COMPUTING SCIENCE

An Algebra of Lines and Boxes

C.M. Holt

TECHNICAL REPORT SERIES

No. 483 June, 1994
An Algebra of Lines and Boxes

C.M. Holt

Abstract

Visual language design is driven by interaction between syntax and semantics: changing the way a concept is represented affects the ease with which it can be understood and related to other concepts. The use of lines, boxes, and icons to augment text affects intuitions about operations and properties. This paper is concerned with characterising syntax by developing a "natural" algebra that tries to reflect it. The nature of boxes gives rise to two kinds of operations, enclosure and adjointness, that can be related to form a ring, which can be embedded in a vector space. Lines connecting boxes can represent an operation that is related to adjointness, but which is more general in arity and the orientation of the edges that can be linked. The decoration of lines and boxes with emblems provides a facility for representing higher-order operations.
Bibliographical details

HOLT, Christopher Martin
An Algebra of Lines and Boxes
(By) C.M. Holt
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 483)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Visual language design is driven by interaction between syntax and semantics: changing the way a concept is represented affects the ease with which it can be understood and related to other concepts. The use of lines, boxes, and icons to augment text affects intuitions about operations and properties. This paper is concerned with characterising syntax by developing a "natural" algebra that tries to reflect it. The nature of boxes gives rise to two kinds of operations, enclosure and adjointness, that can be related to form a ring, which can be embedded in a vector space. Lines connecting boxes can represent an operation that is related to adjointness, but which is more general in arity and the orientation of the edges that can be linked. The decoration of lines and boxes with emblems provides a facility for representing higher-order operations.

About the author

Dr. Holt received the B.A. and M.S. degrees in applied mathematics from Harvard University in 1977, and the D.Phil degree in computer science from Oxford University in 1986. He has worked on the design of parallel algorithms at the Queen's University of Belfast, and was appointed lecturer there in 1984. He is currently a lecturer in Computer Science at the University of Newcastle upon Tyne, where his research interests include the design of constructive logics and their applications to concurrency, specifications and meta-languages.

Suggested keywords

DATAFLOW
PROGRAM SPECIFICATION
VISUAL LANGUAGES

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6424 511.3
U.D.C. 519.767 510.64
An Algebra of Lines and Boxes

C. M. Holt
Dept. of Computing Science, U. of Newcastle upon Tyne NE1 7RU, UK
[chrisholt@newcastle.ac.uk]

Abstract

Visual language design is driven by interaction between syntax and semantics: changing the way a concept is represented affects the ease with which it can be understood and related to other concepts. The use of lines, boxes, and icons to augment text affects intuitions about operations and properties. This paper is concerned with characterising syntax by developing a "natural" algebra that tries to reflect it. The nature of boxes gives rise to two kinds of operations, enclosure and adjoinness, that can be related to form a ring, which can be embedded in a vector space. Lines connecting boxes can represent an operation that is related to adjoinness, but which is more general in arity and the orientation of the edges that can be linked. The decoration of lines and boxes with emblems provides a facility for representing higher-order operations.

1 Introduction

The history of programming language design reflects a continuing tug of war between expressibility ("Can I say what I mean?") and sparseness ("Do I need all those bells and whistles?"), in syntax as much as semantics. The arid elegance of pure Lisp, containing only sequences of symbols nested by parentheses, contrasts with the flexibility offered by extensible languages. The introduction of non-textual elements such as lines, boxes, and icons into the syntax of programming offers such a wealth of possibilities that language designers find it difficult to find, much less agree upon, conventions analogous to those employed in text. There is no obvious equivalent to the Chomsky hierarchy of regular, context-free, and context-sensitive languages, and even if there were the dissatisfaction with syntactic limitations that led to positional languages such as occam could well dissuade many people from use.

The difficulty seems to be in the formal elicitation of good design principles that enable us to characterize one kind of layout as clear and graceful, and another as confused and awkward. We all know art when we see it, but we see different kinds of art. People naturally express themselves in a hodge-podge of styles, and these change with age, experience, and sometimes just a rebellious desire to alleviate boredom. This does not mean that we should just throw up our hands and say "anything goes"; the argument here is that although there is a clear danger of shackling linguistic creativity, there are some straightforward guidelines that can be used to good effect. Furthermore, these can be embedded in the syntax and not just the semantics of a visual language.

There has been considerable research into graph grammars, visual description languages, and visual syntax definition languages; but all too often these merely apply textual approaches to visual objects (e.g. [3,8,9,10]). Attempts to use the syntax of visual languages to express concepts [1,7,11] have generally been focussed on small problem domains, rather than offering the expressiveness needed for a general purpose programming and specification language. This paper proposes that a syntactic basis for a lines-and-boxes language be described in terms of the mathematics of syntactic operations. The intuitions reflect experience gained through the evolution of the visual language viz [4,5,6].

Textual languages have little in the way of syntactic structure. Sequences of symbols simply form a monoid. The addition of parentheses as markers yields sequences of sequences, which provide nested hierarchies, and the underlying conceptual structure of the parse tree. However, that is about as far as textual syntax can go. Boxes are seen here as the visual equivalent of parentheses, offering a hierarchical structuring mechanism; as with parentheses, brackets, and braces, the boundaries they offer should rarely be broken. [Parenthesis structures are generally over-ridden only when used in quotes or comments.] Venn-diagram-like box intersections are considered harmful: they do not extend well to 3D, and make boundaries difficult to determine when applied to more than pairs of elements. The author believes that the visual confusion generated by box intersections has been an inhibiting factor in the spread of the StateMate tool, based on Statecharts [2].
Just as the intersection of two sets is a set in its own right, though it may have been derived from others, so are shared or overlapped elements seen as individual objects, perhaps linked to multiple "owners". This constraint, though perhaps severe, still leaves a considerable amount of freedom to the language designer.

2 Boxes

A box is a finite, closed structure contained in a manifold of one, two, or three dimensions. It is characterized by its boundary, its orientation, and its contents. A boundary is a structure of walls that define a region. If the box is on a line, its walls are points or line segments; if the box is on a plane, its walls are edges or regions; and if the box is in space, its walls are surfaces or volumes. The manifold containing a box need not be Euclidean. A boundary is topologically equivalent to a circle in 2-space or a sphere in 3-space; its shape is defined by its component walls. A wall has an orientation with respect to the box it helps to delimit, and it has curvature: positive, flat and negative curvatures are distinguished, as are the directions in which the curvatures apply. A space has a given granularity of curvature and direction, that determine the extent to which different walls are distinguished. A wall may be decorated with features; these may be boxes or emblems (structures of icons and/or text, corresponding to symbols). A wall may contain (or be) a box of lower dimension; e.g. a surface may contain a 2D box. The box then decorates the surface. The length, width, and thickness of a wall do not in and of themselves have significance (e.g. all rectangular prisms are considered to be shape-equivalent). This is weaker than similarity in the sense of triangles that maintain constant ratios; angles matter only insofar as they indicate orientation. The granularity of the space determines the extent to which different orientations can be distinguished, as for curvature.

Boxes represent an enclosure operation; they can surround lines, emblems, other boxes, and structures consisting of these. Enclosure is associative and commutative with respect to contents that are unconnected in both 2D and 3D boxes; relative position does not matter except among linked components. Since a surface can be a 2D box on the boundary of a 3D box, unconnected decorations on a surface have no relative position; however, decorations on an edge are ordered, as are the contents of 1D boxes. An empty box represents the identity element for enclosure. There is a monadic operation that maps values to their inverses under enclosure; application of this operation is indicated by decorating the boundary of a box with its name (assuming the name is not being "quoted", i.e. used to denote its syntactic structure). Given all this, box enclosure over a domain of values generates a group. Figure 1 illustrates the group generated by enclosure for rectangular boxes, with "~" the emblem denoting the inverse operation.

![Diagram of box operations](image)

Boxes can also be used in representing another kind of operation, that of conjunction, in which the boundaries of two boxes touch one another. If one box is entirely within another and they share part of a boundary, the conjunction is internal; if neither box impinges on the interior of the other the conjunction is external; and if one box is partly inside and partly outside the other, it is a boundary conjunction. [A surface or edge is understood to be recognizable, such that it is possible to determine whether it is being decorated.] A boundary does not break the integrity of its resident decorations by passing through them; rather, the decorations form a part of the boundary, such that it may have thickness. Adjunction is characterized by the patterns in the walls that are shared; different touching patterns (e.g. a surface that is touched by a vertex in one but not in another) represent distinct conjunction operations. A variety of conjunctions for rectangles are shown in Figure 2.

![Diagram of box conjunctions](image)

If one box is within another, their boundaries may be similar in places. A point on a boundary is similar to one on an enclosing boundary if (i) they are on similar boundary components (surfaces, edges, or vertices), (ii) their positions relative to decorations on those components are equivalent, (iii) the given internal component has no external connections, and (iv) the given external component has no internal connections. Components are similar if they have similar curvatures,
orientations, and decorations. Curvatures and orientations are similar if they are indistinguishable within the granularity of the space. Decorations are similar if they are of the same number, kind, and overlay pattern (more than one decoration may be associated with a single point); on an edge, they must have the same relative order.

Similarity plays a powerful role in a box language that has a similarity-connection axiom: this asserts that connections to similar points are connected, and if one connected point is similar to two or more connected points, the structure on which it resides is duplicated for each of those connections. Since two points can only be similar if the inner one has no external connections and the outer one has no internal connections, this means that an external connection to an external boundary is linked to the internal connection of an internal boundary. It causes adjunction to distribute over enclosure, and requires adjunctions along the same dimension to be associative (though adjunctions over different dimensions are not). It also means that the empty box is a zero element for adjunction. Given all this, enclosure with a single kind of adjunction forms a ring. The ring axioms applied to horizontal adjunction for rectangles are displayed in Figure 3.

![Figure 3](image)

Visual languages, being non-linear, can represent relationships among syntactic elements in more than one dimension. Thus, different adjunctions form distinct rings; but these adjunctions may interact. Similarity can apply to more than one component at once, and it can apply in one direction to a structure adjoined in another direction. Unless care is taken, it might happen that the order in which nested enclosures are evaluated could affect the result, which would be unfortunate. The first requirement is for simultaneity; when two or more components of an boundary are similar to those of an enclosing box, the connections apply all at once rather than one after another, and no connections are introduced among external boxes. The second requirement is for non-interference: if a component of one box is partially adjoined to another box (i.e. part of the component is not shared, so e.g. a wall adjoins a component of lower dimensionality), similarity applies to the lower-dimensional component only if it has no other external connections. Similarity among curved surfaces and edges depends on the discretisation of angles in the given space. Some examples of the effects of non-interference and simultaneity are provided in Figure 4.

![Figure 4](image)

3 Links

A link is a connected structure characterized by its topology, its ends, its decorations, and its strength. A 1D link is composed of lines; its topology is determined by the loops and knots it contains. A 2D link is composed of ribbons, so it may have twists as well as loops and knots (e.g. a Moebius strip). A 3D link is composed of pipes, and again may have twists, loops, and knots. A link may have any number of ends; these may be free (unattached) or bound (connected to a box, an emblem, or a link of different dimensionality). An end has an orientation; if it is free, this is the "slope" of the link at the endpoint, while if it is bound it takes the orientation of the element to which it is attached (orientation between connected links is irrelevant). An undecorated line connecting a vertex to a box is equivalent to that vertex being adjoined to the box at the link points; however, links are more flexible than adjunction in that ends may be oriented in ways that could not occur with the adjunction, and links may connect an arbitrary number of elements to one another. A link between walls is weaker than an adjunction if the adjunction causes more points to be shared (which can depend on the granularity of the space). The strength of a link is a numerical quantity; two links with the same endpoints, topology, and decorations are equivalent to a single link with the sum of their strengths. The default strength of a link is 1. Figure 5 shows some examples of 1D links with both free and bound endpoints.

![Figure 5](image)

There is an analogue to the similarity-connection axiom
for links. If a free end within a box is oriented towards an orthogonal surface having no internal connections, then the end is similar to points on the surface at which decorations match those of the end; it links once with each object connected to similar points. If an end is oriented towards an orthogonal edge with no internal connections (either directly in a 2box or via surfaces in a 3box that lead towards the edge), is less than 3D, and (in the case of ribbons) has the right twist, then the end is similar to points of the edge at which the decorations match those of the end. The same principle applies to vertices; if an endpoint is oriented towards a vertex with no internal connections (via edges not themselves orthogonal to the endpoint), and the decorations match, then the endpoint and vertex are similar. In a non-convex box, it is possible for an endpoint to be similar to more than one vertex. A connection to a box containing nothing that can link to it fails; the box acts as a zero element. Examples of decorated endpoint links are provided in Figure 6.

\[
\begin{align*}
\text{X} & \rightarrow \text{Y} = \text{X} \rightarrow \text{Z} \\
\text{X} & \rightarrow \text{Y} = \text{W} \rightarrow \text{Y} = \text{W} \rightarrow \text{Z} = \text{X} \rightarrow \text{Z} = \text{W} \rightarrow \text{Z}
\end{align*}
\]

Figure 6

4 Emblems

An emblem is characterized by the relative positions and sizes of its components, and by its orientation. Primitive emblems can be textual or iconic; the latter are either geometric or imaged, either drawings or shapes. [Emblems in a multi-media language may also be sounds.] A freestanding emblem represents a value that may be used in a structure. An emblem can decorate a link at an end, throughpoint, or junction; it can decorate a box at a vertex, edge, or surface. In any of these cases the emblem can either be directly at the point of decoration or it can be adjacent to it. An emblem may be quoted, such that it represents the value of its own syntactic structure; this is indicated by putting it in a box with a border consisting of curtains, rather than walls. A curtain is denoted here as a dotted edge or surface. An emblem is defined to have the value of a box when it is adjacent to it, separated by a curtain. An emblem is declared for use locally within a box by placing it adjacent to the inside boundary of the box, separated by a curtain. If a boundary is not decorated at the location of a curtain for a definition or declaration, the actual wall of the box may be depicted as though it were a curtain over the region adjacent to the emblem. Figure 7 gives examples of quotes, definitions, and declarations.

\[
\begin{align*}
\text{X} : 365 & \quad \text{the definition of } \text{x} \text{ to be the value } 365 \\
\{ \text{x} \} & \quad \text{the emblems } \text{x}, \text{y}, \text{ and } \text{z} \text{ declared to be local}
\end{align*}
\]

Figure 7

An emblem may be parameterized both internally and externally. An internal parameter binds a name to a component of the emblem; this name can be used in the definition to refer to the component. Such a name may appear more than once in the parameterized emblem; in this case, its value is the unification of its occurrences. The size and shape of a parameter relative to other components of the defined emblem specify the size and shape of the argument. A quote may be included in an emblem by quoting it in toto. A parameter may be parameterized by including quotes in quotes, as long as they do not comprise the entire parameter. A box in an emblem definition has a contents that is evaluated; the result must be part of the successful matching. An external parameter is bound to part of the environment of an emblem; it must be on an outskirt, and is indicated by leaving the curtain off of the side of the parameter’s quote away from the emblem. It may be anonymous. An external parameter is not evaluated when matching. If it is enclosed in the same structure in the definition as it was in the emblem, it remains unevaluated; if it is enclosed in the same structure but with walls substituted for curtains, evaluation is forced. Examples of parameterized definitions are offered in Figure 8.

\[
\begin{align*}
f \{ x+3 \} & \quad \text{f has a parameter x to its right} \\
f 5 & \quad \text{is an emblem standing for the value 8} \\
g \{ x+3 \} & \quad \text{defines the emblem } g \{ x+3 \} \text{ (which is not parameterized) to have the value } x+3 \\
3; f \{ x+3 \} & \quad \text{defines f in the middle of a sequence to be replaced by g, with the right hand side of the sequence being evaluated}
\end{align*}
\]

Figure 8
5 Vectors

A box may contain multiple instances of a value; enclosure is a join for multisets (bags), rather than sets. When values are repeated often, a shorthand can be developed to specify the number of occurrences, by defining coefficients. A coefficient is an emblem consisting of a number together with something indicative of its status; in the following, the character ":=" will follow the number. Some instances of coefficient definitions are given in Figure 9, though in reality the definition would be parameterized by the number as well.

0: \( \underline{\text{3.}} \) something present 0 times
1: \( \underline{\text{3.}} \) something present once
2: \( \underline{\text{3.}} \) something present twice
-1: \( \underline{\text{3.}} \) an inverse

The domain of coefficients can be extended to the rationals, the reals, the complex numbers, and then linear operators, mapping the domain of values to itself. With this last extension, values may be interpreted as vectors, with the operators matrices, and each row and column is indexed by a value in the domain. When the domain of coefficients is a field, primitive values can be interpreted as basis elements of a vector space, in which enclosure is vector addition. It is possible to define distances, magnitudes, complex conjugates and the like, eventually ending up with Hermitian operators and a Hilbert space, but at this point a more interesting line to pursue involves the re-examination of the interpretation of values as vectors.

The usual way to visualize vectors and operations on them in a space involves an image of arrows leaving the origin with a particular direction and magnitude: the greater the magnitude, the longer the arrow. Vector addition involves creating a parallelogram bounded by the vectors to be added, and the result is the point of that parallelogram opposite the origin. The identity element for addition is the origin; it corresponds to the empty box. Figure 10 has an example of this imagery.

Another kind of image is rather different. A vector is viewed as a set of points in a domain space, where each point has a magnitude associated with it analogous to star patterns. Vector addition consists of superposing two such spaces, summing the magnitudes at each point. The identity element for this approach is the empty space, with no points that have non-zero magnitude. An example of addition in this way is given in Figure 11, where points are coloured by their space of origin and magnitudes are omitted.

![Figure 11](image)

The reason for taking this view of the domain of values is that it is then possible to impose a different metric on the space. Such a metric can reflect computational information; for example, the integers could be clustered together, with booleans in a different cluster and characters in yet another. The metric might capture the chances of erroneous values being returned if values are "fuzzed", and understood not as single points in the space, but rather as normalized probability distributions. An imprecise numerical operation increases fuzziness as it causes values to diverge. However, this approach to reasoning about uncertainty is still speculative, and will not be pursued here.

6 Binding Values Together

The enclosure operation is flat, in the sense that when it is applied to singletons it does not generate new values. In this sense it is closer to the lattice join operation than it is to set enclosure; only infinite sets can be fixed points of the set enclosure operation. One problem with this is that it makes manipulation of the identity element difficult; it can only be detected through the absence of every other value. One interpretation of the empty box is as failure, the absence of any successful evaluation that satisfies a given set of constraints. In the world of computing, we often wish to reason explicitly about whether a program has succeeded or failed, and we want to be able to write programs that do this. It is possible to define new values to reflect various kinds of truth values, but in the general case it is desirable to reason about any collection of results that have been generated, as a single unit. For this reason, a binding operation is introduced, that maps any bag of values to a single value. Such an operation requires an inverse, so that a bound collection of values can be examined and
manipulated in terms of its constituents; the easiest way to define such an inverse is as symmetric to the binding operation. Thus, unbinding a bound value yields the original value, as does binding an unbound value. By symmetry, "unbinding" a bag of values yields a single new value, just as binding does; it merely has an opposite polarity. By convention, one operator is preferred for binding; the bound value of the empty box is called "false". The binding level of a value is the extent to which it has been bound and bound again; it is possible to reconstitute Russell’s hierarchy of classes in this way. It is necessary to introduce binding at this rather low level because it is non-monotonic.

7 Dataflow

The discussion of mappings was restricted to definitions of emblems as naming syntactic structures, effectively at the level of rewrite rules. This is not the best approach to writing readable programs, however; for that, we are going to want to add enough semantics to get dataflow, and eventually communicating objects and control flow. Dataflow requires the introduction of a special emblem that can be used to input arguments to mappings, represented as boxes, and to output their results. Instead of a "head-on" view of program evaluation, we are given a side view, a map of the way evaluation proceeds in terms of its dependency graph, with values flowing between boxes via links and conjunction. An arrowhead is chosen to indicate input and output when it decorates the boundary of a box. Multiple arrowheads can be introduced for arbitrary numbers of inputs and outputs; if these are to be ordered, the arrowheads must decorate box edges.

A box with no inputs is anadic, and denotes a value; a box with one input is monadic, and denotes a mapping; a box with two inputs is dyadic; and so forth. Mappings may have any number of results, indicated by the number of outputs; this is one of the main attractions of the dataflow approach over function evaluation leading to replacement rules, since that style does not lend itself well to multiple outputs. A given emblem may be defined to represent more than one mapping without ambiguity if the mappings have different numbers of inputs and/or outputs: the emblem is polymorphic with respect to such decorations. For example, the emblem + can represent both addition and subtraction if it is defined such that the sum of values passing through any given edge must equal the sum of values passing through an opposing edge. The value corresponding to the contents of a box may be output by using a different position for the arrowhead or augmenting it so that the emblem has an additional component. Informal experience suggests that inputs and outputs should both be centered about an edge they decorate, while a contents output is placed on the surface of the box away from an edge and with nothing at the base. The emblem for introducing a value into the contents of a box is similarly placed away from an edge but with nothing at the point. If one link carries a mapping that is to be applied to the value of another link, we can introduce the mapping into a box which has an input and output. As a shorthand, we can further overload the arrowhead. If it occurs along a link, it represents the identity mapping, such that failure on the result side cannot backtrack and affect the argument. If it occurs along a link and has a cross-link, the value of that cross-link is taken as a mapping and applied to the argument. Examples of these basic dataflow patterns are in Figure 12.

![Figure 12](image)

The emblem structure \( x \rightarrow y \) commonly denotes a mapping from \( x \) to \( y \), that fails if the argument is not \( x \). Using this, a convention that "?" stands for the position of an argument in an expression denoting a monadic mapping, and a further convention that truth conditions return their arguments if they are true (and fail otherwise), the recursive factorial mapping can be created. It is displayed in Figure 13.

![Figure 13](image)

If a box has one or more inputs but no outputs, it is interpreted as a proposition whose truth is determined by
its inputs. A box containing an input to a single value is a test; it succeeds (is true) when the input is that value, and fails (is false) when it is not. The empty box is interpreted as the bottom value, representing failure; thus an empty box with a single input and no output is a non-monotonic test for failure. A box with more than one input is a relation. The simplest relation is equality, in which inputs are linked together; inequality is represented by placing a diagonal bar on the link. A box is not necessary for equality; any link that connects outputs fails whenever the outputs have different values, while inequality fails whenever its arguments are the same. Some simple tests and relations are depicted in Figure 14.

( $$\exists$$ succeeds iff its argument is 3)
( $$\forall$$ succeeds iff its argument fails)
( $$\equiv$$ succeeds iff its arguments are equal)
( $$\neq$$ succeeds iff its arguments are different)
( $$\in$$ succeeds iff its argument is an int)

Two propositions that are not linked together are independent of one another. A box containing independent propositions fails only if they all fail, and evaluate to the empty value; thus, unlinked propositions are disjoint. Two linked or adjoining propositions are mutually dependent; if one fails, both do, since failure is evaluating to the empty box, which is a zero for linked boxes. As well as "and" and "or", propositions may be linked by implication; various negations are linked into implication (e.g. $$\neg x \Rightarrow y$$ has a special symbol); this avoids the need for constructing an explicit negation operation. Quantifiers are represented by the usual emblems ($$\forall$$ and $$\exists$$); these may decorate links. Figure 15 has simple examples.

( $$\forall$$ succeeds iff its argument is an int)

This is enough to begin to write specifications. A specification for a mapping has the general form of an implication, in which the antecedent is a check to see that the input is in the appropriate range, and the consequent is (i) a check to see that the output is in the appropriate range, and (ii) a relation that must hold between the input and output. Figure 16 contains the specification for a sorting algorithm.

$$\begin{align*}
\text{sort} & \quad \Rightarrow \\
\epsilon \in \text{sequence} & \quad \Rightarrow \\
\text{ordered} & \quad \epsilon \in \text{sequence} \\
\text{permutation} & \quad (y \in \text{sequence} \& \text{ordered}(y) \& \text{permutation}(x,y))
\end{align*}$$

A given algorithm, e.g. insertsort, would have to make this specification true; this would require demonstrating that the satisfaction assertion of Figure 17 is true.

$$\begin{align*}
\forall & \quad \text{insertsort} \\
\forall & \quad x. \text{sort}(x, \text{insertsort}(x))
\end{align*}$$

8 Conclusions

The design of a visual language requires immense care in selecting syntactic rules governing the kinds of structures that can be constructed. It is very easy to lapse into making ad hoc decisions, because the potential variety of styles is so high. An attempt has been made here to develop the syntax very slowly and carefully, incorporating a surprising amount of the language semantics in the process. If this approach is successful, it can make it very "natural" to think in terms of the meaning of the language when manipulating its components. Once the groundwork is laid, dataflow can be added, together with the logic required for program specification. This did not require departing from the algebraic syntax rules, which gives hope that they provide a reasonable basis for a programming and specification language.

Work continues on incorporating additional features into the language, with the intent of extending it to the point that it can be used for reasoning about real-time systems. This will require considerable tool support, especially in generating a hypertext library of specifications, proofs, and lemmas. It is too early to say whether the language will be up to the task, especially given the rate of change over the past few years.
However, it seems increasingly stable, offering hope that it is on the right track. The importance of giving users freedom in choosing their own notations within a framework should not be underestimated; it is in that direction that future foundational work will be pursued if problems arise.

References


