Papyrus: A Structured History Database for VLSI Design Flow Management

Tzi-cker Chiueh, Randy Katz

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

ABSTRACT

The trend of VLSI CAD tool development evolves towards functionally specialized, methodology-dependent and technology-dependent. Confusion can easily arise during the design process because of the proliferation of CAD tools and the mammoth amounts of design data. This is aggravated by the fact that different CAD tools are usually created by different vendors with different interface styles. As a result, circuit designers have to mentally switch back and forth between various tools in order to get a task done. Previous VLSI design database systems deal only with the static aspect of VLSI design, i.e., the representation and the structure of design data. Almost no attention had been given to the dynamic aspect of VLSI design, i.e., the execution of tools and the derivation of design data. In this paper, we argue why the dynamic aspect of VLSI design plays an important role in building an integrated design management system and propose a structured history model as a unified framework for managing data evolution and tool execution. The paper describes the implementation of a design history database system named Papyrus.

1. Introduction

As the complexity of VLSI circuits increases, so are the design tools that were invented to help circuit designers. Recent trend shows that CAD tools become more and more specialized for specific methodologies and technologies, and are optimized for various design goals (area, performance, power). This results in the proliferation of design tools that are produced by different vendors with distinct design philosophies. Consequently, circuit designers are now overwhelmed not only by the huge amount of design data but also by the complicated CAD tools with which they have to interact. Previous VLSI design database systems deal only with the static aspect of VLSI design, i.e., the representation and the structure of design data. Almost no attention had been given to the dynamic aspect of VLSI design, i.e., the execution of tools and the evolution of design data.

A design flow management system provides a consistent tool interface to users, navigates users through complicated tool invocation procedures, controls and monitors concurrent tool executions, maintains the working contexts for distinct development paths, keeps track of design evolution, and provides automated support for design exploration and cooperative work. A coherent tool invocation interface minimizes the inconveniences caused by a heterogeneous set of CAD tools. Navigating circuit designers through the design tool space by suggesting which tool to invoke next, alleviates the tool proliferation problem. A context mechanism helps a designer to focus his attention only on those design data that are relevant to the on-going activity. It is particularly useful when more than one development path is simultaneously active. The derivation history of a design object offers a dynamic view of the design object. It shows not only what a design object is, but how it was derived as well. Design is exploratory in nature and is almost always a team work. Automated support facilitates exploration and encourages cooperation.

In this paper, a structured design history model is proposed as a unified framework for implementing design flow management facilities. Previous design database systems[4] provides only the most primitive form of design history called version history, in which versions of a design object are organized according to the "is-derived-from" relationship. However, no information is recorded regarding the tool invocation sequence involved in generating a design object. Our history model offers a full-fledged derivation history for each object by recording the tool invocation sequence and the objects involved in creating this object. Version history is static in the sense that only the snapshots of the design objects are recorded. Our history model is dynamic in that both the snapshots and the transformation process are recorded. This paper describes the implementation of a history database called Papyrus that is built on top of the proposed history model and can be best viewed as a meta-CAD system, i.e., a tool that aids users to make better use of CAD tools.

The organization of this paper is as follows. In section two, we discuss why a history facility like Papyrus is useful in the context of VLSI design. A structured design history model is proposed in section three. The implementation of a history facility based on the proposed model is described in detail in section four. Section five concludes the paper with the current status of the system and future research plans.
2. Why History in VLSI Design?

History has long been recognized as a useful tool for supporting programming activity[1]. In software engineering world, there is a tremendous amount of effort directed towards support for UNDO/REDO and replay [2][3]. Considering the evolutionary nature of VLSI design, it seems to be a good idea to build a history facility into the design environment in order to capture the dynamic dimension of design objects. In this section we explain how the history information can be utilized to provide a better support for VLSI design management.

Design in general is exploratory in nature. Circuit designers in particular usually need to explore various design alternatives in order to find the best implementation. A design environment therefore should provide some support for design exploration. In VLSI domain where CAD tools are used extensively, there are generally two kinds of design exploration. The first is operating on the same data set with different procedures(SDDP) and the other is operating on different data with the same procedure(DDSP). Of course, it is conceivable that there are cases which are actually combinations of the two. But let's only consider these two extreme cases. An example of SDDP is applying layout tools of different styles (gate matrix and standard cell) on the same net-list data. An example of DDSP is using the same logic minimizer and PLA generator on different sets of Boolean equations. Support for these two types of design exploration involve historical information. For the former, the system needs to guarantee that two tool execution sequences are exercised against the same database state; for the latter, previous design steps must be "remembered" for later re-use.

Modern window-based workstations encourage multiple operations running concurrently. By associating a context to each development thread, the history manager can separate interleaved design operations and keep track of the progress of individual thread. This ability of allowing concurrency while preserving individual context is especially useful in VLSI domain, where designers need to face an immensely complicated design space. More precisely, there are three distinct object spaces facing the designers: data space, tool space and operation space. The data space is the collection of data objects that a designer created or referenced within a design project. The tool space denotes the set of tools available in the design environment and the relationships among them. The operation space refers to the whole design history a designer has created so far. Every operation represents a binding of data object(s) with CAD tool(s). To help designers cope with this complex design space, a design environment navigates designers through operation space, tool space and data space by showing what has been done, what to do next and what to work on. The design history recorded by the history facility sets up the foundation to provide navigation services.

Incorporating the temporal dimension of design objects into design database also creates a whole new set of structuring mechanisms which are not possible in previous design database systems[4]. In particular, the derivation history of each design object can be matched against a validation script before the object can be checked into archive library[9]. An object's derivation history can also serve as a boundary to limit the extent of change propagation [9] when a new version of the object is created. Moreover, in view of the increasingly higher level of automatic circuit synthesis, design history becomes an important debugging tool for tracking the evolution of a circuit and locating the design defects when the synthesized circuit's performance is unsatisfactory. Design history is needed even when there are no errors involved because human designers may like to know a design is derived before he actually uses it. This is similar to the explanation facilities in most expert systems where human experts will not feel confident in adopting an advice unless he knows how the advice is created, i.e., the reasoning processes involved. Well-documented design history also plays an important role in design expertise transfer.

Figure 1 The Hierarchy of Design History

3. The History Model of Papyrus

3.1. The Hierarchy of Design History

Papyrus features a hierarchical history model. This hierarchy consists of four levels as shown in Figure 1. At the top level, a design process encompasses a designer's entire design history associated with a design project. In our model, there is a design process associated with every member in a project. A designer can be involved in more than one project, and thus can have more than one design process simultaneously.

The second level is design activity, which denotes a portion of a design process that corresponds to a coherent unit in terms of both design operations and design data. Typically a goal is associated with a design activity. All the design operations within an activity are performed to achieve this goal collectively. A simple analog of a design activity is a directory in a file.
system. Only a design activity is more general in that it includes both design operations issued within this activity and the associated design data. Intuitively, an activity provides a "context", which gathers together related data and related operations. For example, a designer can have an ALU activity, which stores all the design operations involved in developing an ALU, together with the design objects referenced and generated along the way. A design process is a set of design activities. This means that the activities within a design process are independent and concurrent.

Within a design activity, there are a set of task invocations, with some partial orderings among them. A task template is a pre-specified tool execution pattern. A task invocation is an instantiation of a task template with proper inputs and parametric options. Task templates are similar in functionality to shell scripts in UNIX system but has a more expressive power. A task template specifies a generic way of accomplishing a task and designers have the flexibility of customizing it by setting preferred parameter values. An activity is a control thread stringing together a set of task invocations. The thread is not necessarily a linear sequence, but can have branching structures. The partial order among task invocations within an activity is mostly determined by temporal relationships, if task invocation A is performed after task invocation B, A is "after" B in that partial order. However, because our model supports the notion of "rework," (explained in section 4.2) the sequence of task invocations can have branches as shown in Figure 2.

Each vertical bar in Figure 2 is a data structure called the history record that corresponds to a task invocation. The numbers on the arrows denote the temporal order of task invocations. Each arrow specifies a partial order between the left and the right task invocations. Those arrows with only left-side history records represent the "frontier" of the activity. Between two task invocations (or history records) defines a design point, which identifies a unique database state associated with the design activity. This state is called the data scope of the design point. A design point's data scope contains the set of design objects referenced and created along the path from the initial design point to itself. A special design point called the current cursor designates the point to which subsequent history records will be appended.

At the bottom of the hierarchy is tool invocation, which is an invocation of a CAD tool with inputs and parameters. The structure of tool invocations within a task invocation is largely predefined in task template, and designers typically have very limited freedom in modifying it. Tool invocations also represent the basic unit of atomic actions. Transaction boundaries are established automatically between tool invocations to ensure failure atomicity and data consistency. In other words, a tool execution either is completed or never takes place.

The structure of the design history reflects several design goals of a design flow management system. The concept of task hides the idiosyncracies of individual tools from users, relieving designers of memorizing the sequence and parameter settings of frequently used tool execution patterns. The concept of activity provides contextual information in the form of related design operations and data. With activities, a design process is no longer a flat collection of task invocations, but a set of designer-organized threads, each presumption corresponding to a concrete design goal. The concept of design process isolates a designer's work from the other. In a sense, it supersedes the idea of workspace and provides a basis for controlled sharing in team design.

3.2. Design Activity And Task Template

One unique feature of our history model is the provision of the activity mechanism. A design activity is NOT just yet another level in the history hierarchy. From users' perspective, an activity forms a context, through which designers can focus their attention on a specific thread of development. In other words, when a designer is in one activity, he/she won't be able to see design operations and data associated with other activities, thereby isolating him/herself from irrelevant data/operations. From system's point of view, an activity represents a unit of activity forms a context, through which designers can focus their attention on a specific thread of development. In other words, when a designer is in one activity, he/she won't be able to see design operations and data associated with other activities, thereby isolating him/herself from irrelevant data/operations. From system's point of view, an activity represents a unit of activity forms a context, through which designers can focus their attention on a specific thread of development. In other words, when a designer is in one activity, he/she won't be able to see design operations and data associated with other activities, thereby isolating him/herself from irrelevant data/operations. From system's point of view, an activity represents a unit of activity forms a context, through which designers can focus their attention on a specific thread of development. In other words, when a designer is in one activity, he/she won't be able to see design operations and data associated with other activities, thereby isolating him/herself from irrelevant data/operations. From system's point of view, an activity represents a unit of activity forms a context, through which designers can focus their attention on a specific thread of development. In other words, when a designer is in one activity, he/she won't be able to see design operations and data associated with other activities, thereby isolating him/herself from irrelevant data/operations.

As mentioned before, a VLSI designer faces three distinct design spaces: Data Space, Tool Space, and Operation Space. The physical contents of a design activity is the composition of three distinct components: activity workspace, activity history and task status. As illustrated in Figure 3, these three components each opens a window into the data space, operation space and tool space, respectively. Together they provide a sense of "context" with respect to the activity in question.

An activity workspace stores the data used or generated within a design activity. The "union" of the activity workspaces associated with a designer forms his private workspace, which is the local, non-sharable database where a designer carries out the design work. As in Version-Server model, we also have group workspace and archive workspace. But we won't pursue them further in this paper. An activity history is a possibly branching thread of history records. Each history record corresponds to a tool invocation or a task invocation and therefore is hierarchical in nature, i.e., a history record can itself consist of a set of other history records with certain structure among them. A history record maintains names of input data, output data, parameters
and possibly other history records. Task status records the progress a user made in the set of currently active task invocations within the same activity. Task status keeps track of the execution status of the concurrent, interleaved task invocations. This information enables the system to provide tool navigation within a task by comparing the task's progress in the task status against the associated task template.

Orthogonal to the contents of a design activity, Papyrus also maintains a derivation history for each design object when it is created. The derivation history of a design object is defined as the sequence of task invocations (restricted to those in the current activity) that are involved in creating the object in question. Those objects that are not created in a design activity is called the foreign objects to the activity. A design object's derivation history is calculated by combining the derivation history of the input objects of the creating task invocation with its corresponding history record. If the input object is a foreign object, its derivation history is regarded as empty. As a result, an object's derivation history is traced back up to the boundary of the creating activity.

The task mechanism offers designers a structured view of tool usage by encapsulating a fixed pattern of tool invocations. A typical task is "create a logic description", which may involve a text editor for entering high-level behavioral descriptions, and a translator to transform that description into an intermediate representation for the logic circuit, and possibly a logic minimizer to optimize the logic expression. The system will navigate through the tool execution sequence specified in the task templates. In contrast, the system does not provide any guidance between the task invocations within a design activity. In other words, users have to decide which task to execute next. The language used to describe task templates allows specifications of parallel tool invocations and arbitrary "restart" points when individual tool execution is aborted for some reason. There are three reasons why pre-specifying task templates is desirable in VLSI design. First, tasks help to automate some of the design steps which can be mechanized. If some of the steps involve human intervention, the specifications in task templates allows the system to offer designers a form of navigation through the tool space. Second, tasks encapsulate the details of the underlying environment and therefore increase flexibility for adding tools to and deleting tools from the environment. Only the design project manager needs to know which tools are available and how they interact with others. Normal designers simply invoke tasks. Third, because designers are required to step through the tool space as specified in task templates, tasks provide a mechanism for enforcing certain design methodologies or design styles, which are codified as sequences and choices of tools.

Our model also provides control operations to manipulate activities and to change the ordering among task invocations within a design activity. A task specification language that has LISP-like syntax and is expressive enough to describe most of the high-level tasks in our VLSI design environment, together with its compiler is also developed. Due to space constraints, the details are left out and can be found in [5] [8].

4. Implementation of Papyrus

Papyrus consists of two parts: a task manager and a history manager. The task manager is responsible for invoking CAD tools, packaging input/output files, navigating designers through the task templates, and handling exceptional situations. The history manager maintains the structure of design activities, computes the data scope of the current cursor for the next task invocation, and prunes or archives "inactive" history elements.

4.1. Task Management

Users invoke tasks and interact with the task manager to run CAD tools. The task manager helps users by leading them step by step through the pre-defined tool execution sequence, prompting for unspecified inputs/outputs, presenting interactive assistance for setting parameters. When something wrong takes place during a tool execution, the task manager will restart the task execution from a pre-specified tool (not necessarily the first tool in that task).

A task template specifies a fixed pattern of tool invocations. Users are committed to follow the specified sequence once invoking a task. The task manager monitors the execution of active task invocations by comparing the progress within a task against the corresponding task template. The task status, of an activity records the progress of currently active tasks. Since more than one task can be active within an activity, and more than one tool can be active within a task, the task status must keep track of multiple threads of tool executions.

For example, Figure 4 shows a general structure of a task template. Remember a task can have subtasks, which in turn can have hierarchical structures. In this example, subtask 3 is a single tool while subtask 2 consists of a linear sequence of tools. Suppose the user is invoking tool 2.3.2 of subtask 2.3 of subtask 2. When the task manager detects that 2.3.2 is completed, it will tell the user that 2.3.3 is the next tool to invoke. More than one
subtask invocation can be active simultaneously, for example, subtasks 2 and 5. The shaded invocations in Figure 4 represent the currently running executions, which are the "frontier" of progress within a task. Whenever a tool (or subtask) completes, the system steps to the next tool (or subtask) according to the specification and advances the "frontier." A hierarchical subtask is considered completed when all its son subtasks are completed. This is why in Figure 4, subtask 2.3 is shaded when 2.3.2 is shaded, and in turn subtask 2 is shaded because 2.3 is shaded. The task manager also checks whether the users are following the execution patterns. Warning messages will be raised once the task manager found a mis-match. This run-time semantic check is essential because the task mechanism can be used to enforce design methodologies, which require designers to follow certain high-level rules during design development.

The task manager basically does the following things:

[1] Keep track of the status of each active task invocation, and compare the progress with the corresponding task template to provide navigation service or issue violation error messages in the case of mismatches. The task manager also maintains the history records for the completed tool invocations within active tasks.

[2] Manage the data correspondence within a task. That is, after a user invokes a tool, all the input and output mappings of subtasks within that task are under the control of the task manager. Designers don’t need to name the intermediate results. This could be viewed as a generalized notion of "pipe" in UNIX because the task specification language allows non-linear structure among subtasks as opposed to a linear "pipeline."

[3] When a tool invocation aborts, the entire task is reinitiated with a pre-specified tool that is called the restart entry of the aborted tool. Each tool within a task can have its own restart entry. By default, the restart entry of a tool is the initial tool within the same task. Suppose the subtask 4 in Figure 4 is aborted and its restart entry is subtask 2, the task manager will re-invoke the path from subtask 2 to subtask 4, but leaves other paths undisturbed. This facility allows a more flexible way of specifying task-specific abort semantics. Instead of restarting the task all over, users can explicitly pre-specify the roll-back point for each tool when it aborts, thus preserving useful intermediate results for long-running task invocations.

[4] When a task execution is completed, the task manager packages up the history record associated with the task invocation and passes it to the history manager, thus committing the effects of the task invocation.

4.2. History Management

At the design activity level, the system provides services for computing the data scope for the current cursor, manipulative operations for combining or forking design activities, maintaining the structure of the activity history and activity workspace, and periodic archiving of the history elements.

4.2.1. The Computation of Data Scope

Rework means restoring the database to a previous state and exploring another design alternative from there. A typical application is: after a designer finishes a part of design with the gate-array design methodology, he wants to explore the standard-cell methodology on the same inputs. In this case, restoring the database state to what it was before the gate-array approach is most appropriate. The notion of "rework" is similar to "UNDO/REDO" notion in interactive systems. But there are two fundamental differences. First, our data model assumes that design objects are write-once, read-many, whereas other interactive systems support modifiable objects. In other words, modification is done on the copy of an object rather than the object itself. In interactive systems, they define the state of the system as an ordered sequence of operations. More precisely, the current state is the result of applying the ordered sequence of operations on the initial state. In contrast, we define the activity state associated a design point as the collection of objects referenced (used as inputs or generated as outputs) from the start of the activity up to that design point. The second difference is concerned with the mechanism of specifying UNDO/REDO operations. Papyrus offers a more flexible way of specifying UNDO/REDO intentions than most interactive systems by providing a browsing capability for the design history. Designers can literally roll back to any previous design point by repositioning the current cursor. Furthermore, users can choose whether to erase the intermediate effects between the new and the old current cursors.

The data scope of the current cursor defines what is visible to a designer when s/he invokes the next task. More precisely, if the designers do not explicitly specify the version of a data
object to operate on, the system assumes the default version of that data object in the data scope of the current cursor. Therefore, moving the current cursor to a design point limits the designer's visibility to the data scope associated with that design point. Changing the current cursor to an intended design point automatically restores the database state to the data scope of that design point. This is exactly what is needed for rework! As a result specifying the rework point is nothing but moving the current cursor to the desired design point.

This simple operation model is in contrast to what occurs in interactive systems, where a bunch of UNDO/REDO operations have to be issued to get the desired effect. One can also view this browsing/cursor-positioning capability as a way to query a history database. That is, instead of specifying absolute temporal coordinates (date/time) in textual form, a flexible and graphical interaction approach is adopted. Our operation model also offers a novel way of specifying versions of an object. Namely, instead of looking for the right versions of data objects, designers simply look for the right design point. The system will take care of the design-point/data-scope mapping automatically. We believe this is an important step in helping designers to explore design space.

Because the activity state associated with a design point is defined as the collection of objects referenced up to the current cursor and the objects in the system are write-once, it naturally follows that the activity state is monotonically increasing as the activity evolves. The support for rework thus boils down in associating the corresponding data scope with every design point in an activity. Formally, the association can be described as a mapping \( F \) from the set \( D \) of design points in an activity to the set of all subsets in the activity workspace \( W \), \( F: D \rightarrow W \).

If the design history within an activity is a linear sequence, the data scope of a design point can be determined simply by tracing back all the data objects associated with the history records that are ahead of that design point, up to the start of the enclosing activity. It becomes complicated when there are branching or merging structures in the design activity. Let's define the set of objects referenced (used as inputs or generated as outputs) within a task invocation as the reference set \( \text{Ref}(H) \) of the corresponding history record \( H \). In Figure 5, the data scopes of design point 1, 2 and 3 can be calculated by definition; they are \{1\}, \text{Ref}(HR1), \text{Ref}(HR1) \cup \text{Ref}(HR2). As for the design point 4\( (5,6) \), the above definition is ambiguous because there is no causal relationship between the branches 4-5-6 and 7-8. We define the data scope of design point 4 \( (7) \) as the data objects associated with task invocations 1, 2 and 4 \( (7) \). The rationale of this definition is to emphasize the notion of "context." Here we treat task invocation sequences 1-2-3-4-5-6 and 1-2-3-7-8 as separate design paths, therefore the associated data scopes overlap with each other only in the part of data space that is associated with the common invocation subsequence, which in this case is 1-2. As for the design point 9, its data scope is the union of the two branches, i.e., the set of data objects associated with task invocations from 1 to 9.

The data structure for the activity history is a doubly linked list. To map a design point to its associated data scope, we can either compute and store the data scope for each design point, or we can compute the data scope on demand incrementally. The lazy approach is chosen for two reasons. First, not every design point will be used as a rework point; so storing every design point's data scope is not worthwhile. Second, since most activity histories exhibit a linear structure, there are a lot of redundancy in the data scopes of adjacent design points. From storage efficiency standpoint, it would be better off to compute the data scopes for those design points on the fly. Computing the data scope is a simple backward traversal of the corresponding history records from the start of the activity to the rework point and pick up the associated data objects along the path. To avoid unnecessary re-computation and improve performance, the history manager caches the data scope of intermittent design points periodically. Therefore the traversal can stop when it hits one of these intermittent design points, thus limiting the data scope computation to a fixed overhead. The data scope of the current cursor is passed to the task manager for invoking the next task.

4.2.2. Maintenance of Activity History

Maintenance of the design history involves two issues. First, there must be a transaction model that determines when a design operation should be recorded in design history. The goal is to ensure that the states maintained by the history manager and the design database are consistent with each other. The second issue is related to the representation efficiency.

In our model, the basic unit of transaction is a tool invocation. In other words, the execution of a tool is either to the completion or not at all. However, from the history manager's viewpoint the task invocation is the basic recording unit. The packaging of task-invocation history records is done by the task manager. If a task is to be viewed as a hierarchical transaction, it actually has a more flexible abort semantics than conventional nested transactions in that every subtask (corresponding to a subtransaction) can specify a restart point once it is aborted. In other words, the abort semantics can be programmed in advance in order to exploit the structure of the task template and to preserve as much useful work as possible upon abortion.
As mentioned before, users can roll back the activity state by positioning the current cursor to a previous design point. Consequently conflicting considerations arise concerning the representation efficiency. On the one hand, the system should allow users to roll back to any point in the activity history. On the other hand, in order to provide only relevant contextual information to designers and to make efficient use of available storage space, it is necessary to prune away unneeded or useless history elements whenever possible. There are four mechanisms currently implemented to achieve the balance between keeping most relevant design history and supporting "meaningful" rollback. Design points in history that are potential candidates for roll-back are called reworkable. They are design points in the activity history that the history manager allows to be used as branch points.

[1] Hierarchical Aggregation: In the history model, a task template can be hierarchical. Not all the details of a task invocation are interesting to users. It is thus desirable to do some sort of "level compaction" of the hierarchy of invocations within a task. The history manager throws away the history records associated with the low-level subtasks within a task invocation, thereby abstracting away unnecessary details. The intermediate details of the abstracted history elements no longer exist and therefore can not be used as rework points. One can specify the desirable degree of relevance in task templates so that only the corresponding level of details specified are maintained by the task manager. For example, task specifications can specify that only the first two levels of the task are interesting; so the task manager will omit the details below the first two levels and treat them as black boxes. Note, however, the task manager allows these "hints" to be overridden at run-time by users for reasons such as debugging. The effect of Hierarchical Aggregation is shown in Figure 6. Here we have a task T, which consists of subtasks T.2 and T.1, which in turn consists of T.1.1, T.1.2 and T.1.3. The double vertical bar denotes composite subtasks, which consists of more primitive subtasks. The single vertical bar denotes indivisible subtasks. After Hierarchical Aggregation (suppose we are only interested in the first two levels), subtask T.1 becomes an indivisible subtask. In other words, the internal details of T.1 become invisible after level compaction.

[2] Filtering: The raw design operation log is filtered to get rid of unsuccessful or "facility" type of tasks. Tool invocations that are not completed successfully will be ignored by the task manager and thus not recorded in the history record. A task invocation is recorded in the activity history only when it is completed. In other words, the history manager defines the order of task invocations in terms of the completion time stamp rather than the start time stamp. Some "facility" tasks such as printing or showing something on the screen have no influence on the design status. Those tasks only provide temporary aids to the development of the design. Since they are not integral parts of the design process, the history manager can bypass the history records associated with these task invocations.

[3] Aging: From users' standpoint, it is generally true that the relevance of history elements to the current design context decreases proportionally to their ages. Outdated history elements should be abstracted away such that only sufficient historical details are preserved. There are basically two ways to exploit the aging mechanisms to reduce the amount of historical information: vertical and horizontal. Vertical aging means level compaction mentioned previously can be performed according to the ages of the task invocations. In particular, the details of past task invocations can be progressively "forgotten" as they get older. For example, in Figure 7 (a), as T1 and T2 become old, their internal details are abstracted as the right hand side of the arrow. Horizontal aging refers to the process of archiving or eliminating the part of design history that are too far back in time. For example, in Figure 7 (b), T1, T2, and T3 are archived or pruned away from the design history completely when they get old. All of these "forgetting" operations (vertical or horizontal) involve throwing away intermediate data set and the associated history records. The history manager "forgets" the history in a safe way by actively reminding users that some part of design history are going to be pruned away. If users permit the pruning operations, the manager then goes ahead and performs checkpoint operations. This active mechanism relieves the users of the burden of archiving while avoiding causing unrecoverable mistakes. Forgotten history elements are no longer visible to designers and may not be used as rework points.

[4] Garbage Collection: The term garbage collection, as used here, refers to a general procedure for finding "abandoned" history elements. There are two kinds of abandoned history elements in our model. In an iterative refinement process, a sequence of task invocations is iterated for several times. Typically only the effect of a single iteration is selected and used later on. The garbage collector aims to abstract the iterative process by a single iteration and eliminate the history elements associated with other iterations. However, the history manager requires users to explicitly specify the sequence of task invocations that are corresponding to an iterative process. For each iterative sequence, the garbage collector finds all iterations (typically only one) whose outputs are used by later task invocations, and prune away all other iterations. The other type of abandoned
history elements is related to dead-end branches. A design activity history can have branching structures that represent alternative development paths. It is possible that some of these branches are no longer needed, representing dead-ends to the corresponding development. The garbage collector recognizes these kinds of branches by keeping a list of frontier branches together with their last activation time, and marks them as dead-ends when the difference between the last activation time and the current time exceeds a certain threshold. Again, the history manager will actively ask for the user’s permission to prune these abandoned history elements.

4.2.3. Replay of History Events

History allows users to re-run part of the past command sequence without explicitly issuing them again. We call such an action as replay. As mentioned before, there are two forms of design explorations: "same data, different procedures" (SDDP) and "different data, same procedure" (DDSP). The "rework" facility aims to supports SDDP while DDSP is where the "replay" facility comes into play. A user can reuse some of the history sequence without re-issuing those commands again. This is a useful facility in iterative refinement process, such as tuning the geometrical parameters of operational amplifiers for circuit simulation. Users specify a replayed history sequence by choosing the REPLAY option and clicking on those task invocations in the activity history. The smallest unit of replay is a task. In addition, users are allowed to change the parameters and inputs associated with the tasks. Therefore users can customize the way the selected history sequence will be replayed.

From implementation's standpoint, the specified replay sequences are nothing but dynamically-composed high-level tasks. That is, the user-specified history sequence are packaged as a task template and passed to the task manager. All the services provided by the task manager can be applied to these sequences. It is also possible to promote some of the history sequences that prove to be useful as permanent tasks and stored in the template database. Later on they can be invoked exactly the same way as pre-specified tasks. This facility of storing some useful history sequence is a handy macro-like facility in circuit design. Moreover it is an important mechanism for transferring the expertise of experienced designers to the entire community, provided that the composed task specifications have been properly verified.

5. Conclusion

The VLSI design management system has seen a transition from a static view of design to a dynamic view of design. Incorporating a history facility into design databases represents the first step towards integrating the management of design tools and design data. In this paper, we have presented a design history model. This model renders a unified framework for providing tool encapsulation and design exploration supports. The detailed implementation algorithms are also described. The facilities of Papyrus as described in this paper have been implemented. We are working on the user interface part of the system. The general system architecture is shown in Figure 9. A human design manager is responsible for composing task specifications through VEM, an interactive graphic editor. He has a simulator or a pre-viewer that can help him to walk through the specifications to make sure that what he specifies is what he means. Normal circuit designers simply invoke tasks. But before he invokes a task, he may like to first take a look at the details of the tasks. This
browsing functionality is provided by the task specification browser. During task invocation, the task manager helps designers to walk through the task and manage data correspondence among subtasks. The history manager maintains the design history and supports rework and replay as discussed in last section. All task specifications and design history are stored in OCT, which is a CAD database developed in U.C. Berkeley.

In terms of future research, there are two possible directions. In order to take full advantage of the history facility, various indexing techniques such as those presented in [6][7] are needed to improve the performance. The indexing algorithms are closely tied to the underlying history model and are currently under investigation. The other research direction is to extend the single-user history model to support cooperative work and project management. So far we are only dealing with single-user design model. It is no doubt that the support for team design support will become more important as the circuits get more complicated. We expect the design history will play an even more important role in this regard because of the structure of design evolution provided by the design history database.

REFERENCE