

Taxonomy for Visual Parallel Programming Languages

P. A. LEE and J. WEBBER¹

*School of Computing Science
University of Newcastle upon Tyne*

Abstract

Visual notations for constructing parallel programs have a number of potential advantages over traditional text-based forms, and several visual parallel programming languages have been proposed. This paper explores the design space for such visual languages, and presents a taxonomy that captures the salient underlying characteristics, with exemplification from a number of existing systems.

1. WHY VISUAL PARALLEL PROGRAMMING LANGUAGES?

The exploration of visual representations as a replacement for the more normal textual form for constructing programs has been a topic of research for a number of years. In the field of parallel programming, that is the construction of software systems whose constituent parts may be executed in parallel, a number of research efforts have also turned to visual representations as a possible solution to mastering the complexities inherent in this activity.

Parallel programming is difficult. For even the most skillful of software developers, the intricacies of developing parallel applications with traditional textual programming methods can be taxing, and error-prone. There are two main reasons why this is so:

- *Linearity of text-based source code*, which does not offer an intuitive representation of the true concurrent flows of control in a parallel application.
- *Parallelism complexity*, where on top of the complexities associated with the problem domain, the developer is forced to shoulder the burden of identifying and controlling the parallelism, sharing data, protecting access to shared resources and ensuring that critical sections of code are properly delimited.

The benefits of applying a language with a visual syntax to parallel programming are that both disadvantages associated with textual parallel programming can be overcome. Firstly, since a visual language is not limited to one dimension, unlike a textual language, multiple concurrent flows of control can appear naturally side-by-side. Secondly, the semantics of appropriate parts of the syntax of a visual language can be adapted to implicitly resolve a number of parallelism and concurrency control issues.

The purpose of this paper is to explore the design space for visual parallel programming languages (VPPLs) and to present a taxonomy that captures the important characteristics that underlie VPPLs. A number of existing VPPLs will be used to exemplify the taxonomy, but the taxonomy will also be of use to future VPPL designers for highlighting possibilities that might not otherwise be considered.

¹ Work supported by the EPSRC: PhD studentship (JW); and the HiPPO project.

Interest here is in VPPLs that take the form of a graphical mechanism for structuring and controlling parallel invocations of textually specified code written in a traditional (non-parallel) language such as C or FORTRAN. The VPPLs are visual coordination languages, in the sense that the Linda system [Gelernter 1985] is a textual coordination language for parallel programming, targeted at producing software that will execute on commodity hardware.

The taxonomy presented here concentrates on the parallel processing aspects of VPPLs, and is therefore different to other taxonomies on visual languages which cover other aspects of visual technology [Myers 1990, Price *et al.* 1993]. A key feature for this taxonomy to explore is the interaction between the visual features that a VPPL provides and the parallelism features required. The fact that some parallelism feature (e.g. synchronization between parallel elements) could be obtained through the use of the textual programming language parts that a VPPL utilizes is not something to be considered. What is of interest is the way in which the visual features of the language help the implementation of the parallel activity desired by the user, and whether those features are appropriate. This interest is reflected by the use of the terms *explicit* and *implicit*, to represent respectively the situations where the user does and does not have to deal with a parallelism feature directly. *Implicit* features are assumed to be helpful to the programmer.

One feature which does not form part of the taxonomy is tool support. Since each of the surveyed VPPLs forms, more or less, a research prototype it is not expected that complete environments for software development are available. Therefore, one of the non-functional requirements of this taxonomy is to distill features pertaining to the visual language, and isolate them from features provided by tool support, since it is the language element, and not the tool support, which is of importance.

2. VISUAL PARALLEL PROGRAMMING LANGUAGES

A number of contemporary visual parallel programming languages form a basis for exemplification of the VPPL taxonomy. VPPLs supporting traditional imperative programming paradigms include CODE [Browne 1992, Browne 1995, CODE], HeNCE [Beguelin 1994, Browne 1995], VPE [Newton and Dongarra 1994], and ParaDE [Allen 1997]. HiPPO [HiPPO] and Vorlon [Webber 2000, Webber and Lee 2001] are more recent object-oriented VPPLs. Like most visual notations, these languages provide “boxes” and “sticks” that can be drawn and connected together in some form of graph. The overall graph represents a parallel application that is going to be executed on some parallel hardware platform, and there is a process by which the parallel code can be automatically generated from the graph. It is not the purpose of this paper to explain these VPPLs and their environments in depth; the references provide detailed information. Here interest is focused on extracting the salient features of these VPPLs and on using the taxonomy to exemplify similarities and differences. (An extensive bibliography of visual language research is provided by Burnett [Burnett 2003]).

3. SYNTACTIC ELEMENTS OF VPPLS

Visually, a VPPL supports a graph structure in which the main programming constructs are:

- *nodes*, that are linked together by
- *arcs* that transport *tokens* of differing types between nodes

The main type of node is a computation node which is where the VPPL user specifies the application-specific computation. Other nodes may be provided by a VPPL to assist the management of the graph's structure. Nodes that are independent may be executed in parallel. As will be seen, nodes themselves have a number of syntactic elements from which they are composed.

Arcs interconnect nodes and provide an overall indication of the interdependencies between the nodes, amongst other things. Such inter-node dependencies are particularly important in a parallel processing environment as they determine the extent to which computation nodes may be executed in parallel.

Figure 1 shows the main syntactic elements of the taxonomy. The arrows in Figure 1 indicate 'has-a' or compositional relationships rather than indicating the syntactic rules that dictate the legal ways in which graphs can be composed.

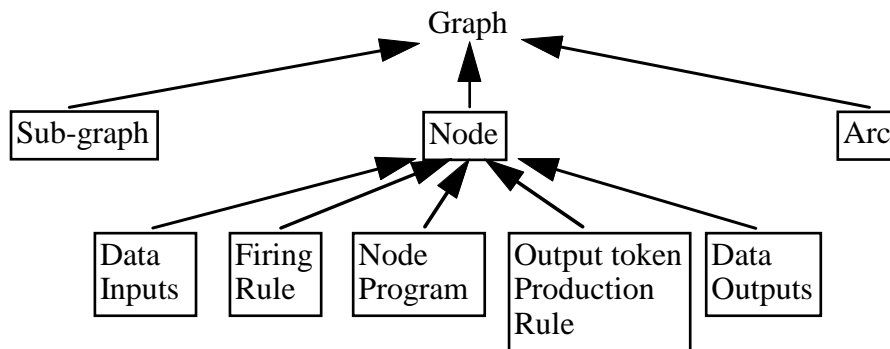


Fig. 1. Main Syntactic Elements of VPPL Taxonomy

In what follows, each of these syntactic elements will be examined in a top-down fashion with respect to:

- what options exist for the semantics of the syntactic element, and some discussion of those options;
- what sub-elements it is composed from (with those sub-elements themselves being examined subsequently); and
- exemplification from the example VPPLs.

3.1 Graph

Table 1 shows the elements that a graph is composed of and the main options for the semantics associated with a graph.

Decomposition or modularization of a complex system into more manageable (and reusable) components is a well-recognized requirement. This requirement is likely to be of particular importance in graphical notations which inevitably occupy more space than compact textual representations. VPPLs can support this requirement by permitting one graph to be composed from other (sub-)graphs. Visual complexity can then be reduced by permitting a node on one graph to refer to a sub-graph constructed separately. This feature also permits graph-reuse (c.f. code reuse), a useful feature in itself for the usual reasons.

Composed of		
Sub-Graphs	Nodes	Arcs

Options	Graph Semantics					
	Structure		Node dependencies		Recursion	
	Directed	Directed	Visible	Invisible	Permitted	Not
Cyclic	Acyclic				Permitted	

Table 1. Graph Elements and Semantics

When sub-graphing is permitted the VPPL will need to provide (visual) means for expressing the way in which connections represented on the parent graph are reflected onto connections in the sub-graph. These connections are usually represented by special types of connector nodes in the sub-graph and appropriate naming conventions to identify the cross-coupling.

The feature of sub-graphs raises the semantic issue of whether recursion (at the graph level) is supported or not. A useful comparison may be drawn here between the issues of macros and functions in languages such as C. Functions may be recursive, requiring the dynamic generation of information at run-time for support. A recursive graph (i.e. a graph that “calls” itself) will also require dynamic support in the form of instantiation of a graph and its arcs at run-time. Non-recursive graphs can be static and completely instantiated at graph-compile time (whether implemented equivalent to a function-call or a macro).

A statically-generated system had the advantage that the complete parallel structure is known before execution commences, which may provide advantages for statically mapping the potential parallelism onto the underlying parallel platform. Dynamic generation has the advantage of supporting recursion, but the disadvantage that dynamic run-time load balancing may be needed to support the evolving structure of the parallel application efficiently. Note however that run-time load balancing may be a requirement in any case if the load on the parallel platform may vary.

As mentioned earlier, the arcs in a graph carry *tokens* (of various types) that capture dynamic aspects of the behavior of the application represented by the graph. Thus arcs show visible node interdependencies and hence impose order relationships on the execution of nodes. As will be discussed subsequently, the dependencies may be of the form of control-flow dependencies (e.g. node B must execute after node A) or data dependencies (e.g. node B needs data produced by node A). Since the graphs capture some of these dependency relationships, all of the VPPLs considered in this paper are *directed*, and may have a *cyclic* or *acyclic* structure. Arcs provide *visible* indications of node dependencies. Alternatively, special node types on a graph may support *invisible* dependencies, in that some links between the nodes do not have a graph-level representation.

Visible and invisible dependencies have advantages and disadvantages. Visible dependencies can be visualized at-a-glance on the graph, while invisible dependencies are “hidden” and cannot be discerned just from the graph’s appearance. However, allowing invisible dependencies on a graph will minimize the need for arcs and therefore reduce a graph’s visual complexity.

Cyclic graphs permit the expression of reusability, in an iterative sense. The nodes enclosed in a cycle essentially represent what would be a repetition loop in a

sequential programming language, permitting (a) a section of the graph to be reused; and (b) feedback of results from one iteration to the next. This is a useful programming paradigm to support even in parallel applications, and VPPLs which are acyclic generally provide other means for expressing iterative repetitions (normally by using a special type of control-flow modifier node).

Graph Semantics				
	Sub-Graphs	Structure	Node dependencies	Recursion
HeNCE	Not permitted	Directed acyclic	Visible & Invisible	Not permitted
CODE	Permitted	Directed cyclic	Visible & Invisible	Permitted
ParaDE	Permitted	Directed acyclic	Visible	Not permitted
VPE	Permitted	Directed cyclic	Visible	Not permitted
Vorlon	Permitted	Directed acyclic	Visible	Permitted
HiPPO	Permitted	Directed acyclic	Visible	Permitted

Table 2. VPPL Graph Examples

Table 2 shows examples of the various graph semantics adopted by different VPPLs. HeNCE graphs are directed and acyclic, and sub-graphs and recursion are not supported. While it was claimed [Browne *et al.* 1995] that sub-graphing was just a support-tool issue, in fact other HeNCE features would make sub-graphing problematical for purposes of reuse (although not for managing visual complexity). The features causing these problems will be explained in Section 3.4.3. Node control-flow dependencies are visible in HeNCE (via arcs), although data dependencies are invisible at the graph level.

CODE provides a true sub-graph, function-call abstraction, and as a consequence can support recursion. While CODE arcs capture some node dependencies, CODE also permits invisible dependencies to occur between nodes in a sub-graph: nodes within the confines of a sub-graph may be involved in a data dependency relation with other nodes, without arcs being present to visualize those relationships. A special node type is provided simply to establish the scope of data that can be used (possibly in parallel) by the nodes in that sub-graph. This node is visible, but the dependency relationships are classified as invisible since it cannot be determined which nodes are accessing these shared data items from the graph structure alone. The approach is clearly meant to improve the visual clarity, and thus scalability, of the notation. Obvious dependencies, in CODE's case being shared memory abstractions, do not require the explicit addition of arcs to each node in a sub-graph. In order to use the CODE shared-memory abstractions the programmer merely has to declare computational graph nodes as readers or writers of a particular shared-memory element in its node stanza (discussed below).

ParaDE supported sub-graphs. The prototype tools only provided support for sub-graphs in reducing visual complexity and not for graph-reuse. However, this was a tool support issue as ParaDE contained the necessary language elements for graph-reuse although not for recursion (discussed later).

Both HeNCE and ParaDE supported the notion of special-purpose iterative graph nodes which could circumvent the normal acyclic graph semantics in order to achieve iteration and result feedback, within a strictly delimited area of the graph. These are discussed in the next section.

Vorlon and HiPPO are similar to HeNCE and ParaDE in that they also provide special nodes which deal with graph-level control-flow issues such as iteration. However, sub-graphing is only used for managing visual complexity of graphs, and not as a means of recursion as in CODE and ParaDE. Instead Vorlon and HiPPO support recursion through recursive method calls.

3.2 Node

Nodes in a graph support the specification of the functionality of the parallel application. While a VPPL may provide a variety of different nodes, those that support user-defined computations are clearly the fundamental building blocks from which a parallel solution is built. The node classification is shown in Table 3.

Options	Node Semantics				
	Type		Purpose		State
Compu- tational	Graph management	Special	General	Retained	Not retained

Table 3. Node Classification

It is convenient to distinguish two types of node: *computational nodes*, that permit a user to define computational behaviour; and *graph management nodes*, that are concerned with controlling aspects of the graph such as:

1. Identifying particular structures within a graph (e.g. node replication or parallelism); and
2. Providing behaviour which would not be obtainable by using the general-purpose nodes (e.g. they may have different firing rules).

Nodes may provide *special-purpose* (i.e. predefined) behaviour or be *general-purpose* (user-programmable, in a textual programming language). They may use a variety of shapes for their representation, the differences in which are not important here.

The *state* issue for a node is a fundamental part of the computational model provided to the user. A node may be able to retain state between executions of that node or may not. The ability to retain state is useful in that it can avoid the overhead of re-obtaining identical input data needed for every execution (e.g. a node repeatedly executing with a varying input and a fixed input). In cyclic graphs the ability to retain state can be used to reduce the visual complexity of the graph in that a feedback arc is not needed. In acyclic graphs state retention can implement the feedback path which otherwise is not available, since arcs cannot form cycles.

When executed, as dictated by the firing rule, a node obtains input data from some other part of the parallel computation, processes that data using the normal features of the sequential language, generates data to be consumed elsewhere, and produces tokens on any outgoing arcs. Discussion below of these constituent parts of a node elaborates on how this behaviour can be achieved.

The majority of the example VPPLs considered in this paper use a sequential programming language as the basis for their general-purpose computational node. The majority permit the use of C or FORTRAN (or both). The CODE system provides a C-like language for node programming, although permits callout from a node to a C function. CODE computational nodes can retain state. Vorlon supports the use of C++ for its object-oriented paradigm. HiPPO, however, does not use an existing language.

HeNCE provides special-purpose graph management nodes to identify pipelines, replicated activity, conditional execution and iterations. HeNCE nodes cannot retain state.

ParaDE provides a graph management node called the loop actor which is capable of repeating the contents of a whole sub-graph iteratively. ParaDE has other graph management nodes such as arc replication and merging nodes, and the “depth actor” node to support data-parallel replication. Thus the depth actor is for both computation and graph management and is general-purpose. ParaDE nodes also cannot retain state.

Vorlon nodes provide a set of graph management nodes similar to those of ParaDE, implementing iterative, data-parallel, broadcast, and selection patterns. Nodes in Vorlon are stateless, but since Vorlon adopts an object-oriented view of an application, the elements which are referenced from Vorlon graphs (objects) can themselves retain state.

HiPPO provides graph management nodes for replicating arcs, for specifying input and outputs, and for specifying iterative and data-parallel computations. The HiPPO computational nodes reflect the object-oriented nature of the language, representing method calls and the like. The (shared) objects manipulated by a HiPPO program retain state, and the nodes on a HiPPO graph are stateless.

3.3 Arcs

Arcs are a fundamental building block of VPPLs and are not decomposable. As discussed earlier, graphs in the surveyed VPPLs are directed; arcs therefore capture interdependency relationships between nodes. Table 4 captures the general semantic options for arcs, together with an additional set of semantics that can be considered when arcs carry some form of data items.

Options		Arc Semantics									
		Tokens Carried		Direction		Capacity		Consumption			Connection
Control	Data	Supply	Demand	Single item	Infinite	When used	Explicit	Other	1:1	N:M	

Options		Data Arc: Additional Semantics				
		Data Tokens		Structure		Transformations
Value	Reference	Single item	Container	Yes	No	

Table 4. Arc Classification

Though arcs are conceptually simple, there are in fact numerous subtle nuances which are worthy of further investigation. Several of the semantic options for arcs dictate the model of execution for the nodes in a graph, although detailed discussion of these models is best left to the section on node firing rules below.

The *direction* characteristic of an arc captures whether an arc is used as a *supply* route from a source node to a destination node that follows it, or as a *demand* route from the destination node back to the source node.

Arcs can carry tokens which represent *control* information, *data*, or some combination of these. Arcs that carry just control information indicate an execution ordering relationship between two nodes. In effect an arc carries a signal indicating completion from the source node to the destination for supply arcs, or a signal which

requests execution with demand arcs. The input data required by the receiving node must then be obtained through some other mechanism; this is discussed in the section below on node data inputs.

Since supply arcs connect independent sources and destinations (whether carrying data or control), it may be possible for a source to race ahead of the consumer. The *capacity* of an arc is therefore an issue to be considered. If an arc can only hold a single item then additional synchronization is implicit in the VPPL using such an arc since a producer cannot generate a new item on an arc before the previous has been consumed. Demand arcs effectively carry only a single item and hence keep source and destination nodes in step. If arcs have an infinite capacity (limited by practical considerations of course) then the VPPL potentially provides a more relaxed execution model with additional parallelism opportunities. Note, however, a VPPL that uses control arcs for node synchronizations coupled with some sharing mechanism for common data is likely to be difficult to program if those arcs have a capacity for more than one completion signal, since there would be a difficulty in keeping the multiple incarnations in order.

The *consumption* aspect of arc semantics addresses the issue of when tokens are consumed from an arc. This may occur implicitly when the tokens are used (e.g. when data is passed to the consuming node), or explicit instructions in a node may be required. Other consumption behaviour could include “never”, in that a token once generated is never removed, or other more complex schemes such as a token is only removed if a replacement token is generated by the sender. Such semantics could help reduce communications overheads if a node’s input is constant and hence does not have to be regenerated every time that node executes.

Arcs normally provide *1:1 connections* between nodes in a graph. However, there may be *N:M* relationships required between the nodes. For arcs carrying control tokens, a 1 to *N* arc effectively fires *N* nodes in parallel (e.g. to obtain data parallelism) while *N* to 1 arcs effectively provide a synchronization point for *N* parallel nodes. For arcs carrying data tokens, a 1:*N* arc may be used to provide the same data to *N* nodes for data parallelism, while an *N*:1 arc would indicate the recombination of a number of separate data streams into one. Such behaviour could also be provided by permitting *N:M* arcs. (Clearly, such arcs would be problematical for data arcs carrying different types of data.)

An alternative to *N:M* arcs that may be adopted in VPPLs is to combine 1:1 arcs with the graph management node types introduced in section 3.2, to split and merge arc contents, or to require the use of a general-purpose node which can be explicitly programmed with replicate- or merge-like facilities.

Initially, one might question whether arcs other than 1:1 were justifiable, given that nodes can be used to achieve splitting and merging. Requiring the user to explicitly program a node to do splitting/merging, while straightforward, is not providing the right level of (visual) language support to the parallel programmer, particularly for the situation of arcs carrying data. Here one would prefer the VPPL to take the burden of managing the data flows; the use of special-purpose replicate/merge nodes goes some way to achieving this, and then the only argument in favour of *N:M* arcs is if they simplify the visual complexity of the graph as compared to that containing extra nodes.

Arcs that carry data are naturally used to carry data that has to be shared between nodes. Data-arcs can be typed in that the VPPL associates a data type with an arc and hence can provide traditional compile-time error detection on node interconnections.

VPPLs using supply data arcs are based on the dataflow model of parallel computation [Treleaven *et al.* 1982] where the execution of a node is determined primarily by the availability of data on its input arcs. Thus data arcs can indicate execution order relationships as well as holding the data a particular node requires, and therefore carry control information implicitly. (This topic is returned to in the discussion of firing rules below.) Demand data arcs would be used by a destination node to signal the preceding node to commence generation of the required data.

Data tokens can be sub-divided into *values* and *references*. References (or “pointers”) permit nodes to share data, thus avoiding the need to partition and copy data items around. Traditionally, references have not been supported on data arcs for two main reasons. Firstly, as VPPLs have been targeted at parallel architectures that encompass distributed memory there would be some implementation difficulties in implementing the abstraction of sharing. A VPPL that targeted just shared memory multiprocessor architectures would not suffer from this implementation problem. Secondly, if data is shared, concurrency-control mechanisms, such as those found in the shared memory abstractions of the CODE language, are required to ensure parallel accesses to that data are properly synchronized. Ideally, such mechanisms would be implicitly invoked.

A further issue with data arcs is whether they carry *single items* or whether they support the transport of *containers* of items. Since most VPPLs are concerned with FORTRAN-like programming languages and numerical applications, arrays are often the only containers supported, although one might imagine that support for C `structs` (i.e. records) could be appropriate when the C language was the textual programming language used in the VPPL. Clearly many other container types exist (sets, bags, lists, etc.) but as these depend on user-level programming conventions, it would be difficult for a VPPL to be able to automatically transform and transport such containers. The limitation in practice of containers being only arrays is not therefore surprising.

Arcs are naturally thought of as conduits without processing capabilities. However, the possibility of an arc performing some form of data *transformations* could be considered. If transforms of single data items were possible, the distinction between an arc and a node becomes less clear. However, considering further the issue of managing data together with the ability for an arc to carry containers such as arrays, other possibilities emerge. When applying parallel processing to arrays of data, it is common to split up those arrays into smaller units which can then be processed in parallel before being recombined back into the output data structure. Matrix multiplication is the typical example. Of course all of the data partitioning can be achieved in a general-purpose node, but that is an additional requirement on the programmer and adds additional complexity (and possible errors) to the code they have to generate. For example, splitting a matrix into rows (or columns) usually requires additional code to permit the rows to be explicitly identified for subsequent recombination into a result matrix. In effect what is required is an arc with partitioning capabilities (simulating 1:*N*, splitting a structure up for parallel processing) and an arc with recombination capabilities (simulating *N*:1).

Another transformation might be for an arc to convert a container of values into a stream of individual items, for example to be fed into a pipeline structure.

Table 5 gives arc examples from the surveyed VPPLs. HeNCE arcs carry control tokens only. Special-purpose replicate (“fan”) nodes in HeNCE provide 1:N and N:1 connections. Clearly, a VPPL such as HeNCE that provides only control-information arcs has to provide some other means for the communication of data between interdependent nodes.

Arc Semantics									
	Tokens Carried		Direction		Capacity		Consumption		Connection
HeNCE	Control		Supply		Single item		When used		1:1 1:N N:1
CODE		Data	Supply			Infinite	When used		1:N N:1
ParaDE		Data	Supply			Infinite	When used	Other	1:1
VPE		Data	Supply			Infinite		Explicit	1:1
Vorlon	Control	Data	Supply		Single item		When used		1:1
HiPPO		Data	Supply		Single item		When Used		1:1

Data Arc: Additional Semantics						
	Data		Structure		Transformations	
CODE	Value		Single item	Array		No
ParaDE	Value		Single item	Array	Yes	
VPE	Value		Single item	Array		No
Vorlon		Reference	Single item			No
HiPPO		Reference	Single item			No

Table 5. Arc Examples

All of the VPPLs have *supply* arcs. Most consume arc tokens when those tokens are used, the exception being VPE which requires consumption to be specified explicitly. The ParaDE system in addition had another mechanism whereby a special arc type had the semantics that a data token wasn’t consumed when used. This could lead to a reduction in communications overheads in some applications, although the semantics of when such a token is removed (e.g. by another token being generated on that arc) are complex to deal with in an application.

In three of the surveyed VPPLs that carry data tokens, both single values and arrays of values were permitted to travel between nodes. However, where the languages differed was in the support provided for decomposition of such structures to feed into parallel processes to exploit data parallelism. The best support is provided in the ParaDE language, where an arc can be equipped with a decomposition template (from a variety supplied with the language) which automatically decomposes and distributes data from an array to a number of parallel processes before recombining the results of those processes back into a single structure. Typical templates would be a column or row of a 2-D array. The level of support provided by the other languages was significantly less than that provided by ParaDE. Although it is possible to decompose, process in parallel, and recompose data in the other languages, it is left to the developer to explicitly implement such schemes.

While the ParaDE decomposition mechanism is in theory arc-based, in practice such arcs could only be connected to a special parallel computation node (the depth actor). It is debatable therefore whether such mechanisms should be associated with an arc or with particular computation nodes.

The CODE language in common with the ParaDE language allows decomposition to be specified at the meta (i.e. graphical) level, though unlike the ParaDE system there is no graphical support for doing such. Instead, arcs in the CODE system are given an index as they are instantiated at run-time, which may be used to identify units of decomposed data to facilitate recomposition.

Only the CODE language actually takes an approach which relies on 1:N and N:1 arcs to achieve container transformations. Since CODE arcs are programmed with the sources and destinations of any data passing through them, it is quite feasible for either the sources or destinations to be the same node for many instances of the arc.

The CODE system with its dynamically instantiated arcs suffers from the fact that not all possible execution paths can be seen in its program graphs, but does allow a program to alter its structure (to the limited extent of adding or choosing not to add more or less of the same arcs specified in a program graph) at run-time.

Only Vorlon and HiPPO amongst the surveyed VPPLs possesses arcs that carry references (to objects) and such references can be to arbitrary types including container types. Conceptually CODE's shared memory abstractions exhibit similar semantics. Some node types in Vorlon are able to automatically decompose such linear structures for parallel processing, in a mode very similar to the depth actor in ParaDE. HiPPO also has a special decompose node for generating parallel processing, with the user able to specify the strategy that a decomposition takes.

3.4 Node Syntactic Elements

In order to understand fully the behaviour of a node in a VPPL, the issues to be considered include:

- when a node is executed (firing rules);
- the data inputs - how a node obtains data from other parts of the application;
- the (textual) program within a node and how that interacts with the visual features;
- the data outputs that a node generates for subsequent processing; and
- the rule for the production of output tokens from a node.

Thus the main elements that comprise a node are shown in Table 6.

Composed of				
Firing Rule	Input Data	Node Program	Output Token Production Rule	Output Data

Table 6. Node Syntactic Elements

3.4.1 Node: Firing Rule

The firing rule specifies the conditions under which the execution of a node is commenced. In turn this may affect the synchronization that will be needed between parallel nodes. The firing rule options are shown in Table 7.

		Node: Firing Rule Semantics	
Options	No rule	Arc-based rule	
		Fixed	User-programmable

Table 7. Node Firing Rule Semantics

As has been discussed, the arcs in a VPPL are directed and provide dependency information in a supply or demand direction. Thus one set of semantics for firing rules is naturally *arc-based*, and dependent upon the arcs that are incident (i.e. directed towards) to the node and their contents. However, some nodes may have *no firing rule*, and instead commence their execution when the graph of which they are a part is “executed”. One set of nodes for which this would be the appropriate behaviour are those which do not have any incident input arcs. Nodes that represent the starting point(s) for the execution of the application represented by a graph using supply arcs (or the end node for a graph using demand arcs), or those that generate tokens on an arc, are cases in point.

For nodes that have incident input arcs, a *no-rule* firing rule is less appropriate. A VPPL that combined a no-firing-rule scheme with arcs carrying control information would not make any sense. A VPPL that uses data arcs with the no-firing-rule is providing little more than a graphical interface to an underlying data communications mechanism (VPE is such a VPPL), and would still requires the programmer to deal explicitly with synchronization aspects (e.g. through explicit message read operations). Thus most VPPLs have implicit arc-based firing rules which simplify what the parallel programmer has to provide.

The options for firing a node based on arc contents, coupled with the arc semantics discussed earlier, give rise to node executions that follow a common classification [Treleaven *et al.* 1982] of *data-driven*, *control-driven* or *demand-driven* firing rules. When incident arcs have:

- “supply” + “data value” semantics, arc-based firing corresponds to the data-driven semantics, where a node can execute when its inputs are available;
- “supply” + “control” semantics, arc-based firing naturally corresponds to the control-driven situation, indicating that a preceding node has completed its execution
- “demand” semantics, arc-based firing would cause a node to fire when outputs from that node are needed. Demand-driven semantics give rise to a lazy form of execution.

In the most general case, a node may have multiple incoming arcs which have to be considered in the firing rule. Of particular importance is the rule that applies to computational nodes, since the programmer has to understand the rule in order to use the system. The rule may be *fixed* by the VPPL, the simple possibilities being that the node fires when *all* or *any* of the arcs contain values. (Other special-purpose nodes may break the fixed rules in order to provide different semantics.) Flexibility would be provided in a *user-programmable* scheme which would permit the programmer to specify the exact firing rules based on logical combinations of arc contents. While this flexibility appears on the surface to be desirable, in practice it means that the overall behaviour of the graph can only be discerned by examining all of the firing rules in detail. A VPPL with fixed firing rules may therefore be easier to comprehend from the graph level.

If a computational node can fire when only some of its incoming arcs contain values then the VPPL has to provide some means for the program in the node to differentiate between the various cases that could arise - for example, to determine which data arcs do or do not contain valid values. This is clearly more complex than the “fire when all present” case.

The ability for a node to fire when only some inputs are present provides one form of conditional execution within a node in a graph. With the “fire when all present” rule conditional execution in a graph can be achieved through the conditional production of tokens on output arcs. This is discussed in the output token production rule section below.

The CODE system supports a user-programmable scheme for defining the firing rules for a node, while all of the other VPPLs have fixed firing rules, although different firing rules can be imposed by different kinds of special-purpose node. For example, ParaDE contains two nodes that have no firing rule: a source node that corresponds to the standard input stream of a sequential program, and a file node that interfaces with the host system’s file store. ParaDE computational nodes have a fixed arc-based rule of firing when all incident arcs contain tokens. HeNCE computational nodes are similar, in that a node can fire when all of its predecessors have completed. HiPPO nodes fire when all input arcs contain tokens (object references), while separate true/false graphs are associated with a special conditional node and invoked automatically as required.

3.4.2 Node: Input Data and Output Data

Interest here concerns the model that indicates how a node obtains data from other parts of the parallel computation, and how obtaining that data is supported by the VPPL; the options are shown in Table 8. Issues concerning the mapping of input data to variables in a (sequential) programming language used in the computational nodes are addressed in the next section (Node: Program). Since output data issues are very similar, they are not expounded upon further.

Node: Data Input Semantics				
Options	Copy			
	<table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="text-align: center;">Shared</th> </tr> </thead> <tbody> <tr> <td style="text-align: center;">Implicit</td> <td style="text-align: center;">Explicit</td> </tr> </tbody> </table>	Shared		Implicit
Shared				
Implicit	Explicit			

Table 8. Node: Input Data Semantics

For parallel programming, the key issue concerning non-local data is the issue of *copy* versus *shared* - that is, whether a node receives copies of data values that it is to process, or whether the data is accessed from some shared area. Data copying suffers the overhead of that copying, mitigated by the fact that a node can then access that data locally and without restriction. Shared data can avoid the copying overhead but may require locking overheads to be imposed to ensure orderly use of the shared space. Ideally, locking will be implicitly provided by the VPPL in order to avoid the burden of requiring the programmer to deal explicitly with the locking issues. Of course, the parallel platform may itself impose further overheads on these semantics - for instance, to implement the abstraction of data sharing on a distributed memory machine.

For arcs that carry data, the most natural semantics are those of copying the data. For arcs that carry references, the sharing semantics follow naturally, although HiPPO permits object cloning to be specified by the programmer to provide the advantages of

non-shared objects . For arcs that just carry control information, the data inputs are an orthogonal issue that has to be supported somehow in the VPPL, such as the in the node interfaces in HeNCE, supporting either copy or sharing semantics.

3.4.3 Node: Program

The node program is where the programmer specifies the computation that a particular computational node undertakes. Issues concerning how variables in the programming language used get mapped onto the data inputs that are provided to the computational node, and where any such mappings are specified, are issues to be addressed here. The options are shown in Table 9.

A VPPL can provide a *proprietary* programming language with features that fit into its graphical framework, such as dealing with data inputs. Such languages are likely to be sequential, as the graph-level is where the parallelism of the application is meant to be captured. More commonly, VPPLs support *standard* sequential languages such as C and FORTRAN, because the purpose of the VPPL is to build upon programmer's strengths in those languages while implicitly handling the parallelism features (to a greater or lesser extent).

Node: Program			
Language		Variable mapping	
Proprietary	Standard	Implicit	Explicit

Table 9. Node Program Options

Programming the computational nodes, particularly in standard languages, will require the use of variables, and an issue therefore to be considered is how the variables in the node program are mapped onto the data inputs and outputs. Ideally, a VPPL will provide *implicit mappings* from the data inputs (e.g. those arriving on arcs) to the variables in the node program. Alternatively the VPPL could permit *explicit mappings* to be specified using some form of (proprietary) language. Similar issues arise for data that a node has to generate for its data outputs.

In either scheme, textual and graphical elements of the application must be closely coupled. Conditional execution of nodes based upon arrival of tokens necessitates the construction of node programs which understand the node's interface, which is undesirable.

The issue concerning variable mappings also concerns what in traditional programming are called formal and actual parameters. Programming languages allow a procedure to have formal parameters that are mapped to actual parameters at run-time. This permits a procedure to be reused from a number of places. The same issue arises in VPPLs - to permit a computational node to be programmed in a general manner and then reused, in a parallel application.

CODE provides a proprietary (C-like) programming language, although also supports callouts to functions written in C. HeNCE and VPE use C or FORTRAN; ParaDE also uses C. Vorlon supports the use of C++.

In HeNCE mappings between the local variables used in a computation node and the shared data in the global namespace are specified in a proprietary language. However, there is no mechanism in HeNCE for defining formal parameters for a node. Instead, the actual parameters are hard-coded into the node interface, and from those hard-coded parameters the appropriate variables are accessed from the global namespace.

This is the reason why HeNCE does not readily lend itself to supporting reusable sub-graphs. Once a node has received sufficient control-flow stimuli from predecessor nodes, it begins its execution and at that point it accesses data elements from the global namespace according to the interface specified for that node. Since the interface explicitly names precise elements from the global namespace which are to be used during the execution of that node, there is no means of distinguishing actual and formal parameters, so in effect parameters are hard-coded into nodes. The addition of sub-graphs into the language does not alleviate this problem, since sub-graph nodes themselves, and their contents, must have their parameters similarly hard-coded.

In CODE, ParaDE, and in some sense VPE too, there exist mechanisms for mapping actual onto formal parameters. In CODE, the formal parameters are presented by the number and type of data-flow arcs and the actual parameters by data values flowing along those arcs. A similar scheme is seen in ParaDE, though ParaDE's tool support automates the mapping between graph level parameters and those in the node program, whereas the CODE approach does not.

VPE provides formal parameters in the sense that its port nodes decouple intercommunicating processes. The code residing in a VPE compute node has an environment consisting of a textual interface to each of the port nodes attached to the compute node, and so in some sense has the notion of formal parameters. Actual parameters in VPE are data values exchanged during the message-passing phase of a compute node's program.

Vorlon and HiPPO are object-oriented and as such make a strong distinction between interface and implementation. It is taken as understood that having such interfaces means implicitly that formal and actual parameters for method calls are a part of the language. From the point of view of any textual code within a Vorlon node, the formal parameters appear as appropriately typed variables in that language. For visual code, the formal parameter list appears as a set of object source nodes in a graph.

3.4.4 Node: Output Token Production Rule

Given that the arcs between nodes indicate some kind of dependencies between those nodes, the options for the alternatives by which one node can control the generation of tokens on its outgoing arcs need to be considered, and are indicated in Table 10.

Node: Output Token Production Rule							
Options	Arc Coverage			Transmission		Streamed Output	
	All	Some	None	Implicit	Explicit	Yes	No

Table 10. Node Output Token Production Rule Options

Arc coverage addresses the issue of how many out-going arcs may be used simultaneously by a node. It is assumed that a computational node may be permitted to have multiple outgoing arcs, since a restriction on there only being one arc may not make sense in a parallel application. The rules may require there to be tokens on *all* output arcs, *some*, or *none*. Conditional execution may be supported by *some* arcs, while nodes with no outgoing arcs act as sinks for data (e.g. storing results into a data file, or acting to terminate the execution).

The VPPL has to define when output tokens can be *transmitted*. This could be *explicit*, via some special instruction used in the node program. Transmission could be

implicit by the system taking action when a node program's execution has completed, or even whenever a variable associated with an output arc was updated. The latter, or explicit generation, would permit a node to generate a stream of output tokens (e.g. in a pipeline structure). Again, special-purpose nodes can provide different semantics in order to achieve streaming behaviour.

In CODE, output rules are explicitly specified as part of the node's stanzas, determining the conditions under which the output arcs from the node will become active. As such, CODE supports all three options for arc coverage. Transmission is implicit in CODE since arcs simply appear as variables in the node program, and variable access is then equivalent to moving data onto an arc. Streaming from any arc is permitted since a computation may contain a loop which continuously updates the variable that represents the output arc and thus continuously cause output on that arc.

HeNCE adopts an *all arcs* output rule whereby control-flow signals from a predecessor node will reach all successors. The sending of output tokens is implicit in that the completion of the node program causes the tokens to be produced, and since the node program can, by definition, complete only once, streaming is not permitted at the level of an individual node. (However, special pipeline nodes permit streaming, and hence pipeline behaviour.) Some HeNCE nodes have no output arcs (indicating the end of a graph or sub-graph).

ParaDE's actor node permits output on all or a subset of its output arcs (the latter for conditional execution). Moreover, special sink nodes have no output arcs. Sending is implicit in ParaDE since arcs are bound to textual variables and any of those variables which are updated during the course of the execution of the node program are automatically sent along their output node when the node program finishes. Although streaming is supported by ParaDE graphs, only the loop actor can instigate it since it possesses a per-iteration output semantic.

Arcs in VPE represent possible routes of message-passing, not dependencies per-se and as such can be used as and when messages are to be passed in whatever combination is required at that time. Sending is explicit via message-passing calls in the textual node program (as is receiving). Streams are permitted through asynchronous sends.

In the general case, Vorlon expects output of a single object handle along each node's single output arc per execution. The only exception to this semantic is the conditional execution node which is permitted to output on either one or both of its output arcs depending on whether the node was executed or not. Streaming is not permitted in Vorlon since it is known that streaming in other acyclic languages such as ParaDE was problematic, and often lead to races within graphs.

In HiPPO object handles are transmitted implicitly from a graph when all of the activity in that graph has completed.

4.0 SUPPORT FOR MODES OF PARALLEL EXECUTION

A VPPL language must, by definition, provide means for expressing parallel activity. Some of these activities, such as pipelining, have already been encountered in the discussion whilst others, such as task parallelism, have been alluded to. Each of the languages under examination provides mechanisms which support various parallel activities. The functionality of such mechanisms is the concern of this section. While not of concern for parallelism, it is also interesting to identify how the VPPLs deal

with iteration, conditional execution and recursion, since these have an important impact on the implementation of an application.

Table 11 shows the main execution patterns and whether or how these are achieved in the VPPLs.

VPPL Pattern	ParaDE	HeNCE	CODE	VPE	Vorlon	HiPPO
Data parallel	Depth Actor	Fan-in/out nodes	Node stanza	Replicate Box	Parallel computation Parallel method call	Foreach node (user specified data decomposition)
Task Parallel	Yes	Yes	Yes	Yes	Yes	Yes
Pipeline	Streamed output from loop	Pipeline begin/end nodes	Streamed output from Node	Asynchronous sends in node program	No	No
Iteration	Loop actor	Loop begin/end nodes	Cyclic graphs	No graph-level iteration	Loop Node	Loop node
Conditional	Conditional output token transmission	Conditional begin/end nodes	Node stanza	No graph-level conditional construct.	Conditional node	Condition node with separate true/false graphs
Recursion	No	No	Yes	Yes	Yes	Yes

Table 11. Execution Behaviours in VPPLs

Whether the means of extracting parallel activity are implicit or those means are provided through explicit language features, has an effect on the way the developer builds a parallel application. Given that it is preferable for parallel activity to be implicit within the structure of an application, rather than to have the developer explicitly identifying parallelism, the main issue for the languages is to what degree they encourage the construction of naturally parallel applications (and indirectly the suitability of the underlying execution models), and what degree of support they provide when they require the developer to explicitly invoke parallel computation.

The VPPLs all support task parallelism implicitly, whereby computational elements which are not directly interconnected may execute independently. Data parallelism however is more explicit, albeit with varying levels of support as is discussed below. Given that data parallelism is often the focus of the numerical problem domain that several of the VPPLs target, this balance of explicit/implicit features is perhaps less than optimal.

In addition to task, pipeline, and data parallelism, parallel languages have previously supported parallel activity from logically distinct loop iterations, and have exploited parallelism from recursive operations. Support for iterative style parallel activity is lacking in the VPPLs since data parallel mechanisms are used to achieve the same effect. Furthermore, it has been reasoned by Allen that iteration should be a purely sequential construct in a visual parallel programming language, and that it is a parallel construct in textual languages only because of the limitations imposed by textual syntax [Allen 1997]. Parallelism based upon individual instantiations of a recursive problem is permitted in some VPPLs (those that support formal and actual parameters and calls between graphs), though exploitation of such parallelism is left to the developer, much as it would have been in a textual language.

The CODE language provides an explicit form of instantiating data-parallel activity through the use of “arc topology specifications” which allow a node to communicate with a run-time determined number of successors or predecessors, and to identify which nodes are communicating via the index of the arc through which data is being

passed. Being text-based, the developer has little support in deploying data-parallelism and furthermore must bear the burden of cross-referencing graph- and text-level objects. Conversely, task parallelism is implicit within the structure of an application and requires no explicit developer input, other than the requirement that developers express solutions without introducing unnecessary dependencies into the graphs. Pipelines (and iterative behaviour) are achieved through a combination of intricate node interfaces and feedback loops which can be used to trigger multiple executions of a node. Recursion is supported through the graph call node which can be used to call the current graph. Since computational nodes execute in parallel with other computational nodes, recursion is naturally parallel in CODE.

Where CODE uses combinations of textual meta-programming and graphics, HeNCE uses only graphical components to initiate parallel activity. Areas of data-parallel activity are delimited by special purpose nodes which may take values at run-time to determine the level of replication for the contained computations. Pipelined parallelism is also delimited by special purpose nodes, but there is no need to specify any further control metrics, since they are implicit within the structure of the pipeline. Special nodes also supporting sequential iterative behaviour. Task parallelism is, once again, implicit within the structure of the application. As HeNCE does not support sub-graphing (let alone formal and actual parameters) recursion is not possible.

ParaDE takes a similar approach to HeNCE in that parallelism is considered purely graphically. Data parallelism is achieved through the depth actor node, which also takes responsibility for decomposing data into pieces for each parallel instance of the actor to work on, and for recombining the results from those actors into a single result structure. The process of partitioning the data for the parallel actors is specified graphically, whereby the user selects from a number of templates when constructing the application, which are then applied to the partitioning of data at run-time. Whilst this is conceptually no different to the approach taken by the HeNCE and CODE data parallel mechanisms, ParaDE's approach significantly simplifies implementation, and thus eases the developer's job. Pipeline parallel activity is supported through the loop actor (as is straightforward iteration), though it is now recognized that pipelining in ParaDE could lead to non-determinacy. Task parallelism is implicit within the structure of an application, though recursion is not supported since ParaDE did not provide a mechanism whereby a named graph can be called.

Since it supports an explicit message passing, as opposed to flow-based, approach, VPE is somewhat different to the other languages under consideration here. Data parallelism is supported in a manner which resembles elements from both CODE and ParaDE. In VPE, a specific graphical notation is used to denote the potential replication of a computation (like ParaDE's depth actor), yet communication with instances of that computation are identified through indices (like CODE's arc topology specifications). The result is that at the graphical level, the developer can specify a potentially parallel computation, but then must invoke that parallel computation with appropriate instructions at the textual level. However, unlike other languages both task and pipeline parallelism, are implicit within the structure of the application. Recursion is not supported at the graph level (and thus cannot be exploited in parallel), though sub-graphing is.

The two primary means of exploiting parallelism within a HiPPO application are firstly, by parallel method calls on multiple objects (task parallelism), and secondly through the foreach node where multiple instances of the graph associated with the

foreach node are automatically instantiated based on the availability of object handles generated by a user-provided method. Iterative behaviour is viewed as being purely sequential and hence implemented by a special loop node.

4.0 CONCLUSIONS

This taxonomy has attempted to identify and describe the salient characteristics of visual, parallel programming languages. Such a taxonomy cannot hope to capture all of the possible interactions between the features that a VPPL might provide. Nevertheless, it is hoped that future VPPL designers will benefit from the insights that the taxonomy provides into language alternatives.

REFERENCES

- ALLEN, R.S. A graphical system for parallel software development, PhD Thesis, Technical Report 640, Department of Computing Science, University of Newcastle upon Tyne (1997).
- BEGUELIN A. *et al.*, *HeNCE: A User's Guide*,
<http://www.netlib.org/hence/hence-2.0-doc-html/hence-2.0-doc.html>.
- BROWNE, J.C. AND NEWTON, P. The CODE 2.0 graphical parallel programming language, *Proc. ACM Int. Conf. on Supercomputing*, Washington D.C., ACM Press, (July 1992).
- BROWNE, J.C., *et al.*, Visual programming and debugging for parallel computing, *IEEE Parallel and Distributed Technology*, 3, 1 (1995), 75-83.
- BURNETT, M.M. *Visual Language Research Bibliography*,
<http://eecs.oregonstate.edu/~burnett/vpl.html>
- CODE, The CODE system home page, <http://www.cs.utexas.edu/users/code/>.
- GELERNTER, D. Generative communication in Linda, *ACM Transactions on Programming Languages and Systems*, 7, 80-112.
- HiPPO, The HiPPO (*High Performance Parallel Object oriented*) Research Project,
<http://www.parallelism.cs.ncl.ac.uk/projects/hippo/index.html>.
- MYERS, B.A. Taxonomies of visual programming and program visualization, *Journal of Visual Languages and Computing*, 1, 1, 97-123.
- PRICE, B.A., BAECKER, R.A., AND SMALL, I. A principled taxonomy of software visualization, *Journal of Visual Languages and Computing*, 4, 3, 211-266.
- NEWTON, P. AND DONGARRA, J. *Overview of VPE: a visual environment for message-passing parallel programming*, Technical Report UT-CS-94-261, The University of Tennessee, Knoxville (1994).
- TRELEAVEN, P.C., BROWNBRIDGE, D.R. AND HOPKINS, R.P. *Data-driven and demand-driven computer architecture*. *ACM Computing Surveys*, 14, 1, (1982), 93-143.
- WEBBER, J. *Visual object-oriented development of parallel applications*, PhD Thesis, University of Newcastle upon Tyne (2000).
- WEBBER, J. AND LEE, P.A. *Visual Object-Oriented Development of Parallel Applications*, *Journal of Visual Languages and Computing*, 12, 2 (2001), 145-161