Managing Change in a Computer-Aided Design Database[1]

R. H. Katz and E. Chang

Computer Science Division
Electrical Engineering and Computer Science Department
University of California, Berkeley
Berkeley, CA 94720

Abstract: Object-oriented concepts can make a design database more *reactive* to changes in its contents. By embedding change semantics in the database model, the design engineer can be relieved of managing the detailed effects of changes. However, mechanisms are needed to limit the scope of change propagation and to unambiguously identify the objects to which propagated changes should apply. We propose new mechanisms, based on group check-in/check-out, browser contexts and paths, configuration constraints, and rules, to support a powerful automatic change capability within a design database.

Key Words and Phrases: Object-oriented data models; Computer-aided design databases; Inheritance; Change propagation; Constraint propagation

1. Introduction

Object-oriented [GOLD83] concepts, as embodied in such systems as Smalltalk-80, LOOPS, and Flavors, are becoming pervasive throughout computer science. They provide an appealing way to structure applications and their data. An emerging consensus is that an object-oriented approach can simplify the applications that create and manipulate computer-aided design data. Several groups are using these concepts to structure a CAD database (e.g., [ATWO85, BATO85, HARR86]), as well as databases for office applications [ZDON84].

The elements of the object-oriented approach appear to include: (i) types (classes), in which operations (methods) on data are packaged with the data itself, (ii) inheritance, in which default procedure definitions and values are propagated from types to instances, and (iii) generic operation invocation, via message passing. For us, the first two concepts are the most important: "object-oriented" means abstract data types with inheritance.

Inheritance provides scoping for data and operation definitions through a taxonomic hierarchy of instances belonging to types, which in turn belong to supertypes (i.e., types of types). If a variable is accessed from an instance, and is not defined there, then its associated type is searched for the definition. If it is

[1]Research supported under N.S.F. grants ECS-8403004 and ECS-8352227, with matching support from the Microelectronics and Computer Technology Corporation.

not defined in the type, the process recurses to the supertype and so on until the root of the lattice. More advanced models allow types to be instances of multiple supertypes (i.e., "mix-ins"), where one supertype's definition of a common variable must be specified to dominate the others.

In this paper, we are particularly interested in how object-oriented concepts can be used to manage change and constraint propagation in a design database. For example, inheritance provides an ability to define default values (for example, in a type) that can be locally overridden (in an instance). It can also be used to determine the constraints that apply to new versions. The goal is to embed change semantics within the database structure, so the system can react to changes automatically.

Most previous work has dealt with change *notification* rather than *propagation.* [NEUM82] defined a transaction model for a database of independent and derived design objects. The database is not consistent until changes to independent objects are propagated to their associated derived objects, although the system does not propagate these itself. [WIED82] proposed a mechanism for flagging records that might be affected by a change in a CODASYL-structured design database. It works by traversing backwards from a changed record, recursively marking its ancestors up to the roots of any hierarchies that contain it. [BATO85b, CHOU86] developed a more sophisticated change notification mechanism within an object-oriented data model. It uses time stamps to limit the range of objects to be flagged in response to a change. A related object is flagged only if it has an older timestamp. In these works, only very limited change propagation is supported. For example, a change propagates from a component to its immediate composite, but no further. There has been little discussion of how to handle ambiguity in the set of propagated changes. We concentrate on these new issues in this paper.

The rest of the paper is organized as follows. Section 2 contains an overview of a version data model, implemented in our prototype Version Server. In Section 3, we describe changes in a computer-aided design databases in terms of default values, change propagation, and constraint propagation. An example demonstrating the interplay among these concepts is given in Section 4. Ways of disambiguating the required set of changes are given in Section 5. Section 6 discusses some implementation issues. Section 7 contains our summary and conclusions.

2. A Version Data Model

The model described in this section has been implemented in a system called the *Version Server* [KATZ86a, KATZ86b]. It manages units of design called *design objects,* which roughly correspond to the named design files found in conventional design environments. Objects are uniquely denoted by *object-name[version#].type.* In addition, the Version Server introduces special structural objects with which to organize these representational objects, much as directories are used to organize files in file systems.

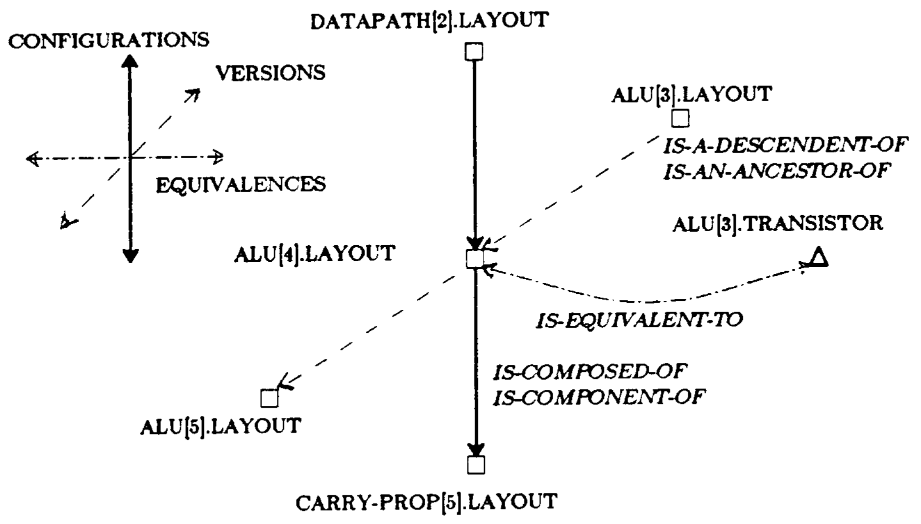

Figure 2.1 -- Version Server Logical Data Model

Design data is organized as a collection of typed and versioned design objects, interrelated by configuration, version, and equivalence relationships. Only representational objects are shown. For example, ALU[4].layout is descended from ALU[3].layout and is the ancestor of ALU[5].layout. It is also a component of DATAPATH[2].layout and is composed of CARRY-PROPAGATE[5].layout. Additionally, ALU[4].layout is equivalent to other objects, such as ALU[3].transistor.

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[4].layout **is-a-descendent-of** ALU[3].layout, both of which are versions of ALU.layout). A structural *version object* is associated with each collection of version instances. Structural *configuration objects* relate composite representational 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. More generally, equivalences can denote arbitrary dependencies among representational objects. They are explicitly represented by structural *equivalence objects.* The Version Server relationships are summarized in Figure 2.1 and the associated data structure is given in Figure 2.2.

Operationally, the Version Server supports a workspace model. Designers *check-out* objects from shared archives into their private workspaces. Changes

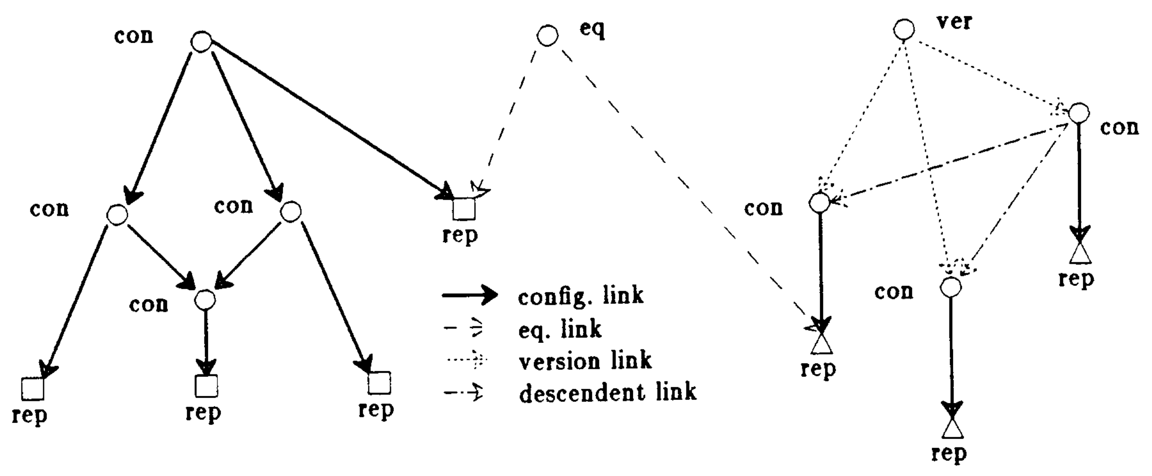

Figure 2.2 -- Version Server Physical Data Structure

Circular objects are structural; square and triangular objects are representational. Relationships are actually implemented by interconnected structural objects, which indirectly reference representation objects. The structural objects are *con*, *ver*, and *eq*, for configuration, version, and equivalence relationships respectively.

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. The Validation Subsystem, invoked on object check-in, analyzes a log of verification events to ensure that the object has been successfully validated before it can be added to the archive. A Browser supports the interactive examination of Version Server databases.

3. Reacting To Changes

3.1. Scope and Ambiguity

Database system implementors have always been reluctant to provide automatic change propagation, because users rarely understand the full (and potentially dangerous) effects of spawned changes. However, aspects of design databases can simplify these problems. Since the design database is append-only, the correct response to change is to spawn new versions of related objects and to incorporate these into new configurations. Note that any new objects are created in private or group workspaces, never in an archive space. Since validation must be performed before these new versions are added to an archive, change propagation can never corrupt the "released" copies that reside there.

However, it is still possible to create a large number of useless intermediate versions. Mechanisms are needed to *limit the scope* of the propagation and *disambiguate* its effects. Ideally, the scope should be limited to the smallest set of objects "directly" affected by the change. Ambiguity is introduced if there is more than one way to incorporate the new versions into configurations. In the worst case, the cross-product of possible configurations could be added to the database, wasting both time and space.

3.2. Default Values

One of the simplest ways to change a design database is by adding to it a new version of an existing object. The designer can fill in its attributes and relationships at its creation time, but it is better if the system can fill these in automatically. Inheritance provides the necessary mechanism. For example, consider the "type" of ALU layouts. All instances of ALU layouts share much in common, perhaps their interface descriptions, or the operations (i.e., add, subtract, etc.) they support. This common data can be factored out of the instances and stored with the type. In creating new instance of the ALU layout, these common attributes can be inherited without being explicitly specified.

However, Smalltalk-style type-instance inheritance provides only one of many possible ways to propagate defaults to a new versions. There are four different kinds of inheritance within the Version Server data model, which can conceivably vary on an instance by instance basis: (i) from version objects to version instances (i.e., the type-instance inheritance mentioned above), (ii) from ancestor to descendent (this is how the Version Server currently operates, since a new version begins as a copy of its ancestor), (iii) from composite object to component (where a version is used within the design hierarchy can determine the value of some of its attributes), and (iv) from equivalent to other equivalents. The propagation of information need not be limited to data; new object instances can inherit constraints, such as equivalence constraints, from their related objects.

3.3. Change Propagation

Once a new version instance is created, it must be incorporated into new configurations to be made part of the design. In most systems (including our prototype Version Server), these new configurations must be laboriously constructed by hand. *Change propagation* is the process that incorporates new versions into configurations automatically. Consider an object A that has been checked out from the archive to create a new version A'. At check-in time, new configuration objects could be created, that form a new configuration of the objects that formerly contained A as a component or subcomponent, but should now contain A'. If the propagations only go up a single level in the configuration hierarchy, then this is essentially the proposal of [CHOU86].

However, it is desirable to propagate changes even further, but this requires additional mechanisms to limit the extent of changes and to keep them

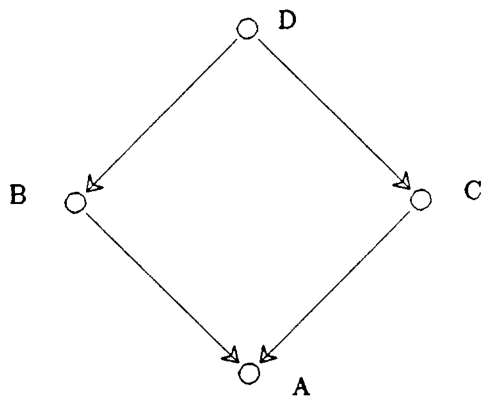

Figure 3.1 -- Configuration Example

Object D is composed of objects B and C. Both B and C use instances of A. Only configuration objects are shown to keep the figures simple. Note that each would contain a reference to a particular representation object.

unambiguous. For example, consider the configuration of Figure 3.1. Object D is configured from objects B and C, which in turn share an object A (only configuration objects are shown). If a new version of object A is created, and changes are naively propagated along both paths, then there are two possible resulting configurations shown in Figure 3.2. This has been called the "multiple path problem" in [MITT86]. Either both paths of changes are merged into a single new configuration of D or two separate new configurations are spawned, one incorporating A' in each of the two original uses of A (i.e., the use of A in B and in C respectively). In general, the former is to be preferred, but there are cases when the latter is what the designer intended. For example, some integrated circuit layout editors support editing in context, which allows a cell to be changed everywhere it is used, or alternatively, just within its current editing context. We will discuss mechanisms to disambiguate changes in Section 5.

3.4. Constraint Propagation

Equivalence relationships model dependency constraints among objects, especially across representations. A new version inherits the equivalences of its ancestor at check-out time. Equivalence is interpreted in terms of the execution of a sequence of CAD verification programs whose success demonstrates that the objects are indeed equivalent. This condition is checked by the Version Server's validation subsystem [BHAT86].

The system currently supports passive enforcement: object check-in fails if any equivalence constraints are left unsatisfied. The obvious extension is to

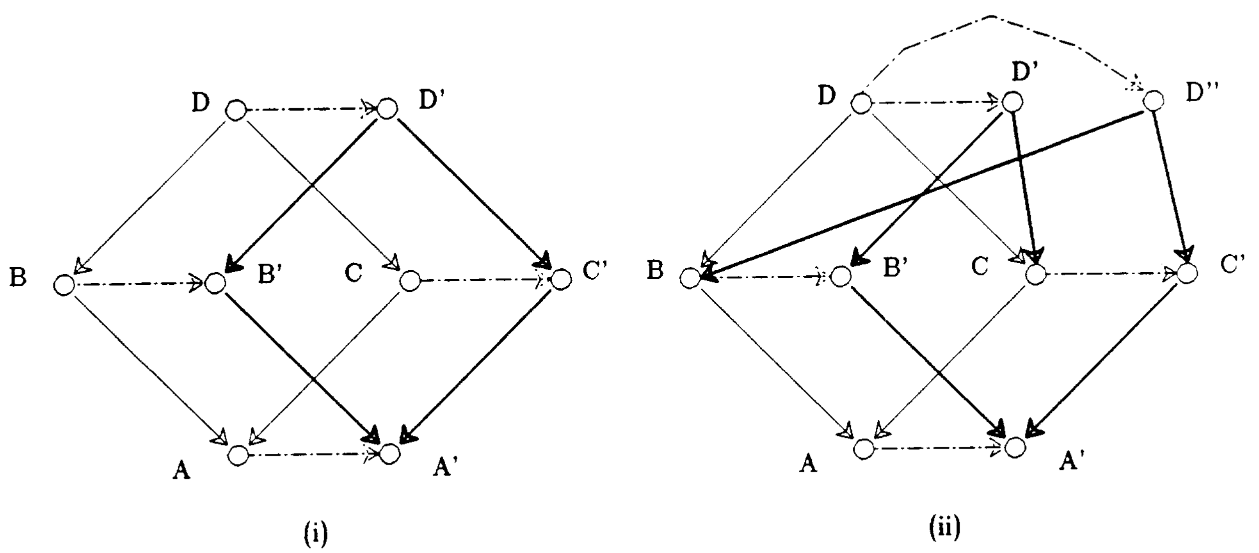

Figure 3.2 -- Ambiguous Change Propagation

A new version of A, A', causes new configurations of B and C to be created through change propagation. Configuration (i) has merged these new objects into a single new configuration of D. Configuration (ii) has evolved separate configurations of D to incorporate the changed components B and C. D' contains the new B and the old C, while D'' contains the new C and the old B. The broken arrows represent descendent linkages.

support *active* enforcement by actually executing the validation script to create a new version of the constrained object. For example, if a schematic object and a netlist object are *actively constrained* to be equivalent, then equivalence is enforced by executing a netlist generator to create a new netlist version when the revised schematic is checked-in. The spawned netlist version becomes a descendent of the original netlist object.

Changes to actively enforced constraints are the only kinds allowed to propagate to configurations of other representations. In other words, if the constraint between objects A and B is passively enforced, then a check-in of a new version of A will not spawn a check-out of B. Of course, the new version of A could not be checked-in unless its equivalence constraint with B had been satisfied.

4. A Detailed Example

We present an example that illustrates the interaction of the three mechanisms described above. Figure 4.1 shows initial configurations of square and triangle objects. Descendent linkages are not shown, to keep the figures uncluttered.

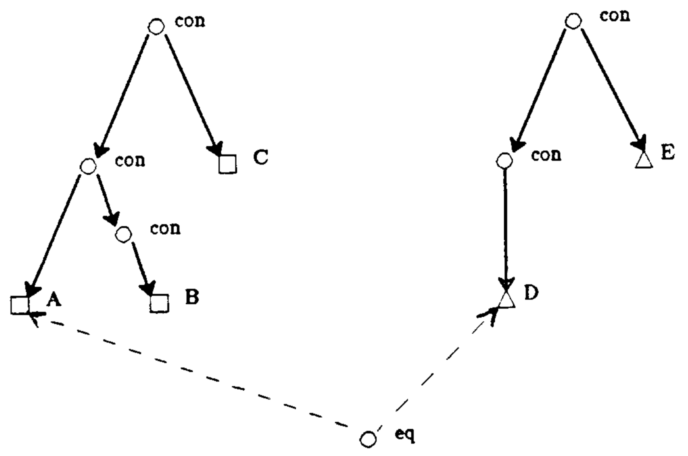

Figure 4.1 -- Initial Conditions

The initial condition has two simple configuration hierarchies, of square and triangle objects respectively, and a single *active* equivalence constraint between them.

The first step is for a designer to check-out A to create a new version A'. A' will *inherit* certain attributes and relationships from its ancestor or objects related to it. Figure 4.2 shows that A' has inherited the equivalence relationship between A and D. We will assume that this relationship represents an active constraint between square objects and triangle objects, i.e., there is a procedure (or set of rules) that describes how to create a new triangle object from a changed square object.

The check-in of the new version A' causes *change propagation*. New configuration objects are spawned to incorporate A' and affect composites up through the root of the configuration hierarchy. The effects are shown in Figure 4.3. Since the constraint between A' and D is actively enforced, a new version of D, D', must be generated. The equivalence relationship is modified to reference this new version. The result is given in Figure 4.4. Finally change propagation must be performed for the configurations that contain D. This is shown in Figure 4.5.

In addition to the multiple path problem already described, the interaction between change and constraint propagation can result in ambiguous configurations. Consider what would happen if object A and E are both checked out for update. If A' and E' are checked back separately, two new triangle configurations will be created: one containing the old D and the new E' and another containing the new D' and the old E. Note that the final configuration is independent of the order of the check-ins. However, if A' and E' are checked in

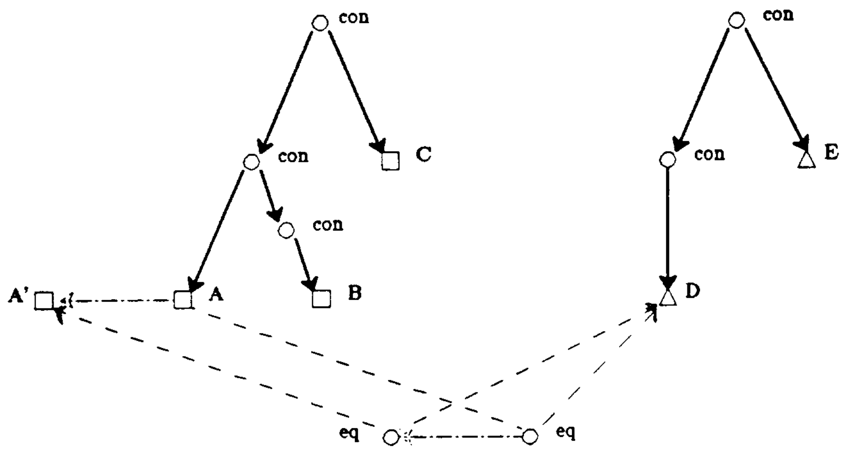

Figure 4.2 -- New Version of Object A

A new version of A is created, A'. Note how A' *inherits* the equivalence constraint from its version ancestor.

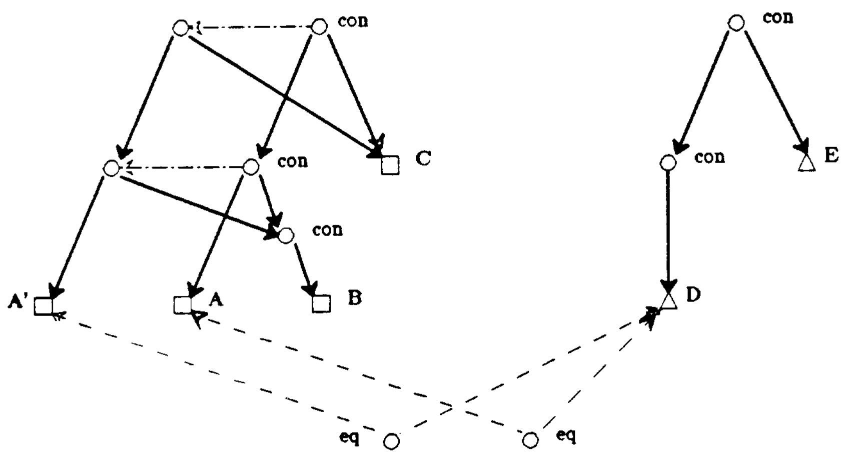

Figure 4.3 -- New Configuration Incorporating A'

Change propagation is realized by spawning new configuration objects upwards towards the root of the square configuration hierarchy. Note that only new configuration objects are created. Existing representational objects are not modified.

as a group, then a single new configuration containing both D' and E' should be made. Group check-in as a method for disambiguating changes will be discussed in Section 5.2.

5. Handling Ambiguity

5.1. Introduction

The system has several options when faced with ambiguity: (1) do not propagate changes if there is any ambiguity, (2) create the cross product of all possible unambiguous configurations, (3) only perform change propagation for the subset that is unambiguous, or (4) provide the designer with the appropriate operational mechanisms to unambiguously describe the effect s/he desires; if that fails, use the browser interface to disambiguate the changes. Choice (1) is the way most systems are built today: they do not support any change propagation. Choice (2) is not really a solution, although some systems have essentially proposed this method [ATWO85]. The systems that do support change propagation usually make the third choice (e.g., [CHOU86]).

In this section, we examine the possible mechanisms for the last choice. Rather than propose a single general purpose approach, we concentrate on more specific "user-oriented" mechanisms. The idea is to provide change propagation effects that make sense to the designers who will be using the system. These may

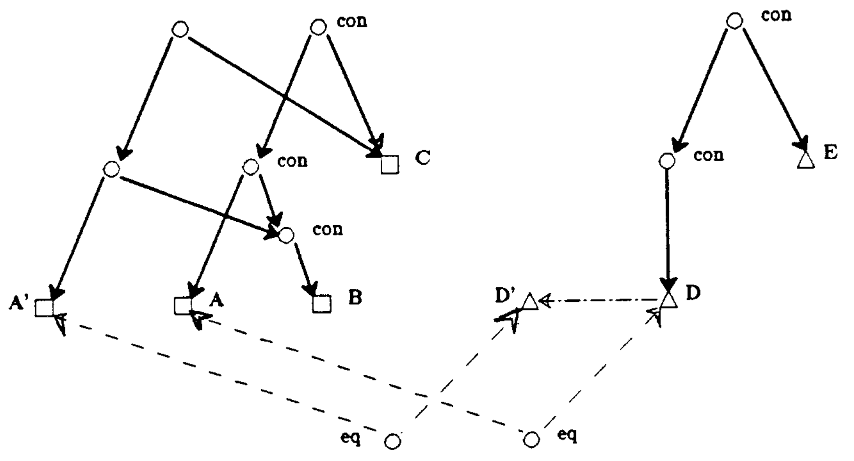

Figure 4.4 -- Constraint Propagation to D

Since the equivalence constraint is *actively* enforced, a new version of D must be spawned. Note how the equivalence object now points to D'.

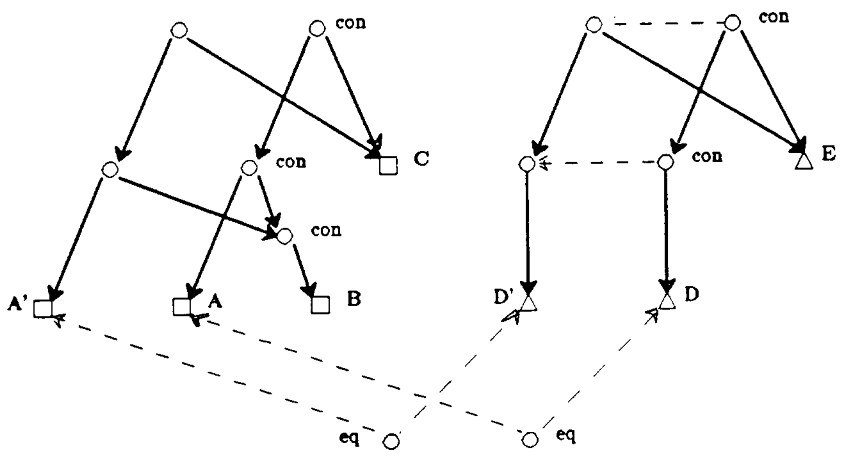

Figure 4.5 -- New Configuration Incorporating D'

Finally, change propagation is once again invoked to spawn new configurations incorporating D'.

be implemented on top of the same underlying general purpose mechanisms, for example, the events and triggers of [DITT84].

5.2. Group Check-in/Check-out

When a single object is checked-out to create a new version, it automatically inherits the equivalence relationships of its ancestor, unless explicitly overridden by the designer. However, consider the situation in which a layout is constrained to be equivalent to a given schematic, and a major design change is underway that will affect both. There is no reason to constrain the new layout to be equivalent to the original schematic, and similarly for the old layout and the new schematic. The desired semantics are provided by *group check-out*. Constraints that range over objects solely within the group lead to new constraints that are limited to the checked-out versions of those objects. Constraints with objects outside the group are inherited in the usual way. Thus, a group check-out of the layout and schematic objects would yield an equivalence constraint between the new versions of the layout and schematic, but no constraints would exist between them and the original versions in the archive.

Group check-in is like a transaction, in that the objects in the group should be added to the database as an atomic unit. In effect, any spawned configurations should merge changes from all members of the group, rather than create new configurations for each. In other words, no more than one new version of a configuration object is created during group check-in, no matter how many change paths touch the configuration it is derived from. The difference between group and conventional check-ins is shown in Figure 5.1.

As long as there is at most one new version of each representation object being checked-in, group check-in is guaranteed to result in an unambiguous final configuration. The sketch of the proof is as follows. At most one new instance can be added for each existing object, either as the result of change propagation (i.e., a new configuration object is spawned) or constraint propagation (i.e., a new representation object version is created as the result of an actively enforced equivalence constraint). It follows that the arcs out of new configuration instances can change at most once. There are two cases. At the time a new configuration object is first created, its arcs either point at old objects that may be superceded later or they already point at the new instances. In the first case, the arcs should change once the new instances are created. In the second case, they need never change during the duration of the check-in. Since each arc changes at most once, the order in which new instances are generated is irrelevant. The same final configuration is obtained.

5.3. Configuration Constraints

A mechanism for limiting change propagation is *configuration constraints*. Several kinds are possible: (1) dependency status constraints, (2) timestamp constraints, (3) interface constraints, and (4) containment and partitioning constraints. This list is meant to illustrate the kinds of constraints that are

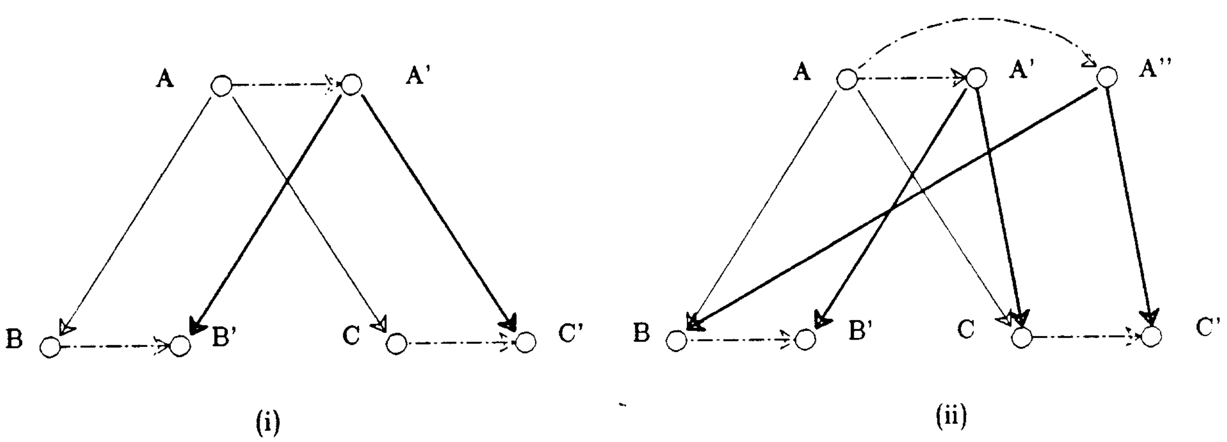

Figure 5.1 -- Group vs. Non-Group Check-ins

Configuration (i) illustrates the result of Check-in (B', C'). Configuration (ii) shows what happens when Check-in (B') is followed by Check-in (C').

reasonable to associate with configuration objects. It is not meant to be exhaustive. Of course it possible to associate more than one such constraint with a configuration object. We will discuss each kind in turn.

The simplest kind of configuration constraint relies on a simple status attribute associated with each configuration object. The value of this attribute can either be ***dependent*** or ***independent***, not unlike independent and derived representations in [NEUM82]. A dependent configuration is one that cannot exist outside another configuration. Change propagation proceeds through dependent configurations, stopping at the first independent configuration it encounters. The designers must incorporate new independent configurations into higher level composites by hand. The first instance of a configuration defaults to being independent, unless explicitly overridden by the designer. Spawned instances can inherit the value of their dependency status in any of the ways described in Section 3.2. Unless changed by the designers, the default inheritance is from their immediate ancestor configurations.

A second kind of constraint depends on timestamps to limit change propagation, as in [BATO85b, CHOU86]. Objects have timestamps that indicate the time at which they were last updated. A configuration's timestamp is inherited from its associated representation object, and is not related to the time at which it was created. A configuration is ***timestamp consistent*** if its timestamp is newer than any of its components, i.e., the composite representation object was last updated after any of its component representation objects. Change propagation proceeds as long as it creates new configurations that are timestamp consistent. This mechanism is well-suited to constructing a valid configuration from a check-in group. However, it explicitly disallows the replacement of a component by a new version within an existing configuration, since the timestamp of the composite representation object will be older than the new version.

Interface constraints depend on the internal details of representation objects. We say that the interface of a new version is *compatible* with its ancestor's interface if it is possible to replace the ancestor in any existing configuration with the new version. The easiest way to ensure compatibility is for the designers to guarantee that the interface portion of the object has not changed across versions. This may be overly conservative since minor changes may not result in an incompatible interface. One can imagine representation-dependent programs that could determine the compatibility of a new version's interface. Change propagation stops when it would attempt to create a configuration from a version whose interface is incompatible with its ancestor.

The last constraint is *representation-dependent containment*. In general, a composite object's configuration is consistent if its components are properly contained within it. This is easy to verify if the representation type defines intersection operations: the intersection of each component with the composite should be the component itself. For example, consider a design type that associates a bounding box with each object, and defines an intersection operation on that box. If a component's bounding box is not properly contained within its associated composite, then the configuration is inconsistent. A related concept is a *partitioning constraint*, i.e., the pairwise intersection of each component is the empty set. Change can be propagated up the configuration hierarchy as long as these constraints are satisfied, and stopped as soon as a violation is detected.

5.4. Browser Paths and Contexts

The propagation of changes is unambiguous as long as each node along the path to the root of the configuration is not referenced from more than one place. When a node is used more than once, as for A in Figure 3.1, then information from the context in which the original was checked-out can be used to disambiguate the change propagation. *Check-in (A')* would result in configuration (i) of Figure 3.2. If A were checked-out along the path D-->B-->A, then a *Check-in (A') along check-out path* would create the configuration rooted at D' in configuration (ii). If the check-out had been along the path D-->C-->A, then the configuration rooted at D'' would be the result. A designer can specify a path explicitly or s/he can select it graphically by using the browser to choose the appropriate configuration arcs. Using the latter method, it is possible to check-in an object along multiple paths. This is particularly useful for objects that are used many places in the design, but the change should be propagated to only a portion of these. An example is shown in Figure 5.2.

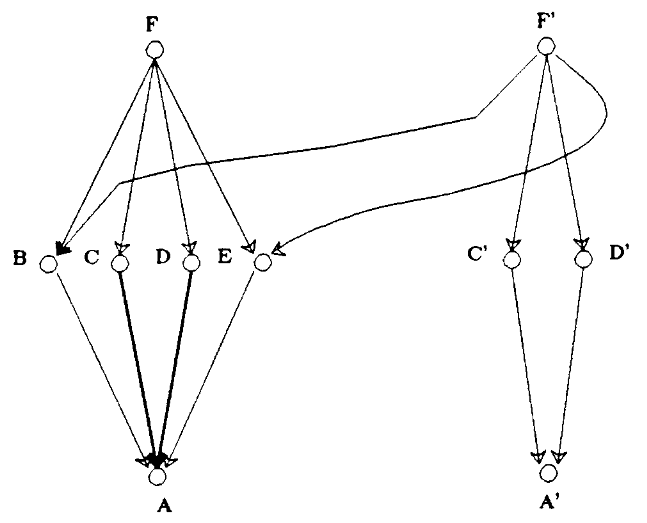

Figure 5.2 -- Check-in Along Selected Paths

Object A is used in four places in the configuration. A new version A' is created, but is to be checked in along the selected paths, which are highlighted (it is possible to select these paths by interacting with the browser prior to check-in). The resulting configuration, rooted at D', is shown on the right. Only the two middle uses of A have been replaced by A'.

Besides disambiguating the path of changes, browser information can limit the scope of changes through the *browser context*. The browser already implements mechanisms for pruning the complex design structure before presentation to the designer. Taking into account the structure of the configuration hierarchy, the browser heuristically determines the neighborhood of interest around an object being browsed. Change propagation can be limited to this neighborhood by issuing a *check-in within context* command.

Figure 4.5 demonstrates that the browsing context by itself is not enough to limit change propagation, because of the effects of constraint propagation across representations. In the figure, the triangle objects need not have been involved in any browser operations involving the original square object A. Mechanisms like configuration constraints must work with contexts to limit the propagation.

5.5. Rule-based Methods

There remains one case in which group check-in does not guarantee an unambiguous result. Consider the example of Section 4, and the case where A and D are checked-out for change, even though D is normally derived from A through an active constraint. A handcrafted version D'' will be created in parallel with the automatically propagated D'. Even if A' and D" are checked-in as a group, the system needs to disambiguate which of D' or D" to incorporate into the new configuration. For example, the system could be given the rule that "a checked-out version always supercedes a spawned version in propagated configurations". Then in the example, D" would be incorporated in new configurations rather than D'. A smart enough system could avoid generating D' altogether.

6. Implementation Issues

6.1. Algorithm for Group Check-in

The semantics of group check-in is that the objects in the group must be merged into a common configuration as the result of the check-in. Note that in Figure 6.1, A' is configured from the old B and C, and the system must build a new configuration if a group check-in of A', C', and D' is issued. Further, if a high-level component and a primitive component are checked-in together, then objects on the path between them must also be contained in any spawned configuration, even if they are not mentioned in the group. The group check-in

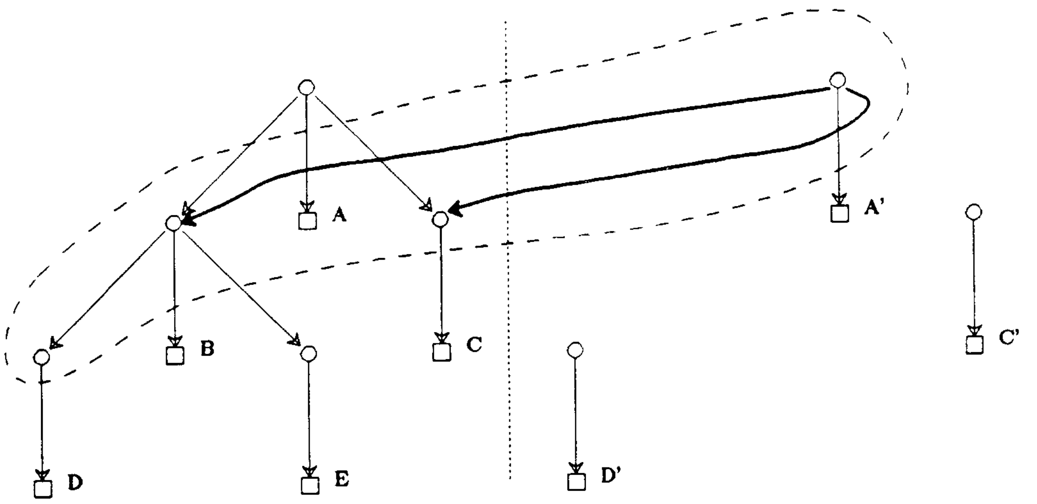

Figure 6.1 -- Before Group Check-in (A', C', D')

The original configuration is shown on the left. Objects A, C, and D have been individually checked out into a workspace on the right, yielding the new versions A', C', and D'. The figure shows the configurations before the execution of *group check-in (A', C', D')*. The minimum spanning configuration subgraph that covers this check-in group is circled in the figure.

algorithm from a source workspace to an archive proceeds as follows:

(1) To keep the discussion simple, we assume each check-in group has a single object that dominates the rest, such as object A' does in the check-in group A', C', and D' in Figure 6.1. This dominating object forms the root of a minimum spanning graph that covers the ancestors of the remaining objects within the group (see Figure 6.2).

(2) Any objects found in this subgraph that do not also have descendents in the check-in list should be marked (e.g., B).

(3) Starting from the configuration of the root object (A'), recursively examine its components and subcomponents: (i) if a node is not marked and does not have a descendent in the check-in list, then the system can ignore the node and any of its components; (ii) if a node is marked, then the system creates a new configuration for it in the source workspace (even though it references a rep object in the archive); (iii) if a node's descendent is in the check-in list, the system will link its descendent's configuration to the configuration being formed in the source workspace.

If there are "holes" in the configurations that need to be filled in this way, then it is likely that the validation subsystem will veto the movement of the resulting configuration into the archive space. However, the full configuration is left in the source workspace, where it can be reverified, and successfully checked-in as a group at a later time.

If the browsing path is specified, the change algorithm is similar to the one above. However, the subgraph would cover only the paths among checked-in objects and the objects specified in the browsing path.

6.2. Algorithm for Group Check-out

The problem of rebuilding the configuration at check-in time can be avoided if the appropriate group of objects was checked-out together. Once again, the appropriate configuration relationships need to be constructed in the target workspace, even if there are "holes" in the group. The basic algorithm is described below:

(1) Determine the root object within the check-out group.

(2) Determine the minimum spanning subgraph that covers the configuration nodes of the checked-out objects and all the paths among them, starting with the configuration of the root object.

(3) Mark all the configuration nodes covered by the subgraph.

(4) Examine the configuration hierarchy, creating configuration objects in the the workspace as follows: (i) if a node is marked, create a configuration object for it; (ii) if a node is also in the check-out list, a new version node is created for it; (iii) if a node is not marked, then it is skipped by the system.

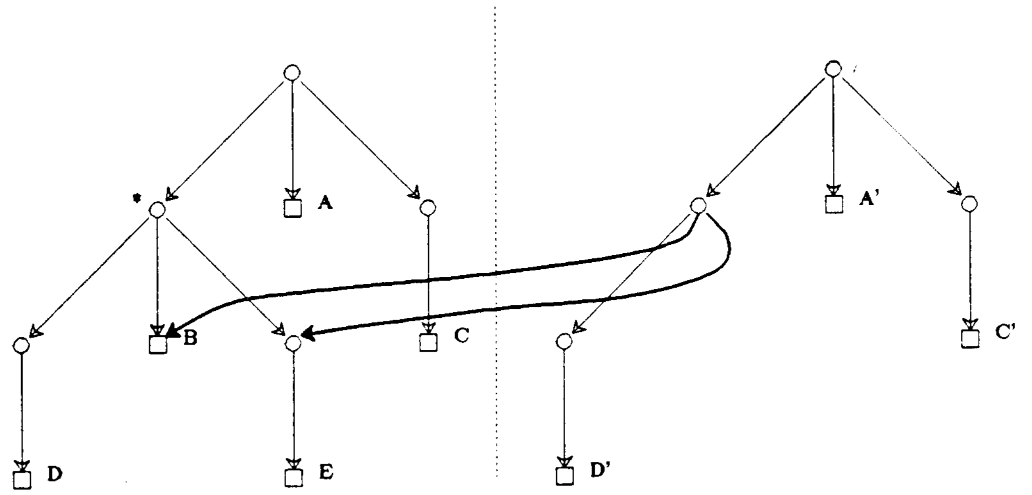

Figure 6.2 -- After Group Check-in (A', C', D')

The configuration node associated with B is marked, because a new configuration node must be introduced into the workspace to complete the configuration. The final configuration is shown on the right.

7. Summary and Conclusions

In this paper, we have described some of the issues in making a design database more adaptive to changes in its structure. Previous work has focused on change notification, i.e., marking objects that might be affected by a change, rather than actually propagating changes automatically. To do so requires new mechanisms for disambiguating what changes are to take place, and for limiting the scope of change propagation. We described specific operational mechanisms that address these issues: group check-in/check-out and browser paths to disambiguate the effects of changes; configuration constraints and browser contexts to limit the scope of these effects. We are implementing these mechanisms in a second edition of the Version Server.

An object-oriented approach helps to limit much of the complexity of change propagation and design evolution. Inheritance makes it possible to identify default values and constraints. By having types with intersection operations, it is possible to support representation-dependent configuration constraints, such as containment and partitioning, without the system needing to know representation details.

We gratefully acknowledge the assistance of Rhajiv Bhateja, David Gedye, and Vony Trijanto, who as members of our research group contributed to the

discussions that led to the work reported here.

8. References

[ATWO85] Atwood, T., "An Object Oriented DBMS for Design Support Applications," Proc. IEEE COMPINT 85, Montreal, Canada, (Sept. 1985).

[BATO85a] Batory, D., W. Kim, "Modeling Concepts for VLSI CAD Objects," ACM Trans. on Database Systems, V 10, N 3, (Sept. 1985).

[BATO85b] Batory, D., W. Kim, "Supporting Versions of VLSI CAD Objects," M.C.C. Technical Report, Austin, TX, (1985).

[BHAT86] Bhateja, R., R. H. Katz, "A Validation Subsystem of a Version Server for Computer-Aided Design Data", submitted to ACM/IEEE 25th Design Automation Conf., Miami, Fl, (June 1987). Also available as UCB CSD Technical Report 87/317, (October 1986).

[CHOU86] Chou, H-T, W. Kim., "A Unifying Framework for Versions Control in a CAD Environment," 12th VLDB, Kyoto, Japan, (August 1986).

[DITT84] Dittrich, K. R., A. M. Kotz, J. M. Mulle, "An Event/Trigger Mechanism to Enforce Complex Consistency Constraints in Design Databases," University of Karlsruhe Technical Report, Karlsruhe West Germany, (November 1984).

[GOLD83] Goldberg, A., D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, (1983).

[KATZ86a] Katz, R. H., E. Chang, R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Database," ACM SIGMOD Conf., Washington, DC, (May 1986).

[KATZ86b] Katz, R. H., E. Chang, M. Anwarrudin, "A Version Server for Computer-Aided Design Databases," ACM/IEEE 24th Design Automation Conf., Las Vegas, NV, (June 1986).

[MITT86] Mittal, S. J., D. G. Bobrow, K. M. Kahn, "Virtual Copies: At the Boundary Between Classes and Instances," Proc. OOPSLA'86 Conference, Portland, OR, (Sept. 1986).

[NEUM82] Neumann, T., C. Hornung, "Consistency and Transactions in CAD Databases," Proc. 8th VLDB, Mexico City, Mexico, (Sept. 1982).

[WIED82] Wiederhold, G., et. al., "A Database Approach to Communication in

VLSI Design", *I.E.E.E. Transactions on Computer-Aided Design*, V CAD-1, N 2, (April 1982).

[ZDON84] Zdonik, S. B., "Object Management System Concepts," Proc. 2nd ACM SIGOA Conference on Office Information Systems, Toronto, Canada, (June 1984).