PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs

Kenneth M. Kahn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

kahn@parc.xerox.com

Abstract

The design and implementation of a visual programming environment for concurrent constraint programming is described. The system is implemented in Strand, a commercially available concurrent logic programming language. Three components are now operational and are described in detail in this report; they are a parser, fined-grained interpreter, and animator of Pictorial Janus programs. Janus, a concurrent constraint programming language designed to support distributed computation, has much in common with concurrent logic programming languages such as FGHC. The design of a visual syntax for Janus called Pictorial Janus is described in [KS90].

Visual programs can be created using any illustration or CAD tool capable of producing a PostScript description of the drawing. The Pictorial Janus Parser interprets traces of PostScript executions and produces a textual clausal version of the parsed picture which can be converted to Strand and run as an ordinary Strand program.

The parser can also produce input to the Pictorial Janus Interpreter. The interpreter accepts as input a term representing the program clauses and query. This term is annotated with the colors, shapes, fonts, etc. used in the original drawing. It spawns recurrent agents corresponding to each agent (i.e. process or goal), rule (clause), message (term), port (variable), link (equality relation), and channel. These agents interact to do the equivalent of clause reduction. The agents also produce streams of major events (e.g. that some message moved and rescaled to the location and size of some other message). These streams are merged and fed into the Pictorial Janus Animator.

The animator generates a stream of animation frames and associated sounds. The resulting frames can then be printed; more importantly, they can be converted to a raster format and recorded on video tape or animated on a work station. The colors, shape, fonts, line weights, used in the original drawing are preserved so that the animation displays these elements in the same graphical terms as they were conceived and created.

Various lessons were learned in the process of constructing the system, ranging from parallel performance issues, to deadlock, to trade-offs between the use of terms and agents.

1 Introduction

This paper presents a software architecture in which concurrent constraint (or logic) programming plays a predominant role. The structure of this software and the programming techniques used are described, and problems that arose and the resulting redesigns are discussed. This paper is primarily about a large concurrent logic program and the fact that the program is one which supports a programming environment for concurrent constraint programming is unimportant to this paper. The purpose of this paper is rather to relate experiences in writing a large, complex, and somewhat unusual application in a concurrent logic programming language. Much of the discussion centers around difficulties in applying, adapting, and choosing between well-known concurrent logic programming techniques. Other papers are in progress which present the visual programming environment.

The software described is part of a "grand plan" in which parsers, editors, source transformers, visualizers, animators, and debuggers all work together to support a programmer in constructing, maintaining, and understanding concurrent constraint programs in a completely visual manner. This work is driven by the belief that such an environment can have a dramatic impact on the way in which software is developed.

The grand plan is to support whole families of concurrent constraint languages, including the familiar Herbrand family, which includes FGHC [Ued85], Strand [FT89], and Andorra Prolog [HJ90]. We also anticipate supporting constraint systems other than the traditional Herbrand constraints of logic programming. Initially the system is being built to support only a pictorial syntax for Janus [SKL90], a concurrent constraint language designed to address some of the needs of distributed computing. Janus most closely resembles DOC [Hir86], Strand, and FGHC.

An important aspect of the pictorial syntax of Janus is that it is a complete syntax (i.e. anything expressible in textual Janus is expressible in Pictorial Janus) and that the syntax is based upon the *topology* of pictures. For example, a port (i.e. a variable occurrence) is represented by any closed contour which has no other elements of the picture inside. A programmer is free to choose any size, color, shape, etc. for the elements of the program. The syntax of Pictorial Janus is discussed in greater detail in [KS90]. A simple example

program that appends two lists is shown in Figure 1.

Computation is visualized as the reduction of asynchronous agents. The rules for each agent are inside it. If at least one of these rules can match, then the agent reduces. A matching rule expands and its "ask" devices (its head and guard) match the corresponding devices attached to the agent. The rule is then removed, and its body remains connected to the pre-existing configuration by links. These links represent equality relations and are collapsed bringing the ports at each end together.

The matching of an append agent with its recursive rule is shown in Figure 2. The matching rule contour expands to match the contour of the agent. The messages and ports rescale and translate to match the corresponding ports and messages of the agents. In Figure 3 the commitment of a rule is depicted. It shows the agent and the matched elements dissolving away leaving the configuration in the body of the rule connected to the configuration of the computation. Figure 4 shows changes which have no semantic meaning and are performed to tidy up the picture. Links in the configuration establish equality relations between ports and can be shrunk to zero, thereby bringing the equivalent ports together. Newly spawned agents are scaled.

2 Pictorial Janus System Architecture

Figure 5 attempts to capture the essential modules and data of a complete Pictorial Janus programming environment. It depicts the various processing stages which take Pictorial Janus program drawings to either a textual form for ordinary compilation or to animations of its execution.

Source programs are drawings in PostScript. PostScript is well-suited for this because of its ability to describe curves, colors, fonts, etc. in a flexible and general manner. Since PostScript is a common page description language for laser printers, every modern illustration or computer-aided design program is capable of producing a PostScript description of a drawing. This is analogous to the situation in textual programming where the text file for a program can be produced by any text editor. An alternative to PostScript input yet to be explored is a custom structure editor that only allows the construction of syntactically correct pictorial programs and can maintain a semantic representation of the program. Another source of PostScript is from automatic tracing tools such as Streamline from Adobe which converts scanned images of hand-made drawings into PostScript strokes.

The problem of discovering the underlying program from a PostScript description is complicated by the fact that PostScript is a full programming language. This is analogous to the situation in conventional languages with sources which require pre-processing. Such sources are not parsed by a compiler; instead the output of a pre-processor run on those sources is. We handle this by executing the PostScript with an ordinary PostScript interpreter in an environment which redefines the graphical primitives that draw strokes,

```
rule(a57(272,485,308,521),
   append(port(p61(box(273,489,276,492))),
          port(p59(box(276,516,279,519))),
          port(p63(box(309,501,312,504)))),
   [equal(c6(312,504,324,516)
          port(p63(box(309,501,312,504))),
          $(port(p65(box(324,516,327,519))))),
    equal(m55(262,488,277,493),
          port(p61(box(273,489,276,492))),
          [])],
   [equal(l5(280,518,324,518),
          port(p59(box(276,516,279,519))),
          port(p65(box(324,516,327,519))))],
          [])
```

Figure 6: Annotated Janus Parse of the base case of Append

show text, etc. to, instead, print a trace of their calls to a file.

The Pictorial Janus Parser is the module which accepts such traces of calls to PostScript graphical primitives and produces a parse in a format called "annotated Janus". This format captures the parse tree of the program picture and maintains correspondences with the original graphical appearances. These correspondences consist of annotations which give the animator guidance in choosing the appearance, position, and scale of various program elements. They are ignored if the program is simply to be compiled and executed without the production of an animated trace. Annotated Janus is the "lingua franca" of the system. It can be produced by the parser, by a visualizer from textual Janus to Pictorial Janus, by a custom structure editor, or by a program transformation tool. It can be used by visual debuggers, animators, or program transformation tools, or it can be converted to textual Janus for ordinary compilation and execution. Figure 6 contains the annotated Janus for produced by parsing the base case rule of append in Figure 1. (Constants such as "p61" also name PostScript drawing procedures.)

A central component of the system is a fine-grained interpreter for annotated Janus. As it interprets the program it produces a stream of events describing activities for each element of the computation (i.e. each agent, rule, port, channel, message, etc.). The event descriptions include a start and end time. By default, the interpreter performs every reduction as soon as possible. This corresponds to a maximally concurrent scheduler. The scheduler can currently be customized to some extent. It can follow a schedule based upon the trace of real execution on a parallel machine or network.

The third major component is the Pictorial Janus Animator. It accepts the stream of event descriptions from the fine-grained interpreter and some layout control and produces PostScript describing each individual frame. This PostScript can be printed, converted to raster for viewing or video taping, or converted to film.

Other components of the system such as the "visualizer" which converts textual Janus to Pictorial Janus and a spe-

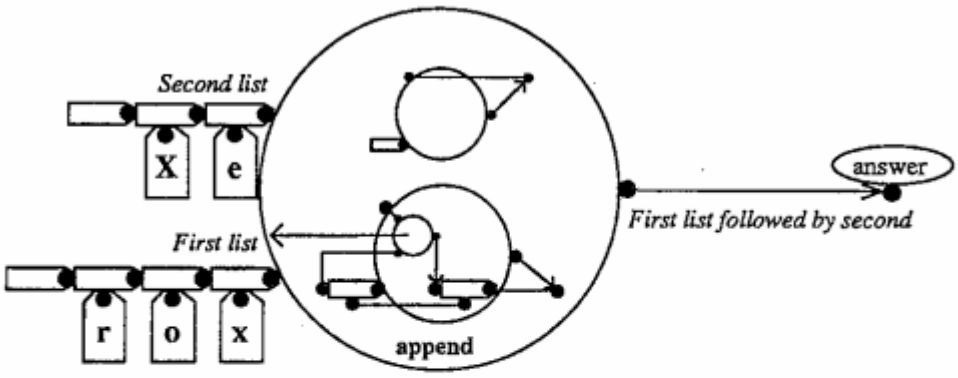

Figure 1: A Simple Example Program to Append Lists

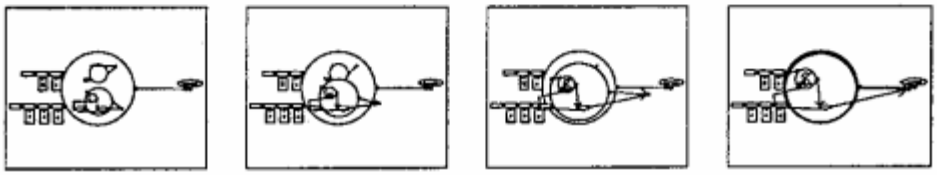

Figure 2: The Animation of a Sucessful Rule Match

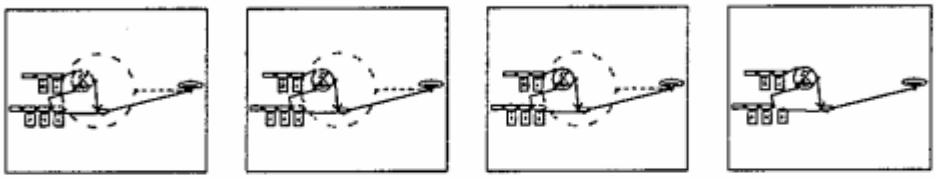

Figure 3: The Animation of a Rule Commitment

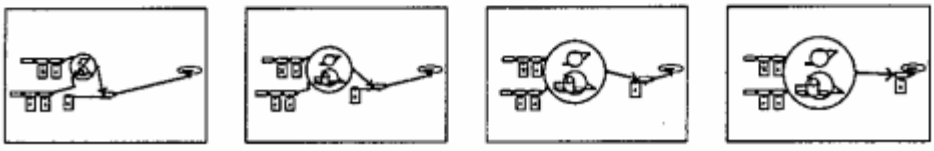

Figure 4: The Animation of Links Shrinking and Agents Rescaling

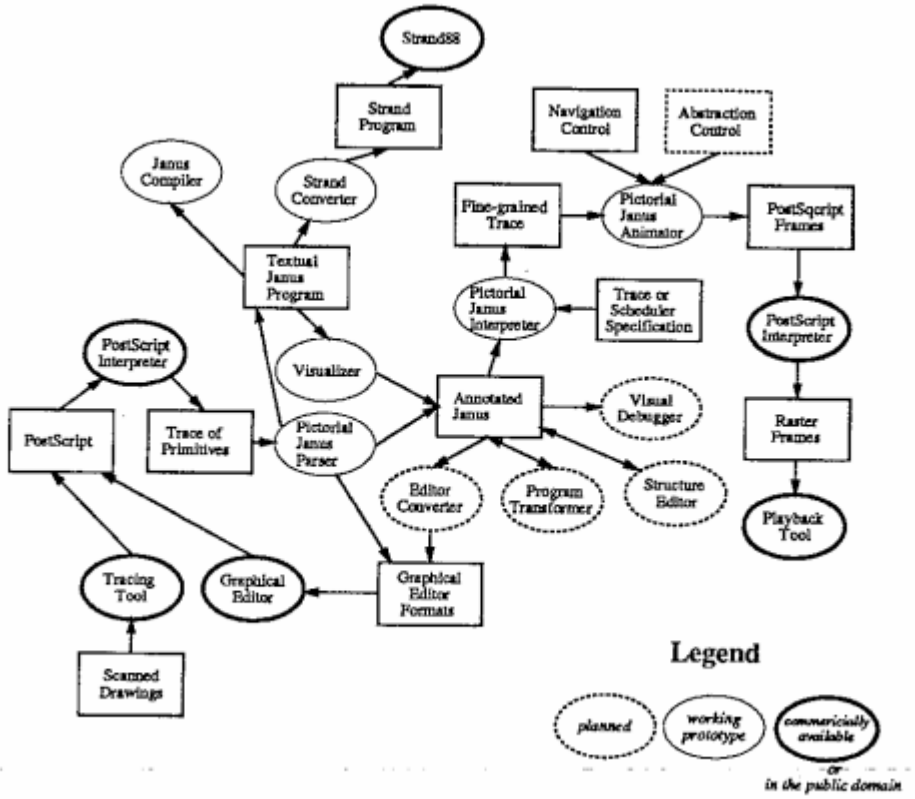

Figure 5: Overall Architecture of the System

cialized pen-based editor for Pictorial Janus are under development and will be discussed in future papers. Additional components such as an interactive visual debugger, a program transformation, and a partial evaluation system based upon Pictorial Janus are only in the planning stages.

The three major components (the parser, interpreter and animator) are operational prototypes and are discussed in further detail in the rest of this paper.

```
ps(stroke,4,
    [moveto(496,483),
     curveto(496,487,485,490,472,490),
     curveto(459,490,448,487,448,483),
     curveto(448,479,459,475,472,475),
     curveto(485,475,496,479,496,483)],
    box(448,475,496,490),
    [eofill,setrgbcolor(1,1,1),setdevicelinewidth(1)]).
```

Figure 7: Sample Trace of PostScript Execution

3 Pictorial Janus Parser

The parser begins with an unordered set of line and curve segments and located text which is the trace from executing a PostScript description of a Pictorial Janus drawing (see Figure 7. These elements are analogous to the "alphabet" of the language. The first phase of the parser is a sort of "tokenizing" where abutting curves are joined, closed curves detected, and arrows recognized. As with most of the phases, some tolerance for sloppy drawings is allowed.

To reduce the complexity of further operations, a containment tree of the elements is constructed. The containment tree associates with each closed curve the sets of closed curves directly contained within, as well as the end points of open curves and arrows and the text directly contained within. Since many parsing decisions depend upon which elements are closest to other elements in the same containment level, the containment tree reduces the amount of search necessary.

The parse proceeds by a series of phases.

- *Identification of Ports.* A port is defined as a closed contour with nothing inside of it. Once the containment tree is completely built this is a trivial test.
- *Classification of Arrows.* Arrows are used for two purposes in Pictorial Janus: channels between ports (i.e. distinguishing between the asker and teller occurences of variables) and the association of agents with rules. Once ports are identified it is easy to distinguish the two cases since channels are arrows between ports while definition arrows connect agents.

- *Attachment of Ports.* Ports cannot be free-standing; they must be attached to either an agent contour, a rule contour, a message contour, or the head of an channel arrow. Essentially the port attachs to the nearest syntactically correct element. This phase also identifies which ports are the internal ports of messages. Messages are identified by having only an internal port and possibly a label inside.

- *Connecting Links.* Open curves depict links which connect ports. This phase determines which port is closest to the end of a link. The connecting port can be up or down one containment level from the level of the link end.

After these phases have completed, the parser can generate Annotated Janus or textual Janus by descending from the top node in the containment tree and collecting information. Agents are distinguished from rules here by alternating containment levels (i.e. the top level contains agents which contains rules which contain agents and so on). Ports of agents, rules and messages are collected into a list by going clockwise from a distinguished port.

Early versions of the parser represented picture elements by terms. Initially, little was known about the terms, so they contained many unbound logic variables for their role, their attached ports, their label, etc.. Lists posed problems since it can't be known beforehand how many elements they have. If tails are left uninstantiated then at the phase where no more elements can be added, some process must find these tails and bind them to the empty list. The lists are constructed in the order the elements were discovered; another logic variable was needed to hold the sorted list. This implementation became more cumbersome and was forced to rely upon some questionable primitives.

Because of these problems, the parser was completely rewritten to represent picture elements by recurrent agents (processes). Lists are no longer a problem since agents can simply recur with a different list. Sorting the list is equally straight forward.

This use of recurrent agents is an object-oriented programming style [ST83]. Unlike traditional object-oriented programming systems, however, the underlying flexibility of concurrent logic programming can be used to incrementally refine the type or class of elements. This might be called "object-oriented recognition". Closed contours, for example, begin as generic "vanilla" nodes. As relationships are discovered, a node may specialize itself to a port or non-port. Figure 8 is a sketch of the code for determining whether a node is a port or not.

Similarly non-ports may be further specialized as messages, rule contours, or agent contours depending upon their local relationships. The locality in this case is between a contour and the elements directly contained within and elements contained in the same contour.

In most object-oriented systems an instance cannot easily (or at all) change to be the instance of another class, even if that class is subclass of the original class.

```
node(In,Contents,State) :-
        In = [identify_ports|In'],
        Contents = [] |
        port(In',State).

node(In,Contents,State) :-
        In = [identify_ports|In'],
        Contents ≠ [] |
        non_port(In',Contents,State).
```

Figure 8: An Example of Incremental Class Refinement

4 Pictorial Janus Interpreter

A detailed trace of every reduction in a computation is needed by the animator. A meta-level interpreter is too coarse for this purpose. Instead a fine-grained interpreter which can report events such as each subterm match is needed.

The fine-grained interpreter of Pictorial Janus is constructed out of recurrent agents. Agents are spawned which represent each element of a program or configuration (programs and configurations are treated identically). There are agents for each rule (clause), port (variable), message (term), link (equality relation), channel (asker and teller pairs) and agent (process). They emulate the ordinary execution at a message-passing level. An agent reduces by spawning an arbiter and sending a message to each of its rules. The rules reduce by sending match messages to each of their ports with streams to the corresponding ports of the agent. If all of the ports respond with a possible match then the rule sends a message to the arbiter. The first rule to send a message to the arbiter then commits and the others eliminate themselves. A committing rule spawns new agent, port, rule, message, channel and link agents.

The agents of the fine-grained interpreter also generate a stream of events. For example, when a rule commits it produces two event descriptions. The first indicates that the rule contour should transform to match the contour of the agent it is reducing. As with all event descriptions, it also indicates the start and stop time for this activity. These times are computed based upon a specification of the scheduler. The second event describes the removal of the rule. All the event streams are merged to produce a time-ordered stream of events.

One problem with the fine-grained interpreter is how it interprets pictorial programs which deadlock. Each rule agent suspends, waiting to hear from its ports how the match went. A port in turn passes the match request to its attached message. The message asks the corresponding port of the reducing agent for a description of its attached message. If there is no message there then the whole collection of rule, message and port agents suspend until a message is

connected to the port. In a deadlocked computation there never will be an attached message. These suspended agents are unable to produce events, which in turn prevents the ordered merge process from producing events; the whole production of the stream of events is cut off.

In Strand it is possible to work around this by relying upon the questionable "idle" guard that suspends until the whole system is idle. The message agents waiting for a response from the corresponding message which are also part of an idle system can then proceed to return a match failure or signal an exception and the interpretation can proceed. It is possible to detect deadlock in a more principled manner [SWKS88], but the price is a significantly more complex and verbose program.

A related problem is controlling the arbiter between competing rule commitments. For example, a merge agent with inputs on both incoming ports can reduce with either rule. Which rule is chosen depends upon which one is the first to get a message to the arbiter of the reduction. Consequently, the fine-grained interpreter selects between competing clauses depending upon the scheduler of the underlying Strand implementation. When run on a single processor this means that the same rule is always chosen. To make more interesting animations a random number generator was needed to remove these biases.

5 Pictorial Janus Animator

The Pictorial Janus Animator consumes the stream of events produced by the fine-grained interpreter. It also can be given layout and viewpoint instructions. It produces a stream of animation frames in PostScript. The animator currently models space as a sequence of ten planes. The graphics of lower planes can be obscured by the graphics of higher planes. The planes are infinite in extent but only a portion is "viewed" at any one time.

The animator accepts event descriptions describing events whose times are described by real numbers. Given a frame rate (i.e. a sampling rate), these are converted to frame numbers. The animator is like a discrete-time simulator where on every "tick" every component needs to compute its next state.

For each kind of event, the animator has methods for depicting it. A typical method might transform one element to gradually match another (currently the transform involves translation, scaling, and rotation). For example, a message matches another message by incrementally changing its position and size until it has the same bounding box as the other message. The other message may be changing and the animator needs to adjust the transformation accordingly. Furthermore, the method must maintain various constraints on the matching message contour so that it remains in contact with other elements. In order for a method to transform an element based upon the position of others, the animator maintains transformation "histories" for each picture element. The history of a visual element is a list of transform matrices, one for each frame. A frame is constructed by selecting from each history the appropriate transformation to apply to the appearance of each element.

The histories are also used to deal with graphical interactions between elements. For example, if a port is to transform itself to match another port which itself is moving, then on each frame the position of the tracking port is a function of where it is, where the other port is, and the amount of time before they meet. An interesting alternative is that it is a function, not of where the other port is on each frame, but where it will be at the time of the meeting. As illustrated in Figure 9, the former corresponds to one port chasing another, while the later is more like a rendezvous. Generally, the rendezvous looks better but it requires "knowledge of the future". With care it is possible to avoid cycles of such requirements of future knowledge that would lead to a deadlock.

The first time the animator was run on a large problem (i.e. one requiring several million reductions), it ran out of memory. Increasing the amount of available memory to 30 or 40 megabytes helped but then it ran out again for somewhat larger tasks. The cause of this kind of problem is very difficult to track down. After much experimentation it was discovered that the problem was that agents inside the animator were producing information faster than other agents were able to consume it. Memory was being used to "buffer" the messages from the producers to the lagging consumers.

This is a well-known problem and there is a well-known concurrent logic programming technique called "bounded buffers" [TF87] for dealing with it. The simple case of a single producer and a single consumer is rare in the animator and a more complex variant was needed to deal with consumers of streams that have multiple producers (typically combined by an ordered merge). This fixed the problem but significantly increased the complexity of the source code. Many subtle bugs cropped up which eventually were traceable to some piece of code not following the bounded-buffer protocol correctly. These were hard to debug because they resulted in deadlocks of thousands of agents.

Another shortcoming of using bounded-buffers is that it is difficult to tune for different language implementations and hardware platforms. Under some schedulers all this complexity is unneeded because the scheduler runs consumers first. To both simplify the code and increase its flexibility, the bounded buffer technique was abandoned and instead each agent was programmed to know the animation frame number it is contributing to and which was the last frame to be completed. Producers are now controlled by an integer indicating how many frames ahead they are allowed to proceed. If this is set to a number larger than the total number of frames in the animation, then buffering is effectively turned off. The use of frame numbers to control producers is easy to generalize to other problem domains such as simulation, but it is not as general as the bounded-buffer technique.

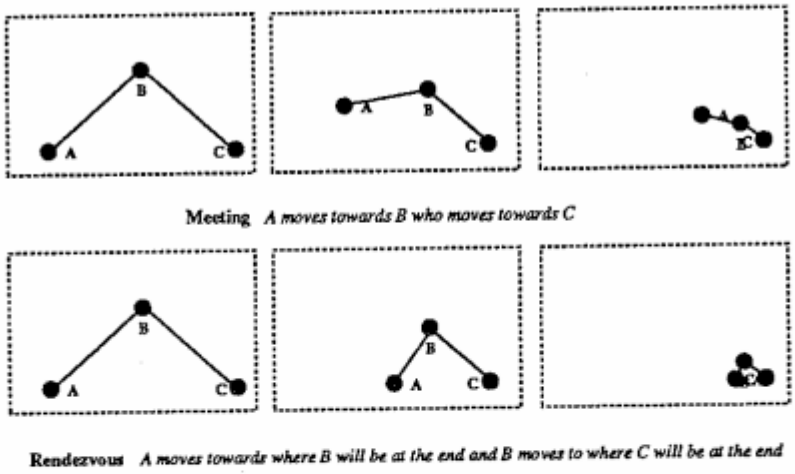

Figure 9: An Illustration of a Chase in Contrast to a Rendezvous

When bounded buffers were first introduced the animator would deadlock sometimes. After a few days of investigation it turned out that the problem was an interaction with the method for having two ports meet. Recall that there are two alternatives: "chase" and "rendezvous". Rendezvous requires knowledge of where the other port will be at the end of the event. It turned out that the system was deadlocking whenever the buffer size was smaller than the number of frames needed to animate a port meeting. Once discovered it was easy to conditionalize the meet method to use the rendezvous style if the buffers are large enough and otherwise use the chase style.

Concurrent logic programs can be written without carefully ordering events since the basic computational mechanism reorders events based upon data dependencies. This greatly simplified the construction of both the fine-grained interpreter and the animator. Each event can be handled independently regardless of whether the data it needs from other concurrent activities has been produced. In a sequential language the programs would have to have been carefully constructed so that, say, the agent contour changes are computed before the dependent changes on their ports.

6 Preliminary Performance Results

The parser, interpreter and animator are implemented in over 11,000 lines of Strand code. The only important component in C is a routine which finds the closest point on a Bezier curve to another point. A typical parse takes a few CPU minutes (on a SUN Sparc 2). The interpreter typically takes a few CPU minutes as well. The animator typically takes tens of CPU minutes (10 to 20 million reductions is not uncommon).

The sequential execution of the system cannot be sped up much by optimizing the Strand code or replacing it with C. For the parser nearly half of its time is spent in the C routine for finding the closest point to a curve. The Strand code of the animator takes a third of the total time to produce an animation; the PostScript rendering to raster takes up the rest.

It would seem that parallel execution should speed up the system significantly. Preliminary results have, however, been disappointing. Possible reasons include:

- *Communication costs.* The coding style used strived for maximal parallelism but little attention was paid to the amount of information passed between agents. On a good shared-memory implementation, this would not be a factor. There are many cases where much of this communication can be programmed away. For example, rather than communicate large shared structures between agents, each processing node could have its private copy and the messages between nodes would just contain tokens referring to elements of these structures. This rewriting has yet to be done. It would also be counter to the dream of concurrent constraint programming (including the special case of concurrent logic programming) that straight-forward high-level portable programs can run efficiently in different environments without major revision. Some rewriting has been done to enable experimentation with parallelism. For example, the output of the animator previously was a large PostScript file and now is a set of files, one for each frame.

- *Agent to Processor mappings.* Experiments to date have used agent-to-processor mapping annotations. While a few different mappings have been tried it is possible that a good one exists but has yet to be discovered. No experiments using a load balancing scheduler have been tried.

Speedups of a factor of 2 to 3 were easily obtained by spawning Unix-level processes to convert PostScript to raster format on separate processors in parallel.

7 Conclusions and Future Work

The building of a large prototype visual programming environment in a concurrent logic programming language was described. The architecture was presented and some experiences and lessons learned were described. These lessons range from the trade-offs between using messages (terms) and recurrent agents, to difficulties with producers getting too far ahead of consumers, to dealing with deadlock.

For sequential executions the overhead of using a concurrent logic programming language was small. For parallel executions on distributed memory machines, speedups are not readily available and appear to require program rewriting and/or very clever distributions of agents and data to processors.

The system is under development. Current plans include extending the animator to deal with both spatial and temporal abstractions. The animator needs to deal better with the layout of elements. The parser needs to be revised to deal robustly with hand-drawn input. Support for primitives and foreign procedure calls are needed. The interpreter needs to be able to accept general scheduler specifications. The animator is currently able to produce a simple sound track synchronized with the animation. The sounds depend upon the kind of activities occurring. This should be extended to differentiate between different elements involved in the activities.

A very challenging direction for future development is to build a "real-time" version of the system that the user can influence as the computation proceeds. This could lead to very powerful debugging tools. It could also be the basis for user interfaces that are simultaneously interactive visual programs. Such a system would need to run on platforms capable of many millions of reductions every second.

8 Acknowledgements

The design of this system benefited from discussions with Vijay Saraswat and Volker Haarslev. I am grateful to Mary Dalrymple, Vijay Saraswat, and Markus Fromherz for comments on earlier drafts of this paper.

References

[FT89] Ian Foster and Stephen Taylor. Strand: A practical parallel programming language. In *Proceedings of the North American Logic Programming Conference*, 1989.

[Hir86] Masahiro Hirata. Programming language doc and its self-description, or, x=x considered harmful. In *3d Conference Proceedings of Japan Society for Software Science and Technology*, pages 69–72, 1986.

[HJ90] Seif Haridi and Sverker Janson. Kernel andorra prolog and its computation model. In *Proceedings of the Seventh International Conference on Logic Programming*, June 1990.

[KS90] Kenneth M. Kahn and Vijay A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the IEEE Visual Language Workshop*, October 1990.

[SKL90] Vijay A. Saraswat, Kenneth Kahn, and Jacob Levy. Janus—A step towards distributed constraint programming. In *Proceedings of the North American Logic Programming Conference*. MIT Press, October 1990.

[ST83] Ehud Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. *New Generation Computing*, 1:25–48, 1983.

[SWKS88] Vijay A. Saraswat, David Weinbaum, Ken Kahn, and Ehud Shapiro. Detecting stable properties of networks in concurrent logic programming languages. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC 88)*, pages 210–222, August 1988.

[TF87] A. Takeuchi and K. Furukawa. *Concurrent Prolog: Collected Papers*, volume I, chapter Bounded Buffer Communication in Concurrent Prolog, pages 464–476. The MIT Press, 1987.

[Ued85] K. Ueda. Guarded Horn Clauses. Technical Report TR-103, ICOT, June 1985.