A PostScript Toolkit for Electronic Design

Sarp Özdemir
Computer Science
SUNY Binghamton
Binghamton, NY
sozdemi2@binghamton.edu

Jennifer Seibert
Computer Science
SUNY Binghamton
Binghamton, NY
jseiber1@binghamton.edu

Mohammad A. Khasawneh
Computer Science
SUNY Binghamton/MathWorks
Binghamton, NY
mkhasaw1@binghamton.edu

Patrick H. Madden
Computer Science
SUNY Binghamton
Binghamton, NY
pmadden@binghamton.edu

***Abstract*—Design automation tools operate on large problems, and often take hours or days to complete a run. Visualizing design data effectively can help track bugs, highlight optimization opportunies, and give a deeper understanding of the impact of subtle algorithmic changes. In this paper, we present a small, lightweight, cross-platform C language interface to the PostScript language, designed for software developers who primarily work in C or C++. Versions of this library have been a key resource for our research group in development of a number of design automation tools. The library is available in open source at *https://github.com/profmadden/pstools*.**

***Index Terms*—electronic design, postscript, graphic display, open source**

I. Introduction

For decades, the size and complexity of electronics have increased; Moore's law continues at it's breakneck pace. To keep up, design automation tools have had to evolve continually.

Often, good visualization of a design can reveal implementation errors in a tool, or make an optimization opportunity readily apparent. User interfaces to many design tools utilize bitmapped high resolution graphic displays - but these can suffer from complex APIs, and can have limited portability across operating systems and development environments. A less freqently used alternative is direct generation of PostScript language figures; this has been a staple in tool development for our research group, but is not widely used. In this paper, we present our approach, and provide an open source set of tools to enable easy use.

II. The PostScript Language

PostScript [1] has become a de facto standard for rendering text and figures; it underlies PDF files, many high resolution printers support it natively, and there are optimized rendering applications for nearly any computer system. Many modern text and graphics editing programs support PostScript directly, enabling the embedding of figures into other documents easily.

The versatility of PostScript comes from it being an expressive *interpreted programming language,* rather than a specific file format. The PostScript programming model supports a flexible coordinate system, primative operators for drawing lines and curves, and an abundance of operators to handle text and different fonts. Bitmapped shapes and textures are also supported. Using these primatives, along with an efficient stack based approach, nearly any image that could be imagined can be translated into PostScript. The language itself is human readable, although it is uncommon to write programs "by hand." In practice, PostScript "programs" are created by other programs, and then rendered by a PostScript interpreter to the screen, or to a printer.

Figure 1 shows a boilerplate header for a PostScript program; this must be formatted correctly for a PostScript interpreter to process a file correctly. While neither complex or cryptic, needing to recall or look up the formatting of this material can be a burden for a software developer focused on another complex task (and working primarily in another programming language). In large part, the library we have developed is to minimize the nuisance of generating PostScript.

```
%!PS-Adobe-3.0 EPSF-3.0
%%DocumentData: Clean7Bit
%%Origin:       0.00       0.00
%%BoundingBox:       0.00       0.00    1000.00    1000.00
%%LanguageLevel: 2
%%Pages: 1
%%Page: 1 1
```

Fig. 1. The header portion of a PostScript program.

The drawing primatives with PostScript are relatively simple, but unless a software developer is actively working with the language, generating code might require frequent trips to the language manual. In Figure 2, we show the creation of a line and a circle in PostScript (with the percent symbol marking a comment line). The language operates off a stack; coordinates for the operations are pushed onto the stack first, and then commands such as *moveto* and *arc* use the stack elements to generate the graphic output.

```
% Create a line from 3, 4 to 25, 37
newpath 3 4 moveto 25 37 lineto stroke

% Create a filled circle with a
% center at 55, 66, radius of 20
55 66 20 0 360 arc closepath fill
```

Fig. 2. PostScript commands to create a line and a circle.

The language itself is not difficult, particularly if the images to be created are relatively straight-forward lines, arcs, and text.

III. PostScript in Design Automation Tools

To illustrate how our research group has used PostScript generation over the years, we include a few figures from published papers. In Figure 3, we show the standard cell legalization step of our fractional cut placement tool *Feng Shui* [3]. Individual cell positions are illustrated, along with red highlighting lines tracking movement from abstract to legal placement. This sort of illustration was helpful in identifying errors and shortcomings in our first placement legalization efforts. Creation of figures such as this required only a half page of C code. The routines to generate the images were created during the development of the placement tool, while the underlying data structures were fresh in mind.

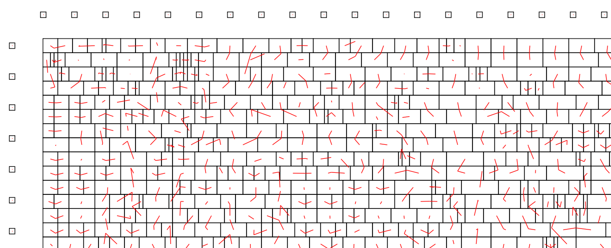

Fig. 3. An illustration of cell movement in the fractional cut recursive bisection placement tool *Feng Shui*. Short red lines indicate small shifts required for legalization.

As a second example, Figure 4 shows global routing congestion maps from our *Chi* router [4], across different iterations of the tool, and using different cost functions to guide optimization. A complete run of a suite of benchmarks could take many hours; generation of PostScript snapshots during each run enabled our group to easily see what occured on any given run, and also to track the impact of code changes across months of development.

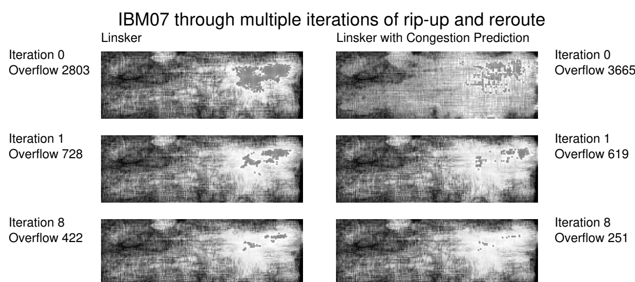

Fig. 4. Congestion maps from the *Chi* [4]. Multiple PostScript files were assembled using a commercial graphics editing package to create the illustration used in the research paper.

Figures 3 and 4 were published in 2003, and we have continued the use of PostScript. In Figure 5, we show illustrations from a detail placement paper published in 2020 [5]. Our focus in this work is on reducing interconnect length, and accomplishing this often required moving dozens of standard cells in synchrony – with the potential solution space being astronomically large. Good visualization was essential to finding and tracking implementation errors, and in spotting potential improvements.

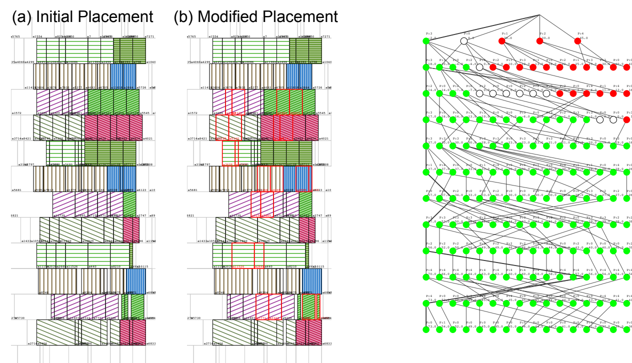

Fig. 5. Our large-window detail placement approach groups standard cells together for permutation; the grouping (and restrictions on permutations) enables large problems to be solved to near-optimality. Cell movement is shown by generating "before" and "after" snapshots, while the tree shows a key internal data structure.

IV. The pstools Interface

To insulate research group members primarily working in C and C++, we have developed the *pstools* library; key routines are shown in Figure 6. The library is neither large nor complex; our objective is simply to "get the job done" without expending a vast amount of software development effort.

```
ps_context *ps_init(char *filename,
                    float origin_x, float origin_w,
                    float width, float height);

int ps_line(ps_context *context,
            float x1, float y1, float x2, float y2);

int ps_circle(ps_context *context,
              float cx, float cy, float radius,
              int stroke, int fill);

int ps_text(ps_context *context,
            float x, float y, char *text);

int ps_note(ps_context *context, char *note);

int ps_finish(ps_context *context);
```

Fig. 6. The programming interface to the *ps_tools* library. **ps_init** opens a file for writing, while **ps_finish** closes it. Commands to create lines, circles, and so on, are straight forward.

The function *ps_init* opens a file under the supplied name, and generates the boilerplate PostScript header. The *ps_context* structure wraps the file handle, and some additional information about the size of the drawing area. When *ps_finish* is called, the file is closed, and allocated memory is released.

Using the library, it's a simple matter to integrate PostScript output into an application program. In practice, we develop routines that produce images of interest – a placement, or routing, for example – and then add calls to this routine at various points throughout the entire automation tool flow. Generation

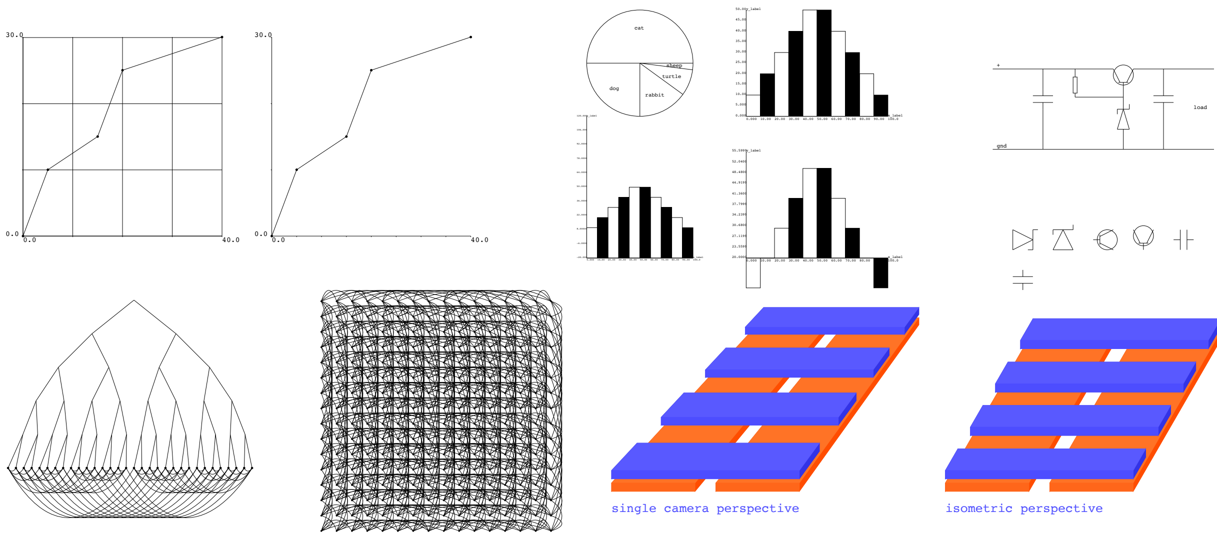

Fig. 7. Example figures included with the *pstools* library. Using the primatives, and also interspersing custom PostScript commands, a wide range of figures can be created easily.

of PostScript files are toggled on and off with debugging flags, or can be enabled and disabled at run-time. We use a simple naming convention in most cases – benchmark name, followed by the date that the file is generated.

An extremely helpful feature of the PostScript language is the insertion of comments (which are lines beginning with a percent symbol). The **ps_note** routine inserts text as a comment into a PostScript file; we normally record the run parameters used, git commit information, and so on, so that all aspects of a run that created a particular image can be replicated at a later date.

Additionally, design information that may be more detailed than one might wish for a graphic display can be embedded – for example, the length of individual interconnect nets might be recorded, with the software developer being able to see the layout in Adobe Acrobat, while simultaneously looking at the notes in the generated PostScript file with a simple text editor.

V. PSTOOLS EXAMPLES

The *pstools* library contains a number of simple examples, which can be compiled and run using **make**. The raw PostScript can be converted into PDF format using an application such as **GhostScript**.

The example output, shown in Figure 7 include a simple plot, pie charts, bar charts, and a preliminary set of routines to draw standard electronic symbols.

As part of our current work, we are focused on combinatorial optimization problems – two of the figures show solution spaces for binary variables. In these, we combine the primatives in the *pstools* library with direct generation of PostScript language, to support curved lines between vertices of a graph.

There is even a simple three dimensional drawing toolkit under development; a set of commands creates three-dimensional rectangular areas, which are queued and then rendered back-to-front. The eventual goal of this portion of our work is to allow simple rendering of transistor-level layouts, including detail routing.

VI. CONCLUSION AND FUTURE WORK

The PostScript language is an excellent platform for creating graphic images, and it meshes well with the needs of design automation tools. Much of the work done by design tools is not interactive in nature – capturing snapshots from hours-long runs is extremely convenient.

Because the library uses only standard C output routines, it is exceptionally portable. The figures produced are vector-based, making them compact, and also scalable – ideal for embedding into research publications. Images can be zoomed electronically, and do not degrade the way bit-mapped images do (an excellent example of this is with Figure 5). We have used PostScript for decades, and we plan to continue work on the library, adding additional features, and futher documentation.

REFERENCES

[1] Adobe Systems Incorporated, “PostScript Language Reference,” (online) https://www.adobe.com/content/dam/acom/en/devnet/actionscript/articles/PLRM.pdf
[2] Binghamton Optimality Research Group, “pstools library,” (online) https://github.com/profmadden/pstools
[3] A. R. Agnihotri, M. C. Yildiz, A. Khatkhate, A. Mathur, S. Ono, and P. H. Madden, “Fractional Cut: Improved Recursive Bisection Placement,” proc. International Conference on Computer Aided Design (ICCAD), pp. 307–310, 2003.
[4] R. T. Hadsell and P. H. Madden, “Improved Global Routing through Congestion Estimation,” proc. Design Automation Conference (DAC), pp. 28–31, 2003.
[5] M. A. Khasawneh and P. H. Madden, “Hill Climbing With Trees: Detail Placement for Large Windows,” proc. Internationa Symposium on Physical Design (ISPD), 2020.