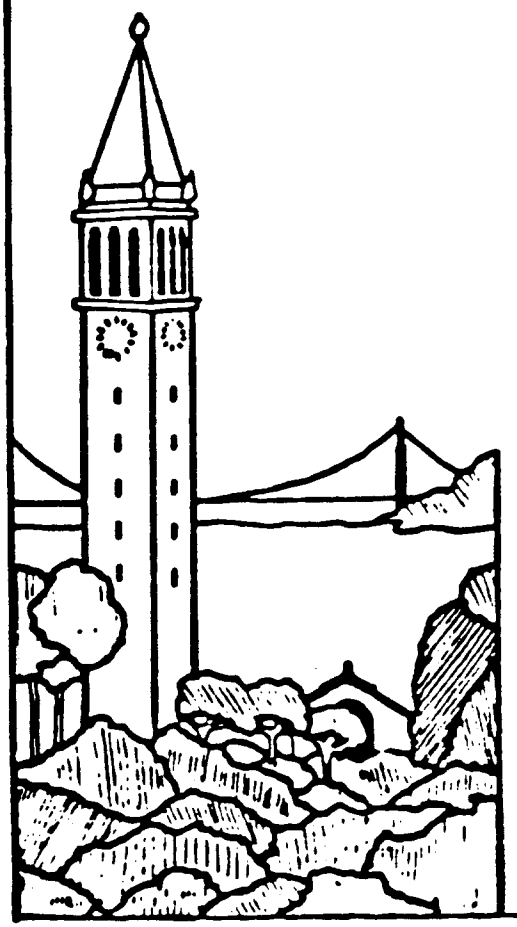
A Validation Subsystem of a Version Server for Computer-Aided Design Data

Rajiv Bhateja and Randy H. Katz



Report No. UCB/CSD 87/315

October 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

A Validation Subsystem of a Version Server for Computer-Aided Design Data

Rajiv Bhateja and Randy H. Katz

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720
(415) 642-8778

ABSTRACT

Design methodologies specify the sequence in which verification programs must be successfully executed to determine a design's correctness. We present a mechanism for assisting designers in adhering to their methodology, specified as Prolog rules that must match a verification event log. A new version cannot be released if a methodology violation is detected. Designers can query for the source of their violation. The system has been implemented within a prototype Version Server.

Key words and phrases: Design Systems and Methodologies (10), Databases (14).

1. Introduction

VLSI design presents an enormously complex data management task. A key challenge is to keep the design consistent across its multiple representations, often created and refined in parallel. *Design database systems* provide the structural framework in which the relationships across representations are captured. Yet this by itself is not sufficient to guarantee the consistency of a design.

Most design projects follow a *design methodology:* a prescribed sequence of steps leading to the completion of a design. These steps are often realized by CAD programs that either transform the design (perhaps synthesizing a piece from existing pieces) or analyze it for correctness. If all the steps have been followed, in the correct sequence, and have resulted in the desired outcomes, then the design is considered complete and correct. With few exceptions [EAST81, NOON82], manual check-off procedures have been used to enforce the methodology. Since the design process depends on computer-based tools to create, analyze, and store the design, it should also use such tools to enforce the methodology itself.

We have been developing a *Version Server* to manage the data of large VLSI design projects, as they undergo revisions over time [KATZ86a,b]. Such a system must provide some mechanism to track the verification status of designs. The Version Server maintains a log of verification events, captured by an extension of the conventional UNIX C-Shell through which designers interact with the system. A design methodology is specified as a proper sequence of CAD program invocations. These sequences must match the log to insure that the methodology has been followed.

Throughout the paper, we use the terms verification and validation interchangeably. In theory, *verification* implies that correctness has been "proven" through formal mechanisms, whereas *validation* implies that it has been demonstrated by evidence. Almost all analysis tools actually validate the design, yet are called verification tools. The Validation Subsystem described in this paper in a sense verifies the correctness of the design by using Prolog to prove verification rules (methodology specifications of program sequences) from verification events (logged program executions).

The rest of the paper is organized as follows. In the next section, we briefly describe the functions provided by the Version Server. Section 3 presents a design methodology that will be used as a running example throughout the rest of the paper. Section 4 describes how design methodologies are specified. The implementation and use of the Validation Subsystem is discussed in Section 5. Section 6 contains our conclusions and status.

2. Overview of the Version Server

The Version Server manages units of design called *design objects*, which correspond to the named design files found in most conventional design environments. The Version Server creates a structural framework in which to organize design objects, without placing restrictions on their internal formats. Design objects are *typed*, e.g., a

design object might be of type layout, schematic, or functional. Objects are uniquely denoted by *object-name[version #].type.*

The Version Server recognizes three possible relationships among design objects: *version histories*, *configurations*, and *equivalences*. Version histories maintain **is-a-descendent-of** and **is-an-ancestor-of** relationships among version instances of the same real world object (e.g., ALU[3].layout **is-a-descendent-of** ALU[2].layout, both of which are versions of an ALU.layout). Configurations relate composite objects to their components via **is-a-component-of** and **is-composed-of** relationships. Finally, equivalences identify objects across types that are constrained to be different representations of the same real world object, e.g., ALU[2].layout **is-equivalent-to** ALU[3].schematic if these are different representations of the same ALU design. These relationships provide alternative hierarchical organizations for design objects. For example, an object can be clustered with its other versions, the composite objects that use it as a component, or the objects to which it is equivalent. The Version Server Data Model is summarized in Figure 2.1.

Operationally, the Version Server supports a workspace model. Designers *check-out* objects from shared archives into their private workspaces. Changes made in private workspaces are not visible to other designers until such objects are *checked-in* to a shared group workspace, where the changes can be integrated with other designers' work. Finally, the modified object is returned as a new version to the shared archive.

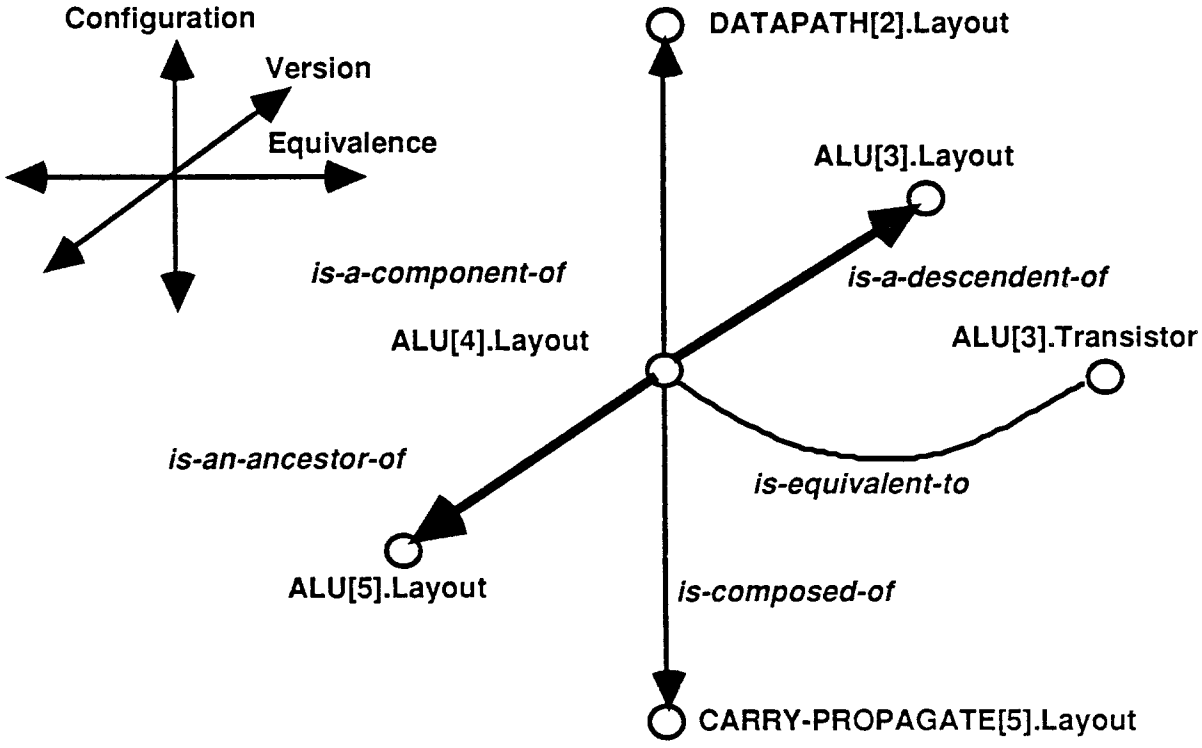


Figure 2.1: Version Server Data Model

The Version Server supports three distinct relationships among design objects: version histories (is-a-descendent-of), configurations (is-composed-of), and equivalences (is-equivalent-to). The latter is used to constrain objects on check-in: they cannot be returned to the archive as new versions unless all equivalences that involve them can be shown to have been validated.

Equivalence relationships are the most important for verification. Besides indicating equivalent objects across types, they represent constraints on the design. The meaning of "equivalence" is defined relative to a particular project's design methodology. Designers must demonstrate that two design objects constrained to be equivalent actually are.

3. An Example Design Methodology

Consider a typical VLSI design project. We will examine a simple design methodology for this project. At least conceptually, a methodology is specified by filling in the chart shown in Figure 3.1. Well-formedness constraints are entered for each type of design object. For each pair of types, the method by which consistency between the types can be demonstrated is entered in the matrix.

In our example, four types of design objects exist: layout, logic schematic, transistor netlist, and functional description. Schematic capture and layout editor tools create their respective representations; a circuit extractor creates a transistor netlist from a layout; a netlist generator creates the netlist from the logic schematic. The verification tools include a netlist comparator, a switch level simulator, a logic simulator, and a functional simulator.

First, we must define the well-formedness conditions. For layout objects, this would be the successful execution of a design rule checker against the object. For logic schematics and transistor netlists, it would be the execution of a connectivity/electrical rules checker. A successful compilation and linking of the functional description would indicate its well-formedness.

The next step is to specify how consistency is determined among representation types. We restrict consistency conditions to those that involve equivalence, since this is

	T1	T2	T3	T4
T1	well-formed constraints	T1-T2 EQ constraints	T1-T3 EQ constraints	T1-T4 EQ constraints
T2		well-formed constraints	T2-T3 EQ constraints	T2-T4 EQ constraints
T3			well-formed constraints	T3-T4 EQ constraints
T4				well-formed constraints

Figure 3.1 -- A Methodology Matrix

A methodology describes how objects of different types can be shown to be equivalent. Well-formedness constraints are entered along the diagonal, as are equivalence constraints between objects of the same type. Pairwise equivalence constraints are entered in the upper triangle of the matrix.

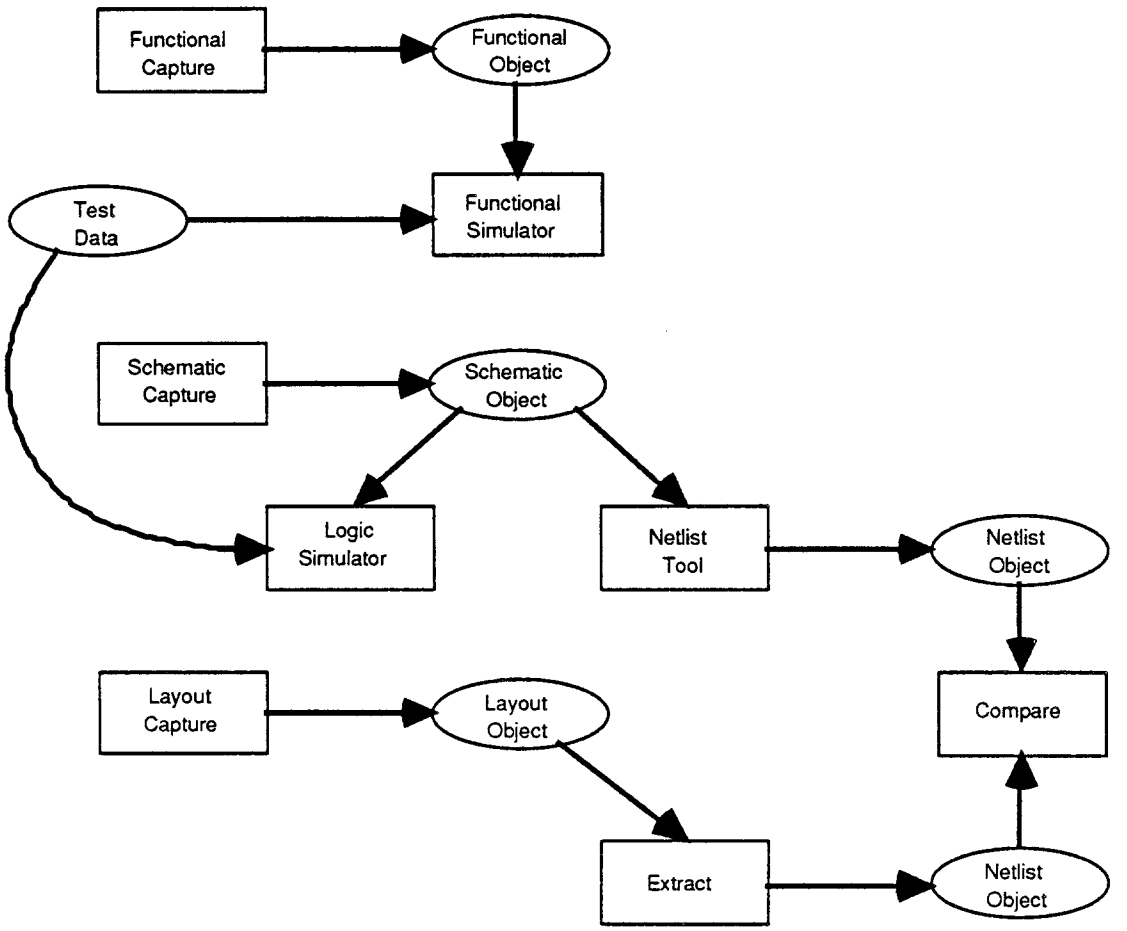


Figure 3.2: Design Process Flow

Ovals represent design data; boxes represent CAD tools. If a CAD tool creates the object, it is placed at the tail of the arrow, with the head at the object. If it consumes the object, then it is shown at the head of the arrow, with the object at the tail.

what existing verification tools concentrate on demonstrating. For layouts and netlists, design objects are equivalent if the netlist object has been extracted from the layout object. Alternatively, a layout and a netlist can be shown to be equivalent through parallel simulation: a netlist is extracted from the layout and simulated with the same inputs as the original netlist. If the outputs are identical, the layout and the netlist are equivalent. Schematic and netlist objects are equivalent if the netlist has been generated from the schematic. Not all equivalence constraints need to span types. Two netlists are equivalent if the netlist comparator determines this to be so. Finally, a functional object and a schematic object are equivalent if under the same test inputs they generate equivalent outputs (perhaps as determined by a designer) from their respective simulations. The design process is shown in Figure 3.2 and the equivalence constraints as they would appear in the database are summarized in Figure 3.3.

Observe the following. Consistency constraints, such as equivalences, are verified through the execution of CAD programs. Sometimes these involve a suite of tools that must be executed in sequence, e.g., first extract, and then simulate. However, the

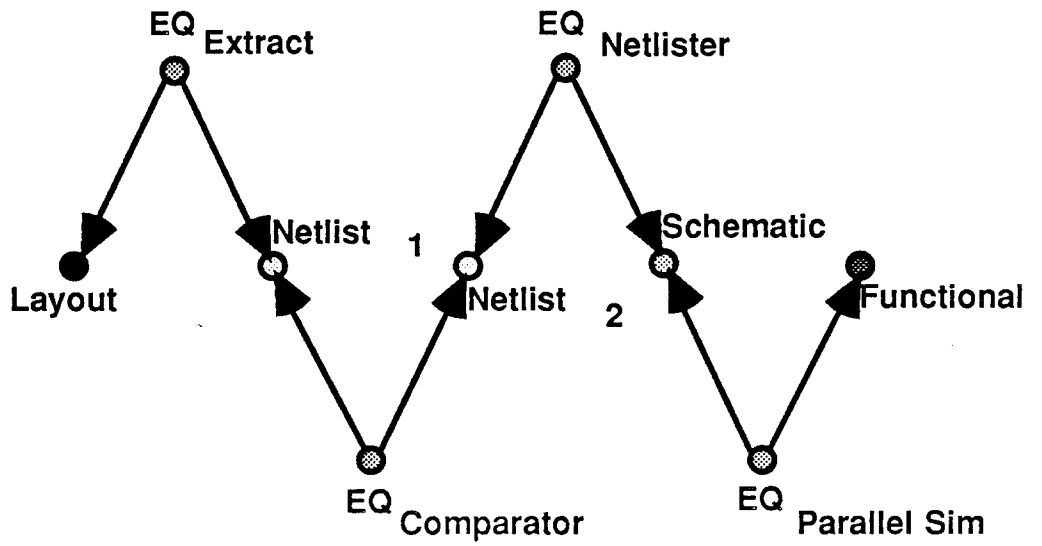


Figure 3.3: Equivalence Constraints Among Design Object Types

Equivalence constraints are explicitly represented by structural objects in the design database. The process flow of Figure 3.2 describes how these can be verified. For example, a layout and netlist are equivalent of the latter has been extracted from the former.

designer cannot be eliminated from the loop: parallel simulation requires someone to adjudicate the objects' equivalence. Often alternative methods for demonstrating consistency exist, such as (i) extract from one representation into another OR (ii) perform parallel simulation. To prove that two objects are consistent, a history of verification events must be maintained, and it must be possible to examine this log to determine whether the tools were invoked on the correct versions of objects and in the proper sequence.

Other forms of consistency constraints among objects may be of interest besides equivalence across types. An example is a configuration constraint that restricts a component layout to fit within an allocated area in the composite layout that contains it. However, equivalence constraints are most readily validated with existing analysis tools. Although we use equivalence and consistency interchangeable, it should be noted that consistency refers to a much richer class of constraints than equivalence.

4. The Validation Subsystem

4.1. Introduction

The Validation Subsystem provides the mechanism for filtering out those objects for which the mandatory constraints have not been explicitly demonstrated. It monitors the verification status of the design and enforces the methodology on object check-in. It is fundamental to the correct operation of the Version Server that the archive not be contaminated with objects of questionable integrity.

The Validation Subsystem expects a methodology to be specified as *validation*

rules. These correspond to the entries made in the methodology matrix discussed above. The rules, in conjunction with the event log, are used to deduce the validation status of design objects. An object is accepted in the archive as a new version only if it can be shown to satisfy the verification conditions specified for its type and all types related to it through equivalences.

At check-in time, the Version Server queries the Validation Subsystem about the status of the object being checked-in. The Validation Subsystem applies the rules to the log. If all constraints are satisfied, the object is checked-in to the archive. Otherwise, the unsatisfied constraints are reported. There will be occasions when design objects will fail validation, but the designers will nevertheless desire to archive them. While the Validation Subsystem can be bypassed in an emergency, this is an option to be discouraged.

4.2 Rule Syntax

A *rule* is a conditional statement similar to an if-statement. It can be expressed as:

if (A) then B

More often, especially in Prolog [CLOC84]), rules are written in the form:

B if A

in which B is called the *goal* or *head* of the rule and A is called the *conditional-part* or *tail* of the rule.

The condition part may contain more than one condition. Thus, a rule may take the form:

D if (A & B & C)

in which the symbol '&' is used to denote the logical **and** operator between the conditions A, B, and C. The goal D is true if each of the conditions A, B, and C are satisfied. In addition, there may be many rules that support the same goal. For example:

R if (A & B & C)
R if (D & E & F)

In this case, R can be proved true by either of the two rules. A rule could contain logical **or** operators. However, we restrict validation rules to being conjunctive. Disjunction merely leads to multiple rules.

4.3 Validation Rules

Validation rules constitute a set of checks that are applied to a design object of a given type to ensure:

(a) its completeness (i.e., nothing is missing; it is well-formed), and
(b) its consistency with other design objects.

Completeness rules refer to type-dependent (well-formedness) correctness. They ensure that an object is of the type (e.g., layout) that it purports to be, and that it is a correct instance of its type. Typically this is achieved by showing that certain performance goals have been met or that certain design constraints (such as the design rules) have been satisfied.

Consistency rules, on the other hand, ensure that a design object is consistent with the objects it is related to in the archive. Most frequently, consistency is defined in terms of equivalence. For example, a consistency rule could state that an ALU functional representation is consistent with an ALU schematic representation if their simulated behaviors are equivalent.

Ideally, consistency rules would require that an object is shown to be consistent with the entire archive. However, all that can be achieved is certain limited consistency checks between the object and some subset of previously archived objects. This subset is its *proof set*, and consists of those objects directly related to it through Version Server supported relationships. If the consistency checks succeed, the object is assumed to be consistent with the entire archive by transitivity. Since most consistency constraints are associated with equivalence relationships, which are binary (i.e., most verification tools compare at most two representations), the proof set consists of the equivalent object.

The reason for assuming that consistency is transitive is based on real world limitations in performing consistency checks. The consistency requirements between a given object and *all* other archived objects are too complex to be represented as rules, or verified in practise. An all-encompassing set of consistency requirements would involve project objectives and design policy decisions that are not easily expressible and may change with time. Such a set of consistency requirements may not be completely understood even by designers.

Note that if there are multiple completeness and consistency rules, an object would need to satisfy only one of the many completeness rules, and only one of the many consistency rules to be considered valid. Consequently, the validation rules are *acceptance* rather than *rejection* rules -- they specify conditions under which objects are considered valid rather than conditions under which they are invalid.

5. Implementation of the Validation Subsystem

5.1 Using Prolog

Prolog was chosen as the language for implementing the Validation Subsystem because of its rule-based structure. A Prolog program consists of rules, facts, and queries. The validation problem maps naturally onto this paradigm, since it consists of rules, a log consisting of program execution facts, and validation queries.

In Prolog, alphanumeric strings beginning with a lower case letter denote constants

(similar to enumerated types in PASCAL), while those beginning with an upper case letter denote variables. In our implementation, objects are identified by the triple (object-name, type, version_number).

Prolog *facts* are written as a constant predicate name optionally followed by any number arguments in parentheses, and terminated by a period. The log contains a history of program executions recorded as Prolog facts. Thus, the log could contain the fact:

```
extractor (alu, layout, 5, alu, schematic, 7, 809).
```

which records that a circuit extractor was executed on ALU[5].layout to obtain ALU[7].schematic. The word "extractor" is the predicate name of this fact, and the other terms (alu, layout, etc.) are arguments of the predicate. The number 809 in the last field is a time stamp indicating when the extraction was completed.

Rules are similar in syntax to facts. For example,

```
consistent_with (Object, schematic, Vs, Object, layout, Vl) :-
        extractor (Object, layout, Vl, Object, schematic, Vs).
```

in which the symbol ":-" is Prolog notation for "if". This rule states that a version of a schematic representation of an object is consistent with a layout version if it was derived from it using an extractor. Note that the extractor is implicitly assumed to be error-free in this rule.

Since we do not use disjunctive rules, we use only the logical "and" operator, denoted in Prolog by a comma between conditions. An example is:

```
consistent_with (L, layout, Vl, S, schematic, Vs) :-
        extractor (L, layout, Vl, Nl, netlist, Vx, Tex),
        netlister (S, schematic, Vs, Ns, netlist, Vy, Tnl),
        net_comparator (Nl, netlist, Vx, Ns, netlist, Vy, ok, Tcom).
```

This rule describes the equivalence between layout and schematic objects in terms of a circuit extracted from layout, netlist generation from schematics, and successful comparison of derived netlists via a comparison tool. Recall that the last field of each condition is a time stamp. Each of the three conditions is pattern matched with facts found in the event log. In this rule object names, version numbers, and time stamps are variables, while types are constants. The conditions are matched in the order in which they appear in the rule.

Data used to determine whether or not conditions are satisfied are gathered from the log. A condition is assumed not to be satisfied unless proven otherwise -- a feature of the closed world assumption in Prolog.

5.2 Example Event Logs and Methodology Specifications

The log contains a collection of time-stamped Prolog facts that correspond to verification events. An example log follows:

```
created_functional (alu, functional, 1, 100).
created_functional (alu, functional, 2, 105).
created_layout (alu, layout, 1, 110).
created_schematic (alu, schematic, 1, 120).
extractor (alu, layout, 1, alu, netlist, 1, ok, 130).
simulated_functional (alu, functional, 2, alu, testdata, 1, ok, 135).
simulated_logic (alu, schematic, 1, alu, testdata, 1, fail, 140).
created_schematic (alu, schematic, 2, 150).
simulated_logic (alu, schematic, 2, alu, testdata, 1, ok, 155).
netlister (alu, schematic, 2, alu, netlist, 2, ok, 160).
comparator (alu, netlist, 1, alu, netlist, 2, ok, 165).
design-rule-checker (alu, layout, 1, ok, 180).
timing_verify (alu, netlist, 1, ok, 185).
```

Event argument lists are position dependent, so the methodology specification and the tools that write into the event log must be closely coordinated.

A typical (abbreviated) design methodology rule file is shown below:

```
consistent_with (L, layout, Vx, N, netlist, Vy) :-
        extractor (L, layout, Vx, N, netlist, Vy, ok, TSx).

consistent_with (L, layout, Vx, N, netlist, Vy) :-
        extractor (L, layout, Vx, Nz, netlist, Vz, ok, TSx),
        comparator (Nz, netlist, Vz, N, netlist, Vy, ok, TSy),
        precedes (TSx, TSy).

consistent_with (L, layout, Vx, N, netlist, Vy) :-
        extractor (L, layout, Vx, Nz, netlist, Vz, ok, TSx),
        comparator (N, netlist, Vy, Nz, netlist, Vz, ok, TSy),
        precedes (TSx, TSy).

consistent_with (S, schematic, Vx, N, netlist, Vy) :-
        netlister (S, schematic, Vx, N, netlist, Vy, ok, TSx).

consistent_with (S, schematic, Vx, F, functional, Vy) :-
        simulated_logic (S, schematic, Vx, X, testdata, Vz, ok, TSx),
        simulated_functional (F, functional, Vy, X, testdata, Vz, ok, TSy).

complete (L, layout, Vx) :-
        created_layout (L, layout, Vx, TSx),
        design_rule_checker (L, layout, Vx, ok, TSy).

complete (N, netlist, Vx) :-
        timing_verifier (N, netlist, Vx, ok, TSx).
```

The rules specify conditions sufficient for establishing the completion or consistency status of objects. More than one rule may be specified for the same types. Thus a netlist representation is consistent with a layout if it has been extracted from it. But they are also consistent if there exists some netlist object extracted from the layout that has been shown to be equivalent to the netlist in question by use of comparator tool.

Consistency and completeness rules can be specified in a sequenced or unsequenced fashion. Sequenced rules specify the order (in time) in which the conditions must be met. To enforce sequencing, the corresponding log entries must contain time stamps. The sequencing is specified via "precedes" predicates as part of the rule. Without such predicates, the clauses of the rule can be satisfied in any order.

5.3 Sample Execution

We have developed an interactive tool that allows designers to query the verification status of their design. It support three commands: **complete**, **validate**, and **status**. **Complete** checks for satisfaction of the completeness rules, while **validate** checks the consistency rules. **Status** reports on which clauses within rules have or have not been satisfied so far. A transcript for the design methodology and event log shown above follows:

```
Welcome to the Validation Subsystem -- Version 1.0

: complete (alu, layout, 1)?
Object complete by rule:
        complete(F, layout, Vx) :-
        created_layout (L, layout, Vx, TSx),
        design_rule_checker(L, layout, Vx, ok, TSy).
**yes

: validate (alu, schematic, 2, alu, functional, 2)?
Objects are consistent by rule:
        consistent_with(S, schematic, Vx, F, functional, Vy) :-
        simulated_logic(S, schematic, Vx, X, testdata, Vz, ok, TSx),
        simulated_functional(F, functional, Vy, X, testdata, Vz, ok, TSy).
**yes

: status (alu, schematic, 1, alu, functional, 2)?
Checking consistency status according to rule:
        consistent_with(S, schematic, Vx, F, functional, Vy) :-
        simulated_logic(S, schematic, Vx, X, testdata, Vz, ok, TSx),
        simulated_functional (F, functional, Vy, X, testdata, Vz, ok, TSy).

Clause status:
        simulated_logic(alu, schematic, 1, alu, testdata, 1, ok, TSx)
        NOT satisfied
        simulated_functional(alu, functional, 2, alu, testdata, 1, ok, 155)
```

```
        IS satisfied
**yes

: status (alu, schematic, 2, alu, functional, 1)?
Checking consistency status according to rule:
        consistent_with(S, schematic, Vx, F, functional, Vy) :-
        simulated_logic(S, schematic, Vx, X, testdata, Vz, ok, TSx),
        simulated_functional (F, functional, Vy, X, testdata, Vz, ok, TSy).

Clause status:
        simulated_logic(alu, schematic, 2, alu, testdata, 1, ok, 155)
        IS satisfied
        simulated_functional(alu, functional, 1, alu, testdata, 1, ok, TSy)
        NOT satisfied
** yes
```

Complete and **validate** report the actual Prolog rule used to verify the completeness or consistency of the argument. **Status** indicates which clauses of a rule have been satisfied and which have not, and prints the clauses with the arguments substituted where possible. For example, the first status check fails because alu[1].schematic was never simulated, while the second fails because alu[1].functional was never simulated. Fortunately, version 2 of both of these objects have been shown to be equivalent, as indicated by the success of the **validate** check.

6. Conclusions, Status, and Future Work

We have defined design methodologies in terms of Prolog rules. Verification events are stored as time-stamped facts in a Prolog database. An object cannot be checked back into an archive workspace unless all methodology rules that involve it can be inferred from these facts. If the rules can be satisfied, then the system considers the object to be consistent and to have passed validation. The Validation Subsystem as described here has been implemented as part of a Version Server for computer-aided design data. The system is just becoming operational now, and we hope to be able to report on user experiences in the near future.

There are many directions for further work. Most project managers would prefer a more friendly user interface in which to specify their design methodologies than Prolog! We are interested in developing a graphical interface for methodology specification that would use Prolog as an internal form. The same interface could be used to graphically portray the current verification status of the project.

There is an important performance issue of how to structure the event log. At the moment there is one event log for the system. This could form a bottleneck in a large project that creates many intermediate versions, especially since the methodology rules need to be pattern matched against the whole log. The alternatives are to maintain a log per object or workspace. But since events span objects (i.e., they touch multiple objects), this could lead to significant additional redundancy, since an event entry must be stored in the log for all objects involved. A log per workspace would probably reduce this

additional redundancy, and is still under study. The real issue is limiting the portion of the log that needs to be examined for any given verification attempt. Techniques such as checkpointing remain to be developed. However, the issue may be rendered moot if verification can be performed in the background.

Another issue is one of active versus passive enforncement of design constraints. For example, a rule stating that X is equivalent to Y if Y is extracted from X would fail if the entry "extractor (X, Y, timestamp)" is not found in the event log. Since X is not equivalent to Y, an attempt to check-in X would fail. This is an example of passive enforcement. A more active mechanism would discover that Y has not been extracted from X, and would then proceed to execute the extraction to force the constraint to be true (this is essentially what the UNIX MAKE facility does). It should not be difficult to extend the system described here to also permit active enforcement, if so specified in the design methology.

The final issue is to identify which equivalence constraints an object should inherit on check-out. At the moment, the Version Server allows the designer to specify this at check-out time. Inheritance appears to be related to issues of type within an object-oriented database, although this is an example of inheritance up rather than down the object lattice. We are developing a semantics of type for design objects that should make it possible to define reasonable defaults for equivalence inheritance.

7. Acknowledgements

The Version Server is being implemented by Ellis Chang, David Gedye, and Vony Trijanto. M. Anwarrudin participated in the early stages of the design while a visiting fellow at the U.C. Berkeley CAD/CAM Consortium on leave from Digital Equipment Corporation. Joan Pendleton and Professor Richard Newton provided many stimulating discussions that helped us understand VLSI design methodologies. This research has been supported under National Science Foundation Grants NSF ECS-8403004 and ECS-8352227, with matching support from the National Semiconductor Corporation and Microelectronics and Computer Technology Corporation.

8. References

[BHAT86] Bhateja, R., "A Validation System for Computer-Aided Design", U. C. Berkeley CS Division M.S. Report, (June 1986).

[EAST81] Eastman, C. M., "Database Facilities for Engineering Design," **Proc. IEEE**, **V** 69, **N** 10, (October 1981).

[CLOC84] Clocksin, W. F., and C. S. Mellish, "Programming in Prolog," 2nd Ed., Springer-Verlag Publishing Co., Berlin, 1984.

[KATZ86a] Katz, R. H., E. Chang, M. Anwarrudin, "A Version Server for Computer-Aided Design Data," 23rd ACM/IEEE Design Automation Conference, Las Vegas, NV, (July 1986).

[KATZ86b] Katz, R. H., E. Chang, R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases," ACM SIGMOD Conference, Washington, DC, (May 1986).

[NEWT81] Newton, A. R., "Computer-Aided Design of VLSI Circuits," **Proc. IEEE, V** 69, **N** 10, (October 1981).

[NOON82] Noon, W. A., et. al., "A Design System Approach to Data Integrity," Proc. 19th ACM/IEEE Design Automation Conference, Las Vegas, NV, (June 1982).

[PEND86] Pendleton, J., "A Design Methodology for VLSI Processors," U. C. Berkeley Ph.D. Dissertation, Memo No. UCB/ERL/M85/88 Electronics Research Laboratory, (November 1985).