Visual Programming, Knowledge Engineering, and Software Engineering

Tim Menzies *

March 4, 1996

Abstract

It is an interesting and exciting challenge to change programming modalities from a traditional text-based approach to a 2-D screen. Based on a survey of current visual programming systems, we find that numerous software engineering and knowledge engineering techniques are required to meet that challenge. Further, we argue that VP systems can benefit from on-going knowledge engineering research on the computational complexity of different representations. Hence, we conclude that the designers of VP systems should be well-versed in a wide range of knowledge engineering and software engineering techniques.

1 Introduction

It is an interesting and exciting challenge to change programming modalities from a traditional text-based approach to a 2-D screen. When faced with such a challenge, we find that a wide range of software engineering and knowledge engineering techniques are required. For example:

- The main problem with constraint-based visual programming languages (e.g. the THINGLAB) is not its visual presentation. Rather, it is the the speed of the underlying constraint satisfaction mechanism. Constraint satisfaction is a well-studied problem in AI (e.g. [13, 14, 29]).
- The designers of the visual expressions (see Section 3) of a VP system often use existing software engineering diagramming tools (e.g. the E-R diagrams of SUPER [8], the structure charts of PICT [15], or petri nets). The use of such familiar software engineering visualisations reduces learning times.

More generally, the central claim of this article is that the design of effective VP languages requires a wide knowledge of software engineering and knowledge engineering techniques. To demonstrate this, we will first define VP and discuss some of its benefits (see Section 2). We will explore the VP field using the framework shown in Figure 1 (taken from a recent survey of the VP field [31]). Figure 1 shows three dimensions along which we can characterise VP systems: the visual expressions (see Section 3), their purpose (see Section 4), and their design (see Section 5). In our conclusion (see Section 8), we will list the numerous knowledge engineering and software engineering techniques used in VP. Further, we will argue that VP designers are ignoring important state-of-the-art knowledge engineering research (see Section 6).

2 What is VP?

2.1 Definition

As a rough rule-of-thumb, a visual programming system is a computer system whose execu-

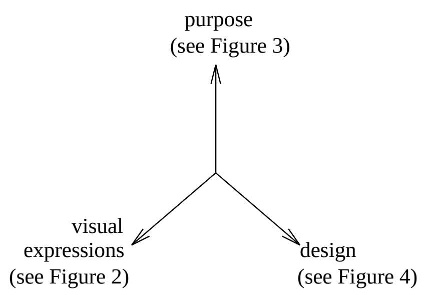

Figure 1: Dimensions of a VP system

*Dept. of Software Development, Monash University, Caulfield East, Melbourne, VIC, Australia, 3145; +61-3-9903-1033; *Email*: timm@insect.sd.monash.edu.au; *URL*: http://www.sd.monash.edu.au/σtimm

tion can be specified *without scripting* except for entering unstructured strings such as `''Monash University Banking Society''` or simple expressions such as `a > 7`. Visual representations have been used for many years (e.g. Venn diagrams) and even centuries (e.g. maps). Executable visual representations, however, have only arisen with the advent of the computer. With falling hardware costs, it has become feasible to build and interactively manipulate intricate visual expressions on the screen.

More precisely, a non-visual language is a one-dimensional stream of characters while a VP system uses at least two dimensions to represent its constructs [5]. We distinguish between a *pure VP* system and a *visually supported* system:

- A *pure VP system* must satisfy two criteria. *Rule #1*: the system must execute. That is, it is more than just a drawing tool for software or screen designs. *Rule #2*: the specification of the program must be modifiable within the system's visual environment. In order to satisfy this second criteria, the specification of the executing program must be configurable. This modification must be more than just (e.g.) merely setting numeric threshold parameters.

- There exists a class of VP systems that are not pure, but are *visually supported* systems. Most commercial VP systems are not pure VP systems, such as Microsoft's VISUAL BASIC, Borland's DELPHI, and IBM's VISUALAGE. For more details on visually supported systems, see Section 4.

2.2 Benefits of VP

Visual programming (VP) is an seen by many as an exciting alternative to traditional text-based computing. For example:

> When we use visual expressions as a means of communication, there is no need to learn computer-specific concepts beforehand, resulting in a friendly computing environment which enables immediate access to computers even for computer non-specialists who pursue application domains of their own. [20]

Green *et. al.* [17] and Moher *et. al.* [33] summarise claims such this as the *superlativist* position; i.e. graphical representations are inherently superior to textual representations. Both the Green and Moher groups argue that this claim is not supported by the available experimental evidence. Further, they argue against similar claims of *information accessibility*; for example:

> Pictures are superior to texts in a sense that they are abstract, instantly comprehensible, and universal. [20]

We will return to the issue of information accessibility below (see Section 3.2). For the meantime, we note that the doubts of Green *et. al.* and Moher *et. al.* regarding the utility of diagrammatic reasoning are not universally accepted:

- Koedinger [23] argued that diagrams can support and optimise reasoning since they can model model whole-part relations.

- Larkin & Simon [25] argue that diagrams can optimise certain types of reasoning (e.g. feature extraction and searching through related concepts). Both the Larkin & Simon and the Koedinger study argue for the computational superiority of diagrams for representing problems that have a two-dimensional component.

- Goel [16] studies the use of *ill-structured* diagrams at various phases of the process of design. In a *well-structured* diagram (e.g. a picture of a chess board), each visual element clearly denotes one thing of one class only. In a ill-structured diagram (e.g. an impressionistic charcoal sketch), the denotation and type of each visual element is ambiguous. Goel found that ill-structured tools generated more design variants (i.e. more drawings, more ideas, more use of old ideas) than well-structured tools.

- Kindfield [22] studied how diagram used changes with expertise level. According to Kindfield, diagrams are like a temporary swap space which we can use to store concepts that (i) don't fit into our head right now and (ii) can be swapped in rapidly; i.e. with a single glance.

3 Expressions in a VP System

3.1 A Spectrum of Visual Expressions

Visual expressions are of at least five types in increasing order of visual extent: text, simple forms, tables, icons, and diagrams (see Figure 2).

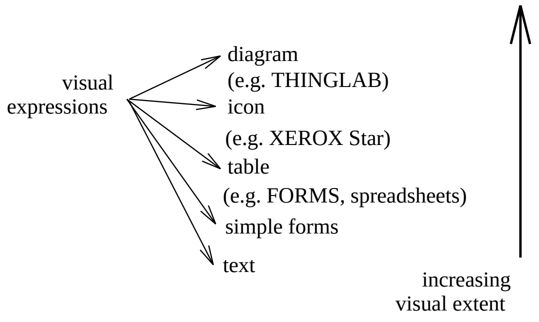

Figure 2: Visual expression types

Purely text-based systems have the lowest-level of visual expressiveness. Simple form-based systems are slightly more expressive. These systems let the user "fill-in-the-blanks" of some prototype specification to generate a more specific specification. Simple form-based systems may require very limited graphical support and can be developed on character-based screens. Such character-based screens limit the representation of (e.g.) two-dimensional graphs. Hence, simple form-based systems are low-end VP systems.

We take care to distinguish between "simple form-based" systems and more general form-based systems such as Ambler's FORMS system [1]. Ambler argues that forms are a really an extension of the tabular/spreadsheet expression. FORMS uses a sophisticated interface in which users can specify the properties of adjacent cells by a single mouse drag across multiple cells.

Tabular expressions make extensive use of the position of their cells. For example, a spreadsheet cell can be the sum of the cell above and the cell immediately to the left. Having mentioned spreadsheets, we stress that the current generation of commercially available spreadsheet packages are weak examples of VP systems. All non-trivial spreadsheet applications require the use of intricate syntax to define formulae or macros. A better example of the use of tabular expressions is the original QBE system [41]. QBE allows a user to "draw" a database query on a character-based screens. The drawing is a little table that reflects the relational structure of the database being queried. Projects and selects can be specified by filling in the cells of the drawn table with restrictions or matches for its values. Joins can be specified by drawing the joined tables, then using the same variable names in the different tables.

In icon-based languages (e.g. XEROX STAR [35]), the position of the icon usually does not effect the services offered by that icon. Typically, users can click on the icon to access a menu of services. However, moving the icon around the screen can represent the transfer of data or the application of some function to some data (e.g. moving a file between a directory).

Diagrammatic systems utilise a wide variety of pictures in their interface. Diagrammatic systems are characterised by "plug-and-play"; i.e. the user creates an diagram by linking up visual components offered from a palette. Often, users can create the visual analogue to a sub-routine by batching up a commonly used diagram into a single icon. For example, a new visual part representing the constraint that a point is (i) on a line and (ii) mid-way between the two end points can be created in THINGLAB [2] by placing these two existing visual constraints into the same "construction view" area. Once there, the net constraints are the union of the individual constraints. This new constraint can be used subsequently in the same manner as the constraints supplied with the start-up system. The new icon for this construct can then be added to a palette thus extending the system's functionality. Alternatively, the icon stays on the screen and only expands out into its full detail if the user clicks on it.

3.2 Which Visual Expression to Use?

In practice, a VP system uses a combination of many of the above visual expressions. For example, MICROSOFT's ACCESS database product implements a QBE variant in which users can specify joins across tables by drawing lines between icons representing the different fields. Projects, selects are specified via a tabular interface. The data dictionary is controlled by a form-based interface while screen designs are specified by an iconic interface.

Interestingly for the claim made in our introduction, many icon-based VP systems adopt diagramming conventions that mimic conventional software engineering practice (e.g. the E-R diagrams of SUPER, the structure charts of PICT, or petri nets). Recall the doubts of Green *et. al.* and Moher *et. al.* mentioned in Section 2.2. Both Green *et. al.* and Moher *et. al.* explored these claims experimentally. Their subjects attempted some comprehension task using both visual expressions and textual expressions of a language. This study rejected the information accessibility hypothesis (i.e. pictures are instantly understood) when they found that novices had more trouble reading the information in their visual expressions

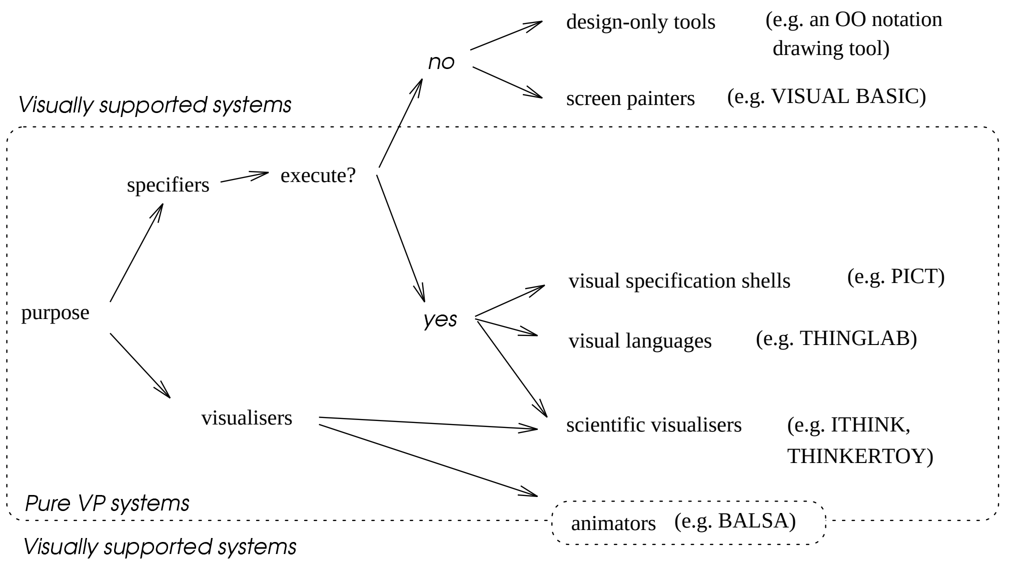

Figure 3: Purpose of a VP system

than experts. That is, the information in a diagram can only be accessed after a training process. The use of standard software engineering diagramming conventions avoids the need for a training program in order to understand the icons of a VP tool.

4 Purpose of a VP System

Figure 3 shows a rough characterisation of the *purpose* of a VP system: i.e. *specifiers* or *visualisers*. Specifiers are tools that let the user record their requirements visually. For example, an interactive E-R diagramming tool could automate the generation of database tables directly from the entered diagrams (e.g. SUPER [8]).

Specifiers may or may not be able to execute their specification within their own visual environment. Two interesting sub-categories of non-executing specifiers are *screen painters* and *design-only tools*:

- *Screen painters* (e.g. VISUAL BASIC) give a developer an interactive point and click environment for placing window widgets (e.g. buttons, list boxes, text fields) on a screen. Typically, these systems can then automatically generate the layout code for these widgets. The developer must then use a textual language to code the semantics and interactions of the widgets

- *Design-only tools* (e.g. an OO notation drawing tool) give a designer an interactive point and click environment for placing design notation icons on a screen. Depending on the tool, relationships between the icons can be specified via specialised edge types. These tools may or may not support automatic code generation. These tools are design-only in the sense that the developer cannot watch the design execute within the environment of the design tool. Note that SUPER is *not* a design-only tool since developers can watch database queries executing within the SUPER environment.

Screen painters and design-only tools do not satisfy *RULE #1* of a pure VP systems (see Section 2.1) since they do not execute. However they are widely used in industry and so, pragmatically speaking, they represent a important category of VP systems. Hence, we call non-executing specifiers visually supported systems.

Executing specifiers can be divided into *visual specification shells* and *visual languages*. A visual shell hides a conventional syntactic language beneath an visual specification environment. The shell executes by translating its diagrams down into the underlying language (for example, PICT converts its structure charts into a subset of PASCAL [15]). A visual language allows the user to specify new language primitives. For example, recall the new constraint added in the above THINGLAB example. Specifiers may include a visual trace facility. For example, whenever control moves to a part of a program, its associated icon may highlight.

Visualisers can be divided into *scientific visualisers* and *animators*. Animators try to represent in a comprehensible way the inner-workings of a program. In the BALSA animator system [4], students can (e.g.) contrast the various sorting algorithms by watching them in action. Note that animation is more than just tracing the execution of a program. Animators aim to *explain* the inner workings of a program. Extra explanatory constructs may be needed on top of the programming primitives of that system. For example, when BALSA animates different sorting routines, special visualisations are offered for arrays of numbers and the relative sizes of adjacent entries.

Animators may or may not be pure VP systems. BALSA does not allow the user to modify the specification of the animation. To do so requires extensive textual authoring by the developer. BALSA therefore does not satisfy *RULE #2* of pure VP system a visually supported system (see Section 2.1). However, there is no theoretical reason why future animation system could not permit visual specification of the animations. Hence we draw animators in Figure 3 on the border of pure VP and visually supported systems.

Scientific visualisers (e.g. THINKERTOY [18] and ITHINK [21]) are tools for support simulations. Arbitrary networks of computational devices can be drawn and executed. Tools are provided for reporting visually the output of the executions. A particular focus of current scientific visualisers is the display of changes to continuous variables over time. Scientific visualisation requires the presentation of large amounts of data. THINKERTOY provides the user with numerous visual tools that support (e.g.) the graphical displays of time-varying values; crystal growth across some 3-D terrain; and the manipulation of data values by user-specifiable filters. Since the user can modify the specification of the simulation within the visualiser's environment, scientific visualisers are also be executable specifiers.

5 Designing a VP System

When building a VP system, designers have to specify a *semantic base*, a *syntactic base* and a set of *basic constructs* which can be used in the start up system. For a description of the *syntactic base*, see the above discussion on visual expressions (Section 3). Many systems permit the extension of the basic constructs (e.g.) in the manner described above for THINGLAB (see Section 3). Note that if a base construct does not include some sort of conditional branching, then the VP system can only ever piece together building blocks defined outside the VP system.

Semantics bases include *control-flow*, *functional*, *data-flow*, *constraint-based*, *logic-based* and *procedural-based* (see Figure 4). Procedural-based systems convert their diagrams into some underlying language (e.g. recall that PICT is converted into PASCAL). While this has some advantages (e.g. simple execution), it implies that the idiosyncrasies of the language have to be handled at the visual level. The other semantic bases listed above strive for a simple uniform view of the program structures. Such uniformity simplifies the interface construction, decreases the amount a user has to learn, and promotes a uniform mental model for the user of the VP system.

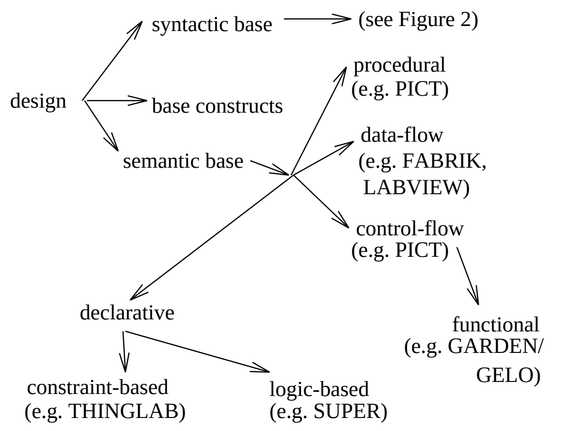

Figure 4: Design choices within a VP system

In a control-flow VP system, users manipulate iterator expressions (e.g. `while-do`, `repeat-until`), conditional expressions (e.g. `if-then`, `case`) and sequence expressions (e.g. `do this, then this, then that`) to explicitly specify the control of the system. Control-flow systems model traditional flow chart systems. Interestingly, PICT is both a control-flow system and a procedural system.

An interesting variant of control-flow systems are functional VP systems (e.g. the LISP-based GARDEN [36]/ GELO [37] systems). In these systems, users manipulate expressions that can recursively contain expressions. All the expressions respond to the same high-level protocol. For example, sending the message `execute` to an `if` expression will result in the conditional part of that `if` being sent the message `execute`. If this returns true, then the `if` expression will send the message `execute` to its action part, which will contain some sequence expressions. These will all be `executed` in turn. The GARDEN system uses the encapsulation of object-oriented programming to implement this common high-level protocol.

A simple trace facility for such functional systems can be implemented as follows. Whenever `execute` is sent to an expression, its visual representation on screen highlights. As this highlight moves over the screen, users can watch the control flow. While simple to implement, this approach has certain intrinsic limitations. Only in a purely functional system can the semantics of an expression be contained in itself and its contained expressions. Certain global processing (e.g. joins across two relational tables) cannot be easily visualised by such a local propagation trace algorithm.

In a data-flow VP system (e.g. LABVIEW [39], FABRIK [27], and the systems reviewed in [19]), control is implicit. Each expression manipulated by the user describes:

- A set of data sources;
- Possibly, a set of conditionals;
- And action(s) to perform when:
  1. All the data sources are available and
  2. The conditions (if any) that use data from those sources are satisfied.

If the actions are performed then the expression is said to have *fired*. After firing, an expression may become an available data source for some downstream expression. At runtime, the pattern of firings ripples out across a network of connected expressions. A simple example of a data-flow system is a petri net. In a basic petri net comprising directed *arcs* and *places* (i.e. edges and vertices respectively), a set of *tokens* move out over the net. A place is fired if out all of its incoming places and none of its outgoing places have tokens. On firing, tokens are removed from each incoming place and one token is deposited in each outgoing place.

Hils argues [19] that a data-flow system is a good design choice for the purpose of scientific visualisation. Given a library of filters that can modify data, it is a simple and intuitive process for users to add filters to data-flow edges in order to transform data. Monitors for data values can be added in the same simple manner.

In a constraint-based VP system (e.g. THINGLAB), the user visually specifies the invariants for each expression. At runtime, a constraint-solver permits manipulations that do not violate the invariants. Declarative constraints can be used to test user-proposed actions or to propose valid-actions. Any user-proposed action that violates the invariants is blocked. Given the current state of the system, a constraint-based system can generate menus of valid actions by generating all variable bindings that would not violate the invariants, given the current state.

Constraint-based systems are a variant on logic-based systems (e.g. SUPER). Such logic-based systems represent their expressions in a uniform recursive manner. Expressions can contain logical variables which can be bound at runtime and only unbound after backtracking on failure. At runtime, a general theorem prover is used to seek a set of bindings that are consistent with the theory. Visual logic-based systems can be traced by updating the display of expressions whenever a variable is bound/unbound. Unlike tracing for functional systems, this logic-based tracing can visualised global variables. For example, all the rows in a database table are global. As the theorem prover searches over the table, the attributes that satisfy the expressions are fetched and displayed.

6 Discussion

In this section we argue that the designers of VP systems should take an active interest in on-going research in other areas. Specifically, we will argue that state-of-the-art research in knowledge engineering can benefit certain VP systems.

6.1 General Explanation Systems

One criticism we have of the BALSA system is that its explanations must be hand-crafted for each task. General principles for explanation systems are widely discussed in AI. Wick and Thompson [40] report that the current view of *explanation* is more elaborate than merely "print the rules that fired" or the "how" and "why" queries of traditional rule-based expert systems. Explanation is now viewed as an inference procedure in its own right rather than a pretty-print of some filtered trace of the proof tree. In the current view, explanations should be customised to the user and the task at hand. For example:

- Paris [34] describes an explanation algorithm that switches from process-based explanations to parts-based explanations whenever the explanation procedure enters a region which the user is familiar with.

- Leake [26] selects what to show the user using eight runtime algorithms. For example, when the goal of the explanation is to minimise undesirable effects, the selected structures are any pre-conditions to anomalous situations. Leake's explanation algorithms require both a cache of prior explanations and (like Paris) an active user model.

Summarising the work of Wick and Thompson & Leake & Paris, we can now diagnosis the reason for the lack of generality in BALSA's explanation system: BALSA lacks (i) the ability to generate multiple possible explanations; (ii) an explicit user model; (iii) a library of prior explanations; and (iv) a mechanism for using (ii) and (iii) to selectively filter (i) according to who is viewing the system.

6.2 Optimisation

AI research could optimise certain VP tasks. For example:

- The major runtime limit a of constraint-based VP systems is the speed of the underlying constraint solver. This is a well-studied problem in AI. For more details, see [13, 14, 24, 29].
- We view the data-flow model used in FABRIK and LABVIEW as a generalisation of the event-driven programming model. Each event handler is like a data-flow node that waits for certain data (events) to arrive. We note that production systems [25] can also be viewed as data-flow systems. Production rules conditions act as demons that await the arrival of certain data elements before executing their conclusion. The AI community has spent considerable effort in optimising production rule systems in both a distributed and non-distributed environments [9, 12].

6.3 Limits to Optimisation

AI has also defined hard limits to certain computational approaches. Seemingly-minor modifications in a representation system can change the runtime speed of that system dramatically [3]. VP designers should therefore be aware that certain seemingly-intuitive interfaces may be undesirable due to the cost of executing them. We expand on this point in the next section.

7 Case Study: Optimising $\mathcal{QMOD}$

Our general case is that VP designers who are ignorant of work in other fields may repeat the mistakes made in those fields. This section presents a case study demonstrating exactly this point.

7.1 About QMOD

QMOD [11] is a qualitative version of the quantitative iTHINK system. The interface to QMOD looks like iTHINK, but with all numbers replaced with the qualitative variables *up, down* or *steady*. When QMOD executes, it must manage the multiple what-ifs generated by executing an under-specified system.

QMOD executes over qualitative theories like Figure 5. Given a set of goal $\mathcal{OUT}$puts and known $\mathcal{IN}$puts, then QMOD can build a set of proof trees $\mathcal{P}$ connecting $\mathcal{IN}$puts to $\mathcal{OUT}$puts. In Figure 5, (i) `x` $\overset{++}{\rightarrow}$ `y` denotes that `y` being `up` or `down` could be explained by `x` being `up` or `down` respectively while (ii) `x` $\overset{--}{\rightarrow}$ `y` denotes that `y` being `up` or `down` could be explained by `x` being `down` or `up` respectively. If we assume that (i) the conjunction of an `up` and a `down` can explain a `steady` and that (ii) no change can be explained in terms of a `steady` (i.e. a `steady` vertex has no children), then we can partially evaluate Figure 5 into the and-or graph of literals shown in Figure 6. This graph contains one vertex for each possible state of the nodes of Figure 5 as well as *and* vertices which models combinations of influences (for example, `gDown` and `bDown` can lead to `fSteady`).

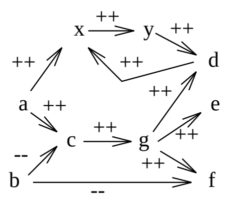

Figure 5: A qualitative theory.

For example, in the case where $\mathcal{OUT}$ = {`dUp,eUp,fDown`} and $\mathcal{IN}$={`aUp,bUp`}, then $\mathcal{P}_1$=`aUp` $\rightarrow$ `xUp` $\rightarrow$ `yUp` $\rightarrow$ `dUp`, $\mathcal{P}_2$= `aUp` $\rightarrow$ `cUp` $\rightarrow$ `gUp` $\rightarrow$ `dUp`, $\mathcal{P}_3$= `aUp` $\rightarrow$ `cUp` $\rightarrow$ `gUp` $\rightarrow$ `eUp`, $\mathcal{P}_4$= `bUp` $\rightarrow$ `cDown` $\rightarrow$ `gDown` $\rightarrow$ `fDown`, $\mathcal{P}_5$= `bUp` $\rightarrow$ `fDown`.

Some of these proofs make assumptions; i.e. use a literal that is not one of the known $\mathcal{FACTS}$ (typically, $\mathcal{FACTS} = \mathcal{IN} \cup \mathcal{OUT}$). Note that some of the assumptions will contradict other assumptions and will be *controversial* (denoted $\mathcal{A}_C$). For example, assuming `cDown` and `cUp` at the same time is contradictory. The key controversial assumptions

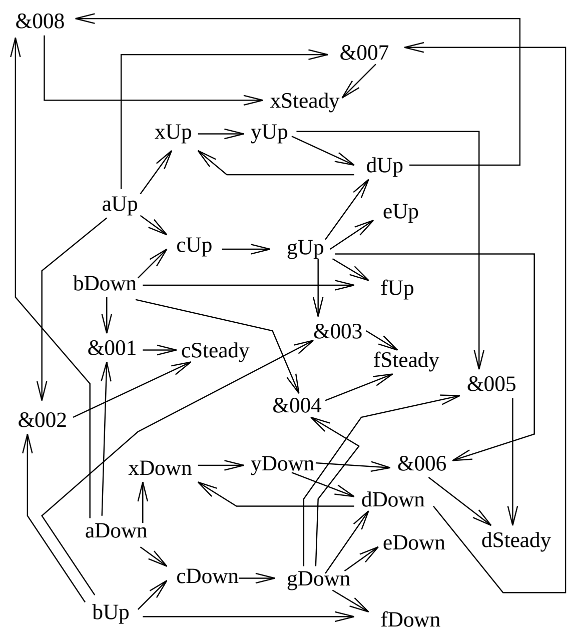

Figure 6: The search space tacit in Figure 5

are those controversial assumptions that are not dependent on other controversial assumptions. We denote these *base* controversial assumptions $\mathcal{A}_B$. In our example, $\mathcal{A}_C$={`cUp`,`cDown`,`gUp`,`gDown`} and $\mathcal{A}_B$ = {`cUp, cDown`} (since Figure 5 tells us that `g` is fully determined by `c`). If we assume `cUp`, then we can believe in the *world* $\mathcal{W}_1$ containing the proofs $\mathcal{P}_1$ $\mathcal{P}_2$ $\mathcal{P}_3$ $\mathcal{P}_5$ since those proofs do not assume `cUp`. If we assume `cDown`, then we can believe in the world $\mathcal{W}_2$ containing the proofs $\mathcal{P}_1$ $\mathcal{P}_4$ $\mathcal{P}_5$ since these proofs do not assume `cDown`. These worlds are shown in Figure 7.

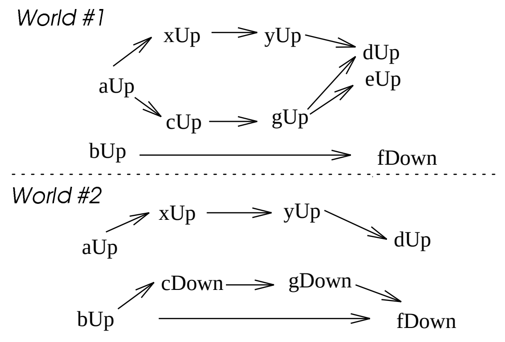

Figure 7: Two worlds from Figure 5

Originally, I agreed with the developers of QMOD that the system was doing something interesting and novel. I read some of the qualitative reasoning and complexity literature, but was unclear if it applied at all to QMOD.

When I began experimenting with scaling up the models used in QMOD, I found that the system could not handle models much bigger than those it was originally developed for. The problem was worlds generation: it did not terminate for models much bigger than those used to develop QMOD. The focus of my research then changed to trying to optimise QMOD.

7.2 Lesson 1

After much experimentation and reading in the literature, I found that I was applying a very dumb algorithm to a very hard task. The slowest way to build QMOD worlds was how I was doing it; i.e. via basic chronological depth-first (DFS) backtracking. I should have known better. Mackworth [28], in 1977, and DeKleer [7], in 1986, offered clear warnings about DFS. If a DFS search algorithm learns some feature of the domain, it can forget it on backtracking and be doomed to re-learn that feature later.

After reading Mackworth and DeKleer, I build a new version of QMOD called HT4 that caches what information it learns about the search space it traverses. This information is learnt via a set of *sweeps*. In the *forward sweep*, HT4 finds $\mathcal{A}_C$ as a side-effect of computing the transitive closure of $\mathcal{IN}$. In the *backwards sweep*, HT4 constrains proof generation to the transitive closure of $\mathcal{IN}$. As a proof is grown from a member of $\mathcal{OUT}$ back to $\mathcal{IN}$, five invariants are maintained. (i) Proofs maintain a *forbids* set; i.e. a set of literals that are incompatible with the literals used in the proof. For example, the literals used in $\mathcal{P}_1$ forbid the literals {`aDown, aSteady, xDown, xSteady, yDown, ySteady, dDown, dSteady` }. (ii) A proof must not contain loops or items that contradict other items in the proof (i.e. a proof's members must not intersect with its *forbids* set). (iii) If a proof crosses an *and* node, then all the parents of that node must be found in the proof. (iv) A literal in a proof must not contradict the known $\mathcal{FACTS}$. (v) The upper-most $\mathcal{A}_C$ found along the way is recorded as that proof's *guess*. The union of all the guesses of all the proofs is $\mathcal{A}_B$.

Once $\mathcal{A}_B$ is known then the proofs can be sorted into worlds in the *worlds sweep*. HT4 extracts all the objects $\mathcal{O}$ referenced in $\mathcal{A}_B$. A world-defining environment $\mathcal{ENV}_i$ is created for each combination of objects and their values. In our example, $\mathcal{ENV}_1$ = {`cUp`} and $\mathcal{ENV}_2$ = {`cDown`}. The worlds sweep is simply two nested loops over each $\mathcal{ENV}_i$ and each $\mathcal{P}_j$. A proof $\mathcal{P}_j$ belongs in world $\mathcal{W}_i$ if its *forbids* set does not intersect the assumptions $\mathcal{ENV}_i$ that define that world.

This algorithm ran two orders of magnitude faster than the original QMOD.

7.3 Lesson 2

Sadly, even with the speed up noted above, this new improved version of QMOD still did not terminate for models merely twice as big as those used in the original QMOD study. In one study [32], 94 new models were made by randomly adding in vertices and edges to the original QMOD models. Figure 8 shows the average runtime for executing HT4 over 94 and-or graphs and 1991 $<\mathcal{IN},\mathcal{OUT}>$ pairs [30]. For that study, a "give up" time of 840 seconds was built into HT4. HT4 did not terminate for $|\mathcal{V}| \geq 850$ in under that "give up" time (shown in Figure 8 as a vertical line). We conclude from Figure 8 that the "knee" in the exponential runtime curve kicks-in at around 800 literals (QMOD was originally developed for a model with 554 literals).

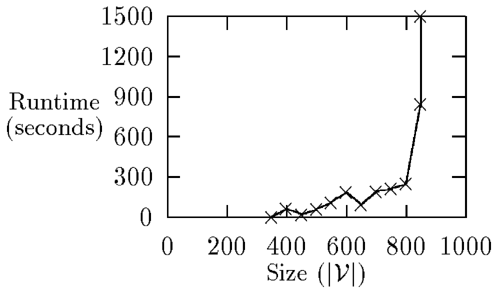

Figure 8: Average runtimes.

Once again, I went back to the literature to find that, formally, the process called HT4 is really *abduction*. One drawback with abduction is that it is slow. Selman & Levesque show that even when only one abductive explanation is required and the theory is restricted to be acyclic, then abduction is NP-hard [38] (i.e. runtimes are likely to be exponential on model size). Bylander *et. al.* make a similar pessimistic conclusion [6]. Computationally tractable abductive inference algorithms (e.g. [6, 10]) typically make restrictive assumptions about the nature of the theory or the available data. Such techniques are not applicable to arbitrary theories.

Therefore, I should not have been surprised that a mere two-orders of magnitude speed up in the runtime of HT4 did not permit the processing of very large models. Exponential runtimes defeat mere polynomial optimisations. Once the knee of the exponential runtime curve kicks-in, then a developer should not expect to be able to tame the runtimes.

7.4 Summary

Twice in the development of a VP system, I could have used existing AI research to (i) avoid certain problems and (ii) recognize what problems were unavoidable. Of course QMOD cannot be scaled up to large theories. Anyone who is familiar with the literature on search, complexity, and abduction would have been able to tell me that. Note that the relevant literature was not in obscure esoteric publications. Rather, it was published in mainstream AI journals and conferences.

8 Conclusion

In the above review, we have seen the use of the following software engineering techniques in VP systems:

- Forms-based interfaces in FORM and spreadsheets;
- Relational databases in QBE and SUPER;
- Numeric simulation in ıTHINK and THINKERTOY;
- Third-generation languages in PICT;
- Object-oriented programming in GARDEN;
- In order to encourage information accessibility, many VP systems adopt standard software engineering diagrammatic notations (e.g. the E-R diagrams of SUPER, the structure charts of PICT as well as petri nets.

We have seen the following knowledge engineering techniques used:

- Constraint-based reasoning in THINGLAB;
- Logic-programming in the PROLOG-based SUPER system;
- Functional programming in the LISP-based GARDEN system;

Also, we have made a case that VP systems can benefit from on-going knowledge engineering research in the fields of constraint-based reasoning, production systems optimisation and explanation. We have also cautioned against using a new representational system without first studying its computational complexity. We have offered a case study in which the develop of a VP system wasted some months because it ignored the published research relating to the basic computational process that it was trying to visualise. Hence, we advise that the designers of VP systems should be well-versed in a wide range of knowledge engineering and software engineering techniques.

References

[1] A.L. Ambler. Forms: Expanding the Visualness of Sheet Languages. In *Proceedings Workshop on Visual Languages, Tryck-Center, Linkoping, Sweden*, pages 105–117, 1987.

[2] A.H. Borning. Graphically Defining New Building Blocks in ThingLab. *Human-Computer Interaction*, 2(4):269–295, 1986.

[3] R.J. Brachman and H.J. Levesque. The Tractability of Subsumption in Frame-Based Description Languages. In *AAAI '84*, pages 34–37, 1984.

[4] M.B. Brown and R. Sedgewick. Techniques for Algorithm Animation. *IEEE Software*, pages 28–39, January 1985.

[5] T.B. Brown and T.D. Kimura. Completeness of a Visual Computation Model. *Software- Concepts and Tools*, pages 34–48, 1994.

[6] T. Bylander, D. Allemang, M.C. M.C. Tanner, and J.R. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.

[7] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.

[8] Y. Dennebouy, M. Andersson, A. Auddino, Y. Dupont, E. Fontana, M. Gentile, and S. Spaccapietra. Super: Visual Interfaces for Object and Relationship Data Models. *Journal of Visual Languages and Computing*, pages 73–99, 1995.

[9] R.D. Doorenbos. Matching 100,000 Learnt Rules. In *AAAI '93*, pages 290–296, 1993.

[10] K. Eshghi. A Tractable Class of Abductive Problems. In *IJCAI '93*, volume 1, pages 3–8, 1993.

[11] B. Feldman, P. Compton, and G. Smythe. Hypothesis Testing: an Appropriate Task for Knowledge-Based Systems. In *4th AAAI-Sponsored Knowledge Acquisition for Knowledge-based Systems Workshop Banff, Canada*, 1989.

[12] C.L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, pages 17–37, 19 1982.

[13] B.N. Freeman-Benson, J. Maloney, and A. Borning. An Incremental Constraint Solver. *Communications of the ACM*, 33:54–63, 1 1990.

[14] E.C. Frueder and R.J. Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992 1992.

[15] E.P. Glinert and S.T. Tanimoto. Pict: An Interactive Graphical Programming Environment. *IEEE Computer*, pages 7–25, November 1984.

[16] V. Goel. "Ill-Structured Diagrams" for Ill-Structured Problems. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 66–71, 1992.

[17] T.R.G. Green, M. Petre, and R.K.E. Bellamy. Comprehensibility of Visual and Textual Programs: The Test of Superlativism Against the "Match-Mismatch" Conjecture. In *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, 1991.

[18] S.H. Gutfreund. ManipIIcons in ThinkerToy. In E.P. Glinert, editor, *Visual Programming Environments: Applications and Issues*, pages 25–45. IEEE Computer Society Press Tutorial, 1990.

[19] D.D. Hils. Visual Languages and Computing Survey. *Journal of Visual Languages and Computing*, 3(1):69–101, 1992.

[20] M. Hirakawa and T. Ichikawa. Visual Language Studies - A Perspective. *Software- Concepts and Tools*, pages 61–67, 1994.

[21] High Performance Software Inc. iThink 3.0.5, 1994.

[22] A.C.H. Kindfield. Expert Diagrammatic Reasoning in Biology. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 41–46, 1992.

[23] K.R. Koedinger. Emergent Properties and Structural Constraints: Advantages of Diagrammatic Representations for Reasoning and Learning. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 154–159, 1992.

[24] V. Kumar. Algorithms for Constraint-Satisfaction Problems: A Survery. pages 32–44, Spring 1992.

[25] J.H. Larkin and H.A. Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, pages 65–99, 1987.

[26] D.B. Leake. Focusing Construction and Selection of Abductive Hypotheses. In *IJCAI '93*, pages 24–29, 1993.

[27] F. Ludolph, Y. Chow, D. Ingalls, S. Wallace, and K. Doyle. The Fabrik Programming Environment. In *IEEE Proceedings Workshop on Visual Languages*, pages 222–230, 1988.

[28] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[29] A.K Mackworth and E.C. Frueder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[30] T. J. Menzies and P. Compton. The (Extensive) Implications of Evaluation on the Development of Knowledge-Based Systems. In *Proceedings of the 9th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge Based Systems*, 1995.

[31] T.J. Menzies. Frameworks for Assessing Visual Languages. Technical Report TR95-35, Department of Software Development, Monash University, 1995.

[32] T.J. Menzies. *Principles for Generalised Testing of Knowledge Bases.* PhD thesis, University of New South Wales, 1995.

[33] T.G. Moher, D.C. Mak, B. Blumenthal, and L.M. Leventhal. Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In *Empirical Studies of Programmers: Fifth Workshop*, pages 137–161, 1993.

[34] C.L. Paris. The Use of Explicit User Models in a Generation System for Tailoring Answers to the User's Level of Expertise. In A. Kobsa and W. Wahlster, editors, *User Models in Dialog Systems*, pages 200–232. Springer-Verlag, 1989.

[35] R. Purvey, J. Farrell, and P. Klose. The Design of Star's Records Processing: Data processing for the noncomputer professional. *ACM Tans Office Inf. Syst.*, 1(1):3–34, 1983.

[36] S. P. Reiss. Working in the Garden Environment for Conceptual Programming. *IEEE Software*, pages 16–27, November 1987.

[37] S.P. Reiss, S. Meyers, and C. Duby. Using GELO to Visualize Software Systems. In *Proceedings of the Second Annual Symposium on User Interface Software and Technology*, pages 149–157, November 1989.

[38] B. Selman and H.J. Levesque. Abductive and Default Reasoning: a Computational Core. In *AAAI '90*, pages 343–348, 1990.

[39] G.M. Vose and G. Williams. LabVIEW: Laboratory Virtual Instrument Engineering Workbench. *Byte*, 11:84–92, 1986.

[40] M.R. Wick and W.B. Thompson. Reconstructive Expert System Explanation. *Artificial Intelligence*, 54:33–70, 1992.

[41] M.M. Zloof. QBE/OBE: A Language for Office and Business Automation. *Computer*, pages 13–22, May 1981.

Some of the Menzies papers can be found at *http://www.sd.monash.edu.au/ ~timm/pub/ docs/papersonly.html.*