Protection and Versioning for OCT

Mário Silva*, David Gedye†, Randy Katz, Richard Newton

Electronics Research Laboratory
University of California, Berkeley
Berkeley, CA 94720

Abstract

This paper describes the extensions made for adding support for group development within the Oct/VEM CAD framework. In our implementation, a set of mechanisms has been incorporated into the Oct library. These contain support for versioning and concurrent access to Oct design objects. The mechanisms can be configured for establishment of specific design management styles. As an example, there is now support for organization of designs in terms of workspaces; but, the number of workspaces or the relationship between them must be established externally by a design management tool. Design management tools configure these mechanisms to establish policies to be followed by the other tools. With this architecture, integration of existing tools with distinct design management styles then becomes feasible.

1 INTRODUCTION

We have added new functionality to the Oct [3] data manager with the objective of adding support for group work. To achieve this, some of the ideas of the Version Server [4] were incorporated into the Oct library.

Oct acts as a central repository for design data so it can be shared among various applications. In the Oct data model, designs are organized in terms of a set of *facets*. Facets are aggregations of design primitive objects. These correspond to the design files found in more conventional design environments. In Oct, a facet is referenced by a composite name with four fields, *cell, view, facet type* and *version*. Oct offers a simple mechanism to version facets, as it allows user qualification of facet names through the specification of a version name subfield.

The Version Server provides version and configuration management functions for design teams. It supports *workspaces*, which can be used for individual private and tentative changes or as an archive of shared design objects. In the Version Server, a higher level of versioning is provided: it is possible to group design objects into *configurations* and there is a name resolution mechanism to bind generic references to specific versions.

*on leave from INESC, Lisbon, Portugal
†currently at Sun Microsystems, Inc.

Our goal was to make Oct a more powerful system by embedding into it some of the concepts from the Version Server, namely support for high-level versioning and protection of design objects. In fact, Oct has been designed with the requirement that in the future these concepts would be added.

The simple versioning at the facet level was impractical in large designs, where multiple versions of each individual objects had to be maintained. There was a need for the introduction of *configurations*. Configurations are a higher level of versioning that allows reference of each individual object version, given its generic name and an entire design version name. In terms of protection, we wanted to provide means to specify access restrictions at a larger level of granularity than individual facets and perform automatic access control during concurrent modifications.

As is common in databases supporting concurrency, each user should have the illusion of working in a single-user system. We wanted to find out how to provide the Oct data manager with these new capabilities while maintaining the same application interface.

Moreover, we wanted to pursue the design philosophy of Oct, providing only the mechanisms for design data management with which particular design styles (or *policies*) could be implemented. For instance, we would provide the means to move or copy Oct facets between workspaces, but we would not enforce any specific verification and/or validation suit normally associated with *check-in* and *check-out* operations [1]. These would have to be implemented by a design management tool, invoking the new mechanisms. In fact, we noticed that there are about as many group design management styles as design teams. In some situations, it is even desirable to change the design management policy while the design evolves. As a result, most of the design management tasks are currently performed manually, as systems either do not support them or enforce policies that conflict with accepted design methodologies within existing design teams.

The rest of the paper is organized as follows. In the next section we describe the architecture of our system. In section 3 a description of the implementation is presented. An example of a simple design manager and how it fits into the OCT/VEM framework is presented in section 4. Section 5 discusses some implementation issues. Section 6 presents our conclusions.

2 ARCHITECTURE

Octane is the set of mechanisms we have added to Oct to provide it with the functionality described in the previous section. These mechanisms have been incorporated into the existent Oct library.

Octane performs two basic functions:

- Protection: it provides means to specify access restrictions at a larger level of granularity than individual facets and performs automatic access control during concurrent modifications.
- Version resolution: if the application program does not specify which version of a particular facet it wishes to open Octane will make this choice based on some higher-level version settings.

Before describing the Octane architecture and how it fits in the Oct/VEM CAD framework, we first need to review Oct implementation. We will then describe the components of Octane and how the mechanisms of Octane are configured. We will then present Octane concepts and how they can be used to satisfy the design goals presented in the previous section.

2.1 REVIEW OF OCT INTERNALS

Oct is implemented as a library of routines. The routines stand between the CAD tools and their data stored on disk.

Facets are the Oct objects that map into files. All the other instances of Oct objects are contained by a facet and are stored inside its corresponding file. The process of loading the data from a facet is explicitly activated from the application level by a call to an Oct function to *open* it. Upon opening, the objects contained in a facet are loaded into memory, and become available for manipulation through the defined Oct procedural interface. When a facet is no longer needed, it can be flushed to disk or simply freed, if changes are not to be saved.

The Oct library is conceptually divided into two layers. The one at the top has the application interface and provides the object manipulation routines. The other one contains the functions that interface with the underlying file system. Between these two layers of Oct there is a well defined interface.

Initialization and termination of a session with a facet always implies a call to one of the functions of the file system interface. One of its functions is the translation of facet names into file names in the underlying operating system syntax. In Oct terminology, the process of obtaining a facet's data from a file, given the facet's name and desired access mode, is called *facet name resolution*, or more succinctly *resolve*.

The opening of a facet is made in Oct by invocation of one of two functions: *octOpenFacet* and *octOpenMaster*. The first has the semantics we just described. The second takes two parameters and is used to open the master of an instance, given the instance and the facet that *contains* the instance. In the resolve mechanism of Oct, a facet opened through *octOpenMaster* is searched in the directory where the facet containing the instance was previously resolved, instead of the default directory.

2.2 OCTANE ARCHITECTURE

The relation between Oct and Octane can be seen in figure 1, which depicts the organization of the new library. The Octane part is represented by the shaded area in the figure. Octane is divided in two parts. The *Octane name resolution* part, intercepts the calls to Oct procedures that go into the *Oct name resolution* part. It does facet names translation according to the defaults established through the *Octane services* part functions.

We use the term *design management tools* to refer to those that link against the new Oct library and use the Octane services. The function of design management tools is basically the setting of defaults for the parameters that configure Octane mechanisms and displaying of its values. These defaults are defined in terms of relationships between Oct facets, the managed design objects. We model these relationships with newly defined objects that describe their organization. These latter objects constitute what is called *meta design data* [9]. The meta design data is, like the design data, stored persistently and is shared by all the tools that cooperate for the completion of a given design.

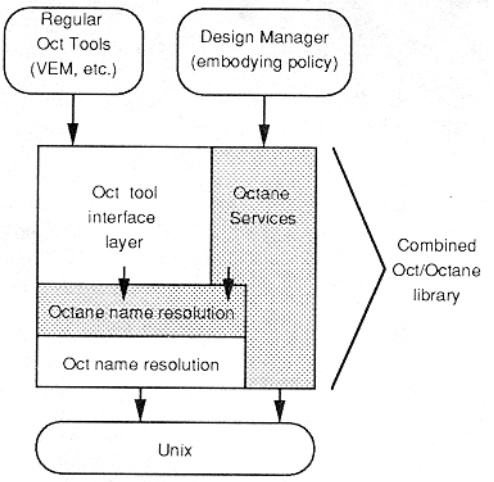

Figure 1: The new Oct library structure, after incorporation of Octane mechanisms

Upon tool initialization the meta design data is loaded to configure the new resolve mechanism. Design management tools impose their management policy by configuring the meta design data. As a result, design tools only have to deal with design tasks, as design management can be accomplished by independent processes. This is illustrated in figure 2.

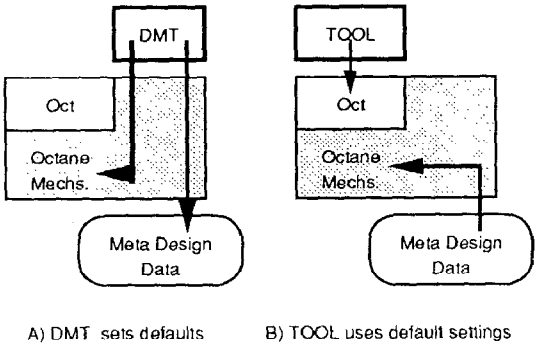

Figure 2: As a result of Octane's architecture, a design management tool (DMT) can configure the mechanisms used by other tools operating as independent processes. From this it also results that changes of design management style would not imply changes in the design tools, even if the DMT had to be replaced.

2.3 OCTANE CONCEPTS

Workspaces

To satisfy our design goals, we introduced into Octane the Version Server [7] concept of *workspace*. In terms of Octane, a workspace is simply a named collection of facets, all

sharing the same protection status. In a design environment each workspace has a unique name. The protection established for a workspace determines access restrictions for all the contained facets. The protection for workspaces follows the Unix file system model: it is possible to grant/deny read and/or write access to the owner, the owner's group and the other. Facets contained in a workspace inherit its protection status. Octane does not allow the specification of access rights to individual facets. To change the protection status of a facet, one has to move it into another workspace.

The criteria for organization of systems in terms of workspaces are left to the design management policy layer. Distribution of design objects into distinct workspaces is a solution to the need to establish levels of protection and/or degrees of verification for design objects. Another possible use for workspaces is as sharable design libraries. Partitioning of design data into workspaces can also be useful to limit the frequency of concurrent access to design data (and meta data) by distinct users. Concurrent access to different objects when they are in the same workspace implies that the workspace meta data is accessed concurrently, and a lock mechanism has to be activated whenever a design object is accessed for modification. One possible policy that could reduce concurrency consists of having a two-level hierarchy of workspaces. *Private* workspaces would contain each user's changes to the design, while the *group* workspace contained approved design data. Contention would then occur only during simultaneous check-in or check-out from the group workspace by distinct users, which is not performed frequently [6,9].

Configurations

In each workspace, facets can be grouped into collections, named *configurations*, for the purpose of versioning the entire workspace. Configuration names are unique in each workspace.

A collection of facets in the same workspace, with the same *cell, view* and *facet-type* names but distinct versions, is called a *facet-family*. The facet-family concept corresponds loosely to a *version history* in the Version Server. It differs because it does not contain ancestor-descendent information. Versioning of facet-families is still performed directly by Oct.

Formally, a configuration is a named subset of the facets in a workspace, with the restriction that no two versions of the same facet-family can appear in the same configuration. It is common however for the same facet to appear in more than one configuration.

Configurations can be used both for checkpointing the state of an entire design (*snapshoting*, in CAD users terminology) and exploring alternative design strategies. A snapshot is a configuration that cannot be changed and is created for later displaying of design status at the moment of its creation. It is the policy layer above Octane that establishes modification criteria for each of the created configurations. In a typical workspace for example, we might find configurations representing "the state of the design on January 10", "the domino logic experiment" and "the chip we fabricated".

In each workspace there is always a configuration defined as the *active* configuration. The active configuration is the single configuration that can be modified. When a new configuration is created, it inherits all the facets that belonged to the active configuration at the time of creation and becomes the new active configuration. It is also possible to *restore* a previously active configuration as the new one. It is this mechanism that permits the exploration of new design strategies and later return back, to re-start changing it from a previous stage. To change the active configuration in a workspace, it is necessary to have write permission over it.

The time interval on which a configuration is active constitutes an *epoch*. An epoch is created whenever a configuration is created or restored in a workspace. Epochs are specified as dates (Octane then finds which epoch contains that date) and are used as alternative ways for specifying configurations in Octane.

To illustrate the use of configurations we present fragments of the code executed during a scenario described in comments. The code would not be executed in one process, but typically in a succession of processes over a period of weeks or months. For clarity some necessary calls were omitted. The evolution of the design status of this example over time is depicted in figure 3

```
/* open the workspace at
 *   current time => current epoch
 *                      => active configuration
 */
octaneWorkspace ws;
octaneOpenWorkspace("spur_cpu", OCTANE_NOW, &ws);

/* create and modify facets in this workspace,
 * using Oct tools.
 */

...

/* Record the status of this design: give a name
 * to this configuration and start exploring
 * a new alternative
 */
octaneSetConfigurationName(ws, "initial");
octaneCreateConfiguration(ws);

/* more facet editing */

...

/* We may later come back to this point.
 * Let's take a snapshot of this new implementation
 * and continue looking for a better solution
 */
octaneSetConfigurationName(ws, "new");
octaneCreateConfiguration(ws);
octaneSetConfigurationName(ws, "better");

...

/* Nope. The initial approach was after all
 * the best. We have to go back to it
 */

octaneRestoreConfiguration(ws, "initial");

...
```

Properties

Octane objects can be annotated with *properties*. Properties can be attached to other Octane objects, such as workspaces, configurations, facets and epochs. Some properties have reserved names and are used for the resolve mechanism parameterization and *tool logging*. Tool logging is performed when an Oct tool accesses a facet, triggering the resolve mechanism. Time stamps and other data is then automatically attached to predefined facet properties registering these events. Other information, such as equivalence relationships between design objects can also be defined using Octane

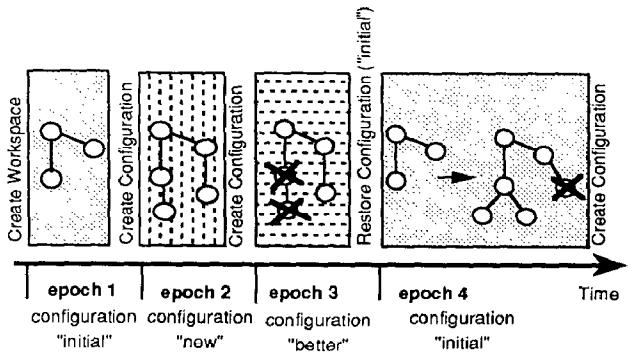

Figure 3: An example of the evolution of a design over time

properties. This functionality constitutes the foundations upon which a verification and validation subsystem [1] could be built.

The Resolve Mechanism

Formally, a facet now becomes a 5-tuple defined as

$$< workspace, cell, view, facet\_type, version >$$

A user specification of a facet will normally omit the workspace and version names, directing the Octane resolve mechanism to find them as a function of the established currency. User's specification of a facet is then

$$< \star, cell, view, facet\_type, \star >$$

In Octane, a *resolve policy* is specified as an ordered list of workspace names. Resolve policies are associated with configurations: each configuration can have its own resolve policy. To change a design management style one simply has to modify the established policy in the *active* configurations of the accessed workspaces. Policies are specified as the values of reserved configuration properties. The values are strings with names separated with ":". When looking for a facet, Octane searches each workspace in the defined sequence until a facet with matching *cell, view* and *facet_type* names is found. In each workspace Octane considers only the facets contained in the *current configuration*. When a facet is opened through *octOpenFacet* a user-specified default resolve policy is adopted.

As an example consider a design on which a user had the accessed facets in three workspaces, `PRIVATE`, `GROUP` and `LIB`. With a resolve policy like

`"PRIVATE:GROUP:LIB"`

the facets in his exploratory configuration in workspace `PRIVATE` would override facets from `GROUP` with the same name which in turn could have some facets checked-out from `LIB` and locally modified.

Facet Movement

Version Server's *check-in* and *check-out* operations are not implemented by Octane, as they are closely associated with policy decisions. Following the philosophy of providing only the mechanisms, Octane offers low-level functions with which all common variants of these operations might be built by a design management tool.

The basic operations provided by Octane to move facets between workspaces are *move* and *copy*. *Move* is the low-level version of *check-in*: it copies a facet into the destination workspace and deletes it from the origin afterwards. *Copy* just duplicates a facet in the destination workspace and is the low-level implementation of the check-out operation. Both functions use the property mechanism to annotate relevant data about the operation, such as the time at which it occurred.

Control of Concurrency

As the meta design data of a workspace is stored in a single file, concurrent writes in the meta design data file of the workspace by two simultaneous tools would result in the loss of the changes made by the first one to write on it. To avoid this, all Octane functions that modify the contents of workspaces automatically lock and release them upon exit. Execution of Octane functions that modify workspaces meta design data is delayed until the release of the locks.

The lock/unlock functions are also visible at the interface level for use of design management tools. They can be used by design management tools to surround every sequence of calls in the same workspace, and are provided for better *efficiency* (avoiding loss of time in successive lock/unlock sequences) and to assure *consistency* (avoiding change of workspace contents between calls).

Octane also offers a mechanism to design managers for rolling back the contents of workspaces meta design data when a complex transaction (as the check-in of all the facets from a configuration) as to be interrupted due to some error and the workspace contents repositioned at its state in the beginning of the transaction.

Time

In a multi-host environment, where a workspace data file can be accessed concurrently from distinct hosts, serious inconsistencies could arise if each tool used the current time of its machine to make design annotations into Octane properties. To avoid this situation, Octane provides a function that returns the current time in the file-server where the workspace data is stored. Design management tools can thus use this service to mark time stamps in Octane properties, guaranteeing that consistency between them is assured. This service is also used by Octane itself when doing tool logging.

3 IMPLEMENTATION

We have made a prototype implementation of the Octane architecture. In our design we wanted the architecture to be independent of the possible implementations. With this in mind, we sometimes have chosen the solution which lead to faster rather then most efficient results.

The meta design data of each workspace is stored in an Oct facet. The Oct data manager has been used to control access to Octane objects. Oct *bags, properties* and *attachements* are used to model the Octane concepts.

In Octane a workspace is implemented as directory that can contain Oct facets. The workspace meta design data is stored in a facet with a reserved name. The placement of the facets of each workspace into a distinct directory was made to have facet name independence across workspaces. It is possible to have facets with the same names but in distinct workspaces.

In the current implementation of Octane, a workspace is referred to via a logical name in a pre-defined name table. The table translates each logical name into the pathname of the directory file that contains the workspace data. The table is stored in a file in a predefined location and can be manipulated with Unix text editing tools. Octane initialization routine reads this table from a file at startup. Instead of using directory pathnames to name workspaces, we used this indirection because design data becomes independent of workspace location. If we would have to change the directory of a workspace in a file system or to move it into another design environment (as it frequently happens to cell

libraries), a simple change in the name table would allow correct reference to the workspace data.

To avoid unnecessary duplication of objects each time a new configuration is created, we implemented a *copy-on-write* mechanism in Octane. When a configuration is created, references to contained facets point to the facets already contained by its ancestor. When there is an attempt to modify a facet that belongs to more than one configuration, the facet is automatically duplicated. The new version of the facet replaces the previous in that configuration.

Concurrency in the access to Octane data is controlled through locks over the file system. However, the workspace data lock mechanism cannot be as simple as a semaphore controlling the access to the meta design data file. As the meta data is organized in terms of Oct objects in a facet, each user accessing it maintains its own copy in its own address space from the beginning until the end of a session, as described in section 2.1. For this reason, before modifying the workspace contents, every Octane function compares the date it was last read from disk with the date of last update of the workspace data file. If the latter is more recent, the data in memory is discarded and re-read again from disk. Also, to make changes visible to other applications, the workspace data is automatically saved into disk whenever it is changed. For efficiency, in each tool there is a pending locks counter associated with each workspace. A file system lock will be set over the workspace data files only when the lock counter changes from *0* to *1* and will be released only when the counter goes from *1* to *0*. In our implementation, the locks are made from a semaphore file, which is created without read permission to any user with the Unix *creat* system call. The technique is described in [8].

4 A DESIGN MANAGEMENT TOOL

VEM is the Oct data manager editor. It offers primitives for design data display and manipulation. It also acts as the graphics shell for editing Oct designs and invoking other CAD tools. Oct/VEM remote procedure call (rpc) applications register their commands into *VEM* upon startup and command execution procedures are invoked when commands are entered in VEM windows.

Lisbon is an example of a design management tool. It is a simple program that invokes *Octane library* functions to perform design management tasks, such as

- browsing the contents of *workspaces*
- creating and restoring *configurations*
- checking-in and checking-out *Oct facets*
- making design snapshots

Lisbon can operate both as a tty-based application or as an rpc application, callable from VEM (figure 4). The version of VEM we use was linked with our Oct/Octane library, instead of the classic Oct library. Any other design manager built on top of the new library could replace *lisbon*, just by its invocation instead of the one formerly used, without need of any change in VEM.

5 IMPLEMENTATION DISCUSSION

In Octane, when a tool modifies the contents of a workspace, there is no mechanism to notify other tools of the changes made. However, this feature does not seem to be absolutely necessary. As described earlier, in our implementation, we use Oct facets to maintain workspaces data. As a result of this and the used locking strategy, the workspace data, as seen from each tool, is guaranteed to be updated only while the workspace remains locked. This does not imply however that each call to Octane, even those that only

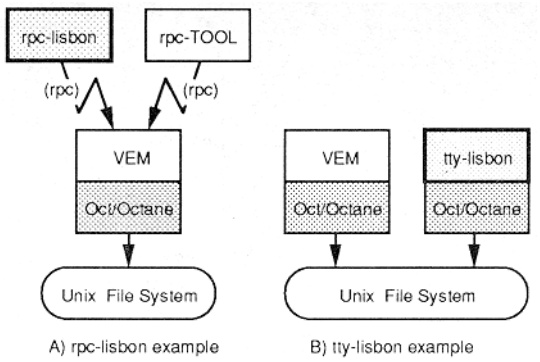

Figure 4: Lisbon is a simple design management tool that operates both as an Oct/VEM rpc application (A) or as an independent process (B).

read the contents of the workspaces data, would have to lock them before returning the data. In fact, most design strategies impose a hierarchal organization of workspaces, with private workspaces used for individual interactive changes. As a result, the changes made into private workspaces only are to be seen by the users who own them. Changes made into group workspaces usually imply the creation of new configurations and require an explicit setting as new *current* configurations by individual users.

However, in some situations, more than one tool is activated simultaneously by a single user to perform one design task. There are multiple examples in the Oct/VEM framework where design tasks are performed by various tools simultaneously, such as the critical path finding tool used in conjunction with the VEM editor to display them. In these situations, one tool has to read the contents of a facet that is modified by another tool. When the tools are coupled through remote procedure calls they share the same facets in memory, and changes are reflected immediately. If they operate independently then changes are not reflected as they are made. For this reason, we think that an implementation of the Octane architecture having a server process per workspace would be better than the one we implemented. With a workspace server, the status of design and meta design data as seen from the tools would be observed in a consistent way, independently of the chosen organization in terms of workspaces.

As we have described, locks to individual facets are made through the setting of properties attached to representations of facets in workspace data. Currently, when a tool exits abnormally the locks the tool has issued have to be removed manually. We use our design manager to remove the properties that register the facet locks and use the operating system commands to erase the workspace lock files. With a server-based implementation, it would be possible to remove automatically the pending facet locks upon tool exit. With the current implementation, we would need a recovery program to run independently, when no tool were accessing the workspace. In a server based approach, clean-ups could be performed automatically by the server, upon detection of disconnection of client programs.

As a consequence of the good design of Oct we could build Octane and have it operating with VEM and other tools without having to change Oct or the source code of the tools. There is however a slight change we would like to see in the Oct interface, to maintain its clearness. Conceptually a workspace encompasses a set of cells, which then encompass

a set of views, etc. We would like that, at the application interface level, a facet could be specified as qualified name with 5 fields, with a new **workspace** field.

In our implementation of Octane, when resolving a facet, we always assume that the workspace and version names are omited. The resolve mechanism of Octane is always invoked whenever a facet has to be opened. This way of accessing the facets is called *dynamic configuration binding* in the Version Server terminology. It can be useful in early design phases, when users are mainly trying to see answers to "what-if" questions. However, changes of the currency established in Octane (for example, a change in the default resolve policy) can lead to unpredicted changes in the appearance of a design, which are generally unacceptable in final design phases. It would be preferable to have *static configuration binding*, to have names always translated into their original resolution, independently of changes in meta design organization. To achieve this, workspace and version names of facets would have to be stored in Oct *instances* upon resolution. The Octane name resolution mechanism could then be bypassed on subsequent openings.

In our definition and implementation of Octane there is no synchronization between the clocks of distinct workspaces. The proposed time service only assures consistency between time stamps logged into the same workspace. We feel that this behavior would not be acceptable for design management tools keeping track of consistency of transactions between distinct workspaces. However, we have just started studying the possible requirements of this class of tools. For this reason we decided to offer only this functionality, as it is of simple implementation.

6 CONCLUSIONS AND FUTURE WORK

We have proposed and implemented some extensions to the Oct design manager to improve the offered support for protection, versioning and concurrency. Many of the extensions were taken from previous experience with the Version Server. While incorporating these concepts into Oct, we have followed its philosophy of providing only general mechanisms with which particular design management styles can be implemented. The conceptual model of the proposed extensions has been presented. We have also described an implementation of the architecture and a simple design management tool that uses it. Both were built mainly to be used as a test bed for our ideas.

This project has demonstrated that the incorporation in design tools of parameterizable mechanisms for the accomplishment of design management tasks is a solution for having them operating with multiple design management styles.

As a consequence of its good design, we verify that Oct did not have to be changed in its interface specification or source code. However, for the sake of consistency, we have proposed slight changes to the interface definition.

We think that with possibly minor changes at the architectural level and some modifications in the implementation, the proposed extensions could be useful to VLSI designers, specially those involved in large projects. Further developments in the design management library would be:

- A server process per workspace to control access to design management data, instead the locks on the file system. Among other reasons, this would offer a consistent way for presentation of updates into workspaces, independently of the organization determined by particular design management styles. It would also permit automatic cleaning of locks upon tools exit by the server.
- Archive support. Users should have the capability of archiving unused configurations.
- Support for concurrent development of alternatives in the same workspace. The implementation would be straightforward, but small changes in the interface would be required. However, this functionality could be simulated by employing additional workspaces.

It might also be necessary to build a new design manager to extend the simple one we implemented. Some of the requirements of the new design manager would be:

- A graphical interface to present a user-friendly representation of the Octane data model. Of particular interest would be the capability of displaying design's evolution over time, in terms of epochs.
- The graphical interface should also be able to present the structure of the design hierarchy maintained by Oct, using techniques explored in [2].
- More sophisticated implementations of check-in and check-out functions, with capability of checking multiple facets simultaneously and rolling back after an error in the middle of these transactions.
- The capability of definition and propagation of relationships between facets [5] and invocation of verification and validation tools [1] to assert the relations.

Acknowledgments

The authors would like to thank the members of the Oct/VEM framework group for the valuable discussions and suggestions for the definition and implementation of Octane. Oct developed under the sponsorship of the Defense Advanced Projects Research Agency; Version Server under the National Science Foundation. This work was partially supported by the FLAD – Luso-American Development Foundation and INESC. Their support is gratefully acknowledged.

References

[1] Rajiv Bhateja and Randy H. Katz. VALKYRIE: A Validation Subsystem of a Version Server for Computer-Aided Design Data. In *24th ACM/IEEE Design Automation Conference*, pages 321 – 327, 1987.

[2] David Gedye and Randy Katz. Browsing the Chip Design Data Base. In *25th ACM/IEEE Design Automation Conference*, pages 269–274, 1988.

[3] D. S. Harrison, P. Moore, R. L. Spickelmier, and A. R. Newton. Data Management and Graphics Editing in the Berkeley Design Environment. In *1986 IEEE International Conference on Computer-Aided Design*, pages 24 – 27, 1986.

[4] R. H. Katz, M. Anwarrudin, and E. Chang. A version server for Computer-Aided Design Data. In *23rd Design Automation Conference*, pages 27 – 33, July 1986.

[5] R. H. Katz and E. Chang. Managing Change in a Computer-Aided Design Database. In *13th VLDB Conference*, pages 455–462, 1987.

[6] Randy H. Katz. A Database Approach for Managing VLSI Design Data. In *19th ACM/IEEE Design Automation Conference*, pages 274 – 282, 1982.

[7] Randy H. Katz, Rajiv Bhateja, Ellis Chang, David Gedye, and Vony Trijanto. Design version management. *IEEE Design & Test*, :12–22, February 1987.

[8] Marc J. Rochkind. *Advanced UNIX programming.* Prentice-Hall, Inc., 1978.

[9] I. Widja, T. G. R. van Leuken, and P. van der Wolf. Concurrency Control in a VLSI Design Data Base. In *25th ACM/IEEE Design Automation Conference*, pages 357–362, 1988.