MARCH 1995

# WRL Research Report 95/1

# Drip: A Schematic Drawing Interpreter

*Ramsey W. Haddad*

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There are two other research laboratories located in Palo Alto, the Network Systems Lab (NSL) and the Systems Research Center (SRC). Another Digital research group is located in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, WRL-2
250 University Avenue
Palo Alto, California 94301 USA

Reports and technical notes may also be ordered by electronic mail. Use one of the following addresses:

| | |
|---|---|
| Digital E-net: | `JOVE::WRL-TECHREPORTS` |
| Internet: | `WRL-Techreports@decwrl.pa.dec.com` |
| UUCP: | `decpa!wrl-techreports` |

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Reports and technical notes may also be accessed via the World Wide Web: `http://www.research.digital.com/wrl/home.html`.

# Drip: A Schematic Drawing Interpreter

**Ramsey W. Haddad**

**March, 1995**

## Abstract

This paper presents a design capture system in which schematics are translated into a procedural netlist specification language. The circuit designer draws schematics with a standard structured graphics editor that knows nothing about netlists or schematics. The translator program analyzes the structured graphics output file and translates it into a procedural netlist specification.

digital **Western Research Laboratory** 250 University Avenue Palo Alto, California 94301 USA

# Table of Contents

# List of Figures

## 1. Introduction

In the history of design capture systems [5], many of them are limited to being either entirely text-based or entirely schematic-based. This is very limiting in that there is no single best choice between these two methods. Typically, some portions of a design are best represented as text and others are best represented as schematics. For example, the low-level unique cells of a design crafted by circuit designers are usually best represented as a schematic. A quick test of this is to show a circuit designer the text-based equivalent design of such a cell and ask what it does. Her first step will be to translate the design into a schematic - only then can she can easily determine the function. This suggests that the schematic was the correct representation in the first place. On the other hand, cells such as control equations are more concise and easily modifiable in a textual representation. A good system must allow the designer to mix between text-based and schematic-based representations on a cell by cell basis.

Another important aspect of many designs is the frequent need to reuse an already existing cell, but with a few small changes. A poor CAD system would require the designer to copy an existing cell, rename it and modify it for the new use. When this happens often enough, the process becomes tedious and updating modifications becomes very difficult. Thus, a good system must allow the designer to specify cells in a *parameterized* manner [7], so that the CAD system can generate many variations from one basic design.

Once the parameterization of cells is allowed, the question arises as to how powerful a description language should be allowed to describe the parameterization. Anything short of a full programming language is awkwardly restrictive. Once the parameterization has the power of a full programming language, the designer is not only designing a circuit, but is also writing a computer program.

Combining all these ideas results in a system similar to one developed at Xerox PARC [1, 2]. With their schematic editor, parameterized cells are designed using the full power of a programming language. One drawback of such a system, however, is that it involved a fairly tight coupling of the graphics editor to the design capture system and used a large number of explicit but non-visible bindings between text and objects that could lead to frequently misleading schematics.

In the layout generation system [8] developed for our microprocessor design projects [4], we modify this approach. A design is represented by a set of procedures. When compiled and executed, they build an annotated hierarchical netlist as internal data structures. To augment this text-based approach, designers can use schematics as a graphical procedure language. The designer creates these schematics with a standard structured graphics editor. The drawing translator *drip* analyzes the schematics created with the graphics editor to generate the equivalent textual description. This approach is similar to that taken by *WireC* [6] and its predecessor, *WireLisp* [3, 10]. *Drip* has a more general evaluation mechanism than *WireC*, and uses a very different understanding of order of evaluation for objects, including an intuitive 2D ordering. This enables us to, for example, allow multiple procedure definitions in one schematic file and to separate the locations of the declaration of an icon and the definition of the procedure corresponding to that icon.

The mechanics and design issues surrounding the drawing translator are the focus of this paper. We will mention aspects of the overall layout generation system that *drip* is a part of only as necessary; another paper [8] describes this overall system more completely.

# 2. Basics

## 2.1. Simple Example

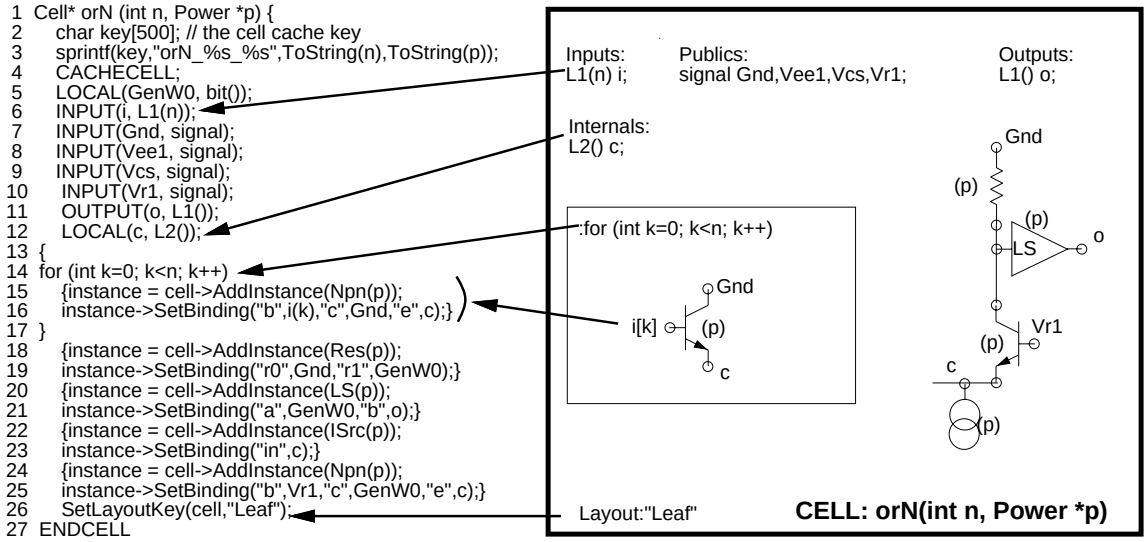


**Figure 1:** Code Generated for "CELL: orN"

Figure 1 shows a sample schematic and the code generated when *drip* analyzes it. The resulting procedure *orN* generates and returns the netlist for an *n*-way OR gate.

Most of the wires are declared in the schematic and hence also in the generated C++ procedure. The code that is generated from the declarations of "Inputs" (line 6), "Publics" (lines 7-10), "Outputs" (line 11), and "Internals" (line 12) is easily identifiable. One wire, connecting the bottom of the resistor to the *npn* transistor, is drawn but not declared in the schematic. Hence, *drip* assigns it a generated name and declares the wire (line 5).

For each of the 5 icons in the schematic *drip* generates a call to the appropriate netlist generation procedure for that subcell (lines 15, 18, 20, 22 and 24). It passes the parameter "p" to these subcell generators. *drip* generates code to bind the external wires of the sub-cells to the appropriate wires in the current cell (lines 16, 19, 21, 23 and 25).

But a number of issues raised in this simple example are not so clear: How does *drip* decide in what order to generate the different pieces of code? How does *drip* know what procedures to call to generate sub-cells? How does *drip* know which wires to bind to which terminals? What tradeoffs are made in answering these questions?

There are a number of philosophies that guided the design of the drawing interpreter.

- The graphic editors should not have to know any of the semantics of the drawing translator or the underlying language.
- The drawings should have no "hidden information": a human should be able to understand the drawing completely by looking at a paper copy.
- The drawing translator should be robust. That is, it should be fairly insensitive to minor changes to the drawing; two drawings that look the same to the naked eye should not have different interpretations.
- The drawing translator should be able to translate drawings from a number of graphics editors.
- The drawing translator should be strict: if some graphical situation is ambiguous, forbid it. This helps ensure the readability of the final graphic language.
- The drawing translator should behave in accord with people's intuition and expectations.

We only break these guidelines when useability or strong traditions demand.

## 2.2. Structured Graphics

One goal is for the schematic translator *drip* to be as independent of the drawing program as possible. As a first step, *drip* is an entirely separate program from our drawing program -- the generic *idraw* variant of *Unidraw* [9]. The only way that the two programs can communicate is through files. The schematic translator reads the output files of the drawing program, much as a compiler reads files written by a text editor.

To force independence, another goal is that the schematic translator be easily retargetable to other drawing programs. *Drip* ensures this by parsing all input files into an intermediate representation and performing all other operations on this intermediate representation. This representation is a *structured graphics* world with a very small set of primitive objects (lines, circles, rectangles, and text) and the ability to *group* primitives together into a single object. All other graphic primitives that may be present in the input file are regarded as ornamental and are filtered out in the process of translating into the intermediate representation. Any graphics editor whose output can be easily parsed into this representation can be used for schematic generation.

This simple structured graphics world prevents any strong coupling between the editor and the translator. It also removes a lot of hidden hints that a translator might otherwise be able to utilize. The only hidden hint is the grouping of objects. Hence, grouping is limited to a very narrow function, delimiting icons, which is discussed in Section 4.1. That the translator has to start with so little information helps to ensure that the resulting graphical language is visually intuitive.

# 3. Generating Procedures

From the outset, we must accept that the goal is to generate computer procedures. It is not the purpose of the schematic system to fool the designer into thinking that she is not really writing a C++ procedure. Rather, the visual layout assists the designer in efficiently specifying the procedure correctly. Also, the schematic provides a single representation of the design, so that all later modifications only need to be made in one place. But, ultimately, what is specified must be a complete and correct procedure. The role of the drawing translator is to provide a schematic world that allows the designer to be concise and yet precise, and to allow easily understood graphical drawings that can be fairly dense. These goals drive the overall architecture of the translation process.

## 3.1. Frames and Evaluation

The first concept that must be understood is that of a *frame*. Every rectangle that is not grouped within an icon is called a frame. Frames partition the page into different regions. Each region is allowed to have its own *region interpretation procedure*. One reason for creating different regions with frames is precisely that the designer wants different regions to have different interpretation procedures. A second use of frames is to group the objects of a region together into a single programming language "statement". A third use of frames is to control the order of interpretation of objects.

Frames partition the page into a hierarchical tree based on inclusion. To ensure this, there is a simple rule: given a frame and any other object, the sides of the frame may not intersect the sides of the bounding box of the object. With this rule, the answer as to whether or not an object is inside a given frame is always well-defined. For each object, *drip* can easily determine the minimal frame in which it lies, and build a tree of objects with that frame as its parent. At the topmost level of the tree, is a *root frame* which encloses the entire page.

Some objects in a frame are text strings of the form "keyword: contents". These text strings perform two different roles depending on the keyword. If the keyword is the name of a region interpretation procedure (e.g. "cell"), then the text string is simultaneously declaring the region procedure to be used for evaluating the contents of its minimally enclosing frame, and the text string is passing arguments to that region procedure. There are two domain-independent region procedures:

- The "comment" procedure is the simplest. Everything inside the frame is treated as a comment; there is no C++ output generated. Its contents are not evaluated.
- The "sub" procedure is the default for the root frame. Its purpose is to ensure that we properly iterate through and evaluate all its children, which may only be text procedures and sub-frames. They are evaluated in the 2D order described in Section 3.2.

If, on the other hand, the keyword is the name of a *text interpretation procedure* (e.g. "Inputs", "Layout"), then the "contents" are being passed to the specified text procedure for translation into generated code. There are three domain independent text procedures.

- The text procedure "code" passes its "contents" straight through to the final C++ program. As a short-cut, text beginning with a colon is equivalent to text beginning with a "code" keyword.

- The text procedure "include" takes a comma separated list of file names. These schematic files are read and translated into C++ code in order by *drip*. For 2D ordering purposes (see Section 3.2), the code is generated as if the schematic files fit in a frame at the same position and size as the text string that contains the "include" keyword.
- The text procedure "comm" generates C++ comments that contain its text arguments.

Text procedures, icons, and sub-frames are directly evaluated and generate code. These are called *evaluation objects*. All other objects in a frame are analyzed only in order to attach parameters and properties on the evaluation objects before their evaluation.

## 3.2. 2D Ordering

Since our graphical representation is really a poorly disguised C++ program, the order in which we generate lines of the program is very important. This section how we order all the objects within a frame for evaluation.

In standard programming languages, the program itself is usually specified as ASCII text. Since these text files are one-dimensional, the order of evaluation is easy: from the beginning to the end of the file. In a graphical language, the user has more degrees of freedom in placing the language constructs. Thus it is less obvious in what order to evaluate the constructs. The first temptation is to sort by the upper-left corners of the bounding boxes of the objects, using the *Y*-coordinate as the most significant key and the *X*-coordinate as the least significant key. This is not entirely satisfactory. It is not as robust as we would like. Nor does it always order objects in the order we'd like.

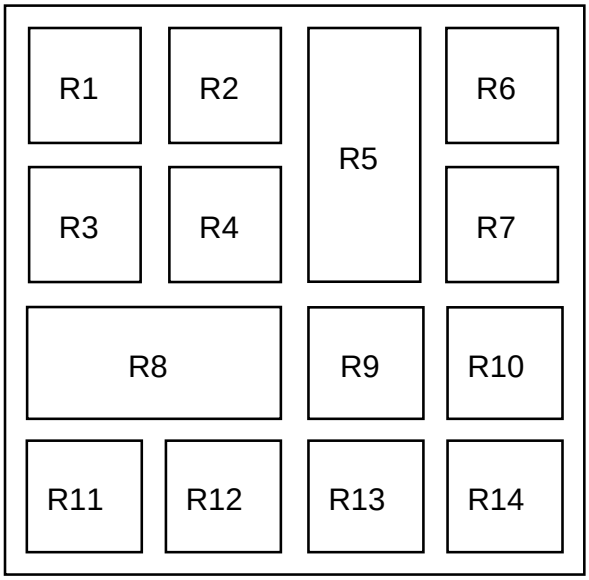


**Figure 2:** 2D ordering of objects

We needed to come up with a better 2D ordering. Our resulting ordering system is easy to explain, robust and closely follows our intuitive reading order. This ordering works very well

for non-overlapping rectangles. When sorting evaluation objects, we use the bounding boxes of the graphical element. An example showing the ordering we use is shown in Figure 2.

At the core of this ordering are these three rules:

Case 1
: When two rectangles can be be separated by a vertical line and their projections onto that line intersect, we order them left to right. Example: in Figure 2, R1 precedes R5.

Case 2
: When two rectangles can be be separated by a horizontal line and their projections onto that line intersect, we order them top to bottom. Example: in Figure 2, R8 precedes R12.

Case 3
: When one rectangle is completely above and to the left of the other rectangle, we order them left to right (or top to bottom, since that is equivalent). Example: in Figure 2, R4 precedes R13.

But, what about R2 vs R3? These are considered incomparable and so we leave them un-ordered for now.

The final ordering is found by the following procedure:

```
repeat until done:
  if only one rectangle has no rectangles that precede it then
    schedule that rectangle
  else
    of all the rectangles that have no rectangles that precede them,
      schedule the one that has the highest upper-left corner
```

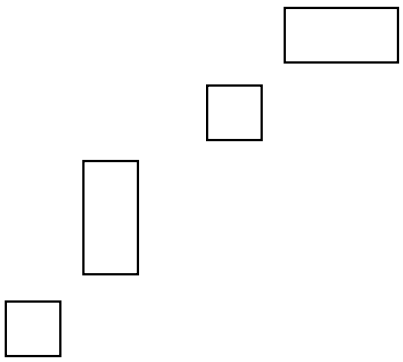

**Figure 3:** Incomparable Rectangles

If we are in the else clause, then the rectangles that have no rectangles that precede them must be stretched out diagonally as shown in Figure 3. In this case, the algorithm picks the uppermost one - which is the same as the rightmost one.

We can see why we didn't add a fourth ordering rule to decide on the incomparable rectangles by looking carefully at Figure 2. Assume that we had said

Case 4
: When one rectangle is completely above and to the right of the other rectangle, we order them from top to bottom. Example: in Figure 2, R2 precedes R3.

Looking at Figure 2, the rules would now say that rectangle R6 before rectangle R4 - yielding a circular order given the previous rules.

Or, similarly, assume we had instead chosen the rule

Case 4
: When one rectangle is completely above and to the right of the other rectangle, we order them from left to right. Example: in Figure 2, R3 precedes R6.

Looking at Figure 2, the rules would now say that rectangle R11 before rectangle R4 - yielding a circular order given the previous rules.

So we cannot add a fourth rule to give us a total ordering without leading to a circular ordering.

## 4. Drawing Interpretation

There is nothing in this evaluation procedure that is specific to the application of schematic capture. It can be used as the basis for graphical programming environments in a number of areas. It is the details of the region and text procedures that define the domain. With a different set of region and text procedures, this graphical programming environment could be retargeted for other uses. Since this paper is mainly concerned with the domain of schematic capture, we now examine some of our domain-specific interpretation procedures.

Some interpretation procedures work on a string of text of the form "key: text", as mnetions previously. Here, the key defines how the text is to interpreted for code generation. The domain-specific text procedures "inputs", "outputs", "publics", "internals" are all used to declare the wires of a cell. The procedures "layout" and "model" generate code to annotate the netlist for later CAD system functions; "layout" specifies what method should be used to generate layout of a cell, and "model" specifies how a cell should be modeled by the simulator. These functions were used frequently enough to warrant a text procedure, rather than forcing the designer to write out the C++ equivalents.

The more interesting interpretation procedures work on the contents of regions. We settled on a simple collection of region procedures: "basic", "block", "cell", "plain", and "icon". We have mentioned that region procedures can be explicitly declared. When there is no label with a specific declaration, a frame’s region procedure is inherited from that of its parent frame.

- The "basic" procedure does our basic schematic evaluation with no extras. It first analyzes all non-evaluation objects in the region: wires, connectors, wire labels, icon parameters. It binds wire names to wires and procedure arguments to icons. It propagates wire names along complex wires and attaches them to icon connectors. Any unlabeled children frames inherit a "block" label. Finally, all children that are evaluation objects are evaluated in the 2D order described in Section 3.2.
- The "block" procedure is just like the "basic" procedure, except that the code it generates is turned into a single C++ "statement" by enclosing it all with {}.
- The "cell" procedure is used at the top level of a netlist generation procedure definition. It first causes the generation of a C++ procedure preamble, then it behaves like the "basic" procedure, and finally it generates a procedure postamble.

- The "plain" procedure is just like the "cell" procedure, except that it causes a less specialized procedure preamble and postamble to be generated.
- The "icon" procedures generates a C++ forward declaration for a the netlist generation procedure which is produced by a "cell" procedure.

## 4.1. Icons

An icon is a grouping of objects which defines the interface to a cell in the netlist. Icons are used in two circumstances. In "cell" regions, an icon represents an instance of a subcell, bound to wires in the current cell at its connectors. In "icon" regions, the identical graphical object generates a declaration for the netlist procedure defined by a "cell". Following the program analogy, "icon" regions are often collected in a drawing which is the graphical equivalent of a C++ ".h" file, defining the procedural interfaces to a collection of cell generators.

A well-formed icon should consist of *n* circles, *n*+1 pieces of text, and any number of other non-circle, non-text objects. The circles are icons connectors, to which wires can be attached. Each connector must be labeled by the name of an external wire of the cell represented by the icon. The text binding procedure described in Section 5.1 will be used to bind *n* of the text objects to the *n* circles. The remaining unbound text is taken to be the name of the icon. Some *conforming* icons are shown in Figure 4.

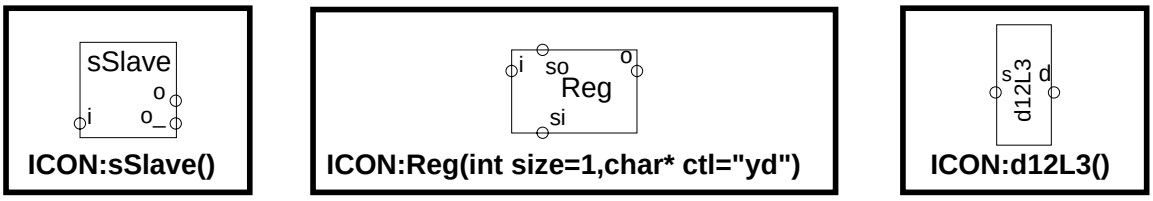


**Figure 4:** "conforming" icons

Unaltered, this scheme imposes some limitations on icons that many users find unacceptable. There are some tricks to get around them that *don't conform* to our guiding philosophies:

- Some common icons are well known, as are the meaning of their connectors, e.g. the *npn* transistors in Figure 5. Requiring a name would cause clutter that is annoying to many designers. The solution in this case is to set the color of the text to "white", thus making it invisible. Unless the icon is very standard, this practice is strongly discouraged.
- Sometimes we need circles that are not connectors in our icon, e.g. the ISrc in Figure 5. These circles must not be bound to text. In this case, the extra circles should be hidden within a sub-group of the icon's group. The icon interpreter and text binder do not search within subgroups of the icon.
- Sometimes we need text that is not a connector or icon label in our icon, e.g. the "+" and "-" in the VSrc in Figure 5. Just as with circles, these are hidden from the interpreter in sub-groups of the icon's group.

By the time that the "basic" region procedure attempts to evaluate an icon, *drip* will already have bound wirenames to the wires in the region and propagated them all the way to the icon connectors, as described in Section 5. So for each icon connector, we know the name in the

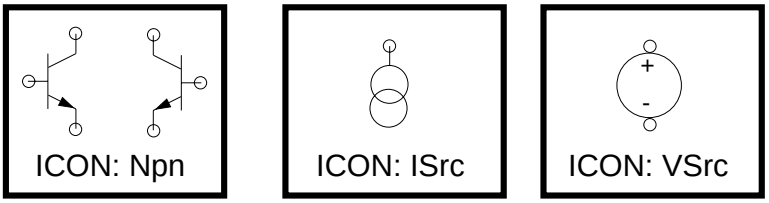


**Figure 5:** Sample "non-conforming" icons

current cell of the wire that is attached to it. From the text binding of the icon contents, we know the name of the wire that the connector represents in the cell generated by the procedure that the icon invokes. Finally, since we bind these two wires by name, we can generate the code shown in Figure 1, lines 16, 19, 21, 23 and 25. Note that the code generated by a single icon is always enclosed with {} and is thus a single C++ statement.

```
1  Cell* sSlave ();
2  Cell* Reg (int size=1,char* ctl="yd");
3  Cell* d12L3 ();
```

**Figure 6:** Code Generated by "Icon" Region Procedure for Figure 4

In Figures 4 and 5 we see the use of the "icon" region procedure. This region procedure has two tasks. First, it generates a forward reference to the corresponding netlist generation procedure. The code emitted is a declaration of the procedure with its arguments, including any default arguments. This code will typically appear in a ".h" file. The code generated by this region procedure for Figure 4 is shown in Figure 6. Second, The "icon" frame procedure analyzes any icons contained in it to make sure that they are well-formed and that the name of the icon is the same as the netlist procedure that is named in the "Icon:" declaration. If the designer wants to copy and paste an icon for use in the current design, an "icon" frame procedure that has already successfully passed through *drip* is a very safe place from which to copy the icon.

## 5. Analysis of Non-Evaluation Objects

During the execution of the "basic" region procedure, the first task is to analyze the non-evaluation objects. These are: lines, which represent wires; circles, which represent wire connectors; and text which is not of the "keyword: contents" form. The role of a particular text object is easily determined from its syntax. Text enclosed in parenthesis represents an argument to be passed to an icon. Text beginning with a "." or a "[" is used for wire subscripting, which will be discussed in Section 5.3. Text that is a normal identifier, or a normal identifier with subscripting, is a wire name.

### 5.1. Binding Text to Objects

The main work in analyzing non-evaluation objects is determining to what objects the non-evaluation text should be attached. Binding text to objects is necessary for several different tasks. For the analysis phase of "basic" we need to be able to bind labels to wires and bind arguments to icons. For icon analysis we need to bind labels to icon connectors. Given three different uses, we choose to use the same algorithm for all three. This makes it easier for the designer to understand the translator behavior.

We do not allow the drawings to have hidden information, so *drip* must base its binding decisions only on the positions of the text in relation to the other objects. The algorithm is straightforward. The binding subroutine will be given a collection of text and a collection of relevant objects to bind the text to. For each piece of text it finds the closest relevant object. If no other text has the same closest object, then the object and the text are bound. When multiple pieces of text have the same closest object, this is usually reported as an error. (We allow one exception: in the case of a wire that has the identical label repeated multiple times, we don't flag an error.) The distance from text to an object is computed as follows: replace the text with the line that results from underlining the text and replace the object with a rectangular bounding box, then find the manhattan distance from the line to the box.

One drawback of such a simple method is that it doesn't bind text to second closest objects in cases that are intuitive to humans. More complicated algorithms were experimented with. They led to sufficient unpredictability at the user level, that they caused more harm than good. The great advantage of our method is that a user looking at the printed copy can easily understand exactly what text is bound to what object.

## 5.2. Wires

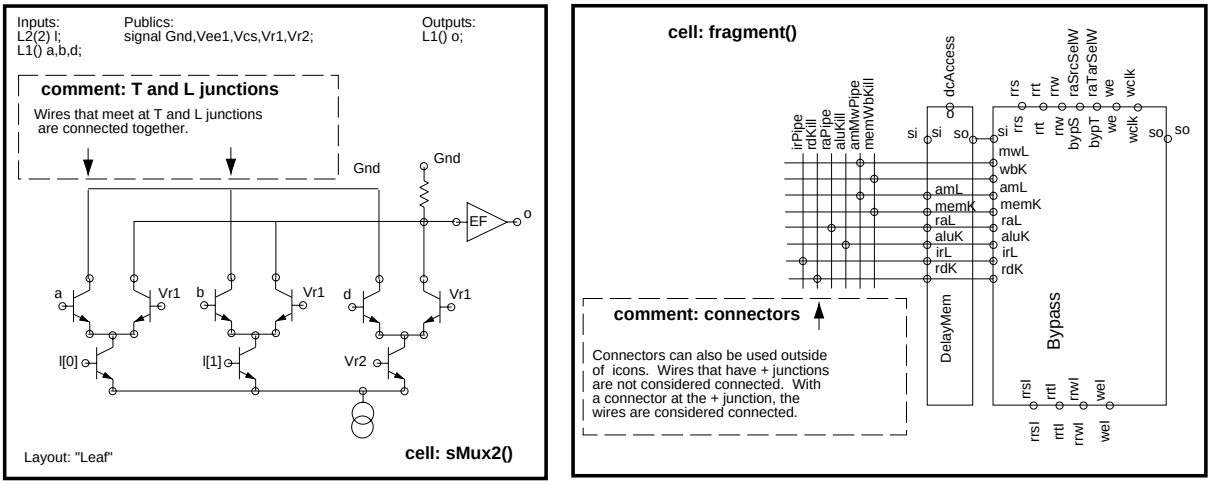


**Figure 7:** Junctions and Connectors

When binding labels to wires, the binding algorithm attaches each label to a single line or circle. However, the translator needs to understand that the label applies to the entire conduction path. A conduction path consist of many lines and circles that are connected together. The rules we use for connectivity analysis are straightforward (see Figure 7).

1. Two wires that meet at a T-junction or an L-junction are connected at that junction.
2. Two wires that meet at a "+"-junction are not-connected.
3. If a wire intersects a circular connector, they are connected.
4. Transitivity: if A is connected to B and B is connected to C, A is connected to C.

## 5.3. Wire Subscripting

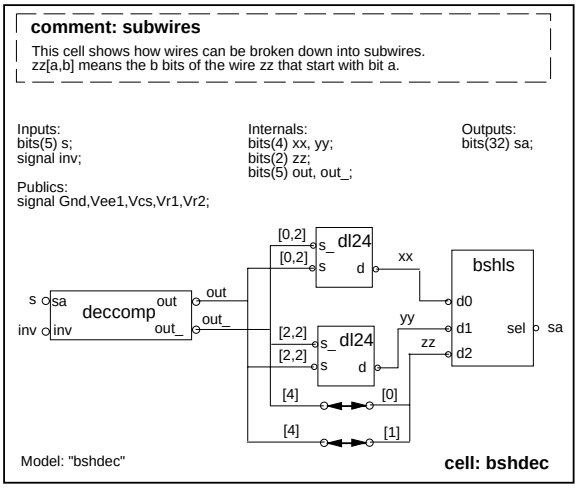


**Figure 8:** Wire Subscripting

In our CAD system, a wire may be an array or complicated structure of sub-wires. In a schematic we may want to tear a wire into sub-wires. An example is shown in Figure 8. Now we will have a set of connected lines, some of which represent a collection of nets, some of which represent a subset of them. Which line represents what nets? For a connected set of conductors that use subscripting, some basic rules must hold:

- The conductors must form a tree.
- There may be exactly one non-subscript label on any of the conductors.
- Each conductor is allowed at most one label, whether it is a subscript label or a non-subscript label.

Given these rules, we can choose the non-subscript labeled conductor as the root of the tree. The label on any conductor is the concatenation of all the labels along the path from the root of the tree to the conductor, inclusive.

This algorithm is intuitive, robust and works for multiple levels of subscripting.

# 6. Error Reporting

The main objection that designers have after the above system is explained to them, is that the decoupling of editor and interpreter makes it harder to deal with schematic errors found by the interpreter.

Since our decoupling is in part modeled on the separation of text editor and compiler that programmers are very used to, we similarly mimic the error reporting mechanisms of *emacs*. When programming with *emacs*, the compiler reports errors in such a format that the programmer can use *emacs* to single-step through the lines in the code that have errors.

Similarly, whenever *drip* encounters an error, it writes an entry into an error file. This entry contains an error message, a source schematic file name, and descriptions of polygons with which to highlight the error region of the schematic.

Since out editor did not have the ability to handle error files, we had to customize it. We added commands to load an error file, and single step through the errors. The editor pulls up the schematic with the error, overlays the highlighting polygons, and pops up a window with the error message. The designer can fix the error and then step to the next one.

## 7. Experiences

This system was successfully used at our laboratory for a period of four years, encompassing two microprocessor projects, BIPS0 [4] and BIPS1. Each design used over one hundred schematically specified netlist generation procedures.

Another bonus effect of the separation between the schematics and the generated C++ code that is provided by the interpreter was that the frequent changes to the CAD system libraries and their interfaces usually only required modifications to *drip*'s interpretation procedures, rather than to every schematic.

## Acknowledgements

*drip* descended from *moog*. *moog* was a drawing interpreter written by Mike Nielsen and Jeremy Dion for use with the *artemis* graphics editor. Under the guidance of Jeremy Dion, Kamal Chaudhary wrote a preliminary drawing interpreter *moo*, that would be more graphics editor independent. Ramsey Haddad extended *moo*. Ramsey Haddad and Louis Monier rewrote it, creating *drip*.

*drip* was further improved by incorporating ideas from a number of users: Mary Jo Doherty, Alan Eustace, Norm Jouppi, Jim Keller, Suresh Menon, Silvio Turrini. Jeremy Dion and Louis Monier made numerous helpful suggestions on the paper.

## References

[1] R. Barth, B. Serlet, P. Sindhu. Parameterized Schematics. In *25th Design Automation Conference*, pages 243-249. Sydney, June, 1988.

[2] R. Barth, L. Monier, B. Serlet. Patchwork: Layout From Schematic Annotations. In *25th Design Automation Conference*, pages 250-255. Sydney, June, 1988.

[3] C. Ebeling, Z. Wu. WireLisp: Combining Graphics and Procedures in a Circuit Specification Language. In *Proceedings of the ICCAD*, pages 322-325. IEEE, 1989.

[4] N.P. Jouppi, P. Boyle, J. Dion, M.J. Doherty, A. Eustace, R.W. Haddad, R. Mayo, S. Menon, L.M. Monier, D. Stark, S. Turrini, J.L. Yang, W.R. Hamburgen, J.S. Fitch, R. Kao. A 300-MHz 115-W 32-b Bipolar ECL Microprocessor. *IEEE Journal of Solid-State Circuits* 28(11), November, 1993.

[5] R. Mayo. Mocha chip: A system for the graphical design of VLSI module generators. In *Proceedings of the ICCAD*, pages 74-77. IEEE, 1986.

[6] L. McMurchie, C. Ebeling. *WireC Tutorial and Reference Manual*. Technical Report 94-09-09, University of Washington, Department of Computer Science and Engineering, September, 1994.

[7] L. Monier. Layout Generation Through Parameterized Schematics. In *7th Australian Microelectronics Conference*, pages 157-164. Sydney, May, 1988.

[8] L. Monier, J. Dion. Recursive Layout Generation. *WRL Research Report* (95/2), 1995.

[9] J. Vlissides, M. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors . *ACM Transactions on Information Systems* 8(3):237--268, July, 1990.

[10] Z. Wu, C. Ebeling. *Drawing Wirelisp*. Technical Report 89-12-03, University of Washington, Department of Computer Science and Engineering, December, 1989.

# WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

‘‘Simple and Flexible Datagram Access Controls for Unix-based Gateways.’’
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

‘‘Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.’’
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

‘‘Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.’’
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

‘‘A Unified Vector/Scalar Floating-Point Architecture.’’
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

‘‘Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.’’
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

‘‘Integration and Packaging Plateaus of Processor Performance.’’
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

‘‘A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.’’
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

‘‘The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.’’
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

‘‘Long Address Traces from RISC Machines: Generation and Analysis.’’
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

‘‘Link-Time Code Modification.’’
David W. Wall.
WRL Research Report 89/17, September 1989.

‘‘Noise Issues in the ECL Circuit Family.’’
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

‘‘Efficient Generation of Test Patterns Using Boolean Satisfiablilty.’’
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

‘‘Two Papers on Test Pattern Generation.’’
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

‘‘Virtual Memory vs. The File System.’’
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

‘‘Efficient Use of Workstations for Passive Monitoring of Local Area Networks.’’
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

‘‘A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.’’
John S. Fitch.
WRL Research Report 90/6, July 1990.

‘‘1990 DECWRL/Livermore Magic Release.’’
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

‘‘Pool Boiling Enhancement Techniques for Water at Low Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 90/9, December 1990.

‘‘Writing Fast X Servers for Dumb Color Frame Buffers.’’
Joel McCormack.
WRL Research Report 91/1, February 1991.

‘‘A Simulation Based Study of TLB Performance.’’
J. Bradley Chen, Anita Borg, Norman P. Jouppi.
WRL Research Report 91/2, November 1991.

‘‘Analysis of Power Supply Networks in VLSI Circuits.’’
Don Stark.
WRL Research Report 91/3, April 1991.

‘‘TurboChannel T1 Adapter.’’
David Boggs.
WRL Research Report 91/4, April 1991.

‘‘Procedure Merging with Instruction Caches.’’
Scott McFarling.
WRL Research Report 91/5, March 1991.

‘‘Don’t Fidget with Widgets, Draw!.’’
Joel Bartlett.
WRL Research Report 91/6, May 1991.

‘‘Pool Boiling on Small Heat Dissipating Elements in Water at Subatmospheric Pressure.’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Research Report 91/7, June 1991.

‘‘Incremental, Generational Mostly-Copying Garbage Collection in Uncooperative Environments.’’
G. May Yip.
WRL Research Report 91/8, June 1991.

‘‘Interleaved Fin Thermal Connectors for Multichip Modules.’’
William R. Hamburgen.
WRL Research Report 91/9, August 1991.

‘‘Experience with a Software-defined Machine Architecture.’’
David W. Wall.
WRL Research Report 91/10, August 1991.

‘‘Network Locality at the Scale of Processes.’’
Jeffrey C. Mogul.
WRL Research Report 91/11, November 1991.

‘‘Cache Write Policies and Performance.’’
Norman P. Jouppi.
WRL Research Report 91/12, December 1991.

‘‘Packaging a 150 W Bipolar ECL Microprocessor.’’
William R. Hamburgen, John S. Fitch.
WRL Research Report 92/1, March 1992.

‘‘Observing TCP Dynamics in Real Networks.’’
Jeffrey C. Mogul.
WRL Research Report 92/2, April 1992.

‘‘Systems for Late Code Modification.’’
David W. Wall.
WRL Research Report 92/3, May 1992.

‘‘Piecewise Linear Models for Switch-Level Simulation.’’
Russell Kao.
WRL Research Report 92/5, September 1992.

‘‘A Practical System for Intermodule Code Optimization at Link-Time.’’
Amitabh Srivastava and David W. Wall.
WRL Research Report 92/6, December 1992.

‘‘A Smart Frame Buffer.’’
Joel McCormack & Bob McNamara.
WRL Research Report 93/1, January 1993.

‘‘Recovery in Spritely NFS.’’
Jeffrey C. Mogul.
WRL Research Report 93/2, June 1993.

‘‘Tradeoffs in Two-Level On-Chip Caching.’’
Norman P. Jouppi & Steven J.E. Wilton.
WRL Research Report 93/3, October 1993.

‘‘Unreachable Procedures in Object-oriented Programing.’’
Amitabh Srivastava.
WRL Research Report 93/4, August 1993.

‘‘An Enhanced Access and Cycle Time Model for On-Chip Caches.’’
Steven J.E. Wilton and Norman P. Jouppi.
WRL Research Report 93/5, July 1994.

‘‘Limits of Instruction-Level Parallelism.’’
David W. Wall.
WRL Research Report 93/6, November 1993.

‘‘Fluoroelastomer Pressure Pad Design for Microelectronic Applications.’’
Alberto Makino, William R. Hamburgen, John S. Fitch.
WRL Research Report 93/7, November 1993.

‘‘A 300MHz 115W 32b Bipolar ECL Microprocessor.’’
Norman P. Jouppi, Patrick Boyle, Jeremy Dion, Mary Jo Doherty, Alan Eustace, Ramsey Haddad, Robert Mayo, Suresh Menon, Louis Monier, Don Stark, Silvio Turrini, Leon Yang, John Fitch, William Hamburgen, Russell Kao, and Richard Swan.
WRL Research Report 93/8, December 1993.

‘‘Link-Time Optimization of Address Calculation on a 64-bit Architecture.’’
Amitabh Srivastava, David W. Wall.
WRL Research Report 94/1, February 1994.

‘‘ATOM: A System for Building Customized Program Analysis Tools.’’
Amitabh Srivastava, Alan Eustace.
WRL Research Report 94/2, March 1994.

‘‘Complexity/Performance Tradeoffs with Non-Blocking Loads.’’
Keith I. Farkas, Norman P. Jouppi.
WRL Research Report 94/3, March 1994.

‘‘A Better Update Policy.’’
Jeffrey C. Mogul.
WRL Research Report 94/4, April 1994.

‘‘Boolean Matching for Full-Custom ECL Gates.’’
Robert N. Mayo, Herve Touati.
WRL Research Report 94/5, April 1994.

‘‘Software Methods for System Address Tracing: Implementation and Validation.’’
J. Bradley Chen, David W. Wall, and Anita Borg.
WRL Research Report 94/6, September 1994.

‘‘Performance Implications of Multiple Pointer Sizes.’’
Jeffrey C. Mogul, Joel F. Bartlett, Robert N. Mayo, and Amitabh Srivastava.
WRL Research Report 94/7, December 1994.

‘‘How Useful Are Non-blocking Loads, Stream Buffers, and Speculative Execution in Multiple Issue Processors?.’’
Keith I. Farkas, Norman P. Jouppi, and Paul Chow.
WRL Research Report 94/8, December 1994.

‘‘Recursive Layout Generation.’’
Louis M. Monier, Jeremy Dion.
WRL Research Report 95/2, March 1995.

‘‘Contour: A Tile-based Gridless Router.’’
Jeremy Dion, Louis M. Monier.
WRL Research Report 95/3, March 1995.

‘‘The Case for Persistent-Connection HTTP.’’
Jeffrey C. Mogul.
WRL Research Report 95/4, May 1995.

‘‘Network Behavior of a Busy Web Server and its Clients.’’
Jeffrey C. Mogul.
WRL Research Report 95/5, June 1995.

# WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.

‘‘The Effect of Context Switches on Cache Performance.’’
Jeffrey C. Mogul and Anita Borg.
WRL Technical Note TN-16, December 1990.

‘‘MTOOL: A Method For Detecting Memory Bottlenecks.’’
Aaron Goldberg and John Hennessy.
WRL Technical Note TN-17, December 1990.

‘‘Predicting Program Behavior Using Real or Estimated Profiles.’’
David W. Wall.
WRL Technical Note TN-18, December 1990.

‘‘Cache Replacement with Dynamic Exclusion’’
Scott McFarling.
WRL Technical Note TN-22, November 1991.

‘‘Boiling Binary Mixtures at Subatmospheric Pressures’’
Wade R. McGillis, John S. Fitch, William R. Hamburgen, Van P. Carey.
WRL Technical Note TN-23, January 1992.

‘‘A Comparison of Acoustic and Infrared Inspection Techniques for Die Attach’’
John S. Fitch.
WRL Technical Note TN-24, January 1992.

‘‘TurboChannel Versatec Adapter’’
David Boggs.
WRL Technical Note TN-26, January 1992.

‘‘A Recovery Protocol For Spritely NFS’’
Jeffrey C. Mogul.
WRL Technical Note TN-27, April 1992.

‘‘Electrical Evaluation Of The BIPS-0 Package’’
Patrick D. Boyle.
WRL Technical Note TN-29, July 1992.

‘‘Transparent Controls for Interactive Graphics’’
Joel F. Bartlett.
WRL Technical Note TN-30, July 1992.

‘‘Design Tools for BIPS-0’’
Jeremy Dion & Louis Monier.
WRL Technical Note TN-32, December 1992.

‘‘Link-Time Optimization of Address Calculation on a 64-Bit Architecture’’
Amitabh Srivastava and David W. Wall.
WRL Technical Note TN-35, June 1993.

‘‘Combining Branch Predictors’’
Scott McFarling.
WRL Technical Note TN-36, June 1993.

‘‘Boolean Matching for Full-Custom ECL Gates’’
Robert N. Mayo and Herve Touati.
WRL Technical Note TN-37, June 1993.

‘‘Ramonamap - An Example of Graphical Groupware’’
Joel F. Bartlett.
WRL Technical Note TN-43, December 1994.

‘‘Circuit and Process Directions for Low-Voltage Swing Submicron BiCMOS’’
Norman P. Jouppi, Suresh Menon, and Stefanos Sidiropoulos.
WRL Technical Note TN-45, March 1994.

‘‘Experience with a Wireless World Wide Web Client’’
Joel F. Bartlett.
WRL Technical Note TN-46, March 1995.

‘‘I/O Component Characterization for I/O Cache Designs’’
Kathy J. Richardson.
WRL Technical Note TN-47, April 1995.

‘‘Attribute caches’’
Kathy J. Richardson, Michael J. Flynn.
WRL Technical Note TN-48, April 1995.

‘‘Operating Systems Support for Busy Internet Servers’’
Jeffrey C. Mogul.
WRL Technical Note TN-49, May 1995.