Program Visualisation for Visual Programs

James Noble and Robert Biddle

Technical Report CS-TR-01/6
August 2001

School of Mathematical and Computing Sciences
Victoria University
PO Box 600, Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Email: Tech.Reports@mcs.vuw.ac.nz
http://www.mcs.vuw.ac.nz/research

Program Visualisation for Visual Programs

James Noble and Robert Biddle

Technical Report CS-TR-01/6
August 2001

Abstract

Publishing Information

Author Information

James Noble and Robert Biddle lecture in Computer Science at Victoria Universtiy of Wellington.

Program Visualisation for Visual Programs

James Noble and Robert Biddle
Computer Science
Victoria University of Wellington
New Zealand
{kjx,robert}@mcs.vuw.ac.nz

Abstract

The Nord Modular music synthesiser systems comprises a stand-alone array of digital signal processors programmed by a dataflow visual langauage and supported by a visual programming environment that runs on commodity hardware. We have investigated applying program visualisation techniques to over 400 Nord Modular programs. Our visualisations make explicit module types and signal flows that are only implicit in the metaphorical graphical syntax adopted by the Nord Modular visual programming language. We have also analysed the programming style used in Nord Modular programs, in particular, the direction of program layouts. While we found that programs tended to arrange signal flow top down and left to right, we found much more variation than we expected, both within individual programs and across the Nord factory program corpus.

1 Introduction

Domain-specific dataflow visual programming languages are now commonplace throughout the world of computing. Although not the earliest form of visual language (that honour, like so many others, of course is due to Sketchpad [25]) dataflow visual languages are now by far the most common form of visual programming language used in practice. Although they have been mostly unsuccessful as general-purpose programming languages, dataflow visual languages have excelled within specialised niche domains, including information visualisation [20], programmable logic controller programming [15, 1], and business process re-engineering [14]. Even in just one domain, computer music, several dataflow visual languages have been successful as products over relatively long terms, including Max [7], Bars-n-Pipes [13], and the topic of this paper, the visual programming language underlying the Nord Modular series of synthesizers [6].

The Nord Modular synthesizer system, by Clavia AB of Sweden, is a digital (re)creation of the traditional analogue modular synthesizer systems common in the 1970s. The key idea of modular synthesis is that sounds are produced a collection of analogue modules, where the sounds the modules produce (or other aspects of their operation) are determined by control voltages produced by other modules.

A typical modular synthesier (see Figure 1) would consist of a number of modules such as voltage controlled oscillators (VCO) to produce sounds, voltage controlled filters (VCF) to shape it, and envelope generators (EG) and low-frequency oscillators (LFO) to produce control voltages for use as input to the oscillators and filters. The modules are connected together using "patch cords" that carry control voltages and audio signals between modules, and eventually to external amplifiers or recording equipment. Modular synthezier configurations (settings of module controls and the patch cord cabling linking them together) were named "patches", after the audio engineering term "patch" meaing to connect, as in "patch cord". The term "patch" has remained current for synthesizer programs of all types.

Although still manufactured by enthusiasts today, analogue modular synthesizers are large, unweildy, unreliable, and expensive. As a result, they were quickly eclipsed by integrated synthesizers, still based on analogue circuitry but where the connections between modules were fixed — typically with one oscillator, filter, a couple of envelope generators, and a couple of LFOs. Ultimately, analogue synthesizers were in turn replaced by digital synthesizers that were also based on fixed algorithms for sound production, generally those algorithms mimiced the sound paths of the integrated analogue synthesizers. Compared with the analogue modular synthesizers, the integrated synthesizers (particularly the digital integrated synthesizers) were small, portable, reliable, and cheap, however, they were much less flexible in sound production than the old modulars.

The Clavia Nord Modular synthesizer system is a recent development to address this problem. Essentially, the Nord Modular implements a flexible, modular system using integrated digital signal processing (DSP), rather than discrete analogue or digital components. Since its implementation technology is the same as the integrated synthesizers, it shares most of their benefits: it is small, portable, reliable, and (relatively) cheap: however, by allowing users to program (or at least reconfigure) the DSP algorithms, the Nord Modular retains most of the flexibility of the old analogue modular systems. This is of course its main liability compared with existing integrated synthesizers with fixed signal flow: users, that is musicians, need to be able to program the synthesizer, rather than simply set parameters to make patches. To address this problem, the Nord Modular uses a visual programming language and environment.

Using the Nord Modular is different from using a physical analogue modular system in a number of important ways. The largest, probably the most significant, is that programming analogue systems is embodied in the physical world, connecting physical patch cables to physical modules, whereas in the Nord, this experience is recreated in software for general-purpose computers. Because the Nord is a software simulation of a modular synthesizer, some constraints are different. In a physical modular synthesizer, users are restricted to a certain number of modules of a given type; if you run of out oscillators, you must do without or buy a new module. In the Nord Modular, patches are limited by the amount of DSP processing power: you can choose to trade off oscillators against filters or envelope generators, but only up to a fixed limit. As a side effect of being

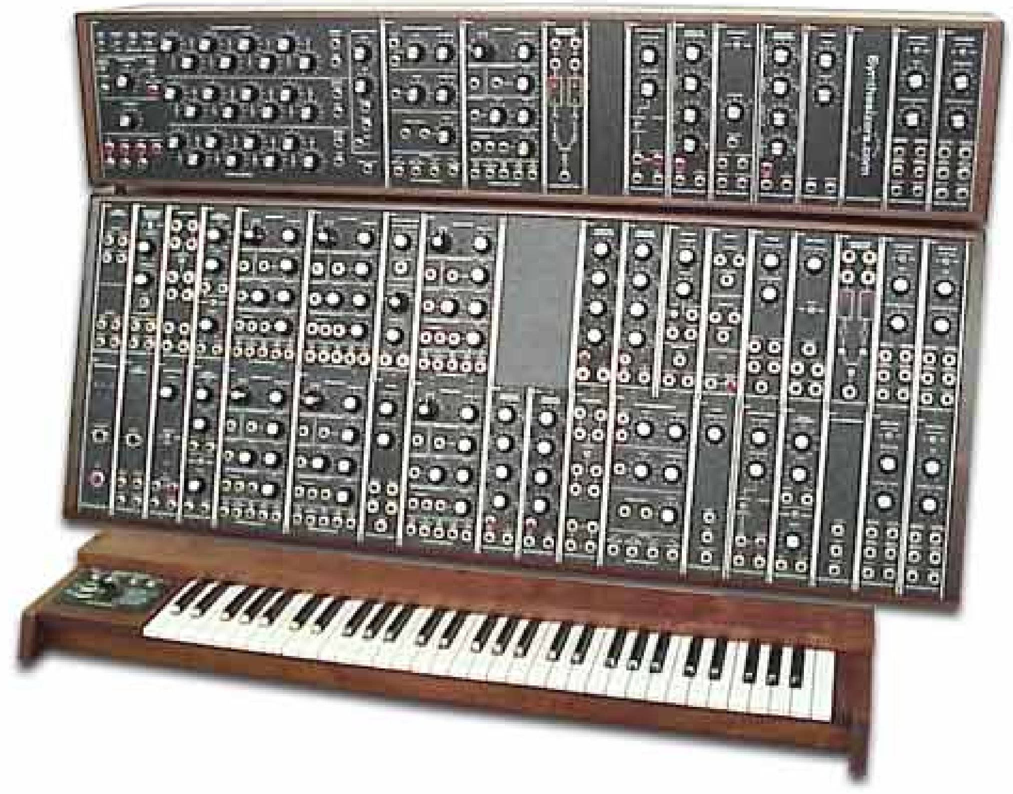

Figure 1: An analogue modular synthesizer

able to choose modules, users can choose the layout of the modules in a patch: as Figure 1 shows, physical module positions are fixed in a traditional modular synthesizer.

The topic of this paper is an analysis of the visual programming language used to create patches on the Nord Modular, with particular emphasis on programming style. Before starting this work, we had several hypotheses about what we could expect from a dataflow visual programming language, compared, say, to a physical modular synthesizer system:

- The programming environment would support the tasks of constructing and debugging modular patches.
- Modules would typically be positioned so that signals would flow left-to-right, top-to-bottom.
- Closely related modules would typically be placed near each other so patch cables would be short.

In this paper, we test these hypotheses by applying program visualisation techniques to the Nord Modular programming language to analyse the programming style with which it is habitually employed. Section 2 begins by describing the Nord Modular system in more detail: the hardware, the programming language, and the programming environment. Then, section 3 describes *patch maps*, our primary visualisation of Nord Modular programs, that addresses the question of debugging support. Section 4 then describes *patch wheels*, our visualisation that shows signal flow direction within programs, that is the core of our analysis of programming style. Section 5 builds upon this visualisation, aggregating information about several hundred patches to give overall information about general trends in programming style. Section 6 then places this work in the context of related work on visual programming, and section 7 concludes the paper.

2 Clavia Nord Modular

In this section we discuss the Nord Modular system. We provide a brief introduction to the hardware, and then concentrate on the programming language and environment.

2.1 Hardware

The Nord Modular synthesizer looks rather different from a traditional modular synthesizer: it's an undifferentiated red box with a few knobs and switches that is connected to a personal computer via a standard MIDI interface (compare Figure 2 with Figure 1).

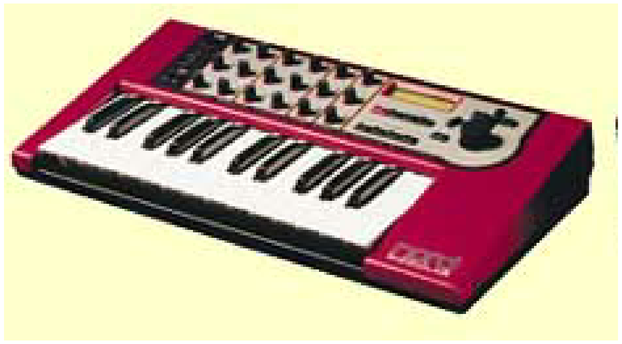

Figure 2: Nord Modular Hardware

The main contents of the box are a number of Motorola DSP56000 digital signal processors which support the synthesis engine: the knobs and switches can be used to control paramters of the synthesizer modules in performance without recourse to a computer. There are three models of the hardware (standard verisons with and without a musical keyboard, and a “mini” version with one processor rather than an array of four or eight), however the programming is the same for each model.

2.2 Software

The key aspect of programming the Nord Modular is designing (or debugging or understanding) Nord Modular programs, known as patches: selecting modules, patching them together with patch cables, and setting internal module parameters. Figure 3 shows a very basic Nord Modular patch. This comprises four modules: an input module named Keyboard1, an oscillator (OscB1), an envelope generator (ADSR-Env1) and an output module (2 outputs1).

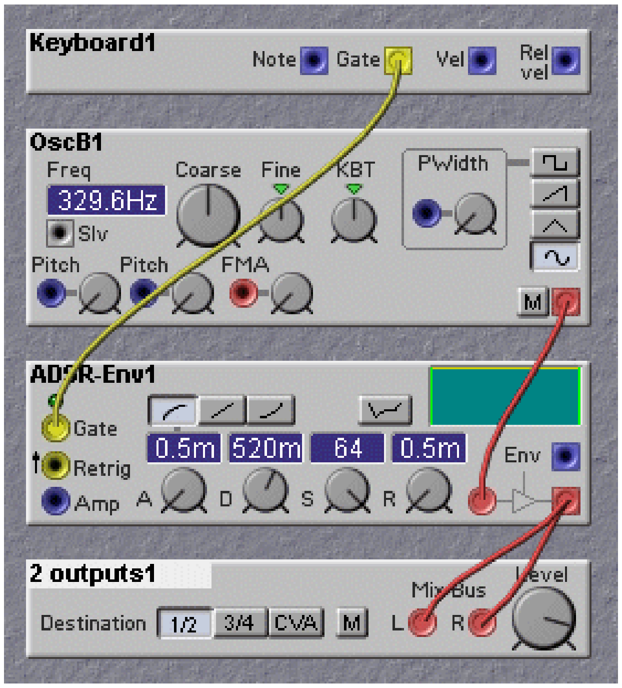

Figure 3: A Simple Modular Patch

The Nord Modular system includes approximately a hundred different types of modules, including oscillators, low frequency oscillators, filters of various categories, audio inputs, envelope generators, clocks, sequencers, logical and arithmetic operations, and audio outputs. Every module is an instance of a particular module type, and can be given a textual name by the programmer.

As the figure shows, programmers select modules, and then lay them out. The editor constrains module positions to one of four columns about thirty rows without scrolling: all modules are one column wide, however, different modules have different heights. Modules are then connected with “patch cables”, from output connectors (square socket icons) to input connectors (round socket icons); inputs may be futher daisy-chained to other inputs. The sockets and patch cables are coloured to represent the type of signal that flows through them: audio signals are coloured red; low frequency signals blue; control signals are coloured yellow or grey.

In Figure 3, outputs from the keyboard input and oscillator input are patched to inputs of the envelope generator, and the envelope's output is patched to the inputs of the output module.

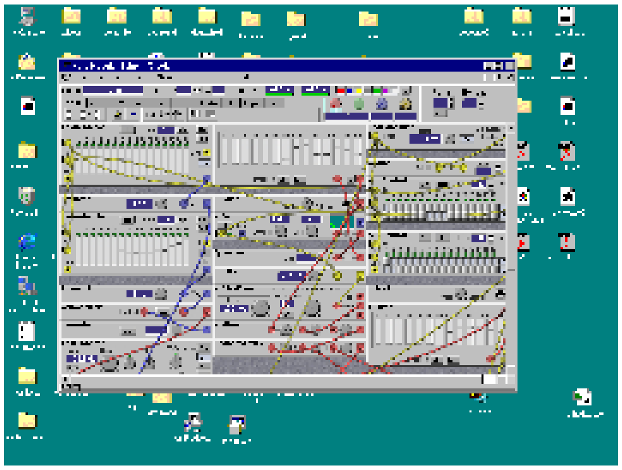

Figure 5: Nord Modular Editor

Each module has a number of parameters controlled by knob or switch icons. These can be adjusted to tune the module's operation, and can also be bound to physical knobs on the Modular hardware so that they can be controlled in real-time. In Figure 3, the oscillator has knobs for parameters named Coarse, Fine, KBT, PWidth, Pitch (two), and FMA, plus buttons to select a waveform, and a mute button marked “M”.

A simple patch, or a single module, can generally be understood quite easily in isolation. More complex patches, such as that shown in Figure 4, are rather more difficult to interpret!

2.3 Programming Environment

Patches are produced using a programming environment called the Nord Modular Editor[1], then downloaded automatically to a Modular synthesizer if one is connect to the computer running the editor. Figure 5 shows the programming environment. This is basically a standard Microsoft Windows (or Macintosh) program that communicates with the Modular hardware using standard MIDI interfaces, protocols, and cables.

As well as providing the interface for users to create and edit patches, the Modular editor also allows patches to be stored to standard file systems. Unlike many other synthesizers, where patch information is only available encoded in MIDI System Exclusive formats, the Nord Modular patches are stored in standard ASCII files (one of the advantages of a system that relies on commodity computer support). This file format is quite simple, and has been designed to be easy to exchange between Modular users. For example, on a web page, a textual link (or snapshot image of a patch in the Editor) can be linked to the patch file. Clicking on the patch will then automatically open the patch file in the editor and load it into a Modular synthesizer, assuming one is attached.

The ease with which patches can be archived and shared has meant that many Modular users have made patches available; the manufacturer, Clavia, also collects user-contributed programs on their web site. This ensures there is a readymade corpus of publically available Modular patches for visualisation and analysis, probably more so than with any other visual programming language.

[1]The Nord Modular Editor is available to download free from http://www.clavia.se/nordmodular.

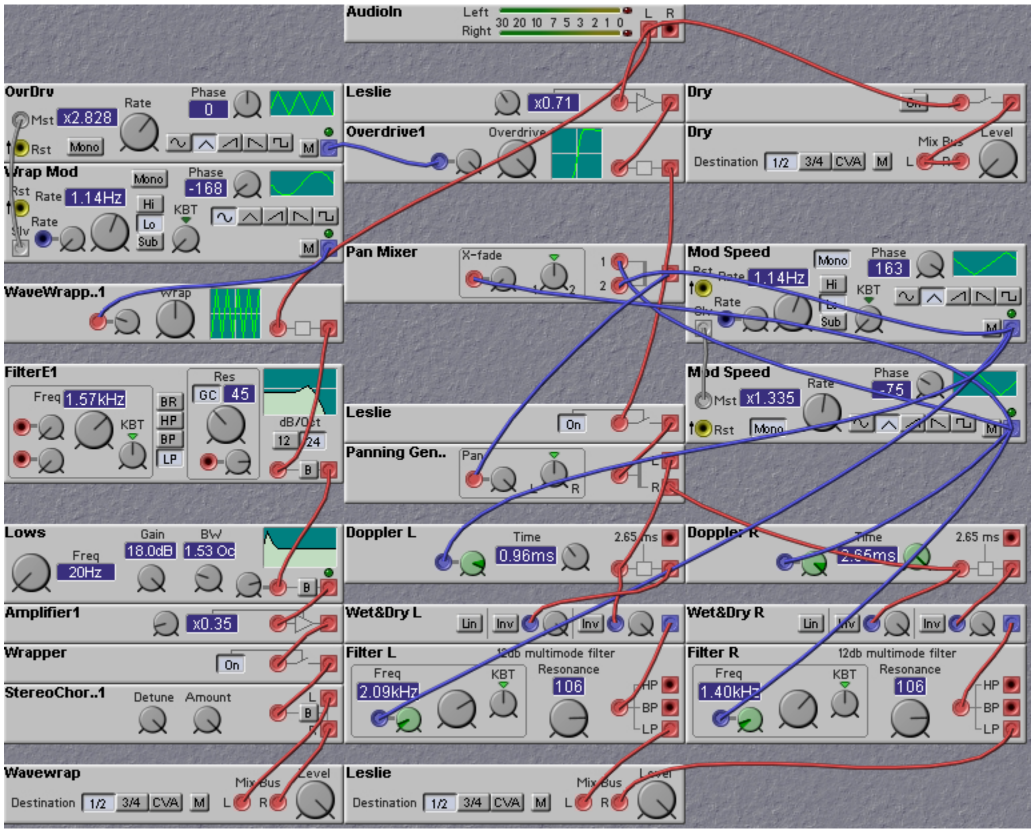

Figure 4: A More Complex Patch

3 Patch Maps

The standard Nord Modular Editor's display is well suited for constructing new patches: selecting modules, wiring them together, and setting their parameters. Often, when first "reading" an unfamiliar patch, however, a programmer needs to trace the patch backwards from the output modules (that produce the sound) through envelopes, filters and audio modifiers back to the osciallators that create the sound. This task can be quite time consuming in the standard display: simply identifying the output modules to work out where the backwards tracing should begin is difficult, as output modules (like all other modules) are uniformly displayed as grey boxes with a partiuclar set of controls.

We attempted to address this problem using software visualisation. Visualisation is often applied to textual programs, say to assist debugging or to highlight important aspects of algorithims, however, we were interested in applying software visualisation techniques to visual programs.

Our first visualisation of Nord Modular patches are what we call *patch maps*. A patch map basically shows all the modules and cables in a patch, but only the modules and cables: it abstracts away any details of the module design, connectors, parameters, and so on. Patch maps use colour coding to support tasks involving comprehension. The patch map displays a fairly simple *module box* for each module in the patch, labelled textually with the module name and type, and draws straight lines (with arrow heads) between modules for each cable. Unlike the editor's display of the patches, where all modules are uniformly grey, every module in a patch map is coloured according to their category: input modules are green, output modules are red, oscillators are blue, low frequency oscillators are dark blue, and so on. Also unlike the standard display, where input and output connectors

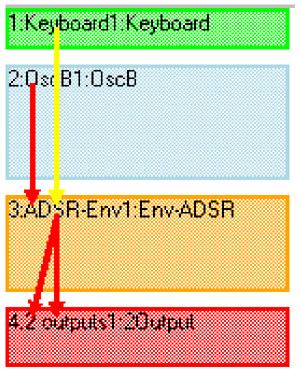

Figure 6: A Patch Map for a Simple Patch

may be located anywhere on a module and are shown as sockets, in a patch map input and output connectors are not shown explicitly and cables begin and end in two rows across the top of each module box. The cables are coloured with the same colour they have in the underlying patch

Figure 6 shows a simple patch map, in fact, a map for the patch shown in Figure 3. As described above, each module is represented by a coloured module box, annotated with the module's name and type, and cables are shown by arrowed lines. Comparing the two figures, the patch map is simpler (and much less impressive) than the patch itself; clearly large amounts of time and effort have been expended crafting the design of the standard displays in the Modular Editor. In terms of the overall topology of the patch, both displays have the same overall geometry: indeed, the relationships between the map and the standard patch display is quite straightforward.

The patch map has a number of advantages over

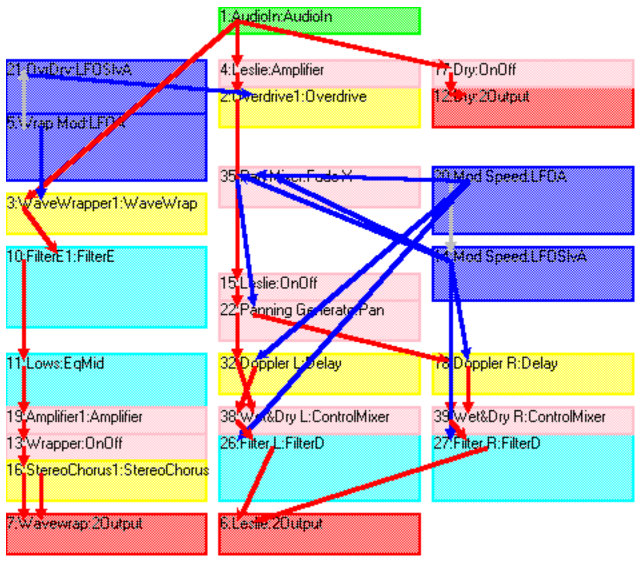

Figure 7: A Patch Map for a Complex Patch

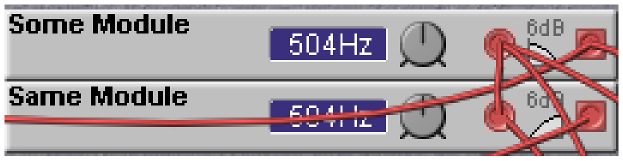

Figure 8: Two Different Module Types?

the patch display itself, along with a couple of obvious major disadvantages: it does not display any information about the details of the modules, and it is nowhere near as graphically !

The first advantage of patch maps is that the colour coding makes it easier to identify particular categories of modules in a patch map than in the standard display. Thus, when reading a patch, users can easily pick out different types of modules — output modules in particular — and then work backwards from there to understand the signal flow of the whole patch. For example, compare identifying output modules in the patch shown in Figure 4 with finding the red-coloured modules in the patch map for the same patch shown in Figure 7, which is printed at similar scale.

Identifying particular type of module (say a 24db filter rather than just all filter category modules) is similarly easier using the patch map rather than in standard display. Although the map does not distinguish between different types of modules visually (only different categories are distinguished using colour) each module box is labelled textually with the name of the type, and users need only inspect the modules of a particular type, rather than all the modules in the program. In the standard display, patch types are distinguished only by fine details of the their graphical design, although a pop-up legend appears when modules are double-clicked. While some modules have textual legends, they are not standard across all modules, and do not necessarily match a module's name (see Figure 8).

The second advantage of patch maps is more subtle: a patch map can be smaller than the standard patch display. This can be seen by comparing Figures 3 and 6. The patch map's simple boxes can be scaled flexibly, while the fine crafted graphics of the Nord editor can be displayed at only one scale. This means that a patch map can display more modules in

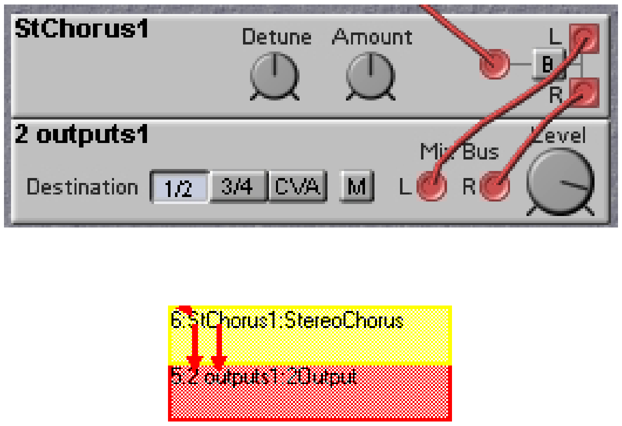

Figure 9: Chorus into Output-2: Excerpts from Patch and Patch Map

a patch than the standard editor in the same amount of screen real estate, thus giving an overview of the structure of an entire patch.

Other advantages of patch maps relate to cables rather than to modules. The position of connectors on patch map module boxes are syntatically simple (left to right across the top of the box) whereas in the standard display connectors are positioned arbitrarily on the face of modules to reflect module semantics. Although outputs are usually to the right and bottom of a module, this is by no means a rigid convention, and inputs may also be along the bottom edge or to the right of a module. As a result, some logically parallel signal flows in a patch can become contorted in the standard display, but will be displayed cleanly in a patch map. For example, Figure 9 compares the connection between the stereo chorus and an output module in a patch and a patch map; the patch displays signals moving diagonally backwards (downwards but to the left), while the patch map makes the parallel stereo signal flow clear.

A final advantage is that patch maps display arrowheads on cables to show the direction of signal flow, while the standard display does not. Although this is a minor visual issue, it is certainly useful when first approaching a patch and its overall topology is unclear, especially if the patch cords in question do not generally flow top down or left to right. We discovered the advantages of arrowheads almost accidently: in fact, only after we had drawn them on patch maps did we realise that the standard display did *not* provide any indication of the direction of signal flows along patch cables.

All these advantages can be seen more readily on a patch map for a larger and more complex patch: Figure 7 shows a patch map for the more complex patch in Figure 4. Comparing Figure 4 and Figure 7 we see some of these advantages more clearly: the map will fit on a small screen where the patch would generally need to be scrolled around the editor window; it is easier to locate important modules (such as the three independent output modules) in the map than in the patch; the overall signal flow is clearer in the map, because the map has fewer distracting details on individual modules, and because the map's arrowheads give the direction of the signal flow.

The key point from the consideration of this Patch Map visualisation is that it provides specialised support for reading modular patches, in particular, when learning to use an unfamiliar patch. Unlike Patch Maps, the Modular Editor colour codes cables, but not modules; and does not draw arrowheads. Both of these graphical features could be included in an extended version of the Modular Editor, which would enable the Editor to provide the same support as currently provided by patch maps.

4 Visualising Patch Cable Layouts

After visualising the layout of whole patches with patch maps, we became interested in the programming style used by Modular programmers. Because Nord Modular patches are quite a simple dataflow language, without any abstraction or definition mechanisms there is not that much scope for style, the degrees of freedom being primarily the layout of modules in a patch and secondarily the names given to particular modules — these two being the only parts of a patch that can be changed *without* altering the sound processing in the patch.

Because patches range in size from two or three modules up to thirty or more, we chose to consider the flow of signals through a patch — that is, the relative positions of interconnected modules — rather than modules' absolute positions. We hypothesized this to be quite straightforward, flowing predominantly left-to-right and also top-to-bottom, as in reading a textual program.

To investigate these cable layouts, we developed a second visualisation that we called the *wheel* visualisation, shown in Figure 10. The wheel visualisation was inspired by the "cartwheels" displayed during one-day international cricket matches to visualise the strokes played by a batsman. Our patch wheels display the relative direction and distance of each cable in a patch, as if every cable in the patch was translated so that it started from the centre of the visualisation.

Figure 10 shows a patch wheel for the simple patch in Figure 3 (compare with the patch map in 6). This "wheel" is degenerate because the signal flow in the patch is straight down. Figure 11 shows a patch wheel for the more complex patch in Figures 4 and 7. The wheel visualisation shows the cables in the same colours as the standard patch (red for audio signals, blue for low frequency modulations, yellow or grey for control voltages).

Both these figures illustrate one important feature of Nord Modular patches: that signal flow is *not* solely rightwards and downwards. Both these patches (indeed, most patches) have significant cables flowing right-to-left, bottom-to-top, or both. Figure 12 shows a patch wheel for a particularly odd patch, where the

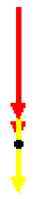

Figure 10: A Patch Wheel for a Simple Patch

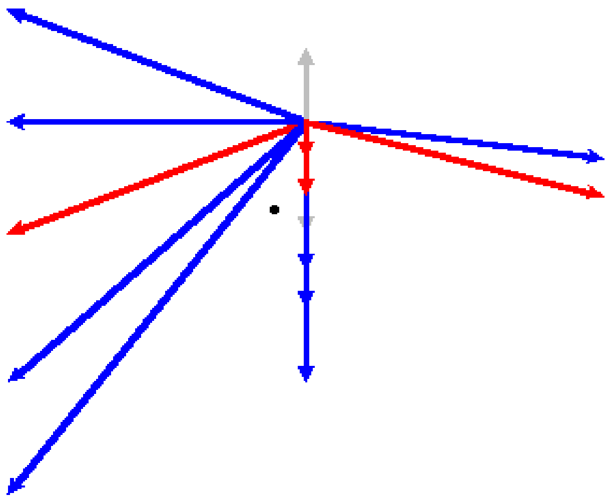

Figure 11: A Patch Wheel for a Complex Patch

predominant flow is rightwards and upwards, it appears: Figure 13 shows the corresponding patch map. To give an idea of the predominant direction of a patch, we extended the basic wheel visualisation with a filled black circular marker drawn to mark the average of all the cable vectors. This shows that the signal flow in the simple patch is downwards (as would be expected), while the signal flow in the more complex patch is primarily downwards, but also flows slightly right to left.

Whatever the dominant cable direction in a patch, there can always be some cables that do not conform closely to the average direction. Some patches produce a characteristic shape on the wheel display, where different kinds of cables typically flow in different directions, as in Figure 14: audio flows rightwards or downwards, LFO modulations flow upwards and leftwards, and control signals flow left and down. Figure 15 shows the corresponding patch: note the sequencer and clock modules in the upper right — the source of rightwards control signals flowing down to the drum synth modules and up to the oscillators.

5 Layout of all Nord Patches

With the patch wheel visualisations giving overviews of the (relative) layout of individual patches, our interest was piqued in the kinds of layouts used generally. The Nord Modular comes with 467 so-called "factory patches" — that is, patches designed and installed when the synthesizer comes from the factory. To investigate this, we modified the patch wheel visualisation to analyse all the factory patches, and draw only the black average vector marker. The resulting visualiastion, shown in Figure 16 is a scatterplot of the average vector for each factory patch: the average displacement for the scatterplot is shown by a red marker from the origin shown as a cross.

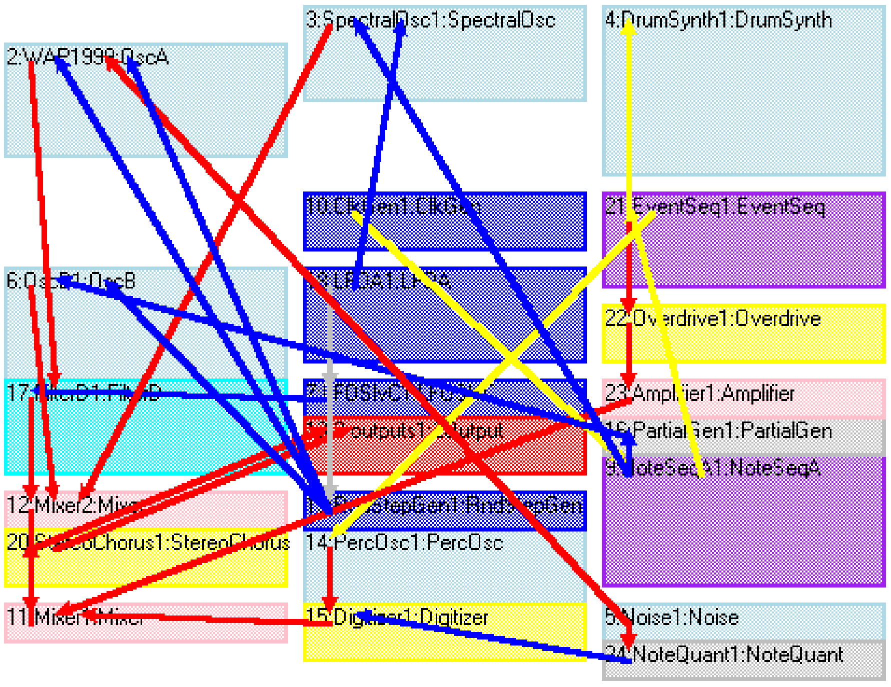

Figure 13: A Patch Map where Flow is Reversed

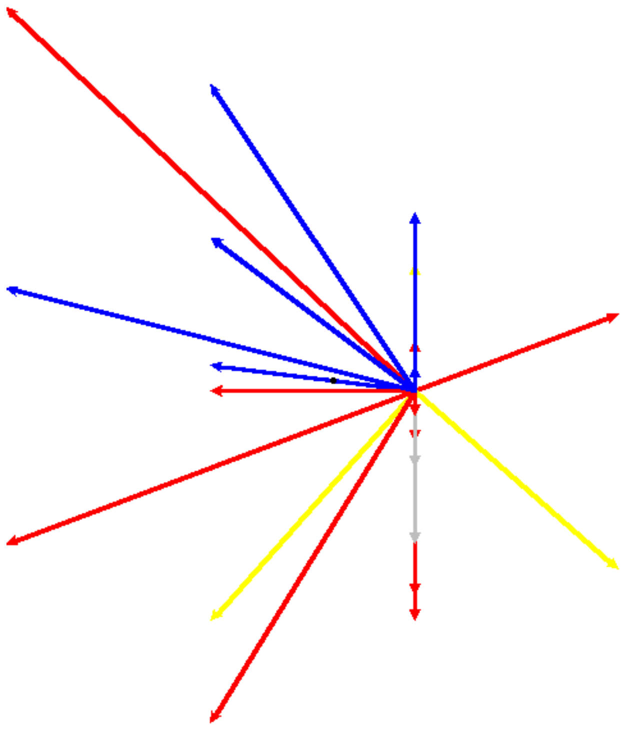

Figure 12: A Patch Wheel where Flow is Reversed

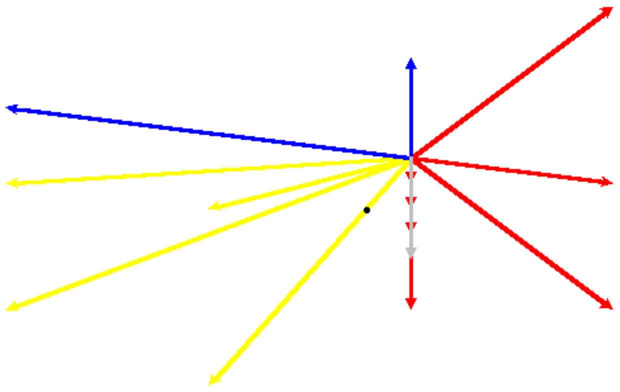

Figure 14: A Patch Wheel where Flow is Segmented

This scatterplot visualisation illustrates two main features of Nord Modular programs. First (as we had hypothesised) the average direction of signal flow in the programs is left-to-right and top-to-bottom. This is shown by the red marker being displaced by 0.178 columns left and 2.85 rows down from the origin. Note that because modules are long and shallow rectangles, one column is approximately sixteen times as large as a row, so the overall angle of displacement is approximately on the diagonal.

Second (and we had not expected this) there is clearly a reasonably large distribution of the average cable vector in modular programs. While in some programs this is quite strongly left to right and top to bottom, in others it is left to right only, or even right to left or top to bottom.

We also determined the average modulus (length) of each cable: at 7.65 this is much larger than the modulus of the average cable vector at 2.86. What this means is that there must be a large distribution of cable directions within each patch — if the cable directions were completely evenly distributed (and if each cable the same length) the modulus of the average vector would be zero, whereas if the cables all flowed in the same direction the modulus of the average vector would be the same as the average modulus of each cable. This is illustrated graphically in many of the patch wheel figures.

To summarise: although cables, signal flow, and layout in Nord Modular patches tends towards left-to-right, top-to-bottom, there is a wide variation, both within individual patches, and across the patch library as a whole.

6 Related Work

Dataflow visual languages are arguably the most common form of visual language, and there are a number

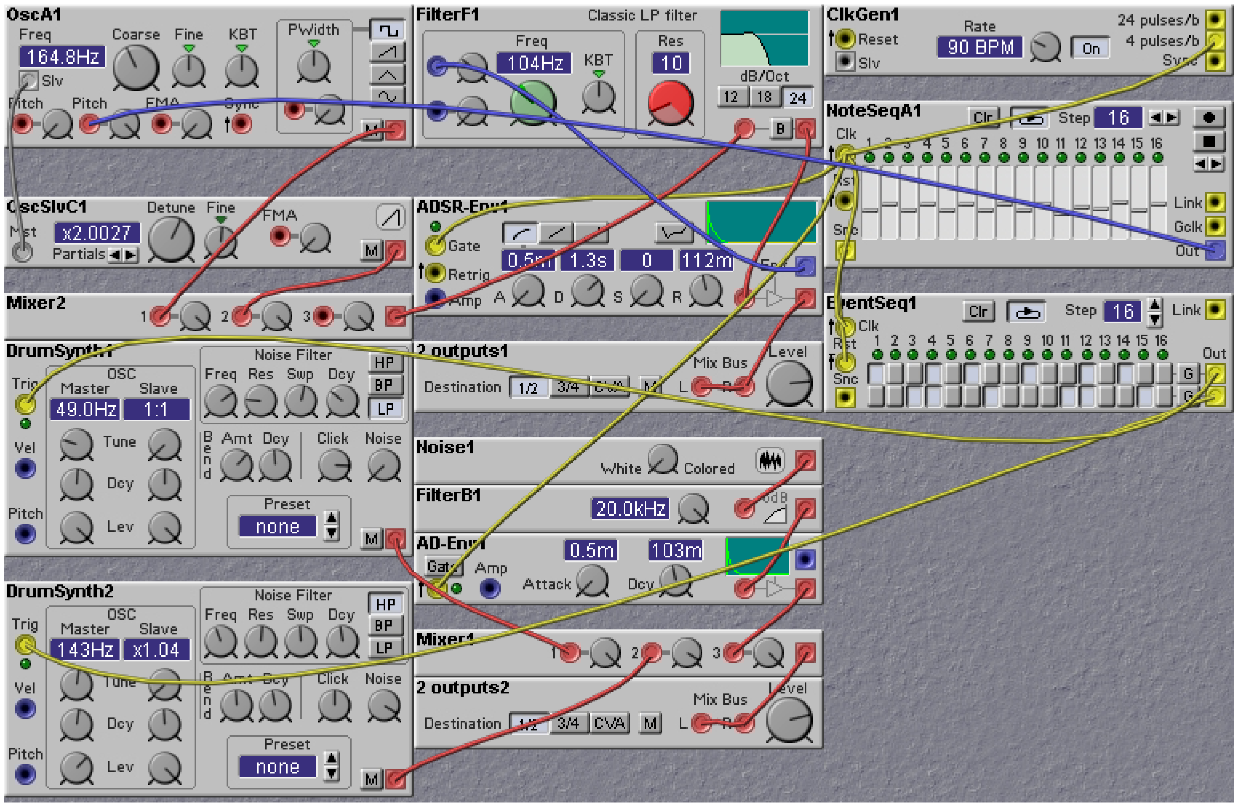

Figure 15: A Patch where Flow is Segmented

of commercial systems based upon such languages, including the IRIS Explorer [20], LabView [1, 19], VEE [15], CAPRE, [14], and MAX [7, 9], as well as the Nord Modular patch language [6]. Given this widespread practical acceptance, it is perhaps surprising that there has been little standalone research on visualising programs in these languages or analysing the kinds of programs these languages are acutally used to write.

Most research on visualisation of visual programs is generally subsumed with research on the visual programming environments themselves — indeed, one of the reasons for the research community moving away from dataflow languages is that the execution of these programs is not easy to visualise. Rather, following Sketchpad [25] once again, many modern (non-dataflow) visual languages incorporate dynamic visualisations directly into the programming model, so that whenever a program runs it is visualised: Toontalk [16], Agentsheets [22], and VIPR [5] are just three examples of this approach.

There has been some work on specialised visualisation of visual programs, however. Burnett has applied software visualisation techniqes to support testing of Forms/3 programs [23], and Grundy and Hosking have applied some program visualisation techniques to software engineering modelling languages [11, 12] — in one case, successfully visualising a “gedanken” notation that was never designed to be executed [10].

Probably because most visual languages do not have a large user base, the practice of the visual languages community has been to adopt empirical usability evaluations to understand how languages are used, or to measure the effectiveness of individual small details of language designs [23, 3], or researchers may participate in programming communities to evaluate their use of langauges [4, 18].

Because they are time consuming, usability evaluations or participant observation are generally limited to tens of subjects working on tens of programs. Surveys can provide information from many more subjects, but surveys cannot engage with actual programs, only programmers opinions and beliefs about their programs [26]. Probably closest in spirit to our work is the empirical analysis of spreadsheet programs, where accountants or auditors work through a corpus to identify features of programs, such as cell error rates [21]. In the mainstream textual language community, analyses of programs are carried out mainly to improve implementations: analyses and critiques of programming style are generally based on single examples, drawing on literary critisism [17, 24] or the patterns movement [2] for models.

In comparison with usability evaluation or participant observation, an approach based on corpus analysis requires a sample of several hundreds of programs, but does not require detailed analysis of the process by which those programs were written. Corpus analysis is best suited to investigating the *parole* [8] of a language — the way it is used in practice — while other techniques can provide more specific information about the design of languages themselves.

7 Discussion and Conclusion

In this paper we have investigated applying program visualisation to the dataflow visual language for patches for the Nord Modular synthesizer. At the start of this exercise, we established several hypotheses about Nord Modular programs: in general these have been borne out by our visualisation research, altough not as strongly was we had expected.

The first hypothesis was that the Nord Modular programming language would support the tasks of constructing and debugging modular patches. While the interface seems to support construction tasks well, it does not provide as much support for debugging Modular patches as we expected. Our alternative visualisation, patch maps, applies colour coding to modules as well as to cables, draws arrowheads on

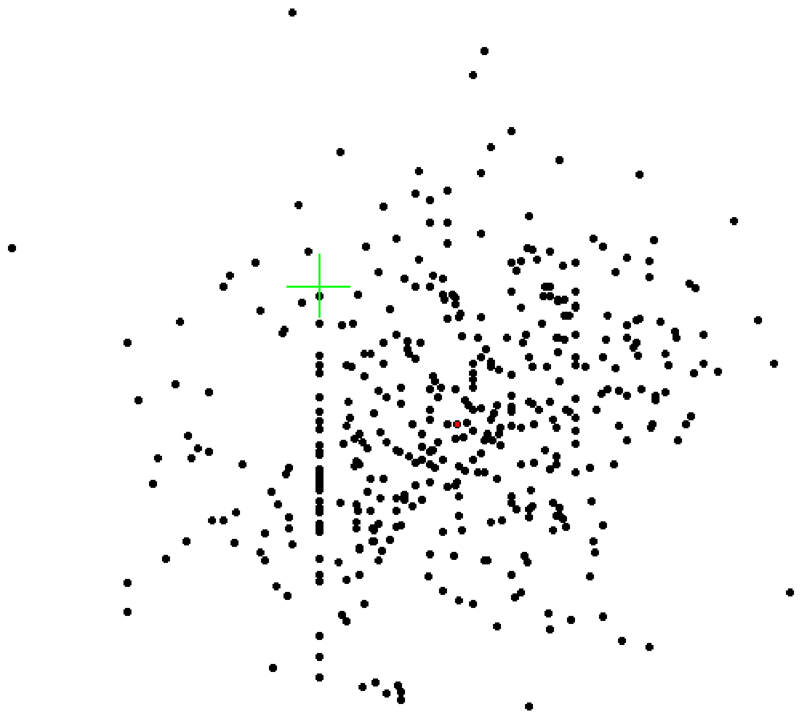

Figure 16: Scatterplot of average cable vector for 467 patches

patch cables, and uses stereotyped locations for input and output sockets. Compared with the standard display, patch maps provide some better support for finding particular types of modules in a patch and for understanding a patch's signal flow, and thus for reading and debugging patches.

The second and third hypotheses were about programming style related to module layout and consequent signal flow — we hypothesized that signals would flow from left-to-right, top-to-bottom, and that closely related modules would be placed near each other so patch cables would be short. Again, these hypotheses were sustained, but rather weakly: the main result of our visualisations of signal flow is that while these hypotheses are true at a large scale, many individual patches have idiosyncratic signal flow and patch cable lengths, and that signal flow is often tightly segmented according to cable type.

The key technical advantage allowing us to produce these visualisations, and to perform corpus analysis in particular, is that the Nord Modular patch files are stored in a simple ASCII format. Performing a similar analysis on many other visual languages would be much more difficult, because we would first have had to parse a much more complex binary file.

We do need to note that these results are somewhat preliminary. For technical reasons due to the Modular patch file format, some patch cables have been omitted from the patch wheel and scatter plot visualisations — it appears this is due to cables that "daisy-chain" inputs to other inputs, rather than outputs to inputs. Also, each Nord Modular program actually consists of two separate patch areas — a polyphonic voice area (PVA) where modules are duplicated for polyphonic patches and a common voice area (CVA) which is shared across all polyphonic voices: our current tools analyse only the polyphonic voice area. We plan to extend our tools to address these issues, but do not expect these to markedly impact our results.

We also plan further visualisation work on Nord Modular programs: indeed, there seems quite some scope for research since only a small amount of standalone visualisation has been performed upon dataflow visual languages, and very little corpus analysis has performed upon visual langauges of any type. We are considering visualising the "real estate utilisation" across patches — that is, which modules tend to occupy which positions in the screen layout, or conversely, how particular screen locations are generally used. We may also investigate the utilisation of particular modules within patches, perhaps comparing this against amount of DSP processing load imposed by the module, and the use of secondary notation, particularly the names programmers assign to modules. More ambitiously, we would like to experiment with providing automatic layout support for modules (to reorganise patches to minimise cable length and cable crossings) and with program slicing (so that all the modules producing one part of a patch could be automatically extracted from a patch making multiple sounds). Finally, to date we have worked only with the four hundred Nord Modular factory patches supplied or gathered by Clavia: we hope to extend this work to analyse the several thousand patches available on the Internet.

References

[1] Ed Baroth and Chris Hartsough. Visual programming in the real world. In Margaret M. Burnett, Adele Goldberg, and Ted G. Lewis, editors, *Visual Object-Oriented Programming.* Prentice-Hall, 1995.

[2] Kent Beck. *Smalltalk Best Practice Patterns.* Prentice-Hall, 1997.

[3] Alan F. Blackwell. Pictorial representation and metaphor in visual language design. *Journal of Visual Languages and Computing*, 12(3):223–252, 2001.

[4] J. Carroll and M. Rosson. Paradox of the active user. In J. Carroll, editor, *Interfacing Thought:*

Cognitive Aspects of Human-Computer Interaction. MIT Press, 1987.

[5] Wayne Citrin, Soraya Ghiasi, and Benjamin G. Zorn. VIPR and the visual programming challenge. *Journal of Visual Languages and Computing*, 9(2):241–258, 1998.

[6] Clavia DMI AB, Sweden. *Nord Modular Manual*, v3.0 edition, 1999.

[7] Cycling '74. *MAX Reference*, 2001.

[8] Ferdinand de Saussure. *Cours de linguistique générale.* V.C. Bally and A. Sechehaye (eds.), Paris/Lausanne, 1916.

[9] Peter Desain, Henkjan Honing, Robert Rowe, and Brad Garton. Putting Max in perspective. *Computer Music Journal*, 17(2), 1993.

[10] J.C. Grundy and J.G. Hosking. ViTABaL: a visual language supporting design by tool abstraction. In *IEEE Symposium on Visual Languages*, 1995.

[11] J.C. Grundy and J.G. Hosking. High-level static and dynamic visualisation of software architectures. In *IEEE Symposium on Visual Languages*, 2000.

[12] John C. Grundy, John G. Hosking, and Warwick B. Mugridge. Serving up a Banquet: Towards an environment supporting all aspects of software development. In *Software Engineering: Education and Practice (SE:E+P)*, Dunedin, 1996.

[13] Richard Hagen. Blue ribbon soundworks' bars and pipes professional. `http://www.richardhagen.org`.

[14] Gregory A. Hansen. *Automating Business Process Re-Engineering: Using the Power of Visual Simulation Strategies to Improve Performance and Profit.* Prentice Hall PTR, 2nd edition, 1997.

[15] Robert Helsel. *Visual Programming with HP-VEE.* Prentice Hall PTR, 1997.

[16] Ken Kahn. Toontalk — an animated programming environment for children. *Journal of Visual Languages and Computing*, june 1996.

[17] Brian Kernighan and Ken Plauger. *The Elements of Programming Style.* McGraw-Hill, 1974.

[18] Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* MIT Press, 1993.

[19] National Instruments Inc. *LabView User Manual*, july 2000.

[20] The Numerical Algorithms Group Limited, Oxford. *IRIS Explorer User's Guide*, 5.0 edition, 2000.

[21] Raymond D. Panko. What we know about spreadsheet errors. *Journal of End User Computing*, 10(2):15–21, Spring 1998.

[22] Alexander Repenning and Tamara Sumner. Agentsheets: A medium for creating domain-oriented languages. *IEEE Computer*, 28(3):17–25, 1995.

[23] K. J. Rothermel, C. R. Cook, M. M. Burnett, J. Schonfeld, T. R. G. Green, and G. Rothermel. Wysiwyt testing in the spreadsheet paradigm: An empirical evaluation. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 230–239, June 2000.

[24] Suzanne Skublics, Edward J. Klimas, and David A. Thomas. *Smalltalk with Style.* Prentice-Hall, 1996.

[25] Ivan E. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings AFIPS Spring Joint Computer Conference*, volume 23, pages 329–346, Detroit, Michigan, May 1963.

[26] K.N. Whitley and Alan F. Blackwell. Visual programming in the wild: A survey of LabVIEW programmers. *Journal of Visual Languages and Computing*, 12(4):435–472, August 2001.