# Hypertextual Concurrent Control of a Lisp Kernel*

P. David Stotts†
Department of Computer and Information Sciences
University of Florida
Gainesville, FL 32611

Richard Furuta ‡
Department of Computer Science and Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

**Abstract**

Using the Trellis human/computer interaction model as an implementation vehicle, we demonstrate how to use concurrency-supporting hypertext to provide visual displays of the execution flows through a parallel Lisp program. In addition to displays, the hypertext interface allows injection of control flow into an otherwise functional computation, and therefore provides reader control over the order of evaluation of expressions. The resulting system, termed λTrellis, can be thought of as a concurrent control flow browser for composing functional computations, providing a visual implementation of *kernel-control decomposition*. The advantages of λTrellis are ease of exploring program side effects; ease of debugging parallel code; aid in teaching functional languages; and the ability to construct hypertext documents that have parallel execution semantics and flexible browsing behaviors.

**Key words:** functional programming, parallelism, kernel-control decomposition, Lisp, hypertext, execution visualization.

## 1 Introduction

Pratt introduced in 1978 a concept he called *kernel-control decomposition* [9] for separating the program text that defines and executes control paths from the text that performs data state transformations. A more generalized form of this technique is now known as *program slicing* [14, 15, 4, 5]. Kernel-control decomposition can be viewed as a pair of program slices: one on the control variables, and the second the complementary slice. This form of analysis has been used for optimizations and other program transformations.

In this brief report, we illustrate a form of kernel-control decomposition that has evolved in an ongoing visual programming and hypertext project called Trellis. Trellis is based on a parallel automaton (a timed Petri net), and one of its interfaces is a graphical editor for constructing documents. Many other systems have taken a graphical approach to the construction or visualization of parallel computation. Just to name a few, consider Poker [10], Novis [6], and prograph [8, 7]. In a related fashion, the Linda family of languages [2, 1, 3] shares some common design goals with Trellis. Though Linda implementations have not been visual in syntax, they have been based on the idea of superimposing a parallel control notation on an existing kernel computation language such as C.

*This work is based upon work supported by the National Science Foundation under grant numbers IRI–9007746 and IRI–9015439.

†Internet electronic mail address: `pds@cis.ufl.edu`.

‡Internet electronic mail address: `furuta@cs.umd.edu`.

In general, though, Trellis goes beyond other visual parallel languages in that it is intended to support *human-directed* parallel computations. Trellis, then, is inherently a system for specifying human-computer interactions. Since its semantics are those of concurrent computation, Trellis is especially appropriate for collaboration support protocols and other application domains in which multiple users are interacting concurrently within some information structure.

In the following section we briefly explain the Trellis human/computer interaction model and present a specific realization of it called $\lambda$Trellis, which differs from earlier Trellis implementations by containing an embedded Lisp system. In section 4 we discuss one method of using $\lambda$Trellis for visualization of parallelism: expressing parallelized Lisp functions in browsable form to illustrate the parallelizing methods. In section 3 we show a less obvious, but perhaps more interesting, method of visualizing parallelism that we term *hyperprogramming*. Section 5 concludes the report with a discussion of other ways $\lambda$Trellis can be used outside the domain of direct visualization. Due to our past development of Trellis concepts in the hypertext domain, we will use the terms *document* and *hyperprogram* interchangeably throughout the report.

# 2 Trellis and $\lambda$Trellis

The Trellis model has been formally defined in detail elsewhere [12, 13]; we include a brief informal description here to aid the current discussion. Abstractly, a Trellis hypertext (document) is an annotated timed Petri net. The information associated with the places is *document content*. The annotations on the transitions are time limits.[1] The net itself constitutes the *static document structure*. The execution state space of the net defines the *dynamic document structure*.

Browsing the document is analogous to executing the Petri net, except that the normally nondeterministic transition firings are directed with selections made by the readers of a document. During browsing, when a token enters an empty place, the content of that place (if any) is presented for consumption. Any enabled transitions are shown to the readers as selectable links (buttons). If a reader clicks on a button, the associated transition is fired and tokens are moved around.

A Trellis system is a particular realization of the Trellis model with a client-server software architecture. The Petri net model is implemented as an engine that has no visible interface but will respond to remote procedure calls (RPC) for its services. Clients are written to present a visual interface, but they have no inherent behavior; rather, they provide user access to the automaton (Petri net) that defines a behavior. Each document is a separate parallel automaton. Several clients can check in with a document, allowing multiple users to interact within its information structure. The particular semantics of multi-user interaction is defined solely by the behavior of the server automaton. For example, if the server implements a classical Petri net, then each reader will see the same state of the document, and actions by one reader will cause all other readers to have their view altered as well; no direct *interaction* can occur among readers. If, however, the server is a colored Petri net, then the document can differentiate one reader from another by token color, and can provide more sophisticated interactions.

We have implemented $\lambda$Trellis, discussed in this report, from the original $\alpha$Trellis system having a classical (timed) Petri net engine; most of our experimentation in hypertext and browsing semantics has been done with this system. Currently we are building two other variants: $\chi$Trellis and $\sigma$Trellis. These are based on a colored (timed) Petri net engine, and are intended for use in collaboration support systems for astronomy image browsing and software process modeling respectively.

## A Lisp kernel

The $\lambda$Trellis system differs from previous Trellis implementations by having a Lisp susbsystem to provide a rudimentary data state and arithmetic computation kernel. The result is a programming system that has a functional kernel language and a separate parallel control notation. The editing client we have written

[1] Timing is fully explained in [13].

for creating $\lambda$Trellis hyperprograms uses a graphical notation to express the control (Petri net), and normal textual notation to express Lisp fragments.

However, unlike traditional languages, hyperprograms are primarily intended for intensive human/computer interaction, so the computation kernel can only affect the control flow in limited ways, at well-defined places, and in specific circumstances. This restricted arrangement allows better analysis of the possible actions during execution, and is applicable to domains in which control decisions are made outside the computation space. A good example is hypertext, where readers direct control flow in ways that are best modeled as nondeterministic choice among alternatives.

In a $\lambda$Trellis document, each transition can have some sequence of Lisp s-expressions, called an *agent*, associated with it. When a transition is selected to fire, the tokens are advanced in the control net as always, and its Lisp agent (if any) is executed.

We have used an old-style, dynamically scoped Lisp implementation called *xlisp* and adapted it for programmatic invocation rather than terminal invocation. We needed the dynamic scoping primarily to use the Lisp a-list feature as an attribute/value pair mechanism for storing and altering timing (and other) properties of hyperprograms. In addition, using a Lisp system rather than directly implementing simple attribute/value lists provided an arithmetic engine.

### The $\lambda$Trellis implementation

When $\lambda$Trellis is activated, three browsers execute concurrently as interfaces for a document. These browsers provide visual access to different aspects of the net's information structure and behavior. One interface is an editor that shows a graphical representation of the net and its annotations, and allows construction and alteration of the net itself. Another browser is the original content browser that has appeared in earlier version of Trellis. This interface displays the content information (text files) associated with active places (those containing tokens) during net execution. The third interface, the agent browser, has been created specifically for use in $\lambda$Trellis. It is basically a functional copy of the content browser with one difference: instead of displaying the contents of active places, it displays in multiple windows the Lisp agents associated with transitions that are enabled for firing.

Figure 1 shows screen images of the three browsers executing concurrently. The top two interfaces are the browsers; the left one is the place content browser, and the right is the transition agent browser. The bottom interface is the editor. Note that two transitions are enabled, and two Lisp segments are showing in the top-right browser. The one labeled "clock.lsp" is associated with the transition called "clock", and the agent labeled "dwell.lsp" is associated with the transition called "back". The text visible in the top-left browser is the content file associated with the place called "subSys1". The detailed behavior of this document is explained in section 3.

## 3 Concurrent hyperprograms

The most interesting method for using $\lambda$Trellis in visualizing parallelism is to recognize the combination of nets and Lisp as a unique programming notation in itself, a combination of procedural and functional languages in one execution vehicle. Concurrent programs can be designed to take specific advantage of this marriage, and to employ the annotation and information display nature of the Trellis model for visuality. We use the term *hyperprogram* to denote a Trellis structure authored specifically to take advantage of all Trellis features (man/machine interaction, timing, information presentation, parallel control flow, Lisp functions) in providing a highly interactive, hypertextual, browsable computation. In a hyperprogram, the data transformations, if any, are not the primary ends of the computation; in fact, they may not be very significant. In a hyperprogram, the interaction of the reader/user with the parallel control flow and with the information annotations will often be the primary end of a computation. An early form of this concept has been previously presented in detail [13]. In this report, we expand on the earlier concept by discussing the added dimension of a Lisp kernel.

In hyperprogramming the Lisp agents are responsible for, among other things, setting traps and triggers for the control net as they execute. It is a synergistic arrangement[2], in which the control net's execution depends (somewhat) on the behavior of the agents, and in which the agents collective behavior depends (somewhat) on the behavior of the control net. An algorithm designed for hyperprogram realization must take into account not only normal design criteria, but man/machine interactions as well, since these interactions are the main methods of altering control flow.

The visualizing aspect of this form of hyperprogram comes in the emphasis on presentation of information. These programs, inherently concurrent, will be highly visual due to the need to communicate control information to and from the users. The interfaces will tend to provide direct manipulation of the control structure and its annotative information.

## An example: adaptation agents

To illustrate these ideas, consider the situation in hypertext authoring that originally compelled us to include a Lisp kernel in Trellis. *Adaptation* is the automated alteration of a document's characteristics in response to behavior exhibited by users of the document. It involves collecting information during document use, making inferences and decisions based on this information, and creating appropriate physical changes in the documents structure and parameters at appropriate times. One characteristic to adapt is the *behavior* of a document, that is the timing of sequences, the provision of automated help, the representation of collections in parallel vs. sequentially, etc. Another type of adaptation is to alter the *structure* of a document, that is the information relation described by links. In this form of adaptation, sections of a document can be hidden or made visible, or preferred paths identified.

Adaptation agents are a restricted form of script associated with links. Agents are invoked by button clicks, and while executing use Lisp variables for their own purposes. An agent will define its variables, associate them with appropriate net components for storage, and access them as needed for information collection and decision making.

In the current implementation of $\lambda$Trellis, we allow an author to associate an adaptation agent with any transition in the net. This means that information gathering and decision making for adaptation will occur only when a link is actually traversed. However, note that an agent may associate variables with any net component; since these variables are actually stored as global Lisp atoms in the engine process, they are persistent and exist from one execution of an agent to the next.

In the previous $\lambda$Trellis document example, the adaptation of help window popup timing is done with three agents. The initialization of variable values is done with an agent called *init.lsp* associated with a detached (0,0) transition that is enabled when the document is first browsed:

```
(setq ccount 10)
(setq dwell 0)
(setq clock 0)
(setq clicks 0)
```

The counting of events is accomplished with a document clock implemented with a detached self-loop, which increments every five Trellis clock cycles. This agent is called *clock.lsp*:

```
(setq clock (+ clock 5))
```

The main work is done by an agent called *dwell.lsp*, attached to all other transitions that are not auto-timed. It computes the running average dwell time from the document clock, and decides when to reset the values of attributes like *min* and *max*:

```
(setq clicks (+ clicks 1))
(cond ( (eq clicks ccount)
```

[2] As contrasted with classical programs, in which there is only tight coupling between the kernel and the control parts.

```
                (setq clicks 0)
                (setq dwell (/ (+ dwell (/ (- clock oclock) ccount)) 2))
                (setq min (* 3 dwell))
                (setq max (* 6 dwell))
                (setq oclock clock)
        )
    )
```

As the example shows, the goal is to provide simple agents which, using the timing mechanism as described in the next section, provide useful forms of adaptation.

In the editor window of Figure 1 locate the transitions named "help". It is not visible in this screen, but these transitions have the Lisp atoms *min* and *max* associated as their timing attributes. During browsing, whenever a transition becomes enabled, the engine checks to see if variables are bound to its timing attributes. If not, then the times are constants and the author-supplied values are always used. If variable are attached, then the values of the variables are obtained from the Lisp a-list, and these values are substituted as the new timing figures for the transition. Note that though $\lambda$Trellis contains a full Lisp sub-system, it is really used in this example as a convenient way (via the a-list) to get a variable/value structure into a document. We do not consider the agents in this technique to be full-blown Lisp programs. They are assumed to be small enough to execute rapidly, in the time a reader would normally expect link traversal to take during browsing.

## Another example: dining philosophers

As further illustration of concurrent hyperprogramming in Trellis with Lisp, consider thye canonical concurrency example of Dining Philosophers. An implementation of four philosophers is shown in Figure 3. Shown is the initial screen when the document is invoked, with a view giving the names of the various components. Throughout this examplel, the content browser is not important, since no significant content has been associated with places in this simple illustration. Thus the screens show the content browser covered mostly by the agent browser, which is in turn covered partially by the editor.

Figure 4 shows the timing view of the initial state. Notice that the transitions (labeled "grab") that would invoke the eating phase of each philosopher are timed as "0,inf". This means that no delay is required before one may be fired, and that an infinite time must pass before it fires automatically; this actually means it will never fire automatically.

The editor allows agent evaluation to be turned off, and it allows timing to be turned off as well. In such a state, the document can be browsed at will, and the concurrent structure of the philosophers simululated at a reader's speed and following a reader's train of thought. However, with timing and agents on, the adaptation implemented in this document causes a round-robin schedule to take place, using the timing to trigger each philsopher in turn. When the button labeled "init" in the first screen is invoked, the agent labeled "init.lsp" is executed, causing the timings on transitions to chage to the ones shown in Figure 5. Now philosopher one is set to automatically fire after 4 seconds, and all others set for 40 seconds. Thus philosopher one will fire before any others time out.

The adaptation agents shown in these figures indicate what happens when a philosopher enters the eating phase. For example, the agent called "thinking1.lsp" will run when the "grab" button is invoked, moving philosopher one into eating. This agent changes the timings on its own button back to "1,40" like the others, and then changes the timings on philosopher one's "full" button to "2,4". Thus after 4 seconds, philosopher one will give up the forks and return to thinking. The agent "thinking1.lsp" will run when this event happens, causing the timings on the "full" button to revert to "40,40" and the timings on philosopher two's "grab" button to become "2,4". In this way control passes around the net. Figure 6 shows the state of the document after philosopher one has passed to philosopher two, which has then begun eating.

The buttons labeled "slower" and "faster" have agents that will appropriately alter the "speed" variable from which all the timing values are determined. Thus a browser can speed up or retard the execution of the round-robin. Figure 7 shows the timings after two clicks on the "slower" button.

Other scheduling policies can be simululated by writing appropriate agents to adjust the timing triggers as needed. We also repeat that no contents have been given for places to keep the example fairly simple. However, appropriate documents for the actions of philosophers while "eating" and "thinking" can be displayed. In addition, if real action is needed, then the execution of Lisp segments as content of places can be simulated as shown in Figure 8. Here an extra place and an extra transition is inserted into each philosopher between the display stages of "thinking" and "eating". This new place neads no content and no name, as it serves simply to enable the new transition. The transition has the desired Lisp segment (called here "eating.action.lsp") attached as its adaptation agent (though adaptation is not really the goal in this case), and the timing on the transition is set to always be "0,0". When the "grab" button is invoked, the intermediate transition fires instantly (after 0 time units have passed, in effect), running the Lisp action and then passing control on to the display section called "eating". This intermediate action is invisible and unknown to the reader.

# 4 Visible parallel Lisp by extraction

Another method for using λTrellis to visualize parallelism is by generating Trellis structures as by-products of compilation. A simple parallelising Lisp system can be created, for example, by generating appropriate Petri net structures to cause concurrent evaluation of argument expressions. Parallel execution can be simulated then by executing (browsing) the Trellis structure. As tokens are moved around the control flow net, the corresponding Lisp segments (attached to the transitions) are evaluated. As an example of this approach consider the Lisp function calling tree shown in Figure 2. The Petri net structure representing this calling tree effectively passes control down to the bottom nodes, and as control is passed back up, the translation-manufactured Lisp code fragments attached to the transitions are invoked. These evaluate the arguments and compute partial functions, storing the values in the a-list for use higher in the tree as control passes further back to the root. The final click evaluates the top level function using the results of the argument evaluations.

As control passes back up the tree, the Lisp browser displays the text of the Lisp functions that are about to be manually invoked. In our examples, we use the annotations at the places to display the original Lisp functions, those that were altered to save intermediate argument values.

By varying the order in which the concurrent alternatives are invoked in the λTrellis interface, the execution effects of different argument evaluation strategies can be explored. A breadth-first exploration of the calling tree would simulate eager evaluation, whereas a more selective exploration would simulate lazy evaluation.

This method is similar in spirit to our previous experiment in browsing various aspects of CSP programs (like the message flow, for example) [11].

# 5 Conclusions

We have illustrated a visual programming system that operates in the domain of hypertext and hyperprorams. This system contains a parallel control flow structure imposed on a functional computation kernel. As a concurrent control structure is browsed, Lisp segments that are elegible for execution are displayed, along with the primary content information in the document. Some interaction of the Lisp with the control structure is allowed, particularly in the setting and alteration of timing triggers. The resulting hyperprogramming structure allows browsing of the concurrent structure to be done in a train-of-thought manner, or as a self-propelled demonstration. With hyperprograms we show two main forms of visualization: direct programming in Trellis, using its kernel-control separation as a unique visual programming vehicle; and derivitive visualization, in which a compiler or other translator extracts some concurrency property of interest from a source (such as a full Lisp program) and generates a Trellis document to represent the property for browsing.

## Acknowledgements

The authors wish to recognize Greg Drew for his implementation efforts in modifying the xlisp system for inclusion in our Trellis prototype.

## References

[1] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, August 1986.

[2] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1985.

[3] D. Gelernter. Programming for advanced computing. *Scientific American*, 257(4):90–98, October 1987.

[4] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3), July 1989.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[6] C. D. Norton and E. P. Glinert. A visual environment for designing and simulating execution of processor arrays. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 227–232, October 1990. Skokie, IL.

[7] T. Pietrzykowski and S. Matwin. Prograph: A preliminary report. Technical Report Technical Report TR-84-07, University of Ottawa, April 1984.

[8] T. Pietrzykowski, S. Matwin, and T. Muldner. The programming language prograph: Yet another application of graphics. In *Graphics Interface '83*, pages 143–145, May 1983. Edmonton, Alberta.

[9] T. W. Pratt. Program analysis and optimization through kernel-control decomposition. *Acta Informatica*, 9:195–216, 1978.

[10] L. Snyder. Parallel programming and the poker programming environment. *Computer*, 17(7):27–36, July 1984.

[11] P. D. Stotts and R. Furuta. Browsing parallel process networks,. *Journal of Parallel and Distributed Computing*, 9(2):224–235, 1990.

[12] P. David Stotts and Richard Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Transactions on Information Systems*, 7(1):3–29, January 1989.

[13] P. David Stotts and Richard Furuta. Temporal hyperprogramming. *Journal of Visual Languages and Computing*, 1(3):237–253, 1990.

[14] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, July 1982.

[15] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, July 1984.

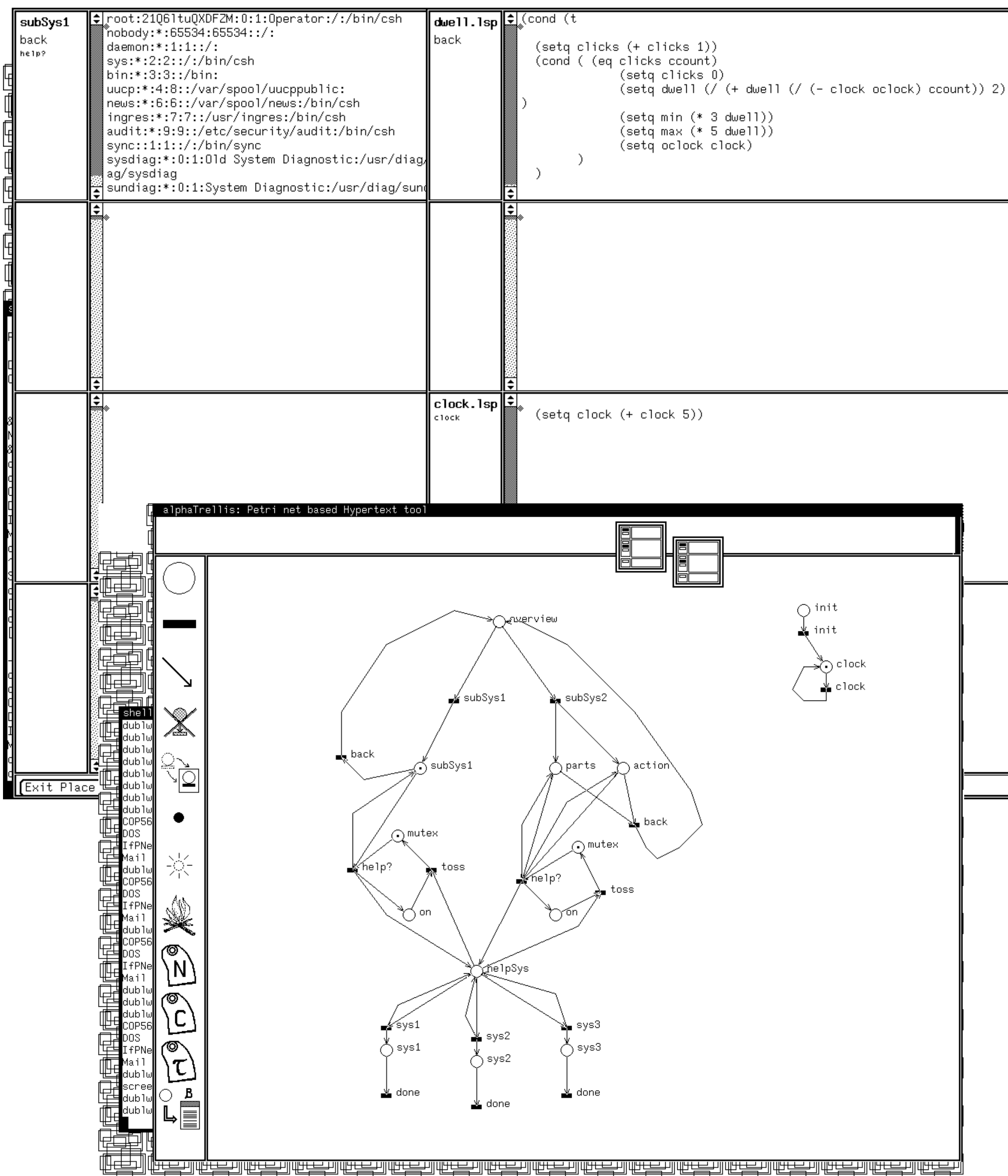
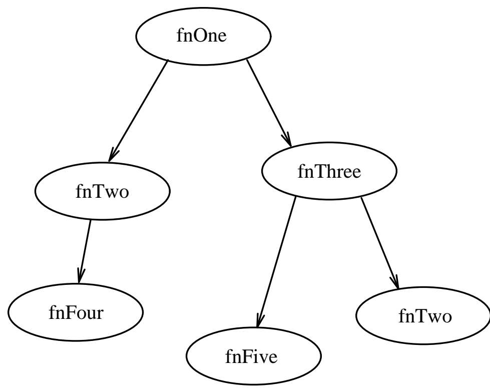



Figure 1: Three $\lambda$Trellis browsers.



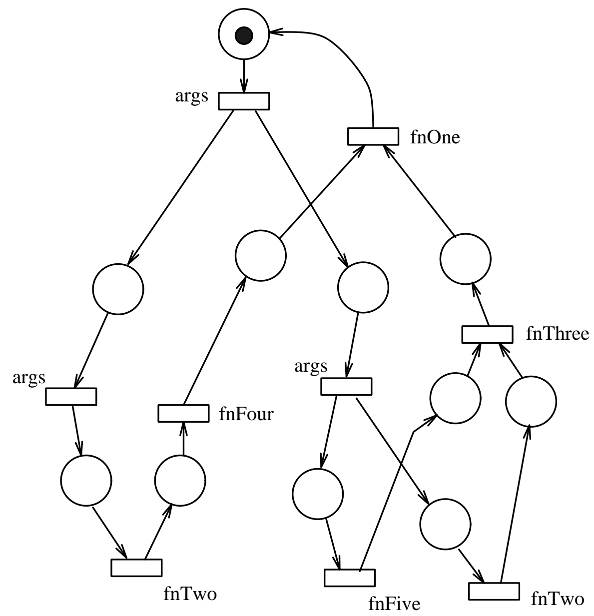




Figure 2: Calling tree for a Lisp function.

Figure 3: Initial Dining Philosophers screen.

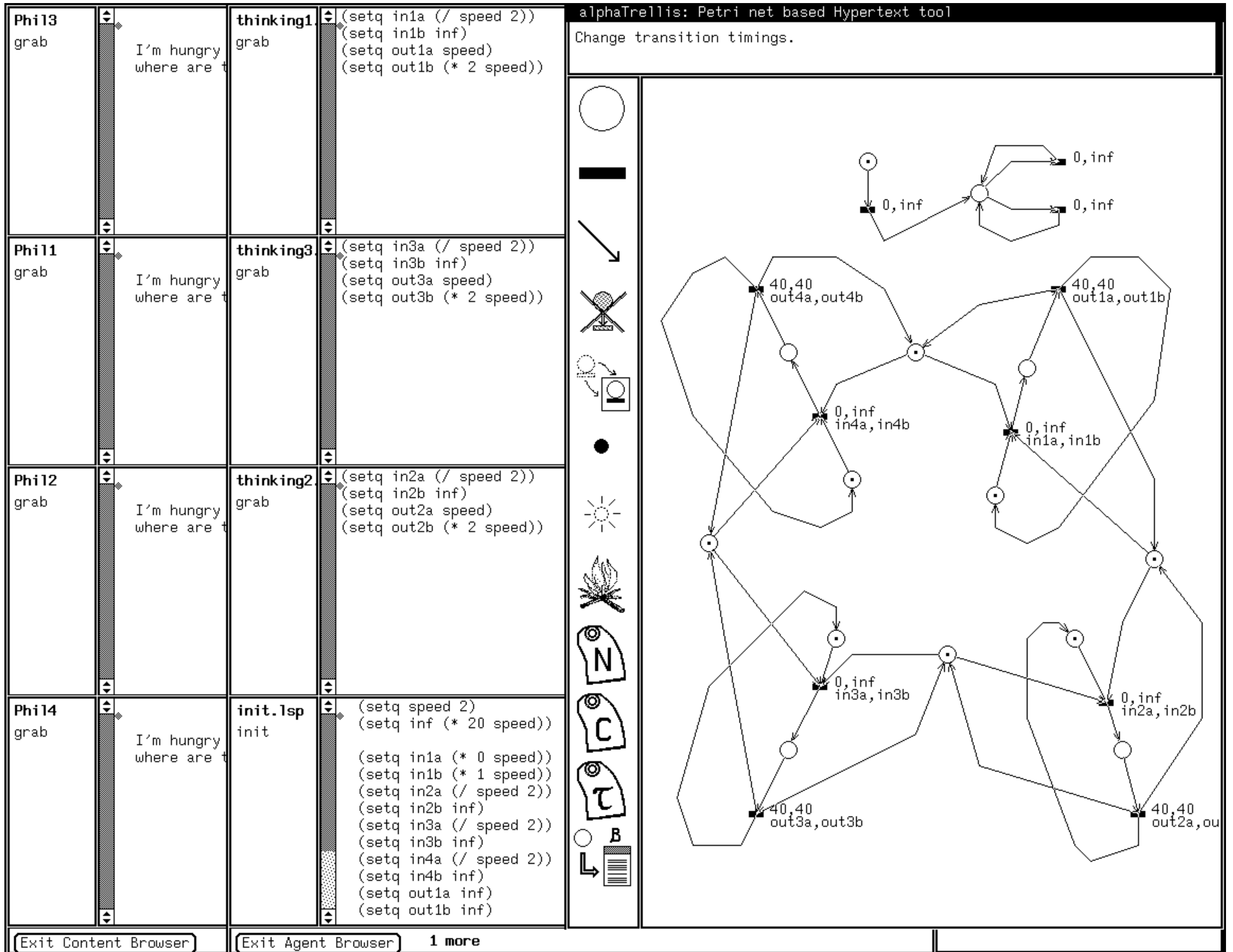



Figure 4: Timings for initial Dining Philosophers.

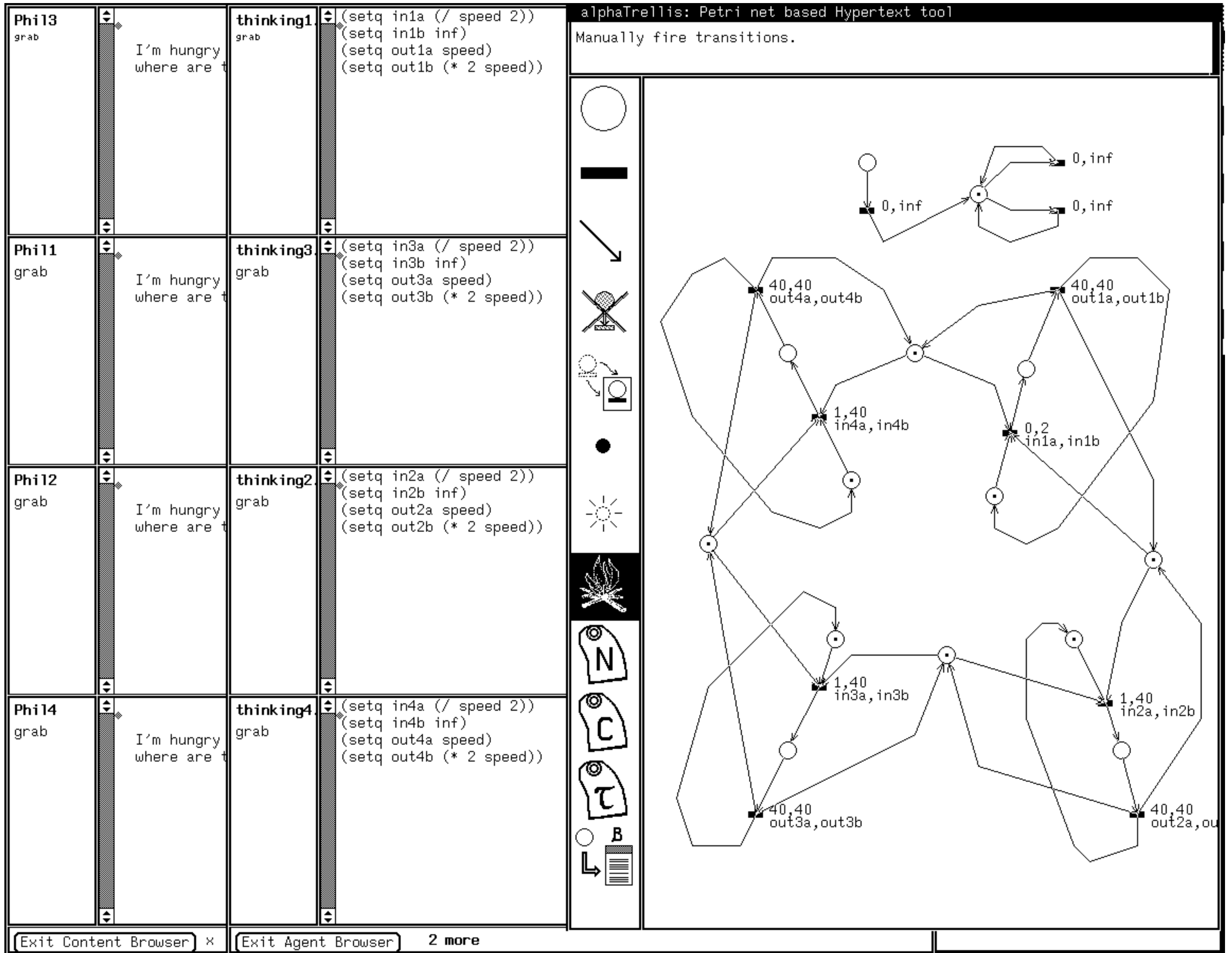



Figure 5: Timings after firing Init button.

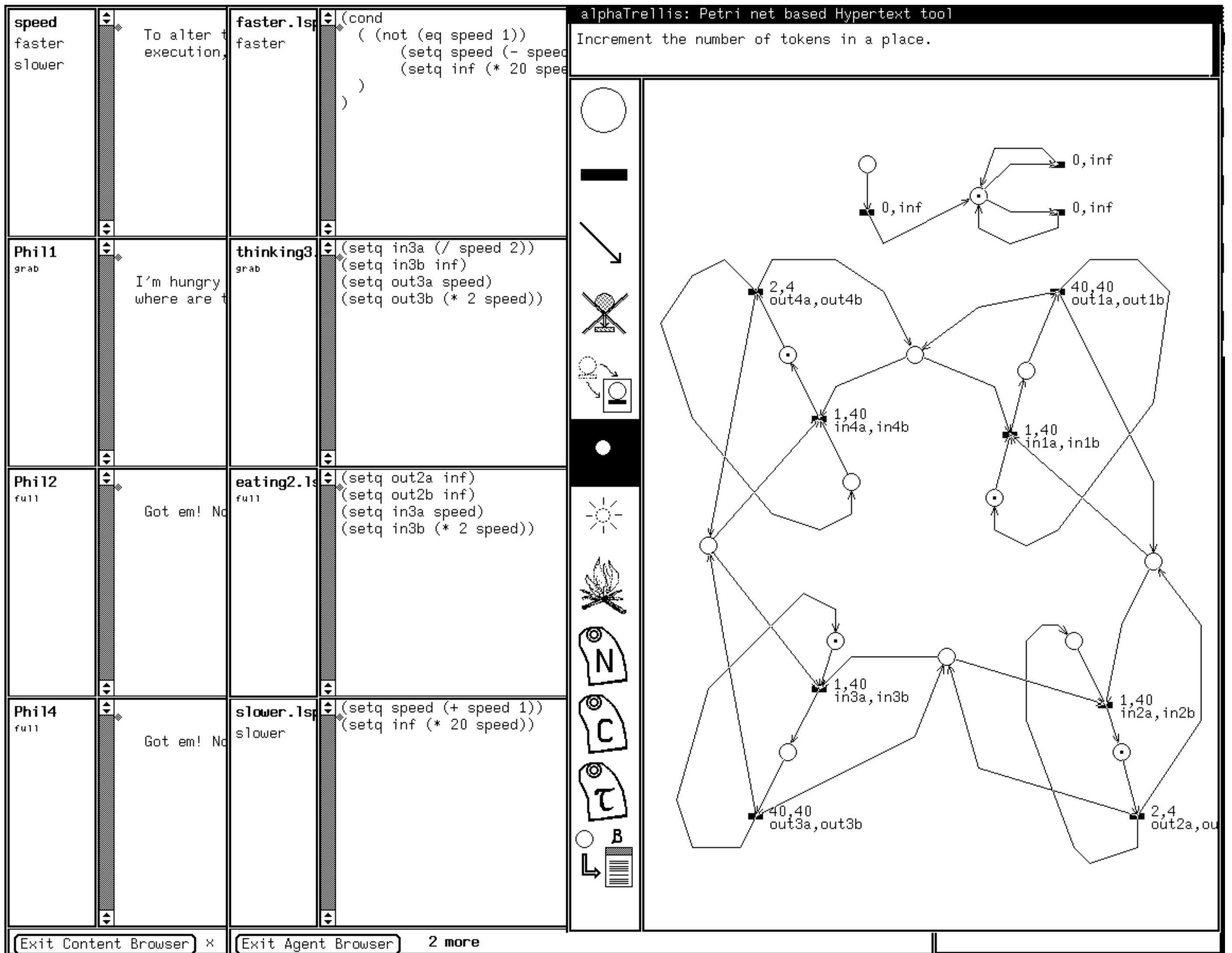



Figure 6: State after timed execution.

Figure 7: State after two clicks on "slower" button.

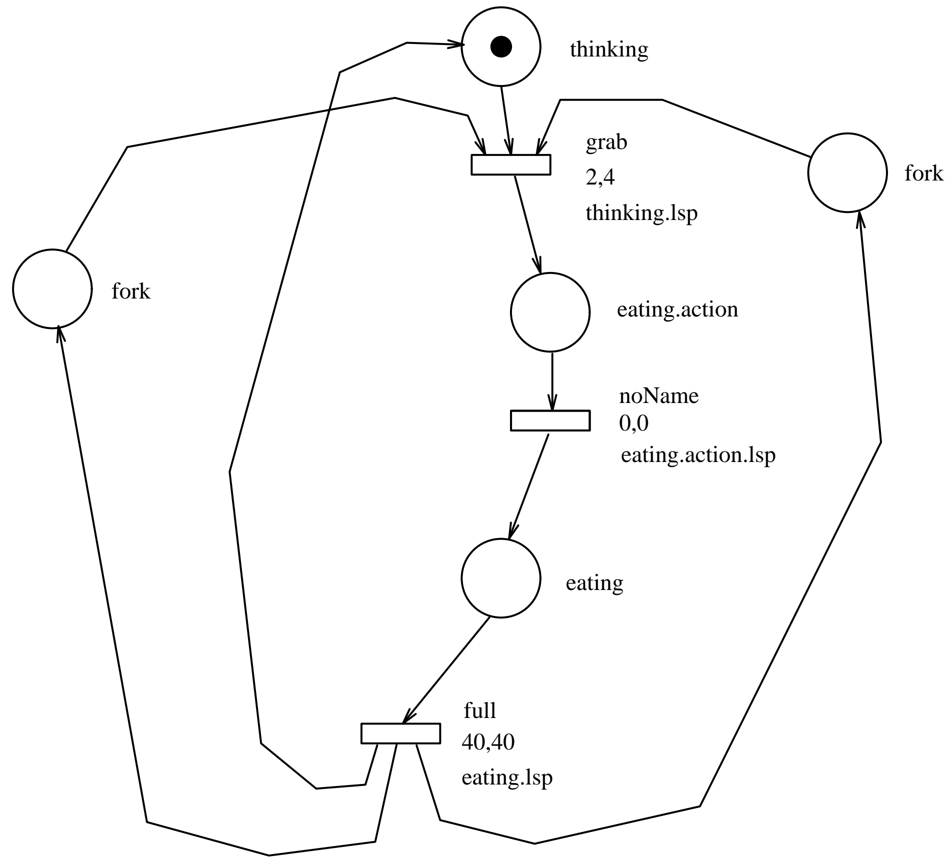



Figure 8: Simulation of Lisp as place content.