Browsing Parallel Process Networks

P. David Stotts* and Richard Furuta†
Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

A hypertext is a non-linearly organized, browsable information structure. The importance of browsing distinguishes hypertext from other network information systems. In this paper we demonstrate the use of the αTrellis hypertext system and its associated parallel browsing semantics for visualization and simulation of the parallel control flow and message network of CSP programs. This is accomplished by automatically translating the text of a CSP program into a Trellis hypertext document. The Trellis model employs the dual nature of Petri nets to formally express in one structure both the linked information elements of a hypertext and the reader/document interactions during browsing. Translation of CSP involves generating a Petri net to represent the parallel threads and synchronization of the concurrent processes, as well as generation of highlighted text fragments for display during browsing.

The advantages of using αTrellis for browsing CSP programs stem from Petri nets being a natural concurrency model. αTrellis contains a palette of net analysis algorithms that can be used to analyze CSP programs for deadlock, trap states, and other properties of interest to parallel programmers. Moreover, this tool is not a special purpose code browser, but rather it is the application of a general net-based hypertext system to an especially appropriate domain. A Trellis-based visualization environment for other parallel languages, like Ada, can similarly be produced by designing an appropriate Petri net representation for its features. Translation into an αTrellis document would then be most effectively implemented as a by-product of program compilation.

Key words: hypertext, visual programming, browsing semantics, Petri nets, Trellis, formal model, parallel programming, concurrent computation, CSP.

CR categories: I.7.m (hypertext), D.1.3 (concurrent programming), F.1.1 (Petri nets)

---

*Supported in part by the USRA Center of Excellence in Space Data and Information Science, and by the University of Maryland Institute for Advanced Computer Studies.

†Supported in part by a grant from the National Science Foundation, CCR–8810312.

1 Program browsing, hypertext, and Trellis

Hypertext is a growing research field that, loosely defined, encompasses study and exploration of the construction (authoring) and use (navigation, browsing) of non-linearly linked information structures. Conklin's recent article [1] will give the interested reader a comprehensive introduction to the concepts and practical applications of current hypertext research. It is within this context of ongoing hypertext research, specifically the authors' Trellis project [10], that a browsing technique for parallel programs has been developed. This report illustrates the technique with a specific visualization environment based on Hoare's CSP language [4, 5]. These browsing ideas can readily be adapted to other languages, though, and easily implemented with only a parser and a Trellis-based hypertext system.

The browsing tool described here presents a hypertextual abstraction of a CSP program, representing the creation and deletion of its parallel processes, the control flow within them, and the message traffic among them. Data states and value transformations are ignored. The reader (the system designer or program user) can simulate execution of the process network by selecting hypertext links to follow, representing transfer of control from one statement to the next, or the synchronized transmission of a message. In this simulation, the reader performs the role of the data state, directing branching and parallelism according to exploratory whims. Multiple windows in the tool provide simultaneous views of the activity within each active process. Within each window the text of a process is shown, with the currently active statement highlighted. A graphical representation of the program network is also available, though in the current tool it requires some reader manipulation after generation to make its appearance neat. Activities are depicted in this graph corresponding to the loci of control in the various active processes.

Hypertextual exploration of a CSP program will help an author (or a user) to visualize the interrelationships among the components of the program, and will help uncover unexpected behavior or problems in the code. The goal of the visualization environment is to augment the reader's intuition about, and static knowledge of, the code by providing an operational feeling for the individual and collective behaviors of the concurrently executing program components. Using the Trellis formalism and the αTrellis system (explained below) to do this provides an added advantage, in that Petri net analysis techniques can be employed with the browser to examine a CSP program in ways other than exploratory.

Our visualization tool extends the concepts of program browsing from the sequential domain into the domain of concurrent computations. Parallel program browsing requires the ability to examine message connections and visualize concurrent control threads. Our project is also unique in that it is not a special-purpose application, but is instead constructed by applying a general net-based hypertext model to parallel programs. As suggested previously, the browsing and analysis facilities developed for the readers of hypertext documents are immediately useful when applied to programs.

In the following sections, we give a brief overview of the Trellis hypertext model [10], and a description of a prototype system called αTrellis [12, 11] with which the CSP browser has been implemented. This discussion assumes that the reader is somewhat familiar with basic Petri net notation and execution semantics. For those who are not, Murata's recent survey [7] provides an excellent overview of the topic; for a deeper treatment of net theory, Reisig's book [9] is comprehensive and thorough.

The Trellis hypertext model

Trellis is a formal model of interactive documents. The model uses a Petri net to capture in one notation the two most important aspects of hypertext: static linked structure, and reader/document interaction during browsing. Being a directed graph, a Petri net expresses with its nodes and arcs the relationships among information elements. Being an automaton, it also expresses with a standard execution semantics the behavior of a document when it is browsed by a reader. Petri nets have the additional utility of being an automaton with inherent concurrency.

A Trellis document has several abstract components. A Petri net represents its linked structure and browsing semantics. A logical information framework is then imposed on the net. An information fragment, or "content element" (text, graphics, executable engine, empty, etc.) is mapped to each place in the Petri net, and a selectable "button" is mapped to each transition. During execution, a content element is displayed for reading or interaction whenever a token is present in the place representing it. A button is displayed for reader selection whenever the transition it is associated with is enabled for firing. Selecting a displayed button (with the mouse) fires the associated transition in the Petri net. This moves tokens around in the net, and so changes the information elements that are displayed for viewing. In this way, a reader browses the Petri net, subject to the behavior the net will allow by its structure and marking (the distribution of tokens over places).

Within this simple framework, complex browsing behaviors can be specified. In the hypertext domain, Trellis is unique with its integral concept of parallel browsing activities, provided by the Petri net automaton. Net structure and execution rules can be used to provide document partitions, access control for classes of readers, multiple readers in one document, multiple threads of exploration for single readers, mutual exclusion and other synchronization of concurrent activities, and hierarchical document composition and browsing. In addition to rich browsing semantics, Trellis' abstract basis allows formal reasoning to be applied to a document. Numerous analysis techniques have been developed for Petri nets. and these solutions readily apply to the hypertext problem domain. Authors (and readers) of Trellis documents are able to detect deadlocks among concurrent activities; to answer questions about node reachability; to arrange screen layout according to the maximum number of windows a document will require; and to perform property-preserving document transformations via net morphisms, to name just a few of the capabilities analysis can provide.

The αTrellis hypertext system

The physical representation and display of information is an aspect of the Trellis model that is separated from the logical aspects of content, interrelationships,and abstract browsing behavior. The model requires the logical form of a document to be filtered through what is essentially a user interface management system to obtain a form consumable by readers. Since the physical "look-and-feel" is the most obvious distinguishing characteristic of a hypertext system, different logical to physical mappings in the Trellis model will give rise to different hypertext prototype systems, i.e., different realizations of the model.

αTrellis is one such realization. It is a document construction and browsing environment based on the Trellis hypertext model, written for the SunView window environment under the Unix operating system. It provides graphical manipulation facilities for Petri nets, and a browsing facility for concurrently displayed windows of text and selectable buttons. Figure 7 shows a screen from αTrellis. It is divided into two main windows, representing its two communicating components. The right

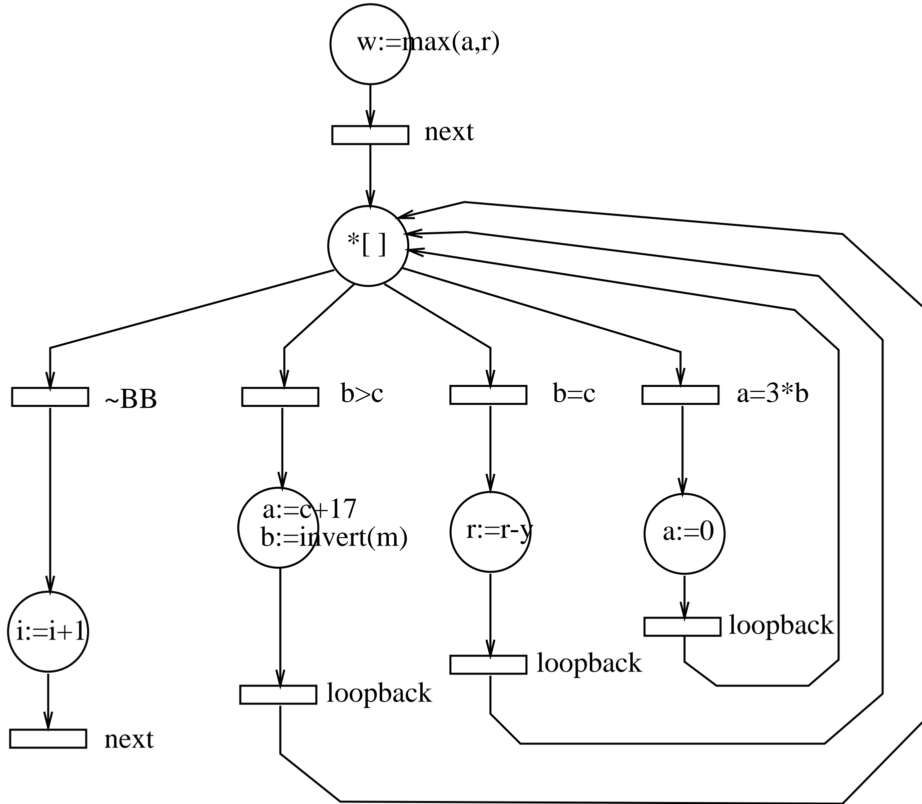

Figure 1: Translation of loop statement.

side is the engine, which allows Petri net construction, and shows graphically the linked document form and the token movement during execution; the left side is the text browser, which displays the information elements and buttons in accordance with the net markings obtained during engine execution. A reader follows a hypertext link by firing an enabled transition. This can be done by clicking the mouse on a transition in the itself, or by clicking on a button displayed in one of the browser windows.

2 Translating CSP to Trellis

Producing the CSP visualization and simulation environment required two main modeling tasks. The first was to adequately represent as Petri nets the portion of CSP semantics we wished to capture for interactive browsing. The second was to map the resulting abstractions into the Trellis hypertext representation. We gathered some modeling ideas from recent papers [2, 3, 8] describing Petri-net-based operational semantics for CSP and CCSP (a combination of Milner's CCS [6] with abstract CSP). These papers deal with complete representations of CSP as a computation model, though, whereas we were not trying to deal with data issues such as variable values, type matches in messages, and the behavior of data structures like arrays. The representation we settled on is designed to capture from a process network its parallel control flow, the message connections, and the synchronization inherent in sending and receiving messages. We wanted to present these aspects of a CSP program for interactive exploration by system designers.

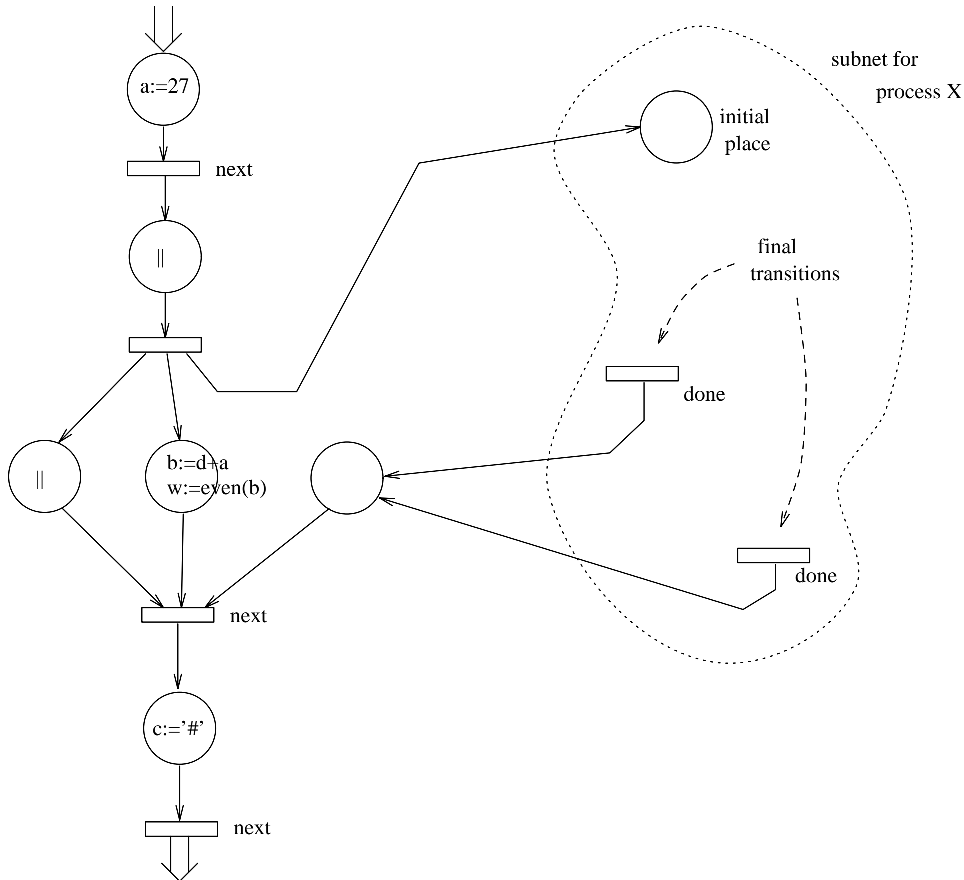

Figure 2: Translation of parallel statement and process invocation.

These goals were met quickly and neatly by designing a translation algorithm that generates an αTrellis hypertext document from a CSP specification. αTrellis is especially well-suited for this application because of its formal basis in Petri net structure and semantics—a natural and well-known concurrency model. Other hypertext systems would be much less successful for constructing parallel code browsers because of the primitive (or completely absent) concept of parallel browsing activities they offer.

Several key associations drive the translation of CSP into αTrellis. First, every CSP guard becomes the label on a net transition, and hence appears as a selectable button during browsing. A reader, by browsing and selecting buttons, causes the display to change (simulating execution of statements) and thereby performs the branching control normally provided by the data state during execution of a program. Secondly, no attempt is made to model data values. We are constructing a control flow model only. When branch points are encountered, the browser allows all branches to be explored, even though during actual execution certain paths may be unattainable due to data constraints. This means that analysis of the net for properties like deadlock will be conservative: if no errors are found, then no errors exist; if errors are identified, then there is a chance that they actually may never arise during program execution with data. Thirdly, every atomic executable statement becomes the displayed contents of a Petri net place (though, to reduce the complexity of the display, we collapse consecutive assignments into a single place). To provide visual context, the

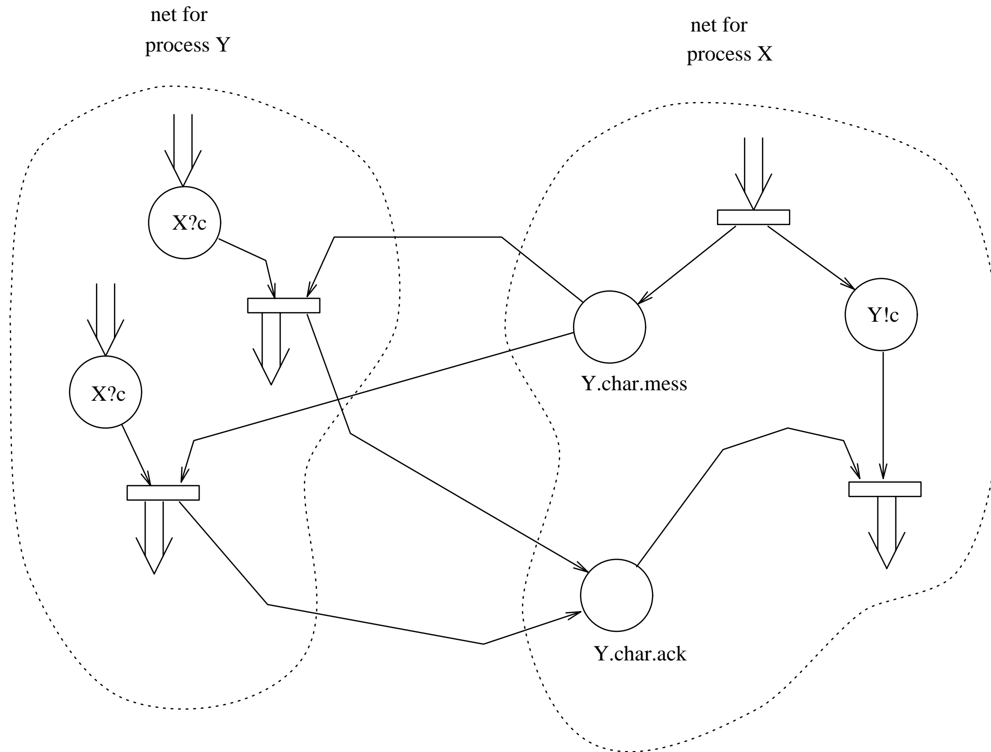

Figure 3: Message structure for interprocess communication.

statement in each place is displayed in the browser as a highlighted region in the text of the process containing it. Lastly, we label each place with the name of the process it appears in.

Given these conventions, we construct a subnet for a CSP process by composing net fragments formed from its various syntactic structures. In general, a net fragment for a statement will have one initial place and one (or more) final transitions. To compose two statements in sequence, the final transitions of the first net fragment are all connected to the initial place of the second. Final transitions of net fragments will in general be labeled with explanatory words like "next" (after assignments and conditionals), "done" (at the end of a process), and "loopback" (return in a loop).

Alternatives and loops

Translation of an alternative statement is done by creating an initial place with as many outgoing transitions as there are alternatives in the statement. The guard on each alternative is the label for the hypertext button that will trigger it. The subnet created for the statements in each alternative is linked after its appropriate "guard" transition. Translation of a loop, then, is straightforward, since its body is a single alternative statement. The semantics of CSP specify that a loop cycles until all guards in its body are `false`, at which point it exits to the statement following the loop. These semantics are modeled by first performing the translation for the alternative statement, and then adding an "extra guard" (called `~BB`) to it, representing the termination condition of all guards being false. The loop subnet is completed by sending an arc from the end of each guarded clause back to the head place. The `~BB` guard is then the outgoing connection for the statement after the

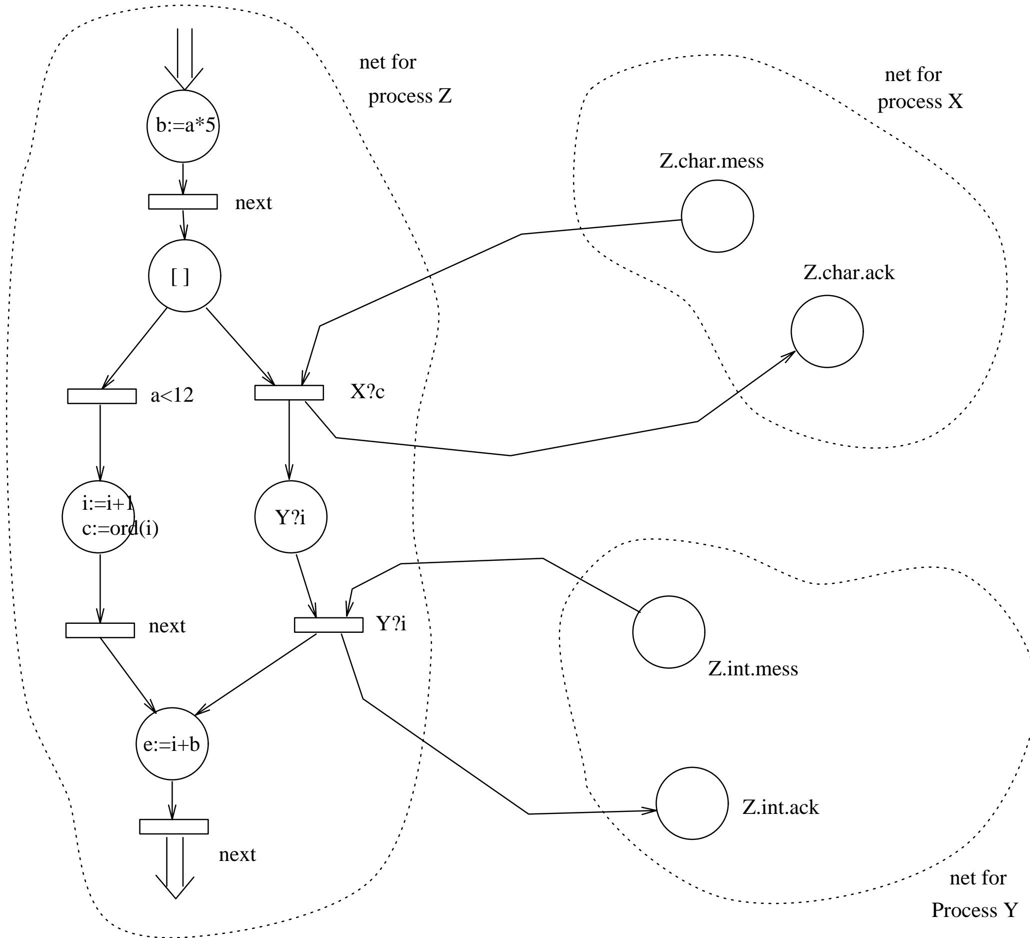

Figure 4: Translation of input guards and input statements.

loop. As an illustration of loop (and alternative) statement translation, consider this CSP fragment:

$w := max(a, r);$

$*$ [

$b > c \;\rightarrow\; a := c + 17;\; b := invert(m)$

▯

$b = c \;\rightarrow\; r := r - y$

▯

$a = 3 * b \;\rightarrow\; a := 0$

]

$i := i + 1;$

Figure 1 shows the Petri net structure and labeling produced from this code. The program text associated with each place represents its display contents. Place names have been omitted to keep the diagram uncluttered.

```
CONWAY
  [ west::DISSASSEMBLE || X::SQUASH || east::ASSEMBLE ]

DISSASSEMBLE
  * [ cardimage:(1..80) character; cardfile?cardimage →
      i:integer; i := 1;
      * [ i ≤ 80 → X!cardimage(i); i:=i+1 ]
      X!space
    ]

SQUASH
  * [ c:character; west?c →
      [ c ≠ asterisk → east!c
      [] c = asterisk → west?c;
           [ c ≠ asterisk → east!asterisk; east!c
           [] c = asterisk → east!uparrow
  ] ] ]

ASSEMBLE
  lineimage:(1..125) character;
  i:integer; i := 1;
  * [ c:character; X?c →
      lineimage(i) := c;
      [ i ≤ 124 → i := i+1
      [] i = 125 → lineprinter!lineimage; i := 1
  ]   ];
  [ i = 1 → skip
  [] i > 1 → * [ i ≤ 125 → lineimage(i) := space; i := i+1 ];
               lineprinter!lineimage
  ]
```

Figure 5: CSP program for Conway's problem.

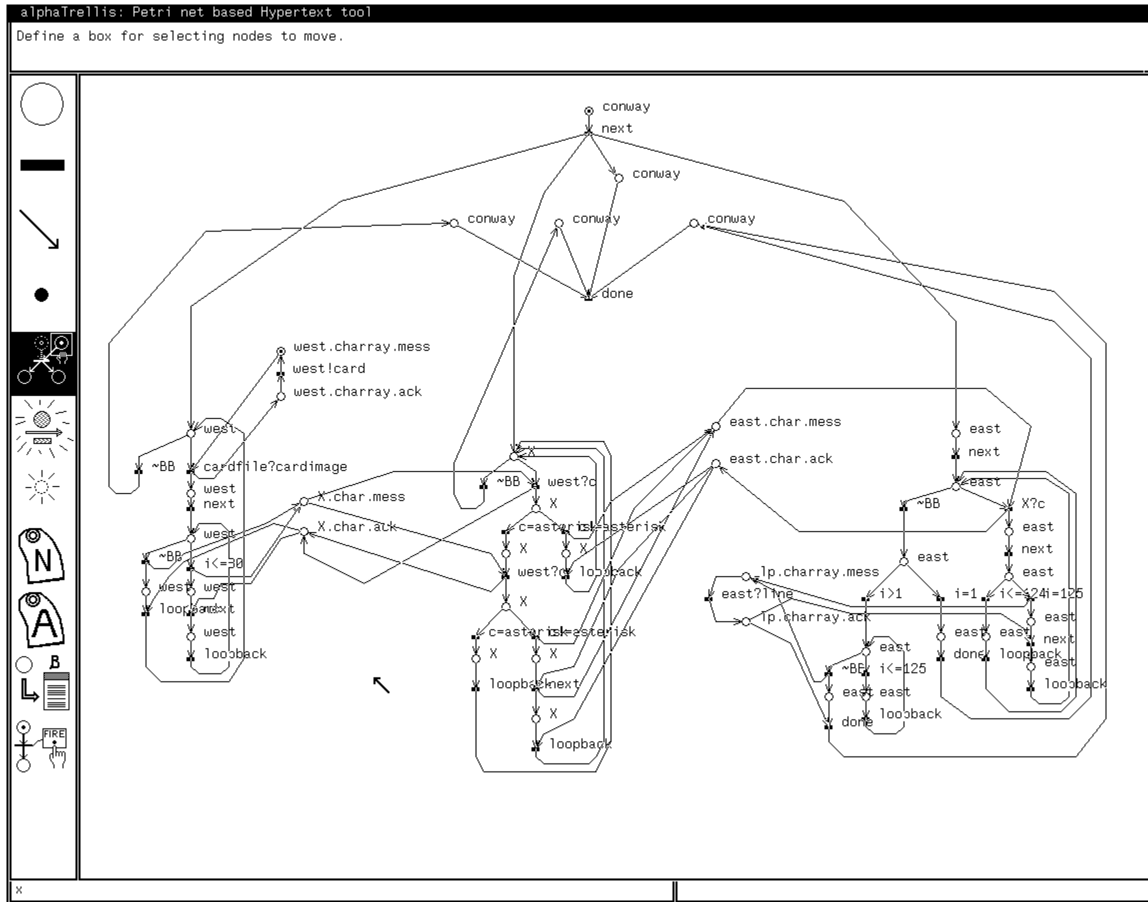

Figure 6: Petri net translation for the Conway program.

Parallel statement and process invocation

Two other translations that require some explanation are those of the parallel statement and process invocation. Consider this CSP fragment:

$a := 27;$

[

$b := d + a;\; w := even(b)$

$\|$

X

]

$c :=$ '#';

The translation of this code is shown in Figure 2. The semantics of the CSP parallel statement require the parent process to suspend until all the concurrent clauses terminate, so the resulting net structure has a synchronizing transition at the end of the substructures representing each clause. For a parallel statement, the translation creates one place to show the parent code with the statement highlighted, (labeled "||" in the figure), and then creates a subnet for each process in the statement. A place with null contents (no visible display) is added for each process to perform synchronization

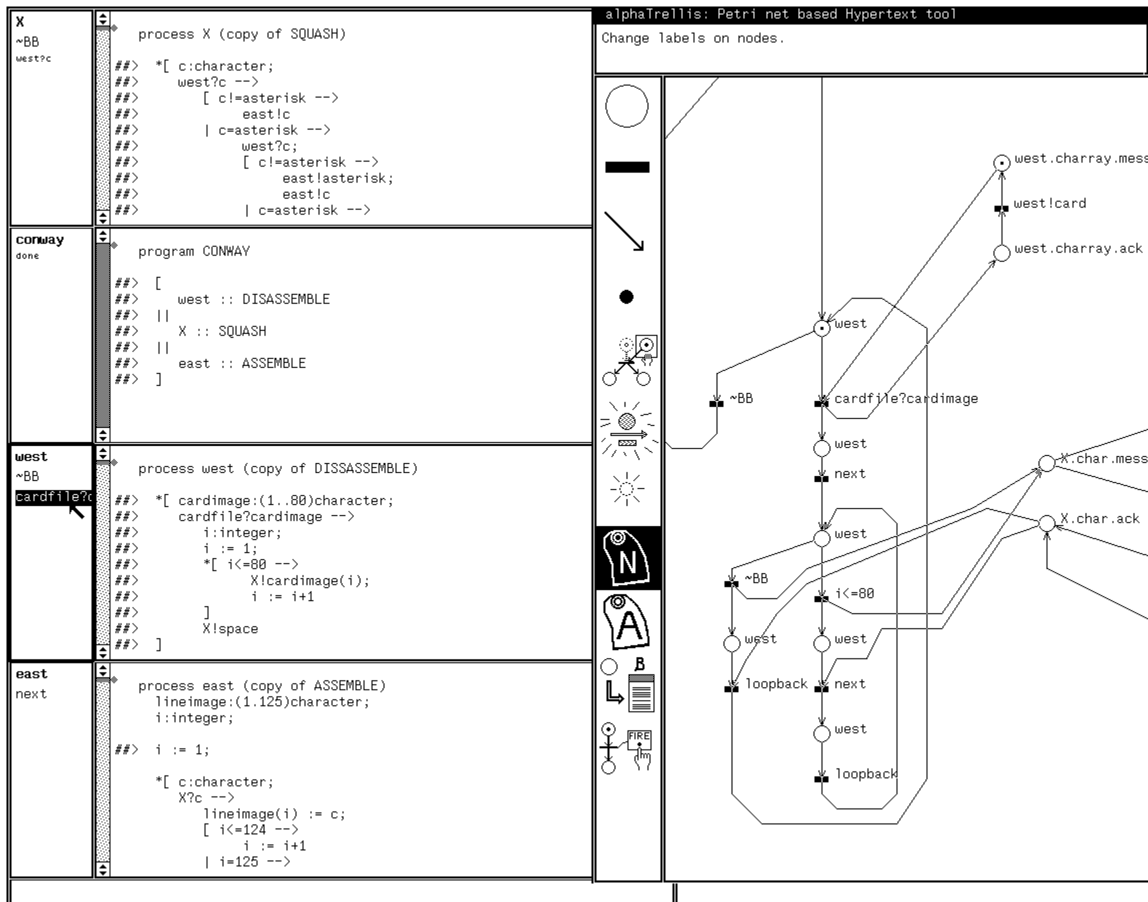

Figure 7: Browser views after three processes are invoked.

upon termination of the call. The final transition for the statement preceding the parallel statement is then connected to the place showing the call, and to the initial place of the subnet generated for each process being invoked. Each final transition (if any) of each process subnet is connected back to its empty place, each of which is connected in turn to the synchronizing transition for the parallel statement.

Message structures and output

Once the simple structures of each process are modeled in this way, the process subnets are interconnected into a message network according to the communication statements they contain. For each output statement in a process, the place representing it is attached via a synchronizing message structure to all potentially corresponding input statements in the named correspondent process. Similarly, each input statement is connected to all potentially matching output statements in the named process.

In the subnet for a single process, one message structure is generated per message type per distinct process named in its output statements. For example, consider process `X` with four output statements: one sends a character to process `Y`, two send integers to process `Y`, and one sends a character to process `Z`. Three message structures are needed in the model for `X`. A message structure is composed of a place to represent a message waiting to be sent, and another place representing a synchronizing "ack." The structure is connected in its own process net in such a way that whenever

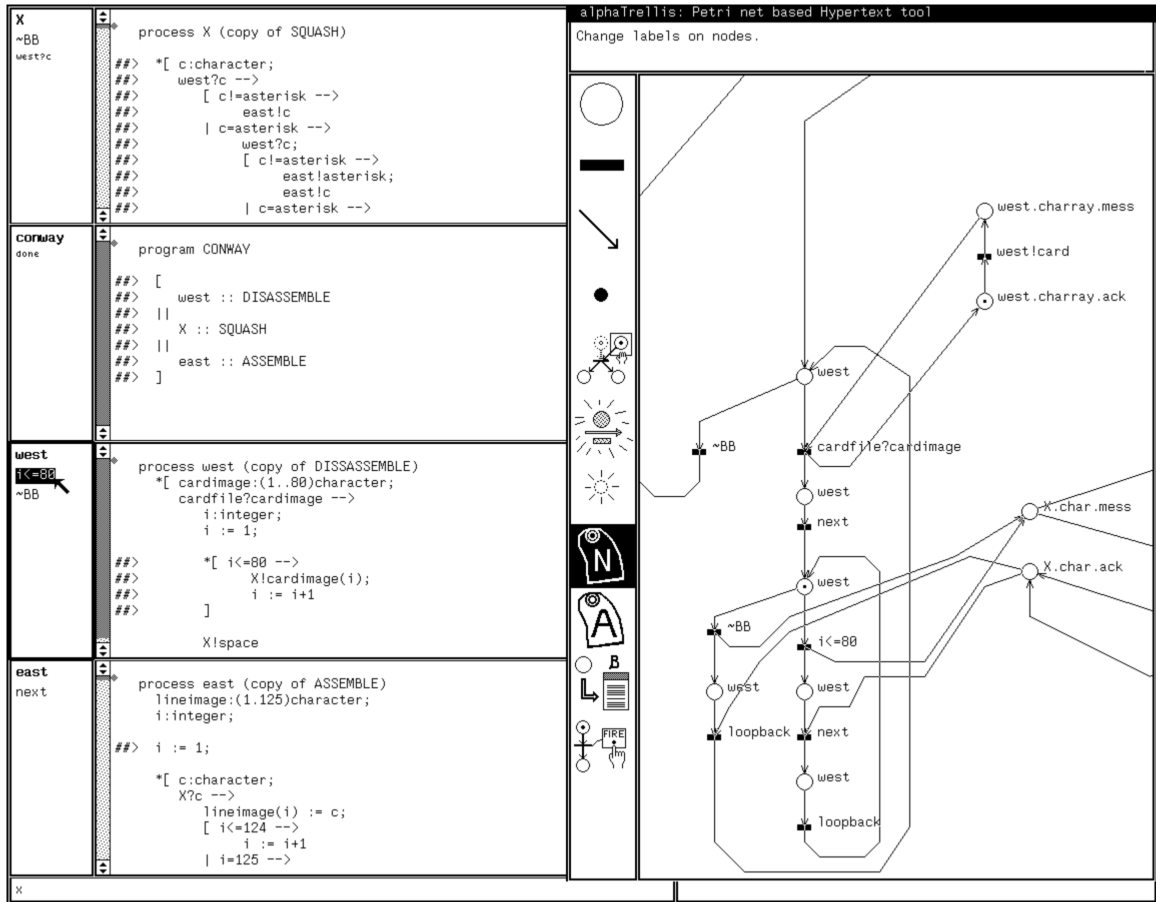

Figure 8: `West` sending a message to `X`.

a transition feeds a token to a place containing an output statement, it also feeds a token into the message place. The "ack" place is connected as a synchronizing input to all transitions following the output statement place. In addition, the message place is connected to the "ack" place via input statement transitions in the communicating process. Figure 3 illustrates a message structure and its interprocess linkage. The display contents of the places in a message structure are set to "null" so they have no visible presence during browsing.

Input, and input guards

One complicating factor in this message translation scheme is that input statements can be guards, which we have previously stated get mapped to transitions (hypertext buttons). Our translation handles this by creating an extra place to represent the input statement, and then following it with a transition labeled by the same statement as a guard. The synchronizing message structure ensures that the guard will never be true (i.e., the transition will never be firable) until a message is received from another process. As a further illustration of translating input statements and input guards, consider this CSP fragment from a process `Z`:

$b := a * 5;$

`[`

$a < 12 \;\rightarrow\; i := i + 1;\; c := ord(i)$

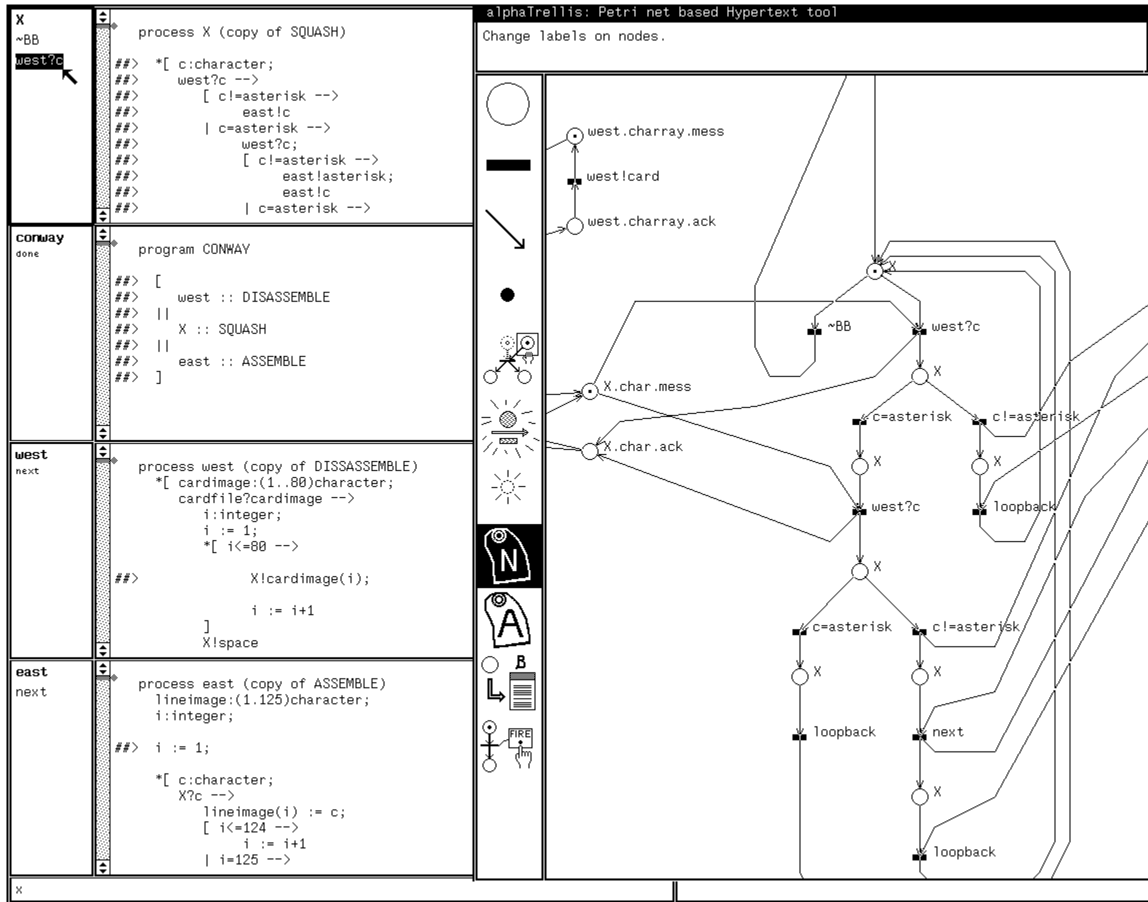

Figure 9: `X` receiving a message from `west`.

$\square$

$X?c \ \rightarrow \ Y?i$

]

$e := i + b;$

Figure 4 shows the Petri net and associated labels generated from this code. Notice especially the difference between the structure for the input guard and the structure for the regular input statement.

To summarize, translation of an entire CSP program proceeds by construction of a single Petri net model of the parallel control flow for the system. First a subnet is produced as described above for each individual process in the program. Next, the subnets for parent and child process pairs are interconnected as described above at invocation points. Finally, the subnets for communicating process pairs are interconnected with message structures at matching input and output statements. In addition to the program net, content elements are associated with each place for display during browsing. Most are drawn from the program text, though some are synthesized to make the hypertext self explanatory. The examples in the following section more clearly show this aspect of the translation.

Figure 10: X sending a message to east.

3 Example: A browsable CSP program

The sample program we use to illustrate parallel program browsing is taken from Hoare's original paper on CSP [4]. It is there referred to as "Conway's Problem." The program invokes three process in parallel: one process (west) reads a character stream in card image format, inserting a space at the end of each card; another process (X) replaces pairs of consecutive asterisks with a single up-arrow; and a third process (east) outputs the character stream in 125 column format. The text of this program is given in Figure 5. We use Hoare's notational convenience of naming the statement lists constituting the definition of each process, to emphasize the components of the main parallel statement.

Figure 6 is a screen from αTrellis showing the entire Petri net produced from this CSP program. The browser half of αTrellis has been occluded in this figure, to show more of the net. Notice that the external process "cardfile" in the program has no definition, so the entire process is modeled in the Petri net as simply an appropriate message structure with an extra transition connecting the "ack" place back to the "mess" place. In the figure, this structure is labeled west.charray.mess, west!card, and west.charray.ack, and it is linked into the input statement for process west. One way to use this structure is to have browsing start with a token in the "ack" place. A message can then be simulated by the reader whenever needed by firing the connecting transition, causing the token to move into the "mess" place and enabling the communication in west. In our implementation, we instead use a timing feature of αTrellis to make the transition west!card fire automatically when-

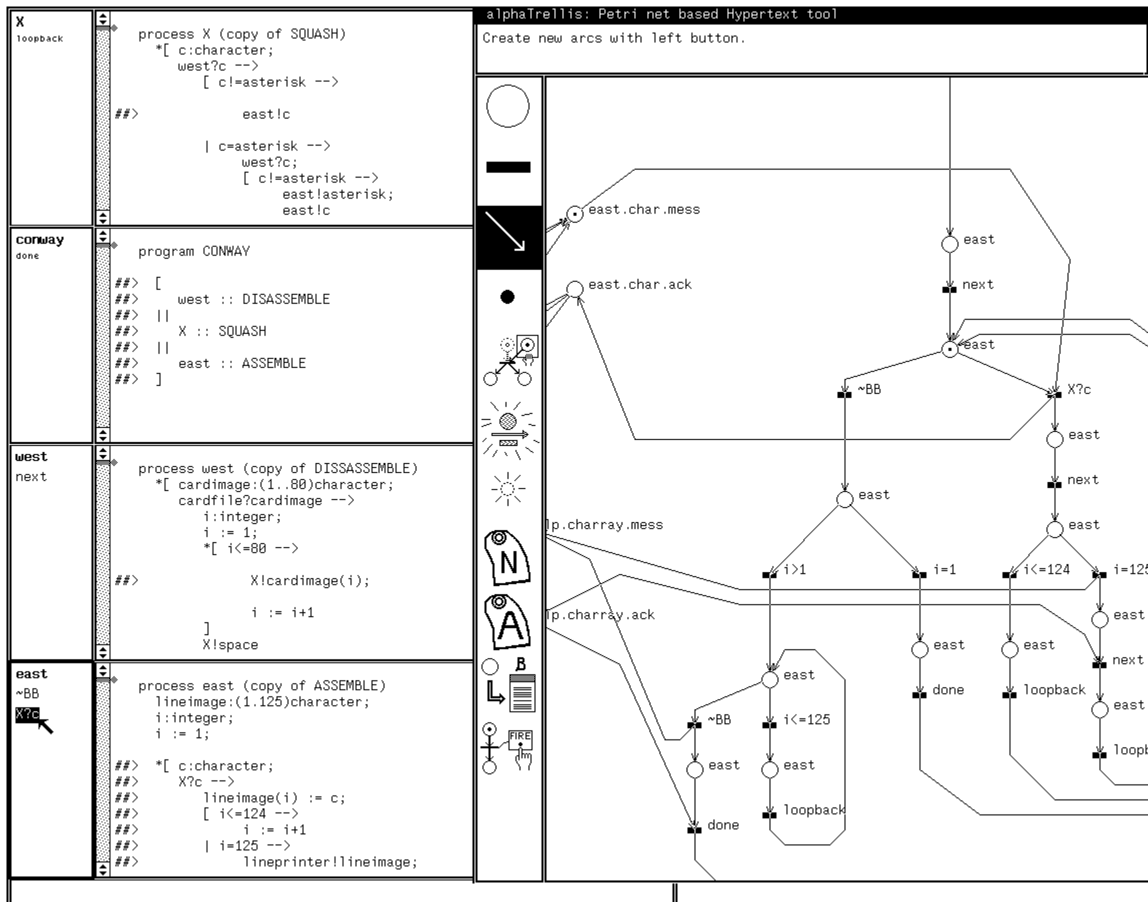

Figure 11: `East` receiving a message from `X`.

ever it is enabled, so the reader does not have to be involved in simulation of the cardreader process. Similarly, the external process "lineprinter" is represented as a single transition (labeled `east?line`) connecting the "mess" and "ack" places in the corresponding message structure of process `east`.

Several notes about special use of αTrellis features are in order before explaining the example. First, the timed events just mentioned were especially useful in freeing the reader from having to explicitly simulate external processes, although such simulation can certainly be allowed if it is desired or appropriate. Secondly, in our translation, all the net places in message structures (and, indeed, all synchronization places) are given "null" contents, a special case in which αTrellis causes nothing to be displayed when a token is present. This reduces the visual clutter that would result if instead a note like "message waiting" were displayed in a window, and tends to keep the number of concurrent windows down to something on the order of the number of active processes. Finally, a limitation of SunView and Unix allows only four concurrent windows to be open in this tool. If a CSP program calls for more than four, a note is displayed at the bottom of the browser saying how many other windows are "invisible." Clicking the mouse on the name of a window will rotate it out to the invisible area, and bring into view one of the occluded windows. Thus a reader can selectively replace views.

The six screen images in this section should convey some sense of a CSP browsing session in αTrellis. Figure 7 shows the browser and net windows after the reader has invoked the three processes in the parallel statement in `conway`. The engine window in this view has been focused

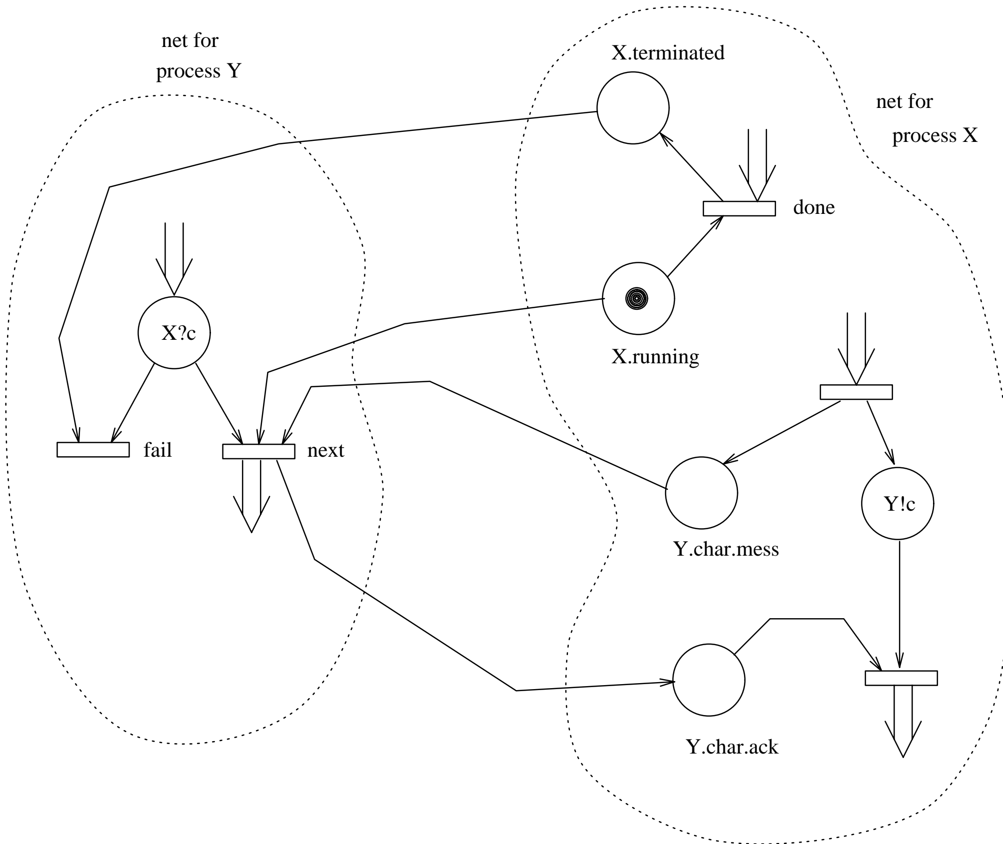

Figure 12: Augmented message structure for termination detection.

on the details of process `west`. This browser configuration was obtained by firing the transition labeled `next` shown enabled in Figure 6. The parent context is shown in window `conway`, and each active process has it own display window. The active statement in each code window is indicated with the highlight markings “`##>`” in the left margin. The small button `done` in the `conway` window indicates a transition that will eventually become selectable, but is not now so. The same holds for the button `west?c` in the `X` window, which is interpreted as `X` waiting for a message from `west`. The screen shows the button `cardfile?cardimage` being selected in the `west` window, getting a message from the `cardreader` process. After firing this transition, and after browsing through the `i:=1` statement in `west`, the screen shown in Figure 8 results. Here, a choice is being made, to follow the `i<=80` alternative and so send a message to process `X`. Notice that in this figure, the token in the `cardreader` message “ack” place has not yet been advanced back to the “mess” place, but in the next screen that event will have occurred.

Selection of the `i<=80` button leads to the screen shown in Figure 9 (the engine window has been shifted to show details of process `X`). Process `west` is now waiting at its communication with `X`, and so the `next` button is small and not yet selectable. In the `X` window, the `west?c` button has become selectable, and doing so as indicated leads to the screen shown in Figure 10. `X` has received the message and deposited a token in the “ack” place, thereby causing the `next` button in window `west` to be enlarged (i.e., selectable). The figure shows the `c!=asterisk` button in process `X` being selected, causing a message to be sent to process `east`. After browsing through the first

statement in east the screen shown in Figure 11 is obtained (the engine window has been shifted to show the details of process east). Notice that a token has been deposited in the message structure for east, causing the X?c button to become selectable. Firing that transition will consume the message, and browsing may continue among the three processes. Eventually the reader would select the BB button on the loops, leading to termination of the individual processes, and enabling the done transition shown back in Figure 6 at the end of the subnet for conway.

4 Discussion and conclusions

The example program illustrates several instances in which the behavior of the browser does not fully represent CSP semantics. This is due to our use of a simplified message network model, which was chosen to simplify the browsing process for the benefit of the program reader. For instance, consider the case where process X checks for a message from process west before west generates a message. Finding no message, X will terminate. Then, when west does produce a message, X is not active to communicate with it. The semantics of CSP say that a communication fails if the other communicant has terminated, so no deadlock will actually happen during program execution (though no computation will occur either). The net we use as a model, though, will indicate a deadlock at such a point when analyzed. This is because we have not attempted in the current prototype to include any detection of process termination in the message structure.

Termination semantics can be added to the model with only slightly more structure. Two places can be created to go with each process subnet: running and terminated. Place running is marked when a process begins, and that token is transferred to place terminated when the process finishes. Any communication statement with another a process will then require two transitions: the normal one (perhaps labeled next), and one called fail. If the running place is marked for the partner in a communication, then the next button will be active and the fail button not. If the terminated place is marked, this situation is reversed and only the fail button would be selectable during browsing. Figure 12 shows this augmented communication structure.

A potential limitation of this system comes from its graphical net representation. Although this is useful, an unconfusing visual layout of the net requires user input in the current tool. Automated layout requires complex algorithms, though it is possible in this context because of the highly structured nature of the Petri nets that represent code. The browser can be operated alone, without the Petri net being visible, and we have found that this mode of exploration still provides a good amount of insight into the concurrent behavior of a program.

In conclusion, we make the following observations.

- We have obtained a useful visualization tool for parallel control flow and communication not by special purpose coding, as with many code browsers, but by application of a general purpose, net-based hypertext system.

- CSP was chosen for this particular browser because it is a simple parallel language and is easily used for demonstrations. For practical applications, the CSP-based browsing tool is almost directly usable in an Occam environment, say on a transputer network, since the net model would be essentially the same for both languages.

- The commonly discussed extensions to CSP, like output guards and message buffers, are easily accommodated in the Petri net model. Adaptation of this tool to these language modifications

would be quick and simple.

- Control flow browsers for other parallel languages, like Ada, can be produced by following the same recipe: generate a Petri net description of the parallel flow and communication synchronization, and design a translation. Realization can then most easily be done as a by-product of compilation.

- An advantage of using αTrellis over other hypertext systems is that the Trellis formalism is based on the Petri net, a natural concurrency model. Thus, the net analysis algorithms in αTrellis (like calculation of the reachability graph), can be immediately applied by programmers to examine the concurrency properties of CSP programs. Note also that the uniqueness of αTrellis in providing this visualization tool for CSP lies not in Petri nets *per se*, but in its foundation on a general concurrent computation model. Other hypertext systems are far less useful for implementation of parallel program browsers because of their lack of any notion of, or foundation in, concurrency.

References

[1] Jeff Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, September 1987.

[2] Pierpaolo Degano, Robert Gorrieri, and Sergio Marchetti. An excercise in concurrency: A CSP process as a condition/event system. In G. Rozenberg, editor, *Lecture Notes in Computer Science 340: Advances in Petri Nets 1988*, pages 85–105. Springer–Verlag, 1988.

[3] Ursula Goltz and Wolfgang Reisig. CSP programs as nets with individual tokens. In G. Rozenberg, editor, *Lecture Notes in Computer Science 188: Advances in Petri Nets 1984*, pages 169–196. Springer–Verlag, 1984.

[4] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Addison-Wesley, 1986.

[6] Robin Milner. *A Calculus for Communicating Systems.* Springer-Verlag, 1980.

[7] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[8] Ernst-Rudiger Olderog. Operational Petri net semantics for CCSP. In G. Rozenberg, editor, *Lecture Notes in Computer Science 266: Advances in Petri Nets 1987*, pages 196–223. Springer–Verlag, 1987.

[9] Wolfgang Reisig. *Petri Nets: An Introduction.* Springer-Verlag, 1985.

[10] 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.

[11] P. David Stotts and Richard Furuta. αTrellis: A system for writing and browsing Petri-net-based hypertext. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 312–328, June 1989. Bonn, W. Germany.

[12] P. David Stotts and Richard Furuta. Temporal hyperprogramming. *Journal of Visual Languages and Computing*, 1990. To appear.