Chapter 8

System Issues and Software Engineering

8.1 Introduction

The physical design of integrated circuits is an extremely arduous task. In order to reduce the difficulty, physical design is divided into simpler stages. The physical design of an integrated circuit will typically involve all of the stages presented in Figure 1.4. However, the flow of data and the interaction between stages will vary depending on the technology and design style. Most of today's CAD systems put the burden on the users - requiring them to control the execution of stages. However, the interactions between stages are often complex and prone to human error. In this chapter, we present an automatic program and data flow system which maintains the proper program parameters and sequences. This system eliminates any human error and is able to react to changes allowing incremental design.

Even with automatic design management, each phase poses a formidable challenge. In addition to the complexities of algorithmic development are *software engineering* issues: the implementation and testing of the software. The physical design is divided to create smaller more manageable pieces. If each piece has a well defined input and output, implementation and testing are more manageable. In this chapter, we also present the many software engineering techniques used to alleviate the software implementation and maintenance problems.

8.2 Previous Work

8.2.1 Design Management Systems

Recently, several systems have been proposed to maintain design methodologies. The Flowmap tool, developed at Delft, features both methodology coding and design state tracing [223]. Activation is manual, and parallel execution is not supported. Another similar system was presented by van den Hammer [230]. A dataflow graph is constructed and used to supply information about the state of the design. The system supports versioning and automatic activation, but it does not offer parallel execution.

Berkeley's VOV system captures working methodologies by extracting the methods found in repeated examples [27]. However, it does not support sophisticated parametric methods. It features only basic automatic parallel execution.

The Methodology Management System supports both methodology definition and concurrent execution [3]. Its control mechanism is a procedural definition. The load balancing scheme is fully distributed.

The Flow system, based on our work in this chapter, defines the design methodology using a dataflow graph and allows concurrent execution [106]. It does not migrate large jobs from overloaded to idle workstations. In addition, Flow loses job execution data when the controlling workstation crashes or workstation network is interrupted.

None of the above systems control the graphics environment.

8.2.2 Software Engineering

Software engineering, or an engineering perspective on software implementation and testing, is still in its development stages. Performing less than expected, today's software is often produced late and over budget. It has been shown that programmers spend more than half their time on maintaining software [75]. In addition, the most difficult part of

maintenance is understanding the original programmer's intent [135]. Without a systematic method to control the implementation, testing, and maintenance of large software systems, software will continue to be unreliable.

Software engineering began with the introduction of structured programming. Although the elements of structured programming have evolved since the early 1960s, the concept first achieved widespread recognition due to Dijkstra [51]. Dijkstra commented that there should be a close correspondence between the program text and its execution flow. A program should read top to bottom without any jump (GO TO) statement. Knuth countered that the presence or absence of the GO TO statement is a poor measure of a good program [118]. Knuth had published a series of books extolling the *art of programming* [117]. Myers offered that structured programming is "...the attitude of writing code with the intent of communicating with people instead of machines." [158]

Structured programming is a property of the source code for a program; it is not a methodology. The earliest software methodology was *stepwise refinement*, proposed originally by Dijkstra and later refined by Wirth [52][237]. In stepwise refinement, the definition of a program is iteratively refined from a high level description into lower-level language statements until the entire program has been implemented in the programming language. The advantage of this method is that the goals of the specifications are always maintained. Its disadvantage lies in the discarding of the evolutionary code. This is a major problem and it has only been recently addressed. Knuth has proposed the WEB system where various levels of the program's abstraction are stored as different views of the program [119]. Each view can generate either code or documentation. Such systems are still under development.

In conjunction with stepwise refinement, the theory of modules was developed. The theory of modules and data abstraction began from work by Hoare [93] and was extended to functional semantics by Gannon, Hamlet, and Mills [70]. They proved that if each mod-

ule encapsulates a function that can be verified in isolation, its internal details can be ignored in a proof of its use. This allows large software systems to introduce modules which divide the program into manageable pieces. This is now a tenet of software engineering.

Myers presented a collection of rules and guidelines for enhancing software reliability in 1976 [158]. One important observation, to never micro optimize code, was reiterated by Bentley in a series of books on programming [9][10][11]. Both authors suggested that the optimization is best performed after the program is correct; it is important to tackle *bottlenecks*, the 5% of the code that takes 95% of the time. Both espouse readability over efficiency in non critical sections of the program. They suggested algorithmic improvements as the best optimization method.

In addition, Myers classified testing strategies into six categories: bottom-up, top-down, modified top-down, sandwich, modified sandwich and big-bang testing [161].

In *bottom-up* integration, only the terminal modules are tested in isolation. The next modules to be tested are ones which directly call these modules. This is repeated until the top module is reached. Bottom-up testing requires a *module driver* for each module. A module driver is a method of providing test case input to the module interface. This may be furnished by writing a small driving program for each module or using a module tester which specifies test cases in a special language.

Top-down development reverses the testing order; only the top module is tested in isolation. After the top module is tested, those modules which are directly called by the top module are merged and tested. This is repeated until all modules have been combined and tested. In top-down development, the need for module drivers is replaced with the need for *stub modules.* Stub modules are necessary to simulate the behavior of modules which have yet to be merged. In most programs, true top-down development is impossible since I/O

functions must be implemented before the top level can input test cases. Top-down development allows the feasibility and correctness of the entire program to be explored at an early stage, a strong advantage.

Top-down testing makes it impossible to check error conditions in submodules since top modules only furnish valid data to the lower modules. To overcome this problem, *modified top-down* testing additionally requires each module to be unit tested in isolation before it is integrated into the program. Although this solves the problem, both module drivers and stubs are needed for each module.

Sandwich testing is a compromise between bottom-up and top-down testing. In this method, bottom-up and top-down testing are performed simultaneously. The particular program determines the point at which they merge.

Modified sandwich testing combines bottom-up and modified top-down testing. It solves the problem of detecting error conditions in the upper half of the program.

The *big-bang* approach is probably the most common approach to integration of modules today. In this method, each module is unit tested in isolation before all are integrated at once. This method has many disadvantages. Both stubs and drivers are needed for each module. Modules are not integrated until late in the testing cycle, allowing serious module interface errors to remained undetected. In bottom-up and top-down approaches only one module is added at a time. If an error occurs, the latest module becomes the prime suspect. However, since big-bang testing is not incremental, debugging becomes problematic.

Table 8.1 shows a qualitative comparison between testing approaches. The table also shows Myers's ranking of the integration methods with modified sandwich ranking as the

best and big-bang as the worst. For the most part, the TimberWolf system was developed using the bottom-up and modified sandwich methods.

Weight	Property	Bottom up	Top down	Modified top down	Big Bang	Sand-wich	Modified Sandwich
3	Integration	Early +	Early +	Early +	Late -	Early +	Early +
3	Time to a basic working program	Late -	Early +	Early +	Late -	Early +	Early +
1	Module driver needed	Yes -	No +	Yes -	Yes -	In part	Yes -
2	Stubs needed	No +	Yes -	Yes -	Yes -	In part	In part
1	Work parallelism at beginning	Med.	Low -	Med.	High +	Med.	High +
3	Ability to test par-ticular paths	Easy +	Hard -	Easy +	Easy +	Medium	Easy +
2	Ability to plan and control the sequence	Easy +	Hard -	Hard -	Easy +	Hard -	Hard -
total		+6	-1	+4	-3	+4	+7

Table 8.1 Myers' weighted comparison between test strategies. From [160].

Carey and Bendick discuss a method to segment a complex system into smaller functionally oriented sections called *builds* [25]. A build is a set of *threads* or execution paths through the system. Each thread is described as a transformation from user input to output specifications. Since the threads are derived from the user specifications, thread modules test the user specifications indirectly. Unfortunately, the number of paths through the system grows exponentially with the number of programs. Nevertheless, this technique has been successfully applied to several large military systems.

Software systems evolve to meet the changing needs of its operating environment. Old modules may be expanded beyond their original function, and new modules may be added. *Software maintenance* is the process of modifying and retesting the software. *Regression testing* is a testing methodology applied after the modifications have been

made. Regression testing involves testing the modified program with test cases to re-establish confidence in the program. Studies have been performed to determine typical programming errors during software maintenance [136]. Common errors include interpretation, wrong function, extra function, missing function, and interface errors.

Another software methodology is fast prototyping. Fast prototyping follows the evolution of a top-down system. But unlike top-down development, only a fraction of the program's functionality is constructed. Its benefit is to explore the design space and prove the feasibility of questionable parts of the system. After the feasibility study, the prototype is discarded and an entirely new system is created using the knowledge from the prototype. Unfortunately, the prototype often gets *hacked* into the production tool. In this methodology, the programmer must be forced to discard the prototype.

While the Computer-Aided Software Engineering (CASE) tools of today offer graphics, specialized editors, code navigational aids, and integrated debuggers, they do not enforce a methodology needed to create reliable software [62]. For this reason, the implementation of the TimberWolf system focused on the methodologies of Myers and Bentley.

While advances in software engineering should eliminate the current software methodologies, another alternative is not viable today. Still, progress is being made. Work has been done towards automated extraction of reusable programming components [2], and error localization methods have become semi-automated [206]. In addition, various methods have been proposed for code maintenance, including transformation systems which increase reliability [18][189]. In the future, software reliability will become an engineering methodology.

8.3 System control: twflow

In the TimberWolf system, the flow of data and the interaction between stages is controlled by the master flow program called *twflow*. It maintains both the sequence of place

and route programs and the flow of data between programs. The program *twflow* creates a flow chart in the graphics window (as shown in Figure 8.1) and automatically shows the current status of the layout process by highlighting the current node in the flow chart. A

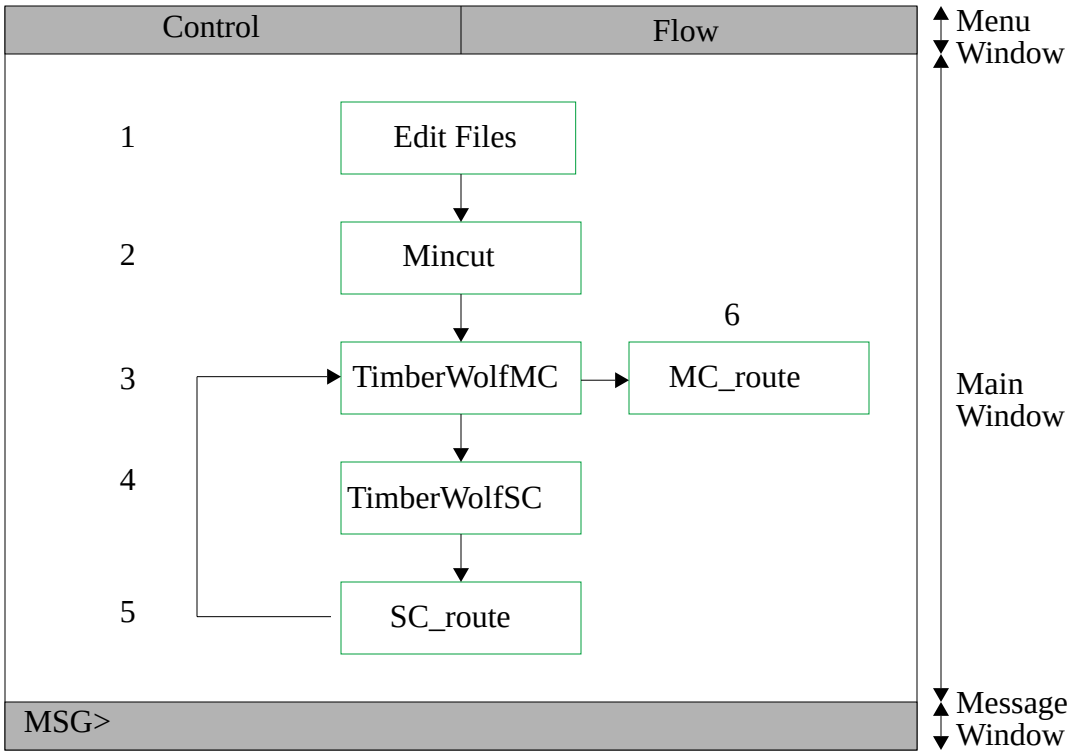

Figure 8.1 The TimberWolf graphical interface.

graph is used to describe the design flow. The programs or shell scripts in the flow are nodes in the graph (flowchart), and the edges between the nodes are valid program execution sequences. The program uses file dependencies in order to determine if a program or shell script needs to be executed, similar in function to the Unix *make*. This allows incremental design; only the necessary programs are executed.

The program *twflow* not only controls the sequence of programs to be executed, but also functions as the master control for the X11 graphics [192]. Figure 8.1 also shows the X-window interface maintained by the *twflow* program. It consists of three windows: the menu window which controls the pull-down menus, the main window where the current

state of the design and/or flow is depicted, and the message window which queries and informs the user of the current status of the design.

The intent was to design a simple interface that was flexible, maintainable, and portable. Flexibility was achieved by not allowing any program to use X11 routines directly. Instead, all graphics requests are handled by an intermediate layer of routines which handle the translation between user coordinates and pixel coordinates. These routines perform the actual requests to the X server. In this way, all programs are written in the user's coordinate system regardless of the graphics system used. If another graphics package becomes available, only the library of intermediate functions needs to be changed.

The graphics system model is shown in Figure 8.2. The twflow program controls access to the X server for all programs. This method creates a modular environment where each program is responsible for drawing data to the screen. This is in contrast to the current commercial CAD systems which link all programs into a single executable program. Although a single executable eliminates most of the redundancy in the graphics routines, the program size typically explodes to between 25 and 64 Megabytes. The program loading stage can take as long as 20 minutes on a state-of-the-art workstation! In addition, a large core memory (RAM) is required to avoid time consuming *disk swapping*. Many of the current systems require a minimum of 64 Megabytes of RAM to operate efficiently. All of this is necessary before any place and route algorithm is executed!

In the TimberWolf system, an executable program rarely exceeds 1 Megabyte (the Motif version maximum is 2 Megabytes). The result is an instantaneous loading of programs. The amount of RAM needed depends only on the problem to be solved, not a systems requirement. The amount of redundant graphics code is mitigated by using a common graphics library. This library handles all screen functions and interrupts. Each program is only responsible for drawing its own view of the physical design. In this way, the TimberWolf system achieves both modularity and flexibility.

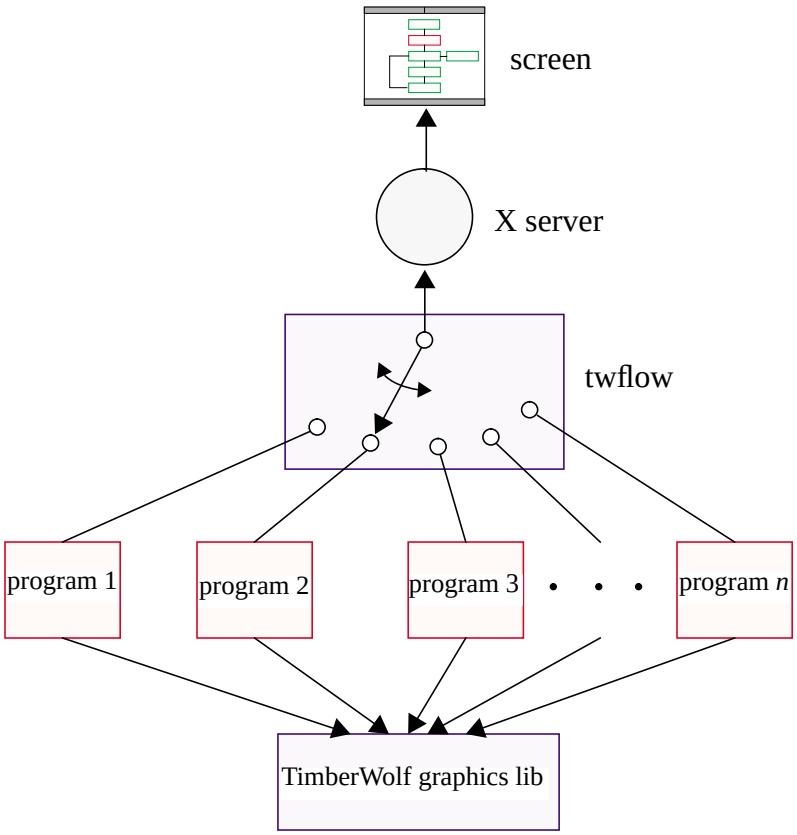

Figure 8.2 The graphics interface. *twflow* allows only one program to talk to the X server at a time. All programs share the same graphic library functions.

In order to make the system flexible from the user's perspective, *twflow's* sequence of programs is not predefined; instead, it is defined by an ASCII input file. A portion of the *twflow* input format is shown in Figure 8.3.

Each program object or *pobject* is numbered as shown in Figure 8.3. The program *TimberWolfMC*, for example, is program number 4, and it may be executed in a valid sequence after program 3 or program 6. The path of the program may be changed from its default place, and the user may choose how he wishes to draw the object. Next, each edge is specified as to how it is executed, and the files it depends on. The $ is a special string substitution character which stands for the design name. For each edge we list the input and output files that are required and generated by the program. By looking at the time

```
numobjects 6
                .
                .
                .
pobject TimberWolfMC 4: 3 6
path:
drawn: 450 450 750 550
edge 3:
ifiles: $.mlib $.mckt $.mcon
ofiles: $.mdat $.blk
args: -w @WINDOWID $
args debug: -dv $
args nographics: -b $
drawn:      600 550 600 650      600 550 620 570      600 550 580 570

edge 6:
ifiles: $_io.mcktl $_io.mlib $_io.mcon
ofiles: $_io.mdat $_io.blk
args: -w @WINDOWID $_io
drawn:      250 500 450 500      250 100 450 100      250 100 250 500
            430 520 450 500      430 480 450 500
                .
                .
                .
```

Figure 8.3 A portion of the *twflow* file.

stamp of each of these files, we can determine if this program needs to be executed. In addition, each edge allows different argument lists to be passed to the executed program. The keyword *WINDOWID* tells *twflow* that the child process is capable of inheriting the graphics interface.

It is very easy to modify the sequence of programs for particular applications. For example, another flow file could be generated which allows the addition of front-end and back-end translation programs, or perhaps another would omit the *WINDOWID* key words for batch mode processing.

8.3.3 Parallel execution

The TimberWolf system supports concurrent execution of the program data flow graph. Each program is scheduled to run on a workstation linked through a local-area network (LAN). Parallel execution at the coarse level is performed using a *remote shell* (/bin/rsh) on a network workstation. A list of all valid nodes is supplied as input to the flow program. The goal is to create a centrally controlled fault tolerant system. This is achieved by

detaching the remote process from the controlling workstation. In the UNIX environment, a remote *C*-shell is executed on *node_remote* by

```
/bin/rsh node_remote (remote_job_csh arg1 arg2... argn) > /dev/null &
```

In this way, the remote shell dies once it has spawned the proper processes on the remote workstation. The remote process is now completely autonomous if the remote job first copies all necessary input files to the local workstation storage. A network failure will not affect the computation at this point.

The remote job also must create *checkpoint* files and *locks* to inform the controlling workstation of the status of its processes. The current state of a program is saved in a checkpoint file in order to allow a restart of the program from the saved state. A lock on the remote input and output files prevent other processes from interfering with the execution. The controlling workstation bases its decisions on the *status checks* of the workstations in the network. A status check is executed on the remote workstation by inspecting the checkpoint files and file locks. The results are piped back to the controlling workstation. Status checks are executed periodically. If the tasks on the remote workstation are check pointed, load balancing is possible. A task may be rescheduled on a less loaded machine by killing the processes on the loaded workstation and rescheduling the task on another machine starting from the checkpoint. Even if the controlling workstation should fail, work which has been started will complete. When the controlling workstation returns to operation, work can progress forward again. In contrast, the current major CAD systems maintain constant communication channels. A single network failure can wreak havoc with these systems. This method overcomes these limitations.

8.4 Software Engineering

Although each of these techniques is by no means novel, taken together they have greatly reduced the number of errors in the TimberWolf system. The techniques can be

Software Development Aids

1. Library routines.
2. Abstract data structure.
3. Library module drivers.
4. Programs use the same data structures.
5. Defensive programming.
6. Peer review.
7. Suffixes for global and static variables. (Coding guidelines).
8. Memory manager.
9. Library debug facility.
10. System dependent code resides in library. (Separated graphics into library.)
11. Source Code Control System (SCCS).

Figure 8.4 Software implementation methodology.

divided into three categories: implementation, system and testing methodologies. Figure 8.4 lists the items which have increased productivity and quality during implementation. The single most important method is to *re-use* code. It is useful to bundle all data structure creation and utility functions into a library of reusable software. From the theory of modules: if a module is correct, its correctness in new programs is assured if side effects do not exist in the code. Likewise, if a problem is detected in a library module, the problem may be rectified in one place since code is not replicated. Figure 8.5 presents a partial list of library functions available in the TimberWolf system. All library functions are designed to minimize side effects as much as possible.

It is also important for all library modules to use abstract data structures. Abstract data structures hide implementation details from the programmer. It is a necessary condition for the theory of modules to be applicable. In the TimberWolf system, each library module

Module	Function
assign.c	linear assignment solver.
buster.c	tile decomposition algorithm - breaks figures into a set of tiles.
cleanup.c	error handler.
colors.c	standardize the colors drawn to screen.
deck.c	builds FIFO and LIFO data structures.
dialog.c	draws a dialog box on the screen.
draw.c	graphics I/O functions.
dset.c	disjoint set data structure.
file.c	file I/O including file compression and decompression.
graph.c	graph utilities include depth-first search, shortest path, and Steiner tree algorithms.
grid.c	forces data to a user specified *x, y* grid.
hash.c	hash table data structure.
heap.c	binary heap data structure.
interval.c	interval tree data structure.
list.c	doubly-linked list data structure.
matrix.c	matrix manipulation routines including multiplication, and LU decomposition.
menus.c	builds menus for graphics interface.
message.c	writes messages to the screen.
mst.c	calculates the minimum spanning tree for a set of points.
okmalloc.c	memory manager which stores the time of allocation.
program.c	program initialization routines.
project.c	computational geometry projection routines.
quicksort.c	quicksort utility.
radixsort.c	radix sort utility.
rand.c	random number generator utilities.
rbtree.c	red-black tree data structure.
select.c	selects *k*th item in linear time without sorting.
set.c	set data structure implemented as an array and linked list.
stat.c	statistical functions including mean, standard deviation, and variance.
stats.c	returns run time statistics for program.
string.c	string utility routines including a string parser.
svd.c	singular value decomposition routines.
system.c	system routines including copying, and moving files and forking new processes.
time.c	returns the system time.
timer.c	sets an elapsed time counter.
trans.c	geometric transformation routines.
xparse.c	command line parser.
ydebug.c	control for debug utility.
ylex.c	replacement for lex.

Figure 8.5 Partial list of library modules and their function.

is required to furnish a driver. The module driver performs two important functions: testing of the module for errors, and documenting the module interface for new users.

Many programs within the TimberWolf system use essentially the same data structures. However, none of the programs use *exactly* the same data structures. Replication of code leads to errors and creates a maintenance headache. The amount of duplicated code can be minimized using the conditional compile features of the *C* language [109]. In this method, the individual programs embellish a minimal data structure. The individual data

structure is built using a set of library functions which use callback functions to customize the data structures. An example of this method is shown in Figure 8.6.

```
/* program_parser.c */
#include <tw/structure_int.h>
#include <globals.h>
#include <tw/structure.h>

VOID init_inst(YINSTPTR inst_p)
{       /* count the number of signal pins for an instance */
        YGPINPTR pin;       /* traverse pins of instance */
        inst_p->numpins = 0;
        for(pin=inst_p->inst_pins;pin;pin=pin->next_inst){
              inst_p->numpins++;
        }
} /* end init_inst()*/

VOID build_data_structure(void)
{       /* call back routines to customize data structure */
        static VOID (*callbacksL[NUM_CALLBACK])() = {
                        init_inst,      /* ADD_INST_FUNC */
                        .
                        .
                        .
        };
        Ystructure_build(...,callbacksL,...);
)}
```

```
/* <tw/structure_int.h> */
#define USER_INST
```

```
/* <globals.h > */
#undef USER_INST
#define USER_INST  INT numpins;
```

```
/* <tw/structure.h> */
typedef struct yinstrec {
   char             *inst_name;
   INT              inst_xc;
   INT              inst_yc;
   INT              inst_orient;
   INT              inst_class;
   BOOL             inst_eco;
   YGPINPTR         inst_pins;
   YMODELPTR        inst_model;
                    .
                    .
                    .
   USER_INST
} YINSTBOX, *YINSTPTR;
```

Figure 8.6 Data structure customization. The data structure has an additional field called numpins. The routine init_inst counts the number of signal pins for each instance as the data structure is built. The <tw/structure_int.h> set the default USER_INST field to the empty set. The user may redefine the fields in <globals.h>. In this case, the field numpins has been added to the end of the struct yinstrec definition in <tw/structure.h>.

Another productive software technique is *defensive programming*. Defensive programming is based on the important premise that the worst thing a module can do is accept incorrect input and produce an incorrect but believable output. To rectify this problem, checks are placed at the beginning of the module to test the input for proper attributes and domain. In this manner, new users of reusable code are alerted to errors at an early stage. In the TimberWolf system, all inputs are syntactically checked using *yacc* and *lex* [193]. In addition, all data and attributes are checked for reasonableness where possible. All library functions must include defensive checks and functional prototypes before acceptance into the system.

In order for software to enter the common pool of library routines, an informal review of software is performed. An environment where peer review occurs on a daily basis is necessary for reliable software. Module interfaces and functionality evolve. It is important that close feedback among developers be established to avoid incompatibilities and duplicate work. Peer review also promotes new ideas through cross-fertilization.

Software reliability is enhanced through source code readability. Comprehension of software reduces the chance of error during maintenance. Readability is greatly enhanced through the use of software guidelines. Figure 8.7 shows the set of coding standards which we have found productive. Code readability is substantially decreased where global vari-

1. End global variables in G, i.e. globalG
2. End static variables in S, i.e. staticS.
3. Begin library functions with Y and module name, i.e. **Yrbtree**_init(...
4. Use meaningful variable names. (Avoiding extremely long and short names)
5. Avoid similar variable names.
6. Use parentheses to avoid ambiguity.
7. All macro (C preprocessor commands) are in all capital letters.
8. Use macros for return codes. [if(strcmp(strg, “test”) == STRINGEQ){]
9. Avoid C language tricks.
10. Do not ignore compiler warnings.
11. Ignore all efficiency suggestions until program is correct - fast prototyping.
12. Let the compiler optimize.
13. Never optimize C code. Tackle efficiency through efficient algorithms.
14. Avoid excessive comments. Comments should answer why not how.
15. Comment variables and their use.
16. Never use a variable for more than one purpose.
17. Be cautious of floating point arithmetic.
18. Always check for integer divide by zero condition.
19. Always add braces for if-then-else constructs. [if(...)**{**...**}** else **{**...**}**]

Figure 8.7 Our coding guidelines.

ables occur; the definition and initialization of the variable may occur in another module. In an effort to warn the reader of a global variable, the variable is emphasized by ending the variable name with a capital G. Likewise, *static* variables, or those variables which are defined for all functions within a module, are emphasized by ending the variable name with a capital S. To further improve readability, we prefix library routines with a capital Y and the module's name. This speeds recognition of routines outside the scope of the current module. The remaining guidelines should be self-explanatory. All guidelines are designed to increase productivity. Other stricter naming conventions such as Hungarian notation [107] have been tried but discarded as non productive. No indentation guidelines have been enforced since word processors such as EMACS allow reformatting to suit the readers' desired format.

Another method to capture errors at an early stage is to replace the system dynamic memory functions (malloc, calloc, realloc, and free) with versions written defensively. Also, a debug mechanism has been designed to localize common errors. All memory words are initialized to -1 in malloc and realloc if the size increases. This alerts the user to uninitialized variable errors. The value -1 has been chosen since it causes segmentation violations if used as a pointer variable. After the memory is released using the free function, the block of memory words are assigned to -1. Words that have been freed erroneously may be used as allocated; this assignment alters the data to display the error. The memory manager also stores the line and file where the memory was allocated. This is a useful device for finding *memory leaks* or places where memory should be released. The memory manager is designed to detect a common *C* language error - overstepping the bounds of an array by 1. The layout of the memory is shown in Figure 8.8. The size is stored at the beginning and end of the array. The memory can be verified with high proba-

bility by testing the two size fields for consistency. This method has discovered numerous memory errors with very little programming effort.

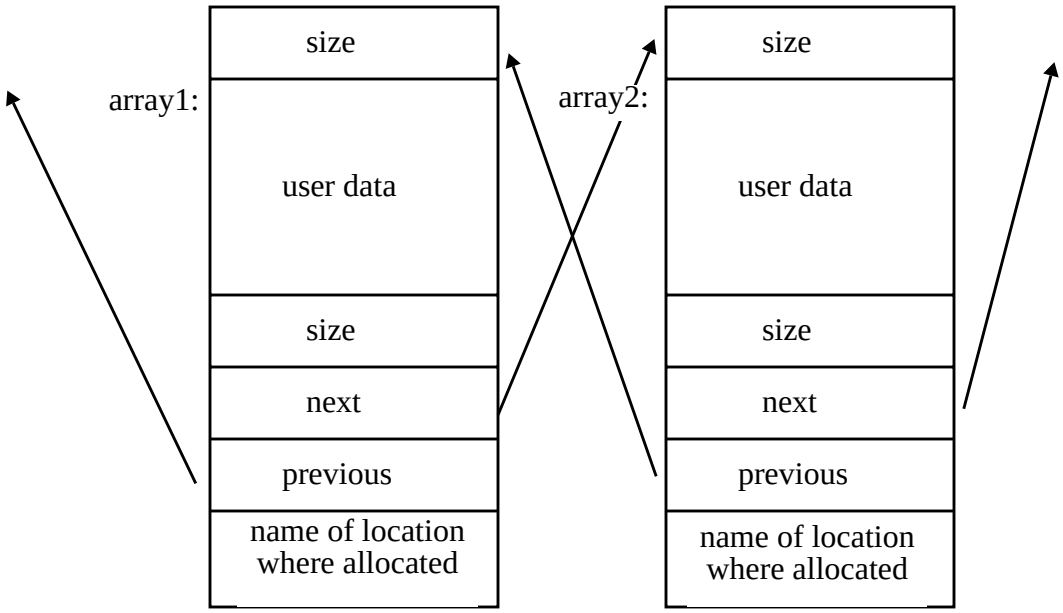

Figure 8.8 Layout of dynamic memory. Allocation location field is only activated during debug mechanism.

The novel debug facility in TimberWolf allows debug code to be activated without recompilation. Figure 8.9 shows a fragment of the debug module. Whenever the debug mechanism is activated, the *dbg* file in the current directory will be read (if it exists). This file maps each debug function to either on or off. Each name is entered into the *debug tree*, a balanced binary tree[1]. Whenever the program enters a debug block, the debug tree is searched for the name of the debug function. If the name is found in the tree, the Boolean state entered in the tree is used to control entry into the debug code. Otherwise, the debug function's name is entered into the tree, and its state is set to off. At the end of an execution, all debug function names encountered will have been entered into the debug tree. At any time, the contents of the tree can be dumped into the *dbg* file for the next debugging

1. A tree data structure was selected over a hash table because of the requirement to maintain an alphabetical ordering of the function names for human convenience.

session. Figure 8.10 displays an example of a debug function block. Using the macro pre-

```
BOOL Ydebug(routine)
char *routine;
{
   ROUTINEPTR data;                 /* the matched data record */
   ROUTINE routine_key;             /* used to test key */
   BOOL return_code;                /* return code so we can set in debugger */

      if(debugFlagS){
            return_code = FALSE;
            if(routine){
                  routine_key.routine = routine;
                  if(data = (ROUTINEPTR) Yrbtree_search(debug_treeS, (char *) &(routine_key))){
                        if(data->debugOn){
                              return_code = TRUE;
                        }
                  } else {/* add it to the tree */
                        /* when adding new routines to an existing file */
                        /* the default is for it to be off */
                        data = make_data_debug(routine, FALSE);
                        Yrbtree_insert(debug_treeS, data);
                  }
            } else {
                  fprintf(stderr, "No debug routine name specified here\n");
            }
      } else {
            return_code = FALSE;
      }
      return(return_code);
} /* end Ydebug() */
```

Figure 8.9 Debug code discriminator. This function returns TRUE if debug label has been entered in the debug tree (set true in the dbg file) and debug is enabled. Notice that the coding guidelines make the function easier to read. It is evident that debugFlagS, debug_treeS, and firstTimeS are static variables; we will need to look at the top of this module to determine their definition and initialization. TRUE and FALSE are macros. Yrbtree_insert is a library module - rbtree (red-black tree). The testing of the routine name is an example of defensive programming.

```
#include <std/debug.h>

VOID findcost()
{
      /* calculate incremental cost */
      cost = calc_incr_cost();

      D("twsc/findcost",
            INT net;
            INT checkcost = 0;
            for(net = 1; net <= numnetsG; net++){
                  checkcost += calc_from_scratch(net);
            }
            if(checkcost!= cost){
                  fprintf(stderr, "ERROR[findcost]: incremental cost %d != global cost %d\n", checkcost, cost);
            }
      );
      .
      .
      .
} /* end findcost() */
```

```
/* <std/debug.h> */
#ifdef DEBUG
#define D(name_xz, func_xz) if(Ydebug(name_xz)) { func_xz ; }
#else
#define D( name_xz, func_xz )
#endif /* DEBUG */
```

Figure 8.10 Example of debug function instance. Debug label is "twsc/findcost". This small fragment warns if the incremental cost calculation differs from an alternate method. This sanity check is enabled by the entering the line "twsc/findcost 1" in the dbg file and using the -d option on the command line to enable the debug mechanism. All debug code can be removed through the DEBUG conditional compile switch.

processing feature in *C*, all debug routines may be completely removed from the code to make a smaller executable[1]. More importantly, a mechanism which allows selective execution of debug code greatly improves productivity. This feature encourages programmers to write debug functions since the code will never be wasted; these functions are always there when a problem arises.

In order to port the system to various workstation platforms, system dependent code is restricted to library modules. This relieves the burden of system programming from the researcher and localizes possible problem areas during the porting process. For example, graphics code is localized in the library. The only interface to the X server exists in the graphics library; graphic system dependent code does not exist in any other module.

In a large system, source code control is necessary to track the software as it evolves. Although many source control systems exist, we chose the Source Code Control System (SCCS) due to its widespread availability.

The next methodology seeks to increase reliability at the system level rather than the module level. Figure 8.11 lists the system methodology guidelines. All of the guidelines

System Development Aids

1. Modular programs.
2. Separate environment for programs. Well defined inputs and outputs.
3. Small executable.
4. Ascii input and output files.

Figure 8.11 Software system methodology.

are based on the modularity of programs. Here the theory of modules is extended to the program level. If each *program* encapsulates a transformation, and it can be verified in iso-

1. This is rarely the case. The ability to selectively choose debug functions without recompilation is far more beneficial than costly. When the debug mechanism is off, the overhead is only a function call and a Boolean test.

lation, then its internal details can be ignored in a proof of its use. Each program is small because we allow only a single albeit complex transformation. In contrast, the CAD vendors have linked all place and route programs into a single executable violating modularity. Testing in such an environment becomes impossible.

All programs in the TimberWolf system transform a set of ASCII input files into a set of ASCII output files. A database scheme was abandoned because it prevents the observation of the transformation.This problem may be solved using databases with a versioning capability. However, the added complexity cannot be justified. The space advantage of a database is lost when the ASCII files are compressed automatically using the UNIX *compress* utility. In the TimberWolf system, the transformation rule facilitates testing of the system, and automatic file compression minimizes storage space.

The final methodology, testing, is crucial to the success of a software system. Testing is not proving the correctness of one's program but as Myers points out, "... the process of executing a program with the intention of finding errors." [159] The programmer's attitude toward his product is related to the principles of *egoless programming* and *cognitive dissonance* presented by Weinberg:

> "A programmer who truly sees their program as an extension of their own ego is not going to be trying to find all the errors in that program. On the contrary, they are going to be trying to prove that the program is correct-even if this means the oversight of errors monstrous to another eye. . . . the human eye has almost infinite capacity for not seeing what it does not want to see." [235]

For this reason, all final program testing is performed by a person other than the author. This technique eliminates a predisposition to ignore an error.

Testing Aids

1. Program testing cannot be performed by author.
2. Regression testing.
3. Testing using several compilers such as cc, gcc.
4. Testing on different machine architectures such as DEC, Sun, and HP.
5. Random seed is saved so you can reproduce the problem.
6. Debug routine library support.
7. Purify.

Figure 8.12 Testing methodology.

Figure 8.12 lists the important facets of testing a large software system. Regression testing in a large system is a necessity. Software systems are not stagnant; their pieces are subject to evolutionary changes. It it imperative to check the system once modifications are completed. In the TimberWolf system, worst-case test examples are designed for deterministic programs, and random test cases are created for stochastic programs. In all instances, the worst possible scenario is chosen.

Field testing is also exploited as a method to test the TimberWolf system. In this method, the system will be exercised not as a programmer would but rather as a integrated circuit designer requires. This catches specification problems and misconceptions about the nature of the problem. Human interface problems also become apparent in the field environment. Any error is reported to the program's author for maintenance.

Another valuable technique is testing with different compilers and platforms. In the TimberWolf system, we test on DEC, SUN, and VAX workstations. In addition, the code has been ported to Apollo, MIPS, Apple Macintosh, HP, IBM 386, and R6000, and IBM mainframe computers. We have utilized all resident compilers on these machines as well as the gcc compiler. Since machine architectures differ, the run time environments differ.

These differences often expose programming errors not detected on the architecture where the system was originally developed.

For stochastic programs, it is important to be able to reproduce the error once it has been detected. It is important that the program be written in such a way that the pseudo random number may be reseeded. In addition, the debug facility in TimberWolf facilitates conditional output describing the program's internal state. This information leads to quick diagnosis of possible programming errors.

Recently, a new CASE tool has been applied to the development and test of the TimberWolf system. The Purify tool helps the programmer discover common programming errors including memory errors. Tools such as this are invaluable in teaching novice programmers software engineering skills.

8.5 Conclusions

All of these techniques have contributed to make the TimberWolf system the industry's most widely applied university tool to be used without modification. After SPICE (an electrical circuit simulator), it is the most prevalent tool used in the manufacturing of integrated circuits. Such a record would not be possible without a systematic approach to implementation and testing. Software reliability techniques have drastically reduced the number of errors in the system as well as increase the productivity of all researchers.