

# Analytic Assessment of Visual Programming Languages

by Lindsey Spratt

Copyright 1996 by Lindsey Spratt. All Rights Reserved.

## Abstract

*In this paper we present new techniques for assessing a visual programming language in terms of the extent to which it exploits its visual representation and its concision compared to other programming languages, both visual and textual. These new techniques are demonstrated by application to SPARCL, a visual logic programming language based on sets with partitioning constraints.*

## Introduction

There are many programming languages and it is difficult to determine the effectiveness of any one of them, absolutely or with respect to other programming languages. This is a notoriously difficult problem for programming languages with textual representations. The problem is additionally difficult for those visual representations. Sethi provides a concise description of what a programming language should provide:

The language must help us write good programs, where a program is good if it is easy to read, easy to understand, and easy to modify.<sup>1</sup>

Two main approaches used in assessing programming languages are the analytic and experimental approaches. In the analytic approach, one analyzes descriptions of a programming language and programs written in that language. In the experimental approach one runs empirical studies involving users using (one or more implementations of) the programming language.

Each of these approaches has its strengths and weaknesses. The experimental approach has the advantage that it measures the desired attributes of a programming language in a fairly direct fashion. However, this approach is very sensitive to the particular implementation of the target programming language and the hardware and software environment in which the experiments are conducted.

These aspects of the experiments are difficult to factor out to develop a critique of the “underlying” programming language. Also, this approach is expensive in several ways: it is time consuming for the researchers organizing and administering the experiments and for the people who are the subjects of the experiments; also, it requires a highly “polished” implementation to minimize the negative environmental effects.

The analytic approach is much less subject to programming language implementation details than the experimental approach. However, the analytic approach produces only indirect measures. Thus, we must infer assessments, a process fraught with difficulties of its own. However, this approach is appealing for its broad and relatively easy application.

There has been little work experimentally assessing visual programming languages, but even less analytically. In this paper we investigate two analytic approaches; diagrammatic and size assessments. In an analytic approach one can assess a programming language in isolation or in comparison with other languages. The diagrammatic assessment is done in isolation: to what extent is the target programming language’s representation diagrammatic versus linguistic? This assessment helps to determine to what extent a visual programming language provides benefits due to its visual representation. The size assessment is done comparatively: does the target programming language produce substantially smaller solutions than do the reference languages? This comparison assumes that shorter solutions are easier to understand than longer ones.

## The target language and programming problems.

Our target language is one created by the authors, named SPARCL. We are interested in comparing it with other programming languages to characterize its strengths and weaknesses and to compare this characterization with our expectations. We expect that SPARCL is valuable in Sethi’s terms particularly when used as an “exploratory” programming language when working on conceptually complex programming problems, from a wide range of

1. p. 4 of [1]. Performance is not in this list, although one might expect it to be. Performance is primarily an attribute of an *implementation* of a programming language. It is only very indirectly an attribute of a language *definition*.

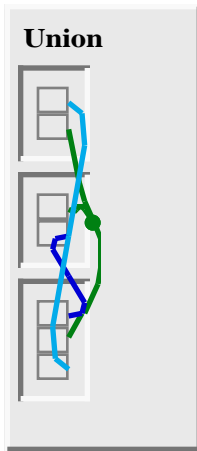


Figure 1: SPARCL Union/3 program.

problem domains.

SPARCL is a set-based language which has a declarative, prolog-like semantics where the outermost boxes are clauses, variables are small circles, empty sets are small black squares, “ur” constants are any text, and sets are solid boxes partitioned by dashed-box parts. Coreferencing terms are connected by (curving) lines. Examples of some of these representations are shown in Figure 1. Here there is one clause. It has the predicate name “Union” and three arguments. The first two arguments are partitioned sets, each with two parts (a part of a partitioning represents a subset of the partitioned set). The third argument is a partitioned set of three parts. There are three coreference links, one connecting the top part of the first set and the bottom part of the third set;. Another link connects the bottom part of the first set, the top part of the second set, and the middle part of the third set. The remaining link connects the bottom part of the second set and the top part of the third set. The first two arguments are sets which have the third argument as their union. The middle part of the third argument is the intersection of the sets of the first and second arguments. The bottom part of the third argument is the *difference* of the first minus the second argument sets, the top part of the third argument is the second argument set minus the first argument set. This interpretation relies on the partitioning constraint that parts of a partitioned set are pairwise disjoint.

We demonstrate the various approaches to measurement using two example programs as solved in SPARCL. These are shown in Figure 1 and Figure 2. We will be comparing these with solutions of the “union” and “classify examples” programming problems with solutions in PROLOG and LISP.

The union problem is to find the set which is the union of two other sets. The *classify examples* problem is to

classify a set of examples according to their values on a given attribute. A classification is a set of associations, where each value of the given attribute is associated with the set of all of the examples for which the given attribute has that value.

## Related Work

### Visual programming languages.

There is little work in assessing a particular approach to representing solutions in programming languages, particularly for comparing linear and nonlinear approaches. What work there is concentrates on user testing—having people use particular programming approaches and evaluating their experiences.

The development and application of various metrics for comparing visual and textual representations is reported in [2]. Work on assessing an aspect of a visual programming language (writing matrix manipulation programs) is reported in [3]. The relative merits of two input devices for editing graphic diagrams is reported in [4].

Sun-Joo Shin presents a unique approach to comparing diagrammatic and linguistic representations of logic in [5]. She is concerned with the logical status of diagrams; in particular, can one reason validly using only diagrams? Historically, mathematicians have believed that one cannot. Shin shows that two diagramming systems (based on Venn diagrams) are logically sound and complete, in the process showing how to approach proving soundness and completeness for diagramming systems in general. She also provides an analysis of some essential differences between diagrammatic and linguistic representations.

### Linear programming languages.

There is considerable work on assessing programs written in linear languages. The vast majority of this work is focused on procedural/imperative languages with destructive assignment, such as pascal, fortran, c, and cobol. A good summary of this work is in [6].

## Diagrammatic Versus Linguistic Representation

Shin’s analysis of the differences between diagrammatic and linguistic representations provides a starting point for an assessment of the extent to which SPARCL’s representation takes advantage of its visual nature. That is, to what extent its representation is diagrammatic versus linguistic. This is an important issue in evaluating a visual language. Since a visual approach is more complex to implement and more computationally intensive than a

textual approach, there must be some compensatory benefit to the visual approach to justify using it.

Shin provides three ways in which diagrammatic and linguistic representations differ: relations among objects, conjunctive information, and tautologies and contradictions. She is clear that a representation is not necessarily diagrammatic simply because it is visual, neither is a representation necessarily linguistic because it incorporates text.

A linguistic visual representation is essentially no more than an illustration for some textual representation, where the information in the visual representation is presented in essentially the same fashion as in the corresponding textual representation. A rebus puzzle is a common example of a visual representation which is essentially linguistic instead of diagrammatic. In such a puzzle some of the words of a phrase are replaced by pictures which are evocative of the replaced words meaning or pronunciation.

### Distinguishing between diagrammatic and linguistic representations.

One of Shin's key distinctions between diagrammatic and linguistic texts is diagrammatic texts rely more on the reader's "perceptual inferences" for understanding than do linguistic texts. Linguistic texts rely more instead on the conventions of the associated representation system being known to the reader. Generally a visual representation system is not wholly diagrammatic or linguistic, but is some of each.

Perceptual inference is the information one extracts more or less directly by the act of perception—no symbol system is involved. Perceiving that a picture is of a known person is a perceptual inference; interpreting a string of characters as a message is not a perceptual inference but rather relies on conventions.

One way in which a representation relies on perceptual inference (instead of convention) is by using spatial relations to model relations among objects. The connection

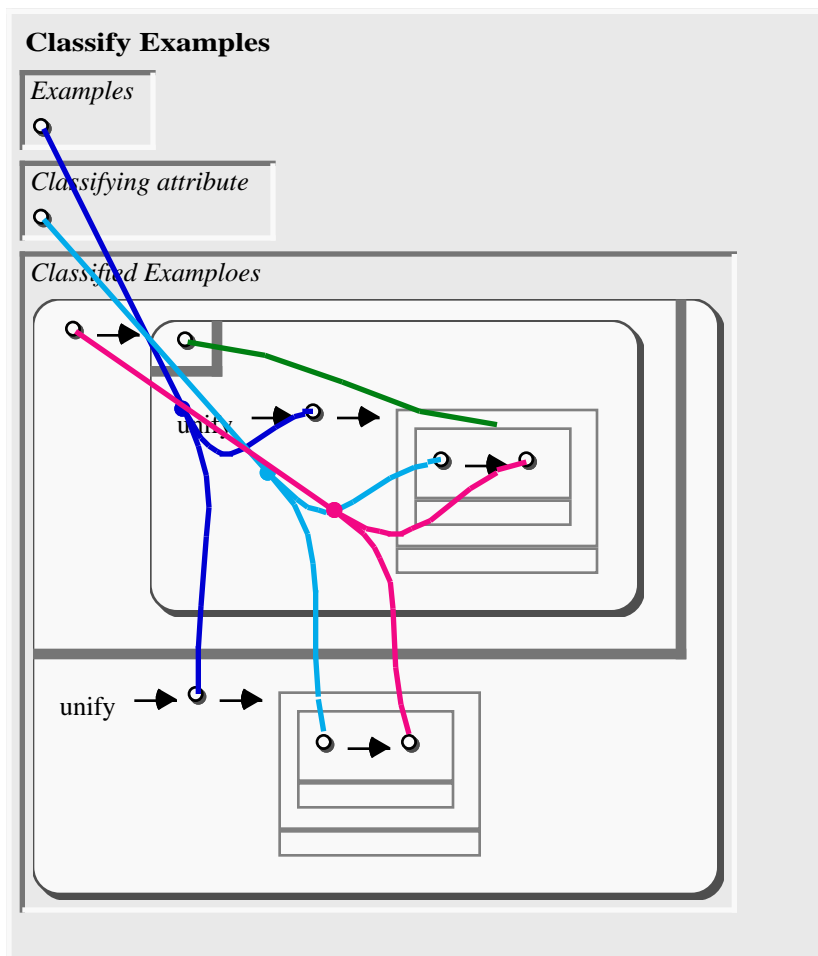


Figure 2: The *Classify Examples/3* program in SPARCL.

between a spatial relation and some other relation is a matter of convention, but the understanding of the spatial relation is a perceptual inference. Shin notes that if the "member of" relation is modeled by putting a dot in a box (the dot being a member of the set represented by the box), then:

...this isomorphism between the spatial arrangement and the "member of" relation is more perceptually obvious than any linguistic symbol that a linguistic representation adopts, since no extra convention involving syntactic devices is required.<sup>2</sup>

Conjunctive information is Shin's second way in which a representation may rely more on perceptual inference. In a diagrammatic representation, the conjunction of facts is represented by simply representing the facts to be conjoined in the same text, no additional representational device is required. In a linguistic representation, there must be some kind of device to represent conjunction (such as '&', 'and', or '^'). Thus, conjunction is a perceptual inference in a diagrammatic representation but is a matter of convention in a linguistic representation.

The third distinction between diagrammatic and linguistic representations is in their handling of tautologies and contradictions. As Shin says: "Tautological information is vacuously true and contradictory information is always false."<sup>3</sup> She claims that diagrammatic systems represent tautologies and contradictions in a more perceptually obvious way than linguistic systems do. A diagram which contains no representing fact is tautologous. However, in a linguistic system a tautology must be represented by a nonempty string, and thus is less obviously "vacuously" true. A contradiction is revealed in a diagram by a violation of some basic rule for valid diagrams, such as no two elements may be in the same location in a diagram, or the same element may not be represented in two different places in the same diagram. Such a violation is easy to perceive. In a corresponding linguistic representation it can be fairly subtle to identify a contradiction, requiring reasoning about properties of the represented relations (such as transitivity and symmetry). The implications of these properties are "perceptually inferred" in the diagrammatic representation.

## SPARCL as a Diagrammatic System

To assess SPARCL's diagrammatic nature, we apply each of Shin's three distinctions. To simplify this discussion we look only at the "basic" form of SPARCL, without the multiple set representations.


### Diagrammatic assessment: relations of objects

Since SPARCL is a thoroughly set-based language, the primary object relationship in SPARCL is that of set membership. We adopt the convention that a set is assigned to a certain kind of basic region, a closed curve in the shape of a rectangle, or a variable represented by a small circle. At this point, the representation is purely conventional and thus linguistic. However, the choice of primitive objects in SPARCL is more constrained than in a linguistic system in that the non-variable representation of sets is limited to closed curves, so that the representation creates an interior and an exterior. However, in a linguistic system any distinguishable symbol may be used.

The membership relationship is represented in SPARCL by the convention that any "term" spatially *in* the representation of a set is a member of that set. This convention connects membership to a spatial relation. Although somewhat arbitrary (other spatial relations can be used for membership, as Shin notes<sup>4</sup>), this is less arbitrary than a purely linguistic device of introducing a symbol/operator such as ' $\in$ '. The use of the spatial relation "appeals to our natural perceptual ability"<sup>5</sup>. Several other membership relationships in SPARCL are represented using the "spatial inside" relationship, including: clauses in programs, literals in clause bodies, predicate name in clause and literal, and arguments in clause and literal.

Another essential relationship in SPARCL is that of the



partitioning of a set, such as , a partitioned set with two parts. A partitioning conveys two kinds of information: that the union of the parts equals (covers) the entire set being partitioned, and that the parts are pairwise disjoint. The representation of sets which are parts of a partitioning of a set is diagrammatic in SPARCL. The relation "part of a partitioning" is represented by a dotted box (the part) being inside a set (a solid box). All of the parts of a partitioning are next to each other so that they completely cover ("tile") the area of the representation of the set: No elements of the SPARCL language can be placed outside of all of the parts of a partitioning but inside of the set being partitioned. This represents in a diagrammatic way that the union of the parts of a partitioning cover the partitioned set. It is a matter of convention that the dotted boxes are sets which are parts of a partitioning. Thus, the representation of the pairwise disjointness relationship is conventional.

2. page 162 of [5].

3. p. 167 in [5].

4. On page 171 of [5], Shin mentions a system by Lambert [7] where a set is represented by a line (segment) and all of the points on the line are in the corresponding set..

5. p. 171 in [5].

Aside from the subset relation as embodied by the parts of a partitioning, SPARCL does not provide direct representation of relationships between sets such as union, intersection, and difference. One can use the partitioned set representation to construct these relationships, but it is not as immediately understandable as a direct representation would be, such as one finds in Shin’s Venn-I and Venn-II. For these relationships in SPARCL one must fall back on coreference links and other more linguistic representations: defining a SPARCL predicate which implements the relationship and using a literal to refer to this predicate. However, the use of partitioned set and coreference links does provide a somewhat diagrammatic representation of these “other” set relationships (union, difference, and intersection).

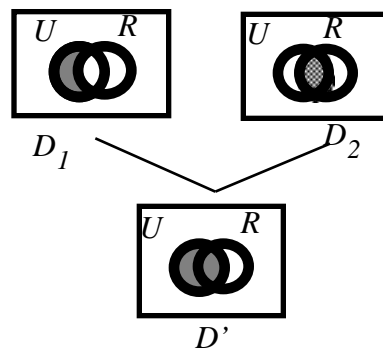
The identification of a term as a predicate name is made by the convention of the term being in the upper left corner of the clause or literal box. Variables are represented by convention as small circles. Ur constants are represented by themselves (i.e. text strings). The empty set constant is represented by convention by a small solid black square. This empty set representation is partially diagrammatic in that it is a rectangle, such as any (potentially) nonempty set, but it is filled in so that no term can be placed inside of it—spatially representing that it has/can have nothing in it.

Syntactically distinct terms can “refer” to semantically the same term. This is term coreference. This is represented in SPARCL by coreference links: lines drawn connecting the coreferencing terms. This is a diagrammatic representation that relies on the perceptual inference of seeing the connections. This connecting of arbitrary terms is relatively clumsy in linguistic representations. One might use a convention such as that done with feature structures<sup>6</sup> where each part of the structure may be labeled (a linguistic constant followed by a colon (‘:’) followed by the substructure being labeled). Those features with the same label are coreferencing features.

### Diagrammatic assessment: conjunction

This aspect of diagrammatic representation is not much used by SPARCL. Additional “facts” are added to a representation by adding additional clauses, much as is done in a linguistic representation. No additional syntax is needed, however; the additional clauses are simply visible and thus known to be in conjunction with with the existing facts. This is more diagrammatic than the linguistic approach, but it is not a particularly strong use.

Shin gives an example of drawing such inferences using her VENN-I system. As shown in Figure 3, the facts “Every unicorn is red” and “No unicorn is red” are con-



**Figure 3:** The “red unicorns” problem in the VENN-I language. The circles are sets.  $U$  = set of unicorns, and  $R$  = set of red items. Shaded areas are parts of sets with no members.  $D'$  is the unification of  $D_1$  and  $D_2$

junctively combined to get a diagram from which it is obvious that there are no unicorns.

### Diagrammatic assessment: tautologies and contradictions

Tautologies in general receive no special handling in SPARCL, and in this regard SPARCL is not specifically diagrammatic. An exception to this regards partitioning. It is vacuously true that a set can be partitioned, since all sets (even empty ones) can be partitioned. This is diagrammatically represented by “vacuous” parts of a partitioning having no contents, but having room in which to place terms (which is different from the empty set which, being a solid square, has no place to put anything).

Some contradictions receive a diagrammatic representation in SPARCL. Some of them are invalid SPARCL representations. Contradictions with partitioned sets would be either the parts not covering exactly the partitioned set (too large or too small), or a term appearing in more than one part of a partitioning. In the first case, the SPARCL programming environment does not allow such a construction, but it would be easily perceived should it occur. In the latter case, one can easily see that the same term is present in different parts, since the parts are spatially close together, as long as the total number of terms and parts is not large. In a linguistic representation there is no “spatial locality”, and thus the invalid sharing of a term is less readily apparent.

Another contradiction is when two terms which are not mutually unifiable (such as two different constants) are said to corefer. Since coreference is perceptually obvious in SPARCL, due to the use of connecting lines, then this contradiction is fairly easy to see. This is much harder to realize when reading a linguistic representation of coref-

6. pp. 105-108 in [8].

erencing.

## Diagrammatic assessment: summary.

SPARCL is appropriately diagrammatic in several important aspects. Thus, the visual representation of SPARCL is beneficial compared to a strictly textual and therefore linguistic representation. Nonetheless, there are areas where more diagrammatic representation is possible; for instance, in the relationships between sets.

## Measuring Program Size

Two of the sets of measurements with the best experimental verification for linear programming languages are those based on “lines of code” and the basic Halstead Software Science<sup>7</sup> (SS) size measurements ( vocabulary, size, and volume). Since lines of code are not measurable for visual programming languages, we adopted an approach based on SS. The key concepts for Software Science (SS) measurements are “token”, “operator”, and “operand”. The tokens are “the basic syntactic units distinguishable by a compiler.”<sup>8</sup> Halstead’s original differentiation between operators and operands was:

“...based on the fact that all programs can be reduced into a sequence of machine language instructions, each of which contains an operator and a number of operand addresses.”<sup>9</sup>

We have used two different methods to develop token counts. One is a “volume”-based approach and the other is to extend the graphic token count of Nickerson to produce operator and operand counts.

In the following discussion we use the standard Software Science notation:

- $\eta_1$  is the number of unique operators (operator types),
- $N_1$  is the total number of operators,
- $\eta_2$  is the number of unique operands (operand types),
- $N_2$  is the total number of operands,
- $\eta$  is the vocabulary (the sum of  $\eta_1$  and  $\eta_2$ ),
- $N$  is the Software Science “size” (the sum of  $N_1$  and  $N_2$ ),
- $V$  is the volume:

$$V = N \cdot \log_2(\eta)$$

## Graphic Token-based Metrics

Jeffrey Nickerson in [2] presents four graphical metrics: graphic token count, diagram class complexity, confusion count, and (graphic) token density. We propose the following definitions based on the graphic token count and diagram class complexity:

$$\eta_1 = \left( \begin{array}{l} \text{number\_of\_edge\_types} + \\ \text{number\_of\_label\_types} + \\ \text{enclosure\_present} + \\ \text{adjoinment\_present} \end{array} \right)$$

$$\eta_2 = \left( \begin{array}{l} \text{number\_of\_node\_types} + \\ \text{number\_of\_textual\_token\_types} \end{array} \right)$$

The “total” counts,  $N_1$  and  $N_2$ , are simple analogs of  $\eta_1$  and  $\eta_2$ , respectively. The terms of these equations are based on three basic attributes of graphic representation: *adjoinment* for *number\_of\_adjoinments*, *linkage* for *number\_of\_edges* and *number\_of\_nodes*, and *containment* for *number\_of\_enclosures*. This approach to graphic token counting only measures the “relationships of objects” aspects of diagrammatic representation identified by Shin. No explicit counting is done for Shin’s other two aspects of diagrams (conjunctive information and tautologies and contradictions).

## Graphic token counts for the SPARCL examples.

The Union/3 program in Figure 1 and the Classify Examples/3 program in Figure 2 have the following basic graphic token counts:

	Union/3	Classify Examples/3
nodes	7	12
node types	1	4
edges	3	7
edge types	1	2
textual tokens	1	4
enclosures	7	13
enclosure present	1	1
adjoinments	7	7
adjoinment present	1	1

To give an idea how these counts were derived, we note that for Classify Examples/3: the node types are variable (8 instances), ‘unify’, intensional set, and ‘+’; and, the edge types are the coreference links and the arrows which are the N-tuple element connectors.

The graphic token count versions of the basic Halstead measures are:

	Union/3	Classify Examples/3
unique operands	2	6
total operands	8	14
unique operators	3	4
total operators	17	27

These measures are combined in the standard way for the simplest Halstead derived metrics:

	Union/3	Classify Examples/3
vocabulary	4	10
size	25	41
volume	50	136

7. See pages 80-87 in [6].

8. p. 37 in [6].

9. p. 37 in [6].

## Volume-based Metrics

The volume-based approach has the strength that it provides a common basis for the development of counting techniques across programming languages regardless of type of concrete representation or underlying programming paradigm. For this approach one develops a canonical (or abstract) representation for the language being studied, then counts tokens in the canonical form of the program of interest. The weakness this has for language-comparison is that it does not yield direct information on the concrete representation of the language. This contrasts with the graphic token count presented above which is explicitly a measure of the visual representation of a program.

Halstead's idea of "volume" for a program was a minimal bit length for a simple encoding of the source of that program. Following this notion for each of the languages of interest (SPARCL, PROLOG, and LISP) leads to a particular approach to counting operators and operands (as well as specialized volume equations). The first step in following this approach is to consider how the source for the target language can be encoded in some "minimal" fashion. This defines the "encoding tokens". These tokens are then categorized into operands and operators, as before, and the various Software Science measurements are applied.

### Volume-based counts for SPARCL.

The SPARCL programs are converted to a basic notation which has a fairly simple mapping onto the display representation, then this basic notation is "counted".

SPARCL clauses are converted to "basic notation" as follows: All N-tuples are converted to their nested, ordered-pair form. Clauses and literals are treated as N-tuples. A *record* in the basic notation has two top fields (**ReferencedFlag ReferencedObject**). The **Referenced-Flag** is 1 or 0; if 1, then the **ReferencedObject** is (**ReferenceID Object**). Otherwise the **ReferencedObject** is simply **Object**. An **Object** has two fields, (**Type Object-Contents**). The **Type** interpretations are:

Type	ObjectContents
UR	UrEncoding {Ur constant}
VAR	empty {Variable}
ES	empty {Empty Set}
OP	( <b>Record1 Record2</b> ) {Ordered Pair}
SET	( <b>Count PartRecords</b> ) {Set}
IS	( <b>Record1 Record2</b> ) {Intensional Set}

The **PartRecord** is (**ReferencedFlag Referenced-Part**). If the **ReferencedFlag** is 0, then the **Referenced-Part** is **Part**. If the **ReferencedFlag** is 1, then the **ReferencedPart** is (**RefID Part**). The **Part** is (**Count**

**Records**).

The operand and operator counts are defined as:

$$\begin{aligned}\eta_1 &= \eta_{set} + \eta_{part} + \eta_{op} \\ N_1 &= N_{set} + N_{part} + N_{op} \\ \eta_2 &= \eta_{ref} + \eta_{ur} + \eta_{var} + \eta_{es} \\ N_2 &= N_{ref} + N_{ur} + N_{var} + N_{es}\end{aligned}$$

( $\eta_{set}$ ,  $\eta_{part}$ ,  $\eta_{op}$ ,  $\eta_{var}$ , and  $\eta_{es}$  are each either 0 or 1. For each of these counts either such an element is present or not.)

The union SPARCL program in (simplified) basic notation:

OP(OP(OP(OP(UR union, SET(1, 2)), SET(2, 3)), SET(3, 2, 1)))

The volume calculation for this representation is:

$$V'' = \left( \begin{aligned} & (\log_2(\eta_{ref}) + 1)N_{ref} \\ & + (3 + \log_2(\eta_{ur}) + 1)N_{ur} \\ & + 3N_{var} + 3N_{es} \\ & + (3 + \log_2(MaxParts) + 1)N_{set} \\ & + (\log_2(\max(MaxPartCard, 1)) + 1)N_{part} \\ & + 3N_{op} + 3N_{is} \end{aligned} \right)$$

The various counts for the SPARCL Union/3 and Classify Examples/3 program are:

	Union/3	Classify Examples/3
$\eta_{ref}$	3	4
$N_{ref}$	7	11
$\eta_{ur}$	1	2
$N_{ur}$	1	3
$N_{var}$	0	10
$N_{es}$	0	0
$N_{set}, MaxParts$	3, 3	4, 2
$N_{part}, MaxPartCard$	7, 0	8, 1
$N_{op}$	3	10
$N_{is}$	0	0

The derived measurements for the example programs are:

	Union/3	Classify Examples/3
$\eta_1$	3	7
$N_1$	13	24
$\eta_2$	4	4
$N_2$	8	24
$\eta$	7	11
$N$	21	48
$V$	59	166
$V''$	55	142

The SS volume measurement,  $V$ , is a little larger (at 58.95) than the specialized volume measurement  $V''$  (at 54.8496).

## Comparison of the Graphic Token-based and Volume-based Counting Methods

We developed basic notations and specialized volume equations based on these notations for PROLOG and

LISP as was done for SPARCL. These counting methods were applied to PROLOG and LISP solutions of the “union” and “classify examples” problems.

The PROLOG program for the *union* program problem is:

```
union([H|X], Y, Z) :-
    member(H, Y),
    union(X, Y, Z).

union([H|X], Y, [H|Z]) :-
    \+ member(H, Y),
    union(X, Y, Z).

union([], X, X).
```

The LISP program for the *union* programming problem is:

```
(defun union (L1 L2)
  (if (null L1)
      L2
      (let ((x (car L1)))
        (if (member x L2)
            (union (cdr L1) L2)
            (cons x (union (cdr L1) L2)))))))
```

The *union* measurements for all of the languages, plus the graphic-based measurements (SPARCL/g) are:

	SPARCL	SPARCL/g	PROLOG	LISP
V (special)	55	50	166	175
V (simple)	59	-	128	168
N	21	25	37	41
$\eta$	7	4	11	15

The SPARCL program is smaller than the other two language programs in all measurements (size, vocabulary, and volume).

The *classify examples* measurements for all of the languages, plus the graphic-based measurements (SPARCL/g) are:

	SPARCL	SPARCL/g	PROLOG	LISP
V (special)	142	136	632	1657
V (simple)	166	-	503	1644
N	48	41	107	303
$\eta$	11	10	26	43

As in the *union* problem, the SPARCL *Classify Examples/3* program is smaller in all measurements than the programs in the other two languages.

The two examples explored in some detail above have the interesting property that the Halstead metrics are very similar values for the two different methods of counting the SPARCL programs. Since these counting methods had very different origins, this gives us hope that the techniques are not too sensitive to vagaries of the counting process to be useful.

We are extending the comparative analysis of these counting methods to problems with larger solutions.

## Summary

This paper presented new methods for analyzing a visual programming language which assesses both the benefit derived from the visual representation and the conciseness of the language compared with other programming languages, both visual and textual. The “target” language of this paper, SPARCL, was shown to exploit its visual representation and to be substantially more concise than the reference languages for two small programming problems.

## References

- [1] *Programming Languages: Concepts and Constructs* by Ravi Sethi. 478 pages. Reading, Massachusetts: Addison-Wesley Publishing Co. 1989.
- [2] “Visual Programming: Limits of Graphic Representation” by Jeffrey V. Nickerson. Pages 178-179 in Proceedings 1994 IEEE Symposium on Visual Languages, October 4-7, 1994. St. Louis, Missouri. IEEE Computer Society Press: Los Alamitos, CA.
- [3] “Is it easier to write matrix manipulation programs visually or textually?” by Rajeev K. Pandey and Margaret M. Burnett. Pages 344-351 in Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- [4] “A comparison study of the pen and the mouse in editing graphic diagrams.” by Ajay Apte and Takayuki Dan Kimura. Pages 352-357 in Proceedings 1993 IEEE Symposium on Visual Languages, August 24-27, 1993. Bergen, Norway. IEEE Computer Society Press: Los Alamitos, CA.
- [5] *The Logical Status of Diagrams* by Sun-Joo Shin. 197 pages. Cambridge, England: Cambridge University Press. 1994.
- [6] *Software Engineering Metrics and Models* by S. D. Conte, H. E. Dunsmore, and V. Y. Shen. The Benjamin/Cumming Publishing Company, Inc.: Menlo Park, California. 1986.
- [7] *Neues Organon I* by Johann Heinrich Lambert. Berlin: Akademie Verlag, 1990.
- [8] “Unification: A Multidisciplinary Survey” by Kevin Knight. Pages 93-124 in *ACM Computing Surveys*, Vol. 21, No. 1, March 1989.