CS-1987-21

Prism: A Distributed VLSI Design System

Neil G. Sullivan, Jonathan B. Rosenberg,
Mark T. Jones, David F. Kotz, R. James Nusbaum,
James W. O'Neil, Hervé Tardif

DEPARTMENT
OF
COMPUTER SCIENCE

DUKE UNIVERSITY

Prism: A Distributed VLSI Design System

Neil G. Sullivan
Jonathan B. Rosenberg
Mark T. Jones
David F. Kotz
R. James Nusbaum
James W. O'Neil
Hervé Tardif

Department of Computer Science
Duke University
Durham, NC 27706

CS-1987-21

June 4, 1987

Contents

1 Introduction

1.1 Purpose of this Document

This document serves two purposes. First, it is intended to be a chronology of decisions made by the Prism design team from January to April 1987. It is also intended to describe some important aspects of the design of Prism, but as a design document, this work is in no way complete.

1.2 Background and Motivation

Chip and cell design take several forms. There are mask-level systems, symbolic-level systems, silicon compilers, and standard cell systems, just to name few. Many of these forms can be used together to help create an entire design.

This design paper describes a symbolic design system called Prism. The motivation for designing Prism arose from the desire to improve symbolic-to-mask compaction — specifically in the VIVID[1] system. Current compactors run as totally batch processes. Running batch, a compactor must either smash the chip hierarchy and compact the entire chip as one cell or compact individual cells, making assumptions about the environment and connections for each cell. In either case, the area of the mask suffers. Also, compactors can take an extraordinary amount of time, and one small change — even if it would make no change in the area of the compacted mask — requires a total recompaction.

Experiences with using and creating VIVID indicated more reasons to build Prism. VIVID is considered one of the best existing symbolic systems, but strides in state-of-the-art communications, user interfaces, and design automation software engineering have left it behind. Prism is a descendant of VIVID, but Prism is a new model for symbolic design.

1.3 Improving on VIVID

1.3.1 Changes to Compaction

Improvements to compaction here do not revolve around improving compaction algorithms.[2] Rather the emphasis is on building a controller around an existing compaction engine. The responsibilities of the controller are making compaction user-guided (rule-based), constraint-based, and incremental. These changes are discussed in detail in other sections.

The reader is asked to keep in mind that it is primarily compaction that has provided the motivation for this work. Many other suggestions grew out of the compaction discussions, and so the Prism design team was organized. For the remainder of this document, the compactor is treated as any other tool in the Prism system.

[1]VIVID is a trademark of the Microelectronics Center of North Carolina.

[2]There are two basic forms of compaction algorithms: constraint-graph and virtual grid. The work described here incorporates the Microelectronics Center of North Carolina's virtual grid compactor, but these compaction improvement schemes are in no way specific to any compaction algorithms.

1.3.2 Other Changes

There are several fundamental flaws in VIVID, and many of them can be directly traced to changes in the state-of-the-art. In terms of the system as a whole, VIVID sits apart from the rest of the design process. It is difficult to use other tools in cooperation with VIVID, and it is nearly impossible to use cell designs created elsewhere in a VIVID design. With the onslaught of new tools and the special requirements of some cells — which cannot be handled by a symbolic system — it is now best to consider a symbolic system as one element in the design path rather than as the only member.

Some flaws in VIVID are based in the use and distribution of data. The lack of a centralized coherent database makes the system incapable of handling large designs — that is, designs of more than 100,000 transistors. This problem stems from the use of ASCII as the data storage format. The sheer magnitude of data makes multiple ASCII files difficult to manage. ASCII files, while providing great generality and access to standard text utilities, are very expensive to read, parse, and write. To make matters worse, no two tools share their data, meaning that to use output data from one tool as the input to another tool, the data must be written out in ASCII and then parsed back in, using expensive *lex/yacc* parsers.

One of the major advantages of symbolic design is technology independence. However, problems have arisen from the implementation of the technology information in VIVID. The implementation assumes too much about some technologies, and setting up a configuration for a new technology is complex and requires an intricate knowledge of C programming and VIVID itself. It is obvious that additions of new technologies must be better supported and more simple. A move away from the C language to one better recognized by designers is the first step. Lisp is the obvious choice since it is similar to the design language EDIF.

In terms of specific tools, there is also room for improvement. The interactive editor of VIVID, ICE, has a reasonable interface, but many functionally similar editors today have a better interface to the user, and some operations can be improved with the use of better data structures.

1.4 Goals

The goals of the Prism design team can be divided into two groups: goals for the system in general and goals for specific tools. The delineation between these groups is usually clear, but there is some gray area. Also, it is important to keep in mind that some system goals will influence design and implementation of specific tools, and some goals for tools will affect the design of the system as a whole.

1.4.1 System Goals

The Prism system must meet the following criteria:

1. provide the user with a wide variety of design tools,

2. allow easy integration of new design tools into the environment and easy integration of this system into other environments,
3. allow tools to run concurrently on separate machines, connected through a network to the user console,
4. support design sharing among multiple users,
5. easily support designs of over 100,000 transistors,
6. do all of the above efficiently.

Prism tools include a symbolic-level circuit editor and a compactor. Other tools will include mask-level circuit editors, simulators, automatic design generators (PLA and memory array generators), and data format translators.

Prism has an open architecture, meaning that tools from different sources, with different data formats, can be easily integrated into the system. This feature is supported by a centralized design database. The database stores design data in a number of formats that represent a middle ground; that is, they can be easily translated back and forth to most of the design data formats currently in use.

The target environment for Prism is distributed workstations. The chip design database is stored on a disk farm, possibly attached to a large central machine. In such an environment, concurrent operation of design tools spreads the computational burden among many machines. To support this environment, Prism has a communication infrastructure, based in the database server.

Multiple invocations of the Prism system share the same design database, although they each have their own database manager. This architecture supports the sharing of design data among multiple users and projects. The data formats used by Prism and the communication architecture allow them to easily support large circuit designs with less overhead than current systems.

The actual efficiency of the system will not be known until it is implemented, but our experience and experimental data suggest that all of the above can be done efficiently enough to make the project worthwhile.

1.4.2 Goals for Specific Tools

Prism has two tools in common with VIVID: the interactive cell editor and the compactor. Other tools are developed for Prism to support the new architecture of the system. The goals for the database server were discussed in the previous section. A browser is also needed so that chip hierarchy can be visualized and manipulated and so that ASCII manipulations can be made on cell designs.

The interactive editor must provide fast response to the user in an agreeable interface. Also, to better incorporate cells designed elsewhere and to provide good compactor feedback, the editor must be able to display both symbolic and mask designs.

To improve compaction, three changes need to be made. The most important improvement is to add user direction. User-directed compaction implies the other two needed changes: compaction must be incremental and constraint-based. Basically these changes mean that the user can compact in a piecewise fashion, choosing the compaction order and selecting the method of compaction, and the user can determine what environment the cell will be compacted in. However, the compactor is still executable as a totally batch process. With this ability, compactors developed elsewhere are usable in the system, and changes in technology for the entire design can be easily tolerated.

1.5 Implementation

These goals are largely met by the system described in this paper, although some issues will not be resolved until an implementation is complete. The system will be implemented on Berkeley-compatible Unix,[3] using the C++ language. Some Unix networking and interprocess communications facilities will be used to simplify the communications architecture. The system will be built on top of an Ethernet, using Unix sockets.

The X graphics standard will be used to simplify the implementation and enhance portability, and support for the EDIF language will help ensure a wider range of applications.

1.6 Requests for Information

Please direct any questions or requests for information to Jonathan Rosenberg, Neil Sullivan, or Jim Nusbaum at Duke University, Department of Computer Science, Durham, NC 27706.

[3]Unix is a trademark of AT&T Bell Laboratories.

2 Prism — A View From The Top

2.1 Prism System

Prism is a symbolic VLSI design system built around a database, and it is run in a networked environment. Prism implements an open architecture. This architecture manifests itself to the user as a system incorporating many disparate tools and to the system builder as a program that easily grows.

The foundation for the architecture is a centralized design database. This database stores design data in a number of formats that represent a circuit as it goes from symbolic design to mask level description. Multiple invocations of the Prism system share the same design database, although they each have their own database manager called *G*.

2.1.1 Topology of the System

The database server, *G*; the compaction controller, *Biv*; the interactive editor, *Roy*; and the browser, *Tom*, form the skeleton of Prism. Each of these tools is discussed in detail in other sections of this paper. They are all discussed briefly in section *2.4 Tools*.

Prism is designed to run in a distributed environment. Tools may be started on any machine connected to the network. They communicate with the user console and the design database through sockets. Figure 1 shows a layout of Prism system.

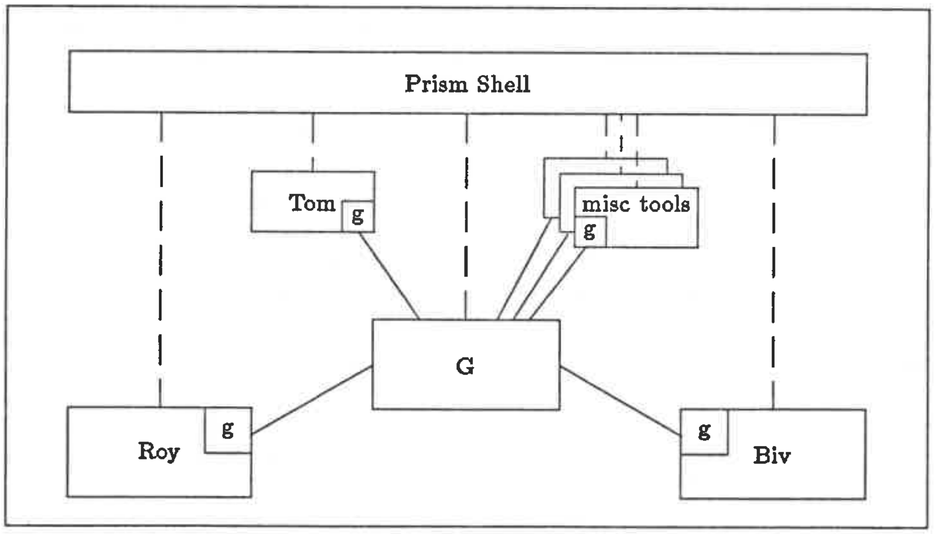

Figure 1: Prism Skeleton

Note, in the figure, that all lines represent communication sockets. The dashed lines that lead into the Prism shell indicate that the socket is only used by the shell to start up the tool and to kill it. If a tool dies before the shell kills it, that information is sent to the shell through that socket.

There will be other miscellaneous pieces of a CAD system that will not be part of the design of Prism but must be integrated with Prism. There will be at least the following: PLA generators, module generators, mask editors (Magic), routers, and standard cell systems.

These miscellaneous applications may wish a different interface to the database; so a translator or compiler between data formats makes tools fit together better. Hopefully, a tool uses a procedural interface to its data. If so, a layer of routines between their procedures and the G data server routines, loaded with the program, provides all the integration necessary. This layer is called the *Adaptor*, and it is discussed in detail in section *4.1 Adaptors*. This layer is not displayed in figure 1, but it would communicate with the *Mini-G* routines shown as the resident portion of the G data server in each tool.

2.1.2 Windows and X Graphics

The user enters the Prism system from a workstation. The user interface is in windows, through graphical and textual interaction with the database. The windows are implemented using the X graphics system developed at MIT, which runs under 4.3BSD Unix and Ultrix-32 Version 1.2. X has been successfully used on MicroVAXes[4] and has been ported to Sun workstations as well.

When a tool starts under Prism shell, the shell creates a window and begins the tool process in that window. Each process may split its own window into subareas. Windows provide the user with an ability to shift between tools, running several at a time (say, a compaction and an editing job) without the tools needing to know about each other.

X graphics provides fast displays and efficient interaction with all the various tools, making more progress towards removing the user's frustration in the process of chip design.

2.2 Prism Shell

The user invokes Prism system by typing the command *prism* into a shell window. This command opens a new window and starts a process: the Prism shell. Associated with that process is a command file, called *.prismrc*, which determines where to call up the database server G and possibly some other tools. Each of the processes has a port into G as shown in figure 1.

2.2.1 Hardware Allocation

Since a Prism session includes using tools on possibly remote machines, a method for starting those processes is necessary. To avoid the complexity of a resource monitor and allocator, the Prism shell takes responsibility for all process initialization. A table, specific to each site, indicates the processor choices for each tool.

[4]Ultrix and MicroVAX are registered trademarks of Digital Equipment Corporation.

The processor choice table is indexed by the process name. For each name there is a list of processors and a threshold associated with each processor. The threshold indicates a load factor. Prism then selects a processor for the tool in the following manner:

```
/*
 * For the given process, check down the list of processors it
 * can be run on.  Select the first processor whose current load
 * is under its acceptable level.  If no processors are usable,
 * select the current processor.
 */

for (i=0; i<=MAX_LISTED_PROCESSORS; i++) {
   curr_load = get_load(choice_tbl[tool_name][i].processor_name);
   thresh_load = choice_tbl[tool_name][i].threshold;
   if (curr_load <= thresh_load)
      return(choice_tbl[tool_name][i].processor_name);
}
return(curr_host);
```

2.2.2 System Startup

The *.prismrc* file contains Prism-level commands to be executed and environment variables to be set upon startup. It indicates which tools a user wants to start with. The file also contains the location of a Prism system file, called *sys_prismchoices*. This file indicates, for all tools that can work on a remote site, a list of the machines they can run on, ordered by preference. Associated with a tool and a particular machine there is a load threshold value useful for hardware allocation. When *sys_prismchoices* is referenced for a particular tool, the hardware allocation proceeds as explained previously in section *2.2.1 Hardware Allocation*.

It is clear, however, that a user may have some preferences on where to start a tool on the network. These preferences might obviously be different than the choices indicated in *sys_prismchoices*. Thus the user is able to specify, for some tools, a list of personally preferred machines. This information is found in a user file called *.prismchoices*.

Whenever a tool has to be started, *.prismchoices* is referenced to find the site where the tool will reside; if no site is given, *sys_prismchoices* makes that decision instead.

Upon startup, the user indicates a project, possibly through *.prismrc*. Associated with each project is a technology and possibly some libraries. They are automatically assigned, if possible. If necessary, the user is prompted to tell which technology is being used. The technology information is not changeable during a Prism session.

The G data server is automatically started by the Prism shell if any tool is started. There is no need to mention it in *.prismrc*, and only one G is ever used under a Prism shell. Prism shell remembers the port number of G when G is started. Immediately after

G, the browser, Tom, is started. Whenever another process is started, it is passed the port number of G, and it requests service there.

The *.prismrc* file also contains other miscellaneous information, such as where to place the windows associated with each tool. A list of commands is given in the next section.

2.2.3 Commands

The following commands are available in the Prism shell:

`source` *filename*

Read and execute a file of commands. (Note *.prismrc* is sourced automatically.)

`status` [*toolname*]

Give some information about the tool itself and the machine it is running on (which machine, load factor, et cetera). If no tool is specified this information is given for all the tools that are running.

`start` *toolname* [*machine* [*threshold_load*]]

Start the tool. If a machine is named, the tool is started on that machine. If a threshold load is given for the machine, the tool is only started there if the load on the machine is under the threshold. If no machine is given, either the *.prismchoices* file or the system's *sys_prismchoices* file determines the machine the tool will be started on. Lacking any instruction, the default is to start the tool on the current machine.

`kill` *toolname*

Remove the tool from the current invocation of the Prism system.

`exit`

End a Prism session. Kill all the processes still alive.

Since a user interacts with Prism at a workstation, the usual window commands provided through mouse interaction are available. These commands include open a window, close a window (that is, map it to its corresponding icon), and kill a tool.

2.3 The Database Server — G

Every invocation of the Prism system spawns a new G process for communications between the design database and the various tools that may be part of that invocation. The decision to have a separate server process for every invocation of Prism — that is, for every user of the system — came after weighing the advantages and disadvantages of having one G per user (many Gs) or one G per site. The table below lists some of the criteria that must be considered.

	Central G	Individual Gs
machine process space	advantage	
socket management		advantage
consistency and coherency of data	advantage	
simplicity of communications		advantage
general efficiency		advantage

Table 1: Database Server Distribution Examination

The issue of process space is not a crucial one. It is assumed that all the memory and processing power required are allotted.

The issue of socket management is important. The Sun operating system, for instance, restricts the number of sockets to 32 per process. In the event that several designers are working on a chip design at the same time, it is not difficult to imagine that a central G server could easily run out of sockets. This fact quickly debilitates the argument for a central G server.

Since the heart of the Prism system is the potentially large design database, concerns for the consistency and coherency of the data must be addressed. Clearly, consistency and coherency are much easier to maintain when there is only one process that will manipulate the data. Keep in mind, however, division of human labor, and note that it would be unusual for several individuals to be simultaneously, yet independently, working on the same part of the same design. In the rare event that several designers are working on the same part of a design, a simple warning-based approach can be employed to handle concurrency and overlapping conflicts (see section *3.5 Avoiding Concurrency Problems*).

Consider the following scheme. When a cell is taken from disk, it can be taken for reading or for reading/writing. The cell is marked accordingly. When the next designer tries to take the cell, a warning is issued to this person currently checking out the cell. It will be the responsibility of the users to resolve any such conflicts that might arise. In relying upon human interaction to resolve consistency problems, the complexity of the database server is greatly decreased; moreover, the feasibility of a discrete G for each invocation of Prism is enhanced.

The communication between the server and the cooperating tools would be quite complex if there was only one database server. If the constraint on the number of sockets per process was not a limiting factor, then the vast number of service requests initiated by the tools would certainly be a bottleneck. As an example, consider a central G server that must communicate with many users. Each user will likely have an editor, a compactor, and perhaps other tools running simultaneously. The bookkeeping involved in serving all these process is expensive, especially if it is to be maintained by one process. Also, much of the bookkeeping facility will be unused if the assumption is correct that the division of human labor will preclude much work from overlapping.

Note that this discussion has been based upon the assumption that there are only two choices for deployment of the G database server. Such is not the case. Several schemes have been investigated, such as having a G server per machine. This scheme would require a daemon on each machine, creating a very complex communication structure between the servers. On top of those troubles, the target environment for this system is (probably diskless) workstations, where cells are saved on a large disk farm off some possibly central machine. In this enironment, the one-G-per-machine scheme degenerates to precisely the one-central-G. Other distributed G architectures presented similar problems.

In summary, a G for every invocation of the Prism system eliminates much of the bookkeeping that is needed in a one server environment. In addition, each user would have a dedicated server, therefore escaping competition with other Prism systems running simultaneously. The communication complexity is also reduced, as each G will talk with only the tools from the Prism session from which it was spawned. Finally, lack of sockets encountered in the one-G-server case, is not a problem when each G-server communicates with only the tools that one user has invoked.

2.3.1 Mini-G

The communication between G and the tools that are part of the Prism system is via sockets. Each tool has, as part of its structure, a library of information retrieval and deposition routines that directly communicate with G by way of the sockets. This library, which is called Mini-G, consists of a set of routines designed to get and put data from and to the design database. It is provided as a common communications foundation for built-in tools and for alien tools that are being integrated into the system. Through Mini-G, all tools, therefore, access the design data in the database. The specific commands of Mini-G and the role of Mini-G as viewed by an arbitrary tool are explained in section *3.7 The Tool's Interface to G — Mini-G.*

2.3.2 Technology Database

One of the advantages of a symbolic design system is that it provides technology independence. The technology information must be stored in a database, though, and G is responsible for the distribution of that data. The technology information system is called *Otis*, and it is explained in detail in section *3.8 Introduction to G Technology Information*

System.

2.4 Tools

The protocol of communication between G and the tools of the Prism system is designed with the concept of an open architecture in mind. The delegation of the data transfer routines to Mini-G has eliminated the requirement that a tool developer be intimately familiar with the inner workings of the system. The library of Mini-G functions frees the developer from data transfer concerns, and in order to integrate a tool into the system, the developer need only know how to translate the new tool's data into the format of the data in the design database and vice versa. The translation will be incorporated as part of a tool adaptor (see section *4.1 Adaptors*). The tool itself then requires little modification to be integrated into the system.

There are some basic tools already part of the Prism system: an interactive editor, a compactor, and a browser. Prism provides these as Roy, Biv, and Tom. There is no need, therefore, for new versions of these tools to be added to Prism, but if a user prefers some particular program, there is no reason why it cannot be integrated and used in place of an existing tool.

Some of the capabilities of Prism's base tools are listed here:

- **Roy** — the interactive symbolic editor
  - display and editing (including edit-in-place) of symbolic design
  - display and editing (including edit-in-place) of floorplans
  - graphical selection of compaction ordering
  - display of mask format
  - routing
- **Biv** — the compactor
  - compaction ordering
  - compaction to constraints
  - worst-case compaction
  - pitchmatching
  - interactive, user-guided compaction
- **Tom** — the browser
  - examine database contents
  - ASCII interface

These basic tools are referenced throughout this document, and a somewhat more detailed description of each may be found in the appendices.

3 Organization Of G

3.1 Hierarchy of Designs

A circuit design is made up of *cells.* A cell is a description of a piece of circuitry. Cells can be divided into two categories: *leaf cells* and *composite cells.* A leaf cell contains a description of the transistors, wires, and other features of an integrated circuit in a number of different data formats. A composite cell is made up of other cells placed in a certain pattern. It contains only cells, not actual circuit description information. This organization suggests a graphical view of a circuit design. A graph representing a circuit takes the form of a tree. Interior nodes of the tree are occupied by composite cells, and the leaf nodes are occupied by leaf cells.

The tree structured view of a circuit imposes a hierarchy on the design elements where each level of the tree is a level in the hierarchy. This hierarchy is called the *design hierarchy.* It is used by many tools in the Prism system and is encoded in the data stored by G, but it is not used by G to organize the storage of data. G does not use the design hierarchy because it has to store many circuit designs at the same time. Many of these designs will share parts of their design hierarchy. This sharing leads to a very complicated graph that cannot be easily managed.

G imposes a secondary hierarchy on cells to order their storage: the naming hierarchy. Each cell is given a name when it is created. Each use of that cell then becomes a reference to that name. Names refer to a cell of a certain type, but each use of the cell must refer to a unique storage location because the environment in which a cell is used can affect the data in the cell. Therefore, each reference in a design refers to a specific *instance* of that name. Each instance is a unique entity.

The naming hierarchy in G can best be viewed as a class system. The creation of a new cell results in the creation of a new class in G. Each use of a cell creates a new instance of that class. Instances share some data that is common to their class, but they also have some data that is unique. For example, a leaf cell, ANDCELL, is created. ANDCELL implements a simple two input logical *and.* ANDCELL is then used in two separate composite cells, FOO and BAR. FOO and BAR contain references to unique instances of ANDCELL. Depending on the way FOO and BAR use ANDCELL, each of these instances may contain some different data, although they will also share some common data.

Each class in the system will also have a special instance, called the *null environment* instance. The null environment instance represents an instance of a cell that is guaranteed not to be referenced by any other instance. In other words, the null environment instance is not contained in any composite cells at a higher level of the design hierarchy. Therefore, no external primitives can affect the size of this instance. This instance of a cell is used in simulation and compaction to get test data and lower bounds on circuit size.

3.2 Namespaces

In order for G to implement a class system that avoids undue name conflicts the concept of ***namespaces*** is used. A namespace is a collection of named classes, providing a way for the same name to be used to refer to different classes.

G divides class names in a hierarchical manner. At the top of the hierarchy are *packages*. A package implements a namespace. A package may represent a certain project or a library of standard cells. Each package has a name. In order to prevent inadvertent or malicious modification of design data, each package has an access system that controls read and write permissions. This system is especially important when a package represents a library.

Within each package are the classes in that package and some package-specific information. Package-specific information includes an indication of the technology associated with the package, package documentation, and data necessary for managing the package access system.

Each class represents a cell and has a name. Class-specific information and instances are stored with the class. Class-specific information includes the type of the class (leaf, composite, or unknown) and whether the class is public or private. Public classes may be used by anyone and modified by anyone who has write permission to the package. Private classes may only be used and modified by those with read or write permission to the package. A symbolic description of the class, documentation on the class, and backup data are also class-specific information.

Old versions of the symbolic and documentation data for a class are backup information. When the data for a class is changed, the old data is stored in a backup location. A type of garbage collection removes unneeded backups from the system when appropriate.

The format of some class-specific data depends on the type of the class. If the class represents a leaf cell, the symbolic data is an actual description of the features of an integrated circuit. If the class represents a composite cell then the symbolic data describes the locations, connections, and names of the cells contained in it. If a class represents a cell of unknown type then the symbolic data is merely a bounding box for the circuit that the class will describe.

The data for each instance of the class has different formats depending on the type. For leaf cells the formats are: a mask level description called *Rex*; compaction constraints, which are imposed on the instance by its neighbors in a design; and a simulation description. If a class is a composite cell then the formats for instance-specific data are the compaction constraints on the cell, simulation data, a list of the specific cell instances contained in it, and the order of compaction for those subcells. (Recall that the class itself defines the symbolic data for the cell.)

Compaction constraints are not actually stored with a class. Since they are easily calculated and will change often, they are recalculated for every access.

3.3 Namespace Implementation

3.3.1 Directory Structure

The Unix file system is used to store the naming hierarchy. Packages are directories. All package-specific information is stored in files in these directories. Classes have subdirectories in the package directories. The data common among all instances of a class is stored in files in the class directory. Each instance of a class has a subdirectory in the class directory. Instance specific information is stored in files in these directories. Backup information is stored in a subdirectory of the class directory. In the backup directory are subdirectories for each backup version.

A pictorial representation of this directory structure is in figure 2. Only the directories are shown, not the files in the directories. Both leaf and composite cells have the same directory structure but different files in their directories and different data in the files.

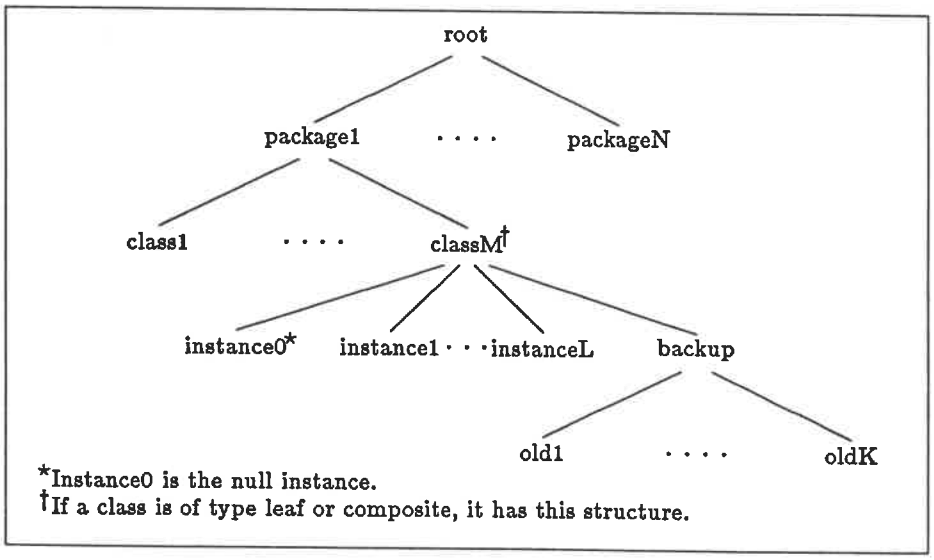

Figure 2: G Directory Structure

G also supports a generic file storage capability. The user can create packages to act as directories and classes to act as generic data files. Generic files are accessed similarly to cell data. A package for a generic file directory contains no subdirectories — only files.

3.3.2 Name References

The following C++ structure, called a *name descriptor*, is used to completely reference data in the system:

```
struct name_descriptor_s {
   char *package;      /* a string for the package name        */
   char *class;        /* a string for the class name          */
   int version;        /* an integer for the version           */
   setofint instance;  /* a set of integers for the instances */
   char format;        /* a character for the format           */
};
```

The package field is a string of characters representing the name of the package. The class field is similar. The version is an integer identifying the version of the data that is desired. This number must be one of the valid version numbers that G has for that class. Version 0 is considered the most recent version. The instance is a set of integers representing the instances that the name descriptor refers to. It is necessary to refer to a set of instances, rather than a single one, in order to permit G to share data among subsets of instances. Format is a character designating which of the data formats for a class is desired. See section *3.4 Data Structure Formats* for a detailed description of the formats. The valid format characters are:

S Symbolic. In a leaf cell, symbolic refers to the description of the actual circuit. In a composite cell it refers to the description of how the subcells are connected, where they are located, and how they are oriented. If a version number other than 0 is specified, G looks in the backup directory for the correct version of this format.

D Documentation. This format is a LaTeXable file describing the cell. If a version number other than 0 is specified, G looks in the backup directory for the correct version of this format.

R Rex. Rex is only valid for leaf cells. It refers to the Rex circuit representation. An instance must be specified for this format.

U Simulation. This format is the simulation representation of the cell. An instance must be specified for this format.

C Constraints. This format refers to the compaction constraints on a cell.

O Ordering. In a composite cell, ordering refers to the compaction ordering and methods for the subcells.

N Names. This format returns all the class names in a package or all the instances of a class.

G Generic. This format indicates that a generic file is being referenced.

I Information. This format gives statistics for a package or class. Statistics on a package include read and write permissions, time information for creation and modification, and the technology associated with the package. Statistics for a class include its type, what versions are available, timestamp information, use information, and whether it is public or private.

A facility is provided to the user to set up a path of packages to search for name resolution. The path is initially set in the *.prismrc* file, but it can be subsequently changed in the browser tool. The path is a sequence of package names that is searched in order to locate a class name if no package name is specified. It allows tools to ignore package names to some extent, and G uses it whenever a name descriptor is encountered with no package name.

Name conflicts are resolved by choosing the first package in the path where the name appears. Of course, the user can always short circuit the path search by specifying the package name explicitly. New names are put into the first package on the path if they are not given a specific package name.

All name references stored by G contain package names, thus removing any ambiguity in stored name references.

3.3.3 File Formats

G extensively uses the Unix file system to organize its name storage. To avoid a shortage of file descriptors, file names are remembered rather than keeping open descriptors for a file that may be inactive after one operation.

G stores the circuit description data in files. The data in these files is grouped into a small number of formats. These formats are based on — if not identical to — the data structures being stored. All formats are stored in a compact linear organization suitable for transmission over a socket.

3.4 Data Structure Formats

The data requirements for G are shown in figure 3. All data structures known by G are shown in ovals, and tools are drawn as rectangles. The direction of the arrows indicates the flow of the data and which tools can change the data. An arrow pointing into a tool indicates that the data structure is required by the tool for proper operation. An arrow pointing into a data structure means that the tool changes data of that form. Data is never changed by G, itself.

The data structures have been carefully chosen in an attempt to minimize the number and size of data formats needed while providing necessary efficiency and flexibility. Thus, some of the formats are a middle ground, approximating the format desired by several tools.

All data stored by G is in one of these formats. The general file format for unstructured data accommodates temporary and other files for tools. Translators should be provided for conversion to and from common external formats, for example, CIF, EDIF, Magic, and Caesar. The tool specifies (via the format field of the name descriptor) the type of data structure it desires from, or is sending to, G (see section *3.3 Namespace Implementation* above).

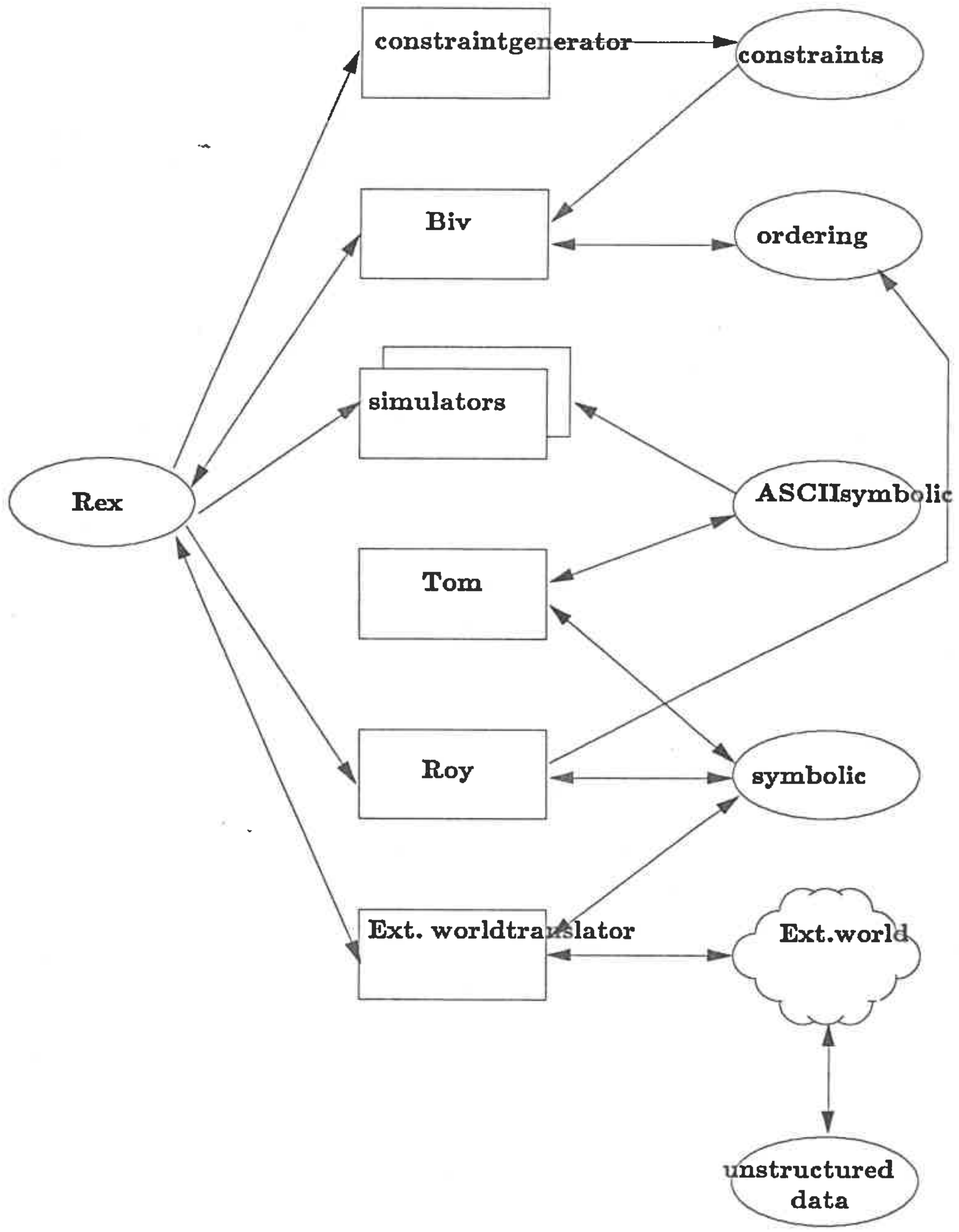

Figure 3: Data Structure Requirements

Note that the G technology information data structures are not shown in the figure. Portions of the technology information are required by every tool shown in the figure. Also, note that the external world cloud includes both data structures and tools.

All data structures are stored in a machine-readable format; that is, no data is converted to ASCII unless explicitly required. So G can transfer extremely quickly from the data base directly to the appropriate socket. The tool can read directly from the socket into its data structures. Some conversion may be necessary in tools running on machines with different word formats. Any further translation to a form desired by the specific tool is the responsibility of the tool (see section *4 Tools*).

3.4.1 Symbolic Data Structure

ASCII symbolic chip designs are basically expressed by ABCD.[5] The symbolic data structure used by G is essentially a compiled form of ABCD. (Clearly, one of the translators mentioned above is an ABCD translator.) This structure is powerful enough to express the hierarchical connectivity of the chip, which is basically the floorplan, as well as the specific devices composing a leaf cell. The data structure consists of a set (that is, an unordered list) of symbolic objects. Each object contains a type and some type-specific information.

In the implementation, the data structure set is given to the tool as a linked list of structures, each of which has the type field first, followed by an arbitrary structure that is type-specific. The types and their associated information are given below. New components, which are not supported by ABCD, are shown in italics.

instance	cellname, reps, direction, orientation, corner, *connect*
device	location, orient, type, W/L, dnet/gnet/snet, *bbox*
wire	point-list, layer, width, net
pin	location, layer, net (signal name)
contact	location, type, net, *orientation*
label	box:[location, size], text
resistor	*location, resistance, net*
capacitor	*location, type, net*

3.4.2 Mask and Compaction Data Structures

One data structure is used to represent arbitrary mask objects before and after compaction. This data structure is called Rex. Conceptually, Rex has three parts:

1. grid: the position, in microns, of each horizontal and vertical grid line, relative to the origin of the local universe;

[5]ABCD is *A Better Circuit Description* language, developed for the VIVID system at the Microelectronics Center of North Carolina and Duke University. Version 2.0 of ABCD is the basis for both the ASCII symbolic and data structure symbolic used in Prism.

2. virtual mask rectangles (VMRs): the set of occupied mask grid points and for each grid point a linked list of rectangles at that grid point — each rectangle has a bounding box, a layer, and an indication for each side telling if the rectangle stops at that point or stretches to another grid point;

3. offsets: the (x, y) distance of each rectangle from its associated grid line — originally zero for every rectangle.

In detail, the Rex data structure is not split as cleanly as the conceptual model. The structure itself contains the following information: the distance, from the origin, of the x and y grid lines and the VMRs. The x and y grid distances are changed by the compactor.

```
struct rex_s {
   int *x_grid_dist;     /* dist of horz grid lines from origin */
   int *y_grid_dist;     /* dist of vert grid lines from origin */
   struct vmr_s **r_vmr; /* virtual mask rectangles             */
} *rex;
```

The VMR structure describes a grid point, and the offset information depicted in the conceptual Rex structure is actually found here. For each grid point, there is a list of rectangles — so abstractly the VMR structure is a two-dimensional array of pointers, each point pointing to a list of rectangles found at that grid point. Each rectangle must have a bounding box, a layer, a net-list node, and an offset from the grid point. If a rectangle stretches to another grid point, the bounding box value in that direction is infinity. For example, if the rectangle is a horizontal wire, both the min x and max x could be infinity. Initially, the offset value for each rectangle is zero for both the x and y directions. The compactor may change those values.

```
struct vmr_s {
   int v_x_posit, v_y_posit;      /* the grid point            */
   struct rect_s {                /* list of rects at this pt  */
      struct bbox_s {             /* bounding box of one rect  */
         int ll_x, ll_y;
         int ur_x, ur_y;
      } rbox;
      int r_layer;                /* layer of this rect        */
      int r_node;                 /* elec node of this rect    */
      int r_x_offset, r_y_offset; /* rect's offset from grid pt */
   } *v_rects;
};
```

Any other mask format can be easily converted to a Rex data structure that has a single grid point at (0,0) from which all rectangles are offset. Symbolic layout can be converted by expanding each primitive into a set of VMRs with zero offset from the primitive's original gridpoint. The distances between grid lines are initially based on a worst-case design rule. This Rex is passed to the compactor, which may change offsets, move grid lines, or even merge or delete rectangles.

Rex is used by tools such as Biv, Roy (for display), mask translators (in to and out of the system), plotters, and simulators.

3.4.3 Compaction Ordering Data Structure

The compaction ordering (see appendix *C.3 Compaction Ordering* for a description) is a list of cell instances in the order that they should be compacted, along with a method of compaction. Instances may be grouped for identical compaction; so the list is an ordered set of pairs *(method, instance-set)*. The compaction method may be compact-to-constraints, pitchmatch, or worst-case-environment. The list is stored as a linked list. Similarly, the instance-sets are also stored as linked lists.

```
struct comporder_s {
   comp_method_t co_method;                /* compaction method */
   struct name_descriptor_s *co_instances; /* instance set      */
} *comp_order;
```

3.4.4 Compaction Constraints

One type of compaction is compaction to constraints. In compaction to constraints, there is an ordering — either explicit or implicit. Consider the example of cell FOO. Cells compacted previous to FOO (preceding it in the order) can affect its compaction, and FOO can affect cells that follow it in the order.

The compaction algorithm is as follows. When it is time to compact FOO, examine previously compacted neighboring cells, noting the positions of all objects on the boundary of those cells. These objects will include any pins that FOO connects to and features such as transistors and wire endpoints. The objects constrain the positions of the corresponding pins and features in FOO. Additionally, the bounding box of FOO, specified by the mask-level floorplan (essentially, whatever hole its neighbors leave it), may be constrained to some amount. Thus, the constraints are a bounding box and an ordered set for each side of the cell. Each constraint in this set is a triple:

```
struct constraint_s {
   int grid_line;  /* virtual grid line                         */
   int offset;     /* required offset from origin of side       */
   char obj_type;  /* object type                               */
};
```

Since the constraints are relatively easy to compute from the Rex for a cell, it seems unnecessary to store them. However, it also seems inefficient to send all of the Rex for all of the neighbors to the leaf cell compactor simply to obtain the constraints. Thus, G has access to a local tool that finds the constraints for a cell, which may then be sent to the compactor.

3.5 Avoiding Concurrency Problems

Because each designer has a G server, but all designers work on the same data, there is always the possibility of concurrency problems. That is, two users may be manipulating the same data at the same time. G has taken the philosophy that concurrency problems indicate larger human interaction troubles. G uses appropriate levels of warnings, leaving final problem resolution to humans.

When data is needed by a tool, G marks the data as taken by the tool for read or for read-write. If another tool then requires the same data, G informs this second tool, at the time of the read, that someone else has the data. If the first tool then tries to write the data back, G warns that other tools have the data, also. There are three development modes of operation:

raw development mode: never inform any user that some previous changes may be lost;

write-protection development mode: inform the user only if someone else has written the data since it was read by this user;

production development mode: always inform the user, whether reading or writing, if any other user has read or written the cell.

Compaction concurrency problems are handled slightly differently. See appendix *C.5 Possible Problems with Compaction Algorithm.*

3.6 G Commands

All interaction with G is through sockets, using a very simple protocol. As G is a server, commands are sent from the tool to G, and G responds. This simple protocol justifies the use of a single-stream (half-duplex) socket. (How the system works on top of the sockets is discussed in section *2 Prism — A View From The Top.*)

Thus a command sent to G consists of a packet, which is constructed and sent as a C structure. Nearly all operations can be expressed as getting or putting a particular name in a particular format, so there are relatively few packet structures. The commands are listed below.

G_login

The user name and password of the user are sent to G, and G uses this information to check permissions on all packages that are accessed. Also, information is sent regarding the service priority that this tool has with G. Note that the strings have to be sent over the socket in a special manner.

```
struct login_packet_s {
   G_command_t command;              /* code for the command */
   char *username;                   /* user name            */
   char *password;                   /* password             */
   int priority;                     /* priority             */
};
```

G_logout

Logout is for terminating a connection to G, specifying that the tool is shutting down.

```
struct logout_packet_s {
   G_command_t command;              /* code for the command */
};
```

G_check_out

G marks the indicated name as being used by the tool and user submitting the command. This marking must be done before a tool may G_put the data associated with a name, and should be done as soon as the tool thinks it may be modifying the data. The access code describes the intentions. The previous value of the code is returned by G. When the tool is done with an object, the tool should check it back in by specifying access code zero.

```
struct checkout_packet_s {
   G_command_t command;              /* code for the command */
   struct name_descriptor_s name;    /* name to checkout     */
   int access_code;                  /* intended access type */
};
```

G_get

The name specifies the object to be obtained, with the format field specifying exactly which data associated with that object is to be returned. Note that some formats may apply only to the fully-specified (instance) name (for example, Rex), whereas others may apply to several levels of the naming hierarchy (for example, "information" or "names"). Consider: if a package is specified with format "information", and all other fields are null, then information at the package level will be returned. The details of names and namespaces are discussed in sections *3.2 Namespaces* and *3.3 Namespace Implementation*.

```
struct get_packet_s {
   G_command_t command;             /* code for the command */
   struct name_descriptor_s name;   /* name to get          */
};
```

G_put

This command is the opposite of G_get. The name must previously have been checked out in this format. Certain formats cause G to perform other actions, such as copying the existing data to a backup version (for symbolic and documentation data) or invalidating the Rex (when putting symbolic data).

```
struct put_packet_s {
   G_command_t command;             /* code for the command */
   struct name_descriptor_s name;   /* name to put          */
   long size;                       /* size of data         */
};
```

Other commands may be necessary. For example, the delete function provided by Mini-G (see section *3.7 The Tool's Interface to G — Mini-G*) may need its own packet type as it does not fit directly with any of the above. Commands from the Prism shell to G may also have to be added (for example, to terminate G). The implementation may also choose to merge the above structures into one or two common forms to avoid proliferation of structures.

The requests are handled by G in a multiphase protocol. On receiving a G_get command, G examines the request for validity. A packet is sent back to the tool containing either an error indication or giving the size of the data to be transferred. When the tool is ready to receive the data it sends an acknowledgement to G. Only once G receives the acknowledgement does it send the data.

When G receives a G_put command, the request only contains the size of the data to be sent. If G determines that the request is valid, it sends an acknowledgement to the requesting tool, which then sends the data. This scheme allows G to allocate an appropriately large input buffer. It may send a message asking the tool to try again later if it is short of memory.

The other three commands, `G_login`, `G_logout` and `G_check_out`, have a simple response packet from G.

The goal is to prevent G from blocking on reads and writes to the sockets. Tools may block as necessary, but G must not block since it needs to service several tools simultaneously. Therefore, G maintains some state about each socket to which it is connected and continuously cycles between them, servicing each as input or output becomes possible. The state necessary for each tool includes a state code (idle, waiting for acknowledgement, receiving data, sending data), the current request packet, and the current input or output buffer with a pointer indicating the current position. If the socket is non-blocking, arbitrarily small portions of the data may be read or written with each call, slowly filling or emptying the buffer.

3.7 The Tool's Interface to G — Mini-G

At the tool end of a socket connected to G is a layer of interface procedures collectively called Mini-G. Mini-G provides a procedural interface that gives the illusion that the tool has its own local database, hiding the sockets and interprocess communication. Mini-G generally maps from a procedure call to one of the G socket commands mentioned in the previous section. Note that the G command name is usually redundant with the format field of the name descriptor. The format field given in these instances is ignored, as Mini-G supplies the appropriate format from the context.

It is intended that Mini-G be rather small and simple, providing a flexible, fully powerful, unassuming interface to G. Further tool-specific enhancements such as data caching, translation, and even semantic changes to the procedural interface are left to the adaptor level of the tool (see section *4.1 Adaptors*).

The needed Mini-G procedures are listed here. The procedures that get information allocate the necessary space for that information, returning a pointer to the information. Note that most procedures return an error indication. Detailed information regarding the error can be obtained with the routine `g_get_error()`.

```
int
g_init (mach_name, listens)
  char *mach_name;
  int listens;
```

Given the address of the G to connect to, specified by a machine name and a socket to listen on, this command initializes the socket interface. The address information is provided to the tool by Prism at startup. This routine returns a 1 if initialization was correctly completed and a 0 if not.

```
int
g_login (user, password, priority)
  char *user;
  char *password;
  int priority;
```

Identify the user to G. This routine corresponds directly to the `G_login` command packet to G. The priority is used to specify the level of service that should be provided. For example, the compactor may receive lower priority than the editor.

```
struct symbolic_s *
g_get_symb (name)
  struct name_descriptor_s *name;
```

Obtain the symbolic data for the given cell, and return a pointer to it. Return `NULL` if unsuccessful.

```
int
g_put_symb (name, sp)
  struct name_descriptor_s *name;
  struct symbolic_s *sp;
```

Write the symbolic data for the cell, pointed to by `sp` and named by `name`. The function returns a 1 if successful and a 0 otherwise.

```
struct rex_s *
g_get_rex (name)
  struct name_descriptor_s *name;
```

Obtain the Rex form of the given cell, and return a pointer to it. Return `NULL` if unsuccessful.

```
int
g_put_rex (name, rp)
  struct name_descriptor_s *name;
  struct rex_s *rp;
```

Write the Rex data for the given cell, pointed to by `rp`. The function returns a 1 if successful and a 0 otherwise.

```
FILE *
g_get_doc (doc_name)
  char *doc_name;
```

Get the documentation for a cell. The function returns a descriptor for a file that can be read and written like any normal file. When finished, write the documentation back using `g_put_doc`, or simply close the file if no changes are desired in the stored documentation. The file is copied to a local temporary file, opened, and unlinked. So the file "disappears" when it is closed. Returns a `NULL` pointer on error.

```
int
g_put_doc (doc_name, fp)
  char *doc_name;
  FILE *fp;
```

Write back the documentation for a cell. The function returns a 1 if it is successful and a 0 otherwise. Essentially this routine copies the file back to G. The file is closed on return.

```
FILE *
g_get_file (package_name, file_name)
  char *package_name;
  char *file_name;
```

Get an arbitrary file from the "generic" file space of the specified package. That is, copy the file stored by G (with no interpretation placed on its format or purpose) to the local file system, and return a file descriptor for it. When finished, the tool should use `g_put_file` to restore the file, or simply close the file if no changes are desired in it. The implementation is similar to that for documentation files. A `NULL` pointer is returned on error.

```
int
g_put_file (package_name, file_name, fp)
  char *package_name;
  char *file_name;
  FILE *fp;
```

Store a file from the local file system in the "generic" file storage area of the given package for later retrieval by the name `file_name`. The remote file will be overwritten if it exists and created if it does not. Closing the local file is up to the caller. The file need not be one that has been obtained from G. The function returns 1 if successful, 0 if not.

```
int
g_get_stats (name, stat)
  struct name_descriptor_s *name;
  struct status_s *stat;
```

Get status information about a cell. The routine takes a cell name and returns information on, for example, bounding box, timestamp, compaction status, memory space needed, access permission (such as read only), and concurrency information (such as "Thomas Lengauer has read this cell"). The actual information returned depends on the format specified in the name descriptor. Returns 1 on success, 0 on error.

```
struct xxxx_s *
g_get_otis_xxxx (tech, ...)
  char *tech;
  ...
```

Get some piece of Otis data for a given technology. Other arguments might be appropriate for some pieces. If `xxxx` is `lisp_list`, the routine returns the technology information in the form of a standard Lisp list, with no data structure conversion. New tools, which have their own data formats, can use this routine and translate any data themselves. See section *3.8 Introduction to G Technology Information System — Otis* for further details on the Otis data formats and procedures.

```
char **
g_get_dir (name, options)
  struct name_descriptor_s *name;
  int options;
```

The function returns an array of names that are children of the given name in the namespace tree. For example, given a project, it would list all of the cell names. Given nothing, it lists all the projects. The options can be used to limit the number of versions listed, et cetera. Returns NULL pointer on error.

```
struct compact_s *
g_get_compact_order (name)
  struct name_descriptor_s *name;
```

Obtain the compaction ordering for the named cell, and return a pointer to the information. Return a NULL pointer on error.

```
int
g_put_compact_order (name, cp)
  struct name_descriptor_s *name;
  struct compact_s *cp;
```

Rewrite the compaction ordering for the named cell. Return 1 on success, 0 on error.

```
struct constraint_s *
g_get_constraints (name)
  struct name_descriptor_s *name;
```

Have G calculate the compaction constraints for the named cell and return a pointer to the information. Returns a NULL pointer on error.

```
int
g_put_constraints (name, cp)
  struct name_descriptor_s *name;
  struct constraint_s *cp;
```

Send the computed compaction constraints to G. This routine is used only by the compaction constraint calculator, and it returns 1 on success, 0 on error.

```
int
g_check_out_cell (name, access_code)
  struct name_descriptor_s *name;
  int access_code;
```

Tell G how the cell will be used. At some point the cell may be thrown away, modified, compacted, et cetera. This function tells G about this event, and G can accurately tell other users what is being done to it. The cell must be checked out before any information can be put back. The name structure provides the specific format to check out. The access code defines the intended uses. The code given will override any previous access code specified by the same user. So checking the cell out later with stronger or weaker access codes upgrades or downgrades the current check out on the cell by a given user. The old value of the access code is returned or 0 on error.

```
int
g_delete_cell (name)
 struct name_descriptor_s *name;
```

Delete the data for the cell with the given name, the format defining the data to be deleted. Return 1 on success. 0 on error.

```
char **
g_get_path ()
```

Return the current set of package names through which G will search for a name that does not specify a package. Return `NULL` on error.

```
int
g_set_path (path)
 char **path;
```

Provide the current set of package names through which G will search for a name that does not specify a package. Return 1 on success, 0 on error.

```
int
g_free_xxxx (cp)
 struct xxxx_s *cp;
```

Free some data structure that had been previously allocated and returned by the Mini-G routines. Routines that are provided are:

```
g_free_symb
g_free_rex
g_free_otis
g_free_dir
g_free_compact_order
g_free_constraints
```

```
struct g_error_s *
g_get_error ()
```

Return a pointer to a static structure that contains information regarding the reason behind the failure of one of the Mini-G routines. The information pertaining to a failure in one routine is available until the next call of one of the Mini-G routines.

3.8 Introduction to G Technology Information System — Otis

The Technology Information System, *Otis*, is the part of G that allows access to various technology dependent parameters. The main goal in the design of Otis is to make G independent of the technology being used. Otis subsumes the functions of the MTF[6] in the VIVID system. The heart of Otis is a Lisp interpreter that is an interface to the Otis database. The technology dependent information, referred to as the Otis database, is in the form of Lisp functions.

3.8.1 Why Lisp?

The MTF system for VIVID is in the form of dynamically loaded C code. Dynamically loaded code is unacceptable because it presents portability problems and wastes space, and it is quite inflexible without a complete linker/loader in subroutine form. However, not all the data in the Otis database can be stored as simple tables, and therefore, there must be some way to write functions to provide technology information.

The Otis database is in the form of Lisp functions, which are interpreted. This approach has several advantages:

1. the database allows technology information to be in the form of functions,

2. the database does not need to recompiled whenever it is changed, and

[6]MTF is the *Master Technology File* system, developed for the VIVID system at the Microelectronics Center of North Carolina.

3. most designers are more comfortable with Lisp, which is similar to EDIF, than with something like C.

A subset of a form of Lisp that is very common, such as Franz Lisp, allows the user access to a well documented language. It is preferable for the sake of portability and ease of interface to Prism that the interpreter be written in C or C++.

3.8.2 Interface to Lisp Interpreter

The interface from G to the Lisp interpreter is a very simple function call of the form:

```
        ret_list = otis_get(func_name, param_list);
```

where `func_name` is the name of the function in the Otis database that is being called, and `param_list` and `ret_list` are pointers to the standard Lisp data structure. After G has made this function call, it processes the information from Otis and packs it into a new data structure. This data structure varies depending on the type of data being sent, but it is a structure that is suitable for transmission over sockets. After building this new data structure, G then sends the information over the socket to Mini-G.

Functions in the Otis database return their data as a list of lists. Essentially everything is in the form of a list of numbers. However, there is a library of Lisp functions that enable a designer to write a new Otis database. This library includes functions that return a properly formatted list of numbers for drawing a rectangle or other polygon. The designer is insulated with functions as much as possible from the fact that integers are really underneath the data returned.

Although a designer can write a new database with little trouble — perhaps for a new technology — the technology information must be extendible. That is, because the required technology information will certainly change, the requirements for the Otis database will also. The structure of the database encourages such growth, and any new information is returned as a Lisp list. Any tool requesting the new information must, however, provide translation routines from the list structure to whatever form the data should be.

3.8.3 Division of Data and Interface to Otis

There are two types of access to Otis: requests for sets of numbers (such as design rules) and function calls (such as how to draw a transistor). In the interests of efficiency, the data in the database that is not function calls is divided into several categories, and when requests are made to Otis, entire categories are shipped. These categories are listed below with the corresponding MTF file in parentheses.

1. Technology data needed by most tools
   (a) declaration of layers (layer_dec.mtf)
   (b) declaration of synonyms for layers and transistors (layer_dec.mtf)

   (c) layers conflicts (ats.mtf)
   (d) declaration of contacts — names and types (ats.mtf)
   (e) declaration of transistor information (ats.mtf)

2. Technology data related to Biv and similar tools (such as a router)
   (a) design rules (cmpctr.mtf)
   (b) anti-feature information (cmpctr.mtf)
   (c) resistance and capacitance information (abstract.mtf)
   (d) router information, such as routing layers and which, if any, to minimize (general.mtf)
   (e) name and type of process (general.mtf)
   (f) virtual grid spacing and mask size (general.mtf)

3. Technology data related to Roy and other graphics tools
   (a) color maps (cmaps.mtf)
   (b) stipple patterns (stipples.mtf)
   (c) write masks, fill styles, and line styles (layer_def.mtf)
   (d) colors for layers (layer_def.mtf)
   (e) declaration of graphics layers (layer_dec.mtf)

4. Simulator information
   (a) model related information (model.mtf)

When a tool makes a request for a particular piece of data, its Mini-G actually receives everything in that category, and Mini-G passes the data to the Adaptor, which caches the data if it wants to.

Function calls, although they logically fit into disparate categories, make a category of their own. Each function, however, is left as a separate call to Otis since the calls cannot be grouped.

5. Functions
   (a) functions for translation of symbolic to VMR (ptrans.mtf)
   (b) functions for drawing symbolic transistors (graph.mtf)
   (c) virtual shape routines (virtual.mtf)
   (d) miscellaneous functions, such as those asking for the length of gate or the area of diffusion (abstract.mtf)

Some of the functional information can be put into tables, but in each case the information required is at least complex enough to require a functional search through a table.

3.8.4 Caching Information

Access to the Otis database is slow because everything needs to be interpreted. Interpretation is not a problem for the data that is in categories because a tool will normally only need to request it once and will cache it for its own use. However, the parts of the Otis database that are functions could be accessed over and over again from a tool like Roy.

These functions deal mainly with translating data in symbolic form to another form, and one of the most important factors is the number of different kinds of transistors in the chip design. A quick survey of symbolic designs at MCNC reveals that only a small number of (well under 50) types of transistors exist in a particular design. Therefore, a tool that needs to access a function in the Otis database many times would cache each request and answer so that it only needs to ask a particular question once.

4 Tools

4.1 Adaptors

4.1.1 Description of an Adaptor

There are many tools associated with the Prism system, performing many different tasks. Some of these tools have been central to the design of Prism and G and are built to be explicitly compatible with G. However, to provide an open architecture, to which other tools may be added relatively easily, G has been designed very carefully to provide maximum flexibility without compromising efficiency. By removing higher-level functions, such as caching and translation, from Mini-G, Mini-G can stay simple and standard across all tools.

The tools interact with G through a layer of subroutines, the lowest level being Mini-G, which insulates the tool from any direct knowledge of the the socket-based interface. Mini-G is meant to be a rather small, unassuming, flexible interface to G. The Prism-specific tools will find Mini-G too simple for their uses. The data may not be in the structure that they prefer, or they may find it more efficient to cache various types of information. External tools will have high-level routines that manipulate their data communication with the outside world in a form that they have defined (say, Caesar). Thus, a second layer, that is tool-specific, provides a buffer between the standardized Mini-G routines and the tool's preferred interface to the world. This layer *adapts* the tool to the G/Prism environment. See figure 4.

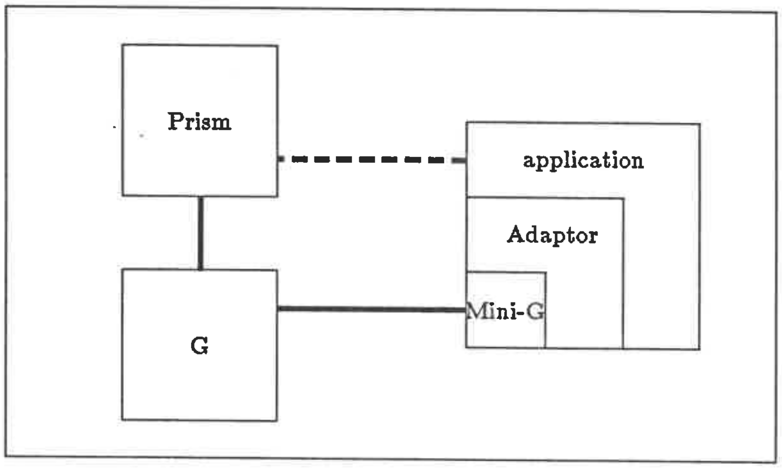

Figure 4: Use of an Adaptor

Note, in the figure, the communication flow between Prism and the tool and between G and the tool. All data communication travels through Mini-G and then through the (possibly non-existent) Adaptor. The dashed line between Prism and the application is only for process creation and termination.

The Adaptor is tool-specific, but certain commonly used functions may be useful to keep in an *Adaptor Library*, which can be used by several tools. The Adaptor Library will be most useful in the home-grown tools, though it is easy to envision data translators (for example, to and from EDIF) that would help outside tools interface with G. For example, the Browser would have reason to use an ABCD↔symbolic translator (and possibly a CIF↔Rex translator). The editor, Roy, would like symbolic or Rex cell caching and translation to two-dimensional data structures. The compactor, Biv, needs to cache constraint information. Some of these routines may be useful to other tools and, if designed carefully, could be kept in the Adaptor library for all to use.

4.1.2 Associative Memory

As previously mentioned, caching and data translation are kept out of Mini-G. The Adaptor for a tool provides the interface to Mini-G, and as such, must translate the data, but the Adaptor also can supply some associative memory, where associative memory is differentiated from caching by virtue of less functionality. Full caching is the responsibility of the tool, itself.

The access and translation routines are built on top of the associative memory routines in the Adaptor. For example, when the tool calls a data access routine to get some symbolic data, the routine works as follows:

```
struct symbolic_s *
adaptor_get_symb (name, access_method, cache, xlate)
   struct name_descriptor_s *name;
   int access_method;
   struct symbolic_s *cache(), *xlate();
{
   struct symbolic_s *symb_data;
   struct raw_symbolic_s *raw_symb_data;

   symb_data = (*cache)(GET, name, access_method);
   if (cache routine fails because data is not in table) {
      symb_raw_data = g_get_symb(name, access_method);
      (*xlate)(symb_raw_data, symb_data);
      (*cache)(ENTER, name, sizeof(symb_data), symb_data);
   }
   return(symb_data);
}
```

Note that the cache and translation routines are assigned by the tools. `Adaptor_get_symb` takes the name of the symbolic data, how it will be accessed (for example, read, read-write, or append), and the two needed routines. If the data is in the cache, it is returned. If not, the Mini-G `g_get_symb` routine returns the raw form of the data, which must then be translated and put in the cache. The translation and cache routines may be effectively null and do nothing.

The comparable put routine is much simpler:

```
adaptor_put_symb (name, symb_data, xlate_inverse)
   struct name_descriptor_s *name;
   struct symbolic_s *symb_data, *xlate_inverse();
{
   struct raw_symbolic_s *raw_symb_data;

   (*xlate_inverse)(symb_data, raw_symb_data);
   g_put_symb(name, raw_symb_data);
}
```

Again, the xlate_inverse may be a null routine if the tool is willing to use the symbolic data in the form used by G.

Finally, note that an adaptor may be as sophisticated as necessary for a tool, or it may not even exist. If there was no adaptor, the tool would call the Mini-G routines directly, and its interface to G, therefore, would be very primitive.

Acknowledgements. The authors wish to acknowledge the help of Lars Nyland whose insight into the workings of VIVID and how VIVID could be improved was invaluable. We also thank the Microelectronics Center of North Carolina for supporting and encouraging this work in many different ways. Finally, we cannot thank enough the members of the spring 1987 CPS265.2 VLSI Seminar. Because of their dedication, good sense, and experience, it is no exaggeration to say that this project never would have been completed had it not been for them.

References

[1] *VIVID 1.3: Designer Documentation, Volume 1.* Microelectronics Center of North Carolina, Research Triangle Park, NC, 27709, December 1986.

[2] *VIVID 1.3: Tool Developer Documentation, Volume 2.* Microelectronics Center of North Carolina, Research Triangle Park, NC, 27709, December 1986.

[3] *VIVID 1.3: Tool Developer Documentation, Volume 3.* Microelectronics Center of North Carolina, Research Triangle Park, NC, 27709, December 1986.

[4] Ackland, Bryan and Neil Weste. An Automatic Assembly Tool for Virtual Grid Symbolic Layout. In *Proceedings VLSI '83 Norway*, August 1983.

[5] Afsarmanesh, Hamideh, Dennis McLeod, David Knapp, and Alice Parker. An Extensible Object-Oriented Approach to Databases for VLSI/CAD. In *Proceedings of 11th International Conference on Very Large Data Bases*, pages 13–24, 1985.

[6] Bancilhon, Francois, Won Kim, and Henry F. Korth. A Model of CAD Transactions. In *Proceedings of 11th International Conference on Very Large Data Bases*, pages 25–32, 1985.

[7] Entenman, George and Stephen W. Daniel. A Fully Automatic Hierarchical Compactor. In *22nd Design Automation Conference*, pages 69–75, IEEE, 1985.

[8] Eurich, John P. A Tutorial Introduction to the Electronic Design Interchange Format. In *23rd Design Automation Conference*, pages 327–333, IEEE, 1986.

[9] Katz, Randy H. Managing the Chip Design Database. *Computer*, 16(12):26–36, 1983.

[10] Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[11] Ketabchi, Mohammad A. and Valdis Berzins. Modeling and Managing CAD Databases. *Computer*, 20(2):93–102, 1987.

[12] Lamport, Leslie. LaTeX*: A Document Preparation System.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[13] McLeod, Dennis, K. Narayanaswamy, and K. V. Bapa Rao. An Approach to Information Management for CAD/VLSI Applications. *Data Base Week: Engineering Design Applications*, 39–50, 1983.

[14] Morris, James H. and et al. Andrew: a Distributed Personal Computing Environment. *CACM*, 29(3):184–201, 1986.

[15] Nyland, Lars. Improving Virtual Grid Compaction Through Grouping. In *24th Design Automation Conference*, IEEE, 1987, Forthcoming.

[16] Ousterhout, John K. and et al. The Magic VLSI Layout System. *IEEE Design & Test*, 2(1):19–30, February 1985.

[17] Rey, H. A. *Curious George Rides a Bike*. Scholastic Book Services, New York, 1952.

[18] Rogers, C. Durward, Jonathan B. Rosenberg, and Stephen W. Daniel. MCNC's Vertically Integrated Symbolic Design System. In *22nd Design Automation Conference*, pages 76–81, IEEE, 1985.

[19] Rosenberg, Jonathan B. CAD Tools for Mask Generation. In *Summer School on VLSI Tools and Applications*, pages 1–64, ETH Institute for Integrated Systems, 1986.

[20] Rosenberg, Jonathan B. and et al. A Vertically Integrated VLSI Design Environment. In *20th Design Automation Conference*, pages 31–38, IEEE, 1983.

[21] Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1986.

[22] Sullivan, Neil and Jonathan Rosenberg. The Cookie Cutter Algorithm for Handling Mixed Hierarchy. *ACM SIGDA Newsletter*, forthcoming, 1987.

Appendices

A Glossary

ABCD. A Better Circuit Description language. Developed at the Microelectronics Center of North Carolina and Duke University as an ASCII description of symbolic layout.

Biv. The compactor for Prism. See section *3.4 Data Structure Formats* and appendix *C The Compactor — Biv.*

browser. A tool, incorporated into Prism, for seeing the hierarchy of a chip and for providing an ASCII interface to cell data. The browser is called *Tom.*

CAzM. A circuit-level simulator developed at AT&T Bell Laboratories and the Microelectronics Center of North Carolina.

Caesar. A mask-level circuit editor. The name of the language used by this tool is also called *Caesar.*

CIF. CalTech Intermediate Format. A language for describing mask-level circuits.

Curious George. A cute little monkey, who always gets into mischief and is looked after by The Man In The Yellow Hat.

EDIF. Electronic Design Interchange Format. A standardized language for describing circuits in mask, symbolic, nets, and other forms.

FACTS. A timing-level simulator developed at the Microelectronics Center of North Carolina.

G. The database server of Prism. See section *3 Organization Of G.*

Magic. A mask-level circuit editor developed at U. C. at Berkeley. The name of the language used by this tool is also called *Magic.*

MTF. The Master Technology File system of VIVID and ancestor of Otis. See section *3.8 Introduction to G Technology Information System.*

Otis. The technology environment of G, developed from the Master Technology File (MTF) system of VIVID. See section *3.8 Introduction to G Technology Information System.*

Prism. The second generation, symbolic layout system, derived from VIVID, with major components Roy, G, Biv, and Tom.

Rex. The data structure used by the compactor in Prism. See section *3.4.2 Mask and Compaction Data Structures.*

RNL. A switch-level simulator with a Lisp interpretor that we just might steal.

Roy. The interactive, graphical editor for Prism, which edits leaf cells and floorplans with mixed mask and symbolic cells. See section *3.4.1 Symbolic Data Structure* and appendix *B The Interactive Editor — Roy.*

Spice. A circuit-level simulator developed at U. C. at Berkeley.

Tom. The browser for Prism. See appendix *D The G Browser — Tom.*

VIVID. Product of the Microelectronic Center of North Carolina for symbolic layout.

VMR. Virtual mask rectangle. Part of the Rex data structure for the compactor, Biv. See section *3.4.2 Mask and Compaction Data Structures.*

B The Interactive Editor — Roy

B.1 Improving on ICE

Roy is an interactive, symbolic editor provided with the Prism system. It is not a mask-level editor; Magic is expected to provide that function. Roy is based on the ICE editor from VIVID but will incorporate many enhancements.

One enhancement is the ability to display cells in both symbolic and Rex formats, allowing cells that were generated outside of the editor, and have no symbolic description, to be displayed. Roy has two display modes. The first displays all cells in symbolic format. Any cells that do not have symbolic data are shown as bounding boxes. The other mode displays all cells in Rex format. All cells have a Rex format. If a mask-level cell is brought into the system, a translator converts its data into Rex so it can be fit into the design, and if a cell is designed in Prism, its Rex is the result of a compaction. If, during a Rex display, Roy finds a symbolic cell with no corresponding Rex, that cell is shown as just a bounding box.

Another enhancement over ICE is the use of a more efficient two-dimensional data structure to store circuit primitives during editing. The data structure is a small, fixed size array combined with linked lists. The array defines a window on the circuit that is being edited. This window covers the area that the user is currently working on. When the user moves the editing area, this window follows. The array holds pointers to the linked lists, and they hold all the circuit primitives that are currently contained in that window.

The purpose of the data structure is to speed up the editing operations of inserting and deleting primitives without using much memory space. The array provides constant time access to points in the window, and access to primitives is linear. Although a linear search for a primitive seems long, experimentation has shown that each point in the user's window contains an average of only one and a half primitives. The array is big enough to hold most leaf cells entirely inside of it.

B.2 Interactions with the Compactor

Since Roy is already an interactive graphical editor, there is one graphical responsibility of the compactor that is logical to incorporate into it. Since compaction is incremental and guided by the user, the user must be able to indicate the order in which cells should be compacted. This order can be clicked out of the floorplan in Roy. The ASCII version of the ordering file can be directly manipulated by Biv. See appendix *C.3 Compaction Ordering*.

C The Compactor — Biv

C.1 Introduction

Biv is viewed by the rest of the system as a compaction controller. Biv is designed as such so that different compactors can be connected to it. This way, integration into the rest of the system will not be necessary. A compactor only has to talk to Biv. The compaction engine that is initially integrated into Biv is the current leaf cell compactor from VIVID. Biv is responsible for coordinating the use of the compaction engine in different methods of compaction.

The job of Biv is to take unmodified Rex data structures in, process them in some way, and return compacted Rex data structures. An incremental approach to compaction is used, as the goal is to compact the minimum number of cells at one time. Biv has its own window, through which it communicates with the user. The designer uses this window to indicate what to compact, and Biv uses this window to communicate problems arising in compaction to the designer.

C.2 Compaction to Constraints

Compaction is constraint-based. The compactor from which Biv is derived does only worst-case environment compaction, meaning that if a cell is to be compacted, all the instances of that cell are viewed and all the design rule interactions with neighbors are considered for a compaction environment. The advantage is that time is saved since there are fewer compactions to do, but a price is paid in layout size.

Constraint-based compaction uses the environment of only one or possibly few instances of a cell for the compaction. The constraints are easily generated from examination of neighboring cells. Constraints include pin positions, cell corners (virtual pins), design rules, and worst-case environment. They can also be width or height, but these can be simulated by using corners. Other possible forms of constraints are for primitives to be equal size or equal distance from some object.

Constraints are easily calculated in the following manner:

```
for (every abutting mask-level cell) {
   Examine pertinent side.
   for (every non-empty grid point) {
      Generate an absolute position for that constraint.
   }
}
```

C.3 Compaction Ordering

The compaction process is incremental to avoid compacting an entire chip every time a small change is made in the design. The goal is to do as little compaction as possible, while ensuring that *everything* is compacted that is affected by a design change. The incremental compaction process is user-guided meaning that the user can interactively specify the order in which cells will be compacted and the method that will be used to compact these cells. The order can be specified in Biv or it can be clicked out on the floorplan while in the editor (see appendix *B.2 Interactions with the Compactor*).

The compaction process should be deterministic. Deterministic compaction means that if a user has interactively specified an ordering that results in a certain compaction, Biv should be able to store that ordering and regenerate the *same* compaction from scratch — without user intervention. A linear ordering of the cells is the most appropriate ordering for compaction. In this ordering, which has been described in section *3.4.3 Compaction Ordering Data Structure*, the method of compaction that was used to compact a cell is also specified. This ordering must be a complete transcript of how a compaction completed. For example, if a cell is compacted and then much later it is pitchmatched to some other cells, the ordering must reflect the fact that the cell was compacted at a certain position in the order, and then later pitchmatched at another position.

The selection of a compaction ordering is rule based. Initially the rule provider is the user. Biv provides an interface for the user to specify this ordering. The interface also allows the method of compaction to be specified. Currently three methods are identified:

1. compaction to constraints,
2. pitchmatching, and
3. compaction to worst-case environment.

The user can easily specify large groups of cells to be compacted to worst-case environment. By using this worst-case scheme, the user can actually choose the environments of particular instances of a cell for a compaction, and other environments can be chosen for another compaction. In this manner, the user can specify the "badness" of the worst-case environment compaction.

Biv provides as much guidance as possible to the user. For example, if the user requests that cell FOO be pitchmatched to cell BAR and this would result in many other cells needing to be pitchmatched, Biv would inform the user. After there is a better understanding of how to specify a good compaction ordering, an expert, rule-based system to generate the compaction ordering can be built.

C.4 Compaction Controller Algorithm

Presented here is the basic algorithm that Biv uses to guide compaction. This algorithm allows the user to make three types of requests:

1. Compact cell FOO, and recompact anything that cell FOO affects in the ordering.
2. Compact cell FOO (and a limited number of cells following cell FOO in the ordering), and flag everything in the ordering after it as needing to be checked for recompaction.

3. Compact cell FOO to the null environment, meaning that the environment consists only of cell FOO (it may be a composite cell), and do not recompact or flag anything else.

Note that these three possibilities do not take into account constraints or pitchmatching. This list simply illustrates how the compaction itself can be employed.

When using method three above, the ordering consists only of cells in subtrees where FOO is the root. The storage of the newly compacted data is handled by G, and it is discussed in section *3.3 Namespace Implementation.* The Biv compaction guidance algorithm follows.

1. Biv receives a request to compact cell FOO. (The type of request does not matter.)
2. If there is not already an ordering from G, request it.
3. Starting at the beginning of the ordering, look for the first cell that has been flagged. A cell is flagged for one of two reasons: either its symbolic has been changed and it has not been compacted since then, or it was flagged during an earlier compaction. Go to the next step, starting with the first flagged cell. Since the cells are being compacted in order, it is guaranteed that we can regenerate this compaction later. The user has the option to skip this step, simply compacting all cells in the order starting with FOO.
4. Request that G generate the constraints on this cell, and check to see if the current compacted version of this cell meets those constraints. If it does, unflag the cell and go step **9** of the algorithm; otherwise continue.
5. Only perform this step if cells following the current cell in the order have not been flagged. Request that G generate the constraints that the current compacted version of this cell is exerting on its neighboring cells. Check those neighboring cells, and flag them if the new constraints no longer match to the cells.
6. Compact the cell according to the method specified in the ordering.
7. If the compaction is successful:
   a. Write out the new Rex to G.
   b. Unflag the cell.
   c. Only perform this step if all cells following the current cell in the order have not been flagged. Request that G generate the constraints that the new compacted version of this cell is exerting on its neighboring cells. Check to see if these constraints are different from those of the previous version. If they are, flag all cells after this one in the ordering as needing to be checked. If they are the same then do not do any flagging.
8. If the compaction is unsuccessful:

a. At this point the algorithm halts, and Biv reports to the user exactly where the compaction failed and, if possible, why it failed. The user then has several options to try to remedy the problem: 1) change the ordering of the compaction, 2) move the cells apart and attempt to route them together, 3) try pitchmatching if it has not been tried already. Biv should eventually give the user some guidance as to which method seems appropriate.

9. Find the next cell in the ordering that has been flagged. Go to step 4 for this cell. If no cells remain in the ordering that are flagged or none remain in the specified range then halt.

C.5 Possible Problems with Compaction Algorithm

This approach is obviously fairly complicated, and the question arises as to whether finding a compaction ordering that will work will be too frustrating or impossible for the user to find. If all the cells in a chip abut together, then yes, this task of finding an ordering may be almost impossible. But, in normal VLSI design a large number of cells rarely abut together. Rather small units, routed together, are the rule. This approach will make finding a compaction ordering a reasonable task, but it points out the need for good routing facilities in Roy.

Keep in mind the fact that the benefits being reaped from interactive compaction process should be large enough to justify the work a designer must do to find the ordering.

The other possible problem is compaction concurrency. If two designers are working concurrently, and each wants to compact a cell that is geographically close to the other, it is possible that these compactions could affect each other. Also, two designers might try to change the compaction ordering simultaneously. There are two concurrency controls: 1) relaxed mode — let both designers know that they are affecting each other's work and let them work it out, and 2) strict mode — only one designer may compact something in the chip at any one time. The general concurrency problem is discussed in section *3.5 Avoiding Concurrency Problems.*

D The G Browser — Tom

A browser, Tom, is provided to allow the user to interact directly with G, thereby bypassing any other tools. The browser allows the user to examine the namespaces and perform high level manipulations on them. Tom interacts with the user through a window on the console screen.

Commands are provided to get a listing of all packages in the system, all classes in the packages, and all the versions and instances of a class. These listings are in an *ls -l* type format.[7] For a package the listing shows the technology associated with the package, the date of creation and last modification to the package, a list of classes in the package, and a list of users with read or write permission to the package. For a class the listing shows whether the class is public or private, what data formats are checked out, the date of creation and last modification of each data format, and a listing of all the versions and instances of that class. For each instance of a class the listing shows the date of creation and last modification of each data format stored with the instance and what data formats are checked out.

The user can do direct modifications on the data stored with a class or instance using an ASCII interface. This interface implements many of the Unix commands for text files, including a standard visual text editor. The browser translates data in G format into ASCII, allows the user to manipulate the data, and then translates the new data back into a G format.

With the browser, a user is be able to modify the package-name path (see section *3.3.2 Name References*). The user can add and remove package names from the path.

The browser also provides a way of reinstating old versions of data as the most current version.

[7]The Unix *ls -l* format displays information about files and directories, including file type; read, write, and execute permissions; size; ownership; and last modification date.

E Simulators

Chip simulation is a vital part of any design system, whether that system be mask-level, symbolic-level, or some other. Although there is no simulator that is a basic component of Prism (as the editor, compactor, and database server are), G supports a simulation data structure for a switch-level simulator, which operates on symbolic layout.

There are, however, many different circuit-level, switch-level, and logic-level simulators that could be integrated into Prism, each of which uses a different data structure form. For this reason, a net-list extractor specific to the simulator is needed between G and the simulator. The extractor will generate whatever form of data is required by the tool, and it should be part of the Adaptor created for the tool. Prism will support at least FACTS, CAzM, RNL, and SPICE.