

CHAPTER 4

CSPN TOOL OVERVIEW AND IMPLEMENTATION DETAILS

And if you don't give up and you don't give in you may just be OK. from "In the living years."

–Mike Rutheford

4.1 CSPN tool overview

The CSP-to-Petri net (CSPN) tool is textual based. The initial specification and parameterization work must be completed using a text editor (see Figures 15, 16, 38, 42, and 43) for examples of P-CSP specifications). Viewing the Petri net's distribution of places and transitions as a graph after a translation is accomplished by setting the "-d" (for *dot*) on the command line.¹ Other command line options are described in Table 4.

The translation rules described in Chapter 3 and enumerated in the Appendix A are codified in the CSPN tool (CSP-to-Stochastic Petri Net). In brief, the mechanism consists of decomposing individual CSP constructions into canonical Petri net structures. The elemental Petri net structures are linked together in a hierarchical fashion according to their adjacency and nesting within the CSP specification. Once CSPN has created this network of linked structures it traverses the net and expands the process descriptions which are represented as sub-Petri nets into larger and larger nets. Also, as CSPN decomposes the CSP constructions, it identifies and records service and failure rate annotations which are embedded in the P-CSP specification. When CSPN encounters failure annotations (and the "-f" command line option is set), it creates supplemental failure transitions with a failure rate as designated in the annotation. When CSPN encounters service rate annotations it will assign those values to the

¹Version 1.0 of CSPN does not automatically invoke the dot program to create the postscript graphic file. To do so use the command: `>> dot -Tps filename.dot > filename.ps`. Dot is available from AT&T Bell Laboratories.

appropriate (timed) transition in the resultant SPNP specification. All of the values assigned from annotations are subject to change if the user so chooses during an interactive CSPN run.

Once the preliminary structure of the Petri net is complete, CSPN must reconcile synchronization points because all CSP input/output actions rendezvous at a particular point. This point is a transition that is named by the message being sent and received. Finally, CSPN generates the Petri net graphic specification and the SPNP Petri net specification file "<file>_snp.c." All of these activities occur at various levels of user controllable interaction as will be described.

4.2 Translation phases of the CSPN tool

There are four basic activities (parts) involved in the context of Figure 26. The first part (1) involves specification. The second part (2-7) involves running CSPN which invokes any of the available command line options (see Table 3). See Appendix C for the Composition Phase 4 algorithms. The third part (8-10) is interacting with CSPN to direct how the SPNP analysis is run (setting the SPNP run parameters) and to parameterize the elements of the translation (e.g., assign rates and probabilities to the resultant transitions). The fourth and last phase (11-12) concerns the structural and stochastic analysis of the Petri net.

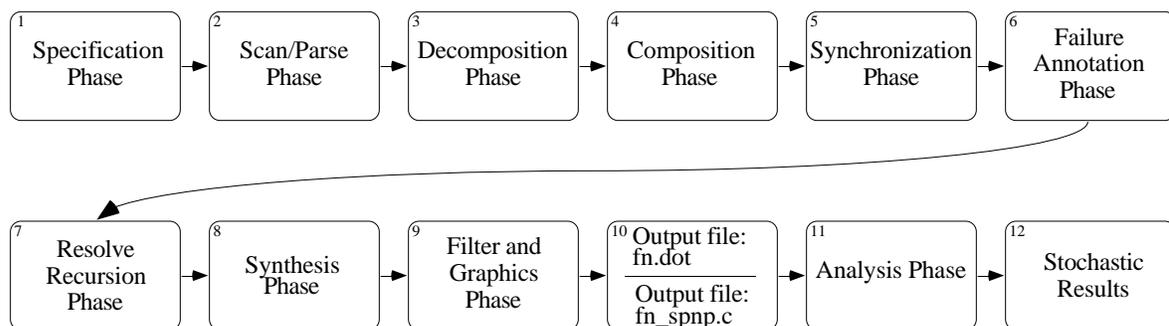


Figure 26. Activities associated with the translation phases of the CSPN tool.

Structural analysis involves viewing the distribution of places and transitions of the

graphical representation of the Petri net.² The *stochastic analysis* involves running SPNP to derive dependability and performance results based on the work from phase three (i.e., parameterizing the model) and relating the results to the graph and back to the original specification. The SPNP specification file may be edited to finely tune specific values of the parameters or other characteristics of the SPNP specification prior to running the analysis.³ Once SPNP is run, the results can be considered in the process of conducting further analysis.

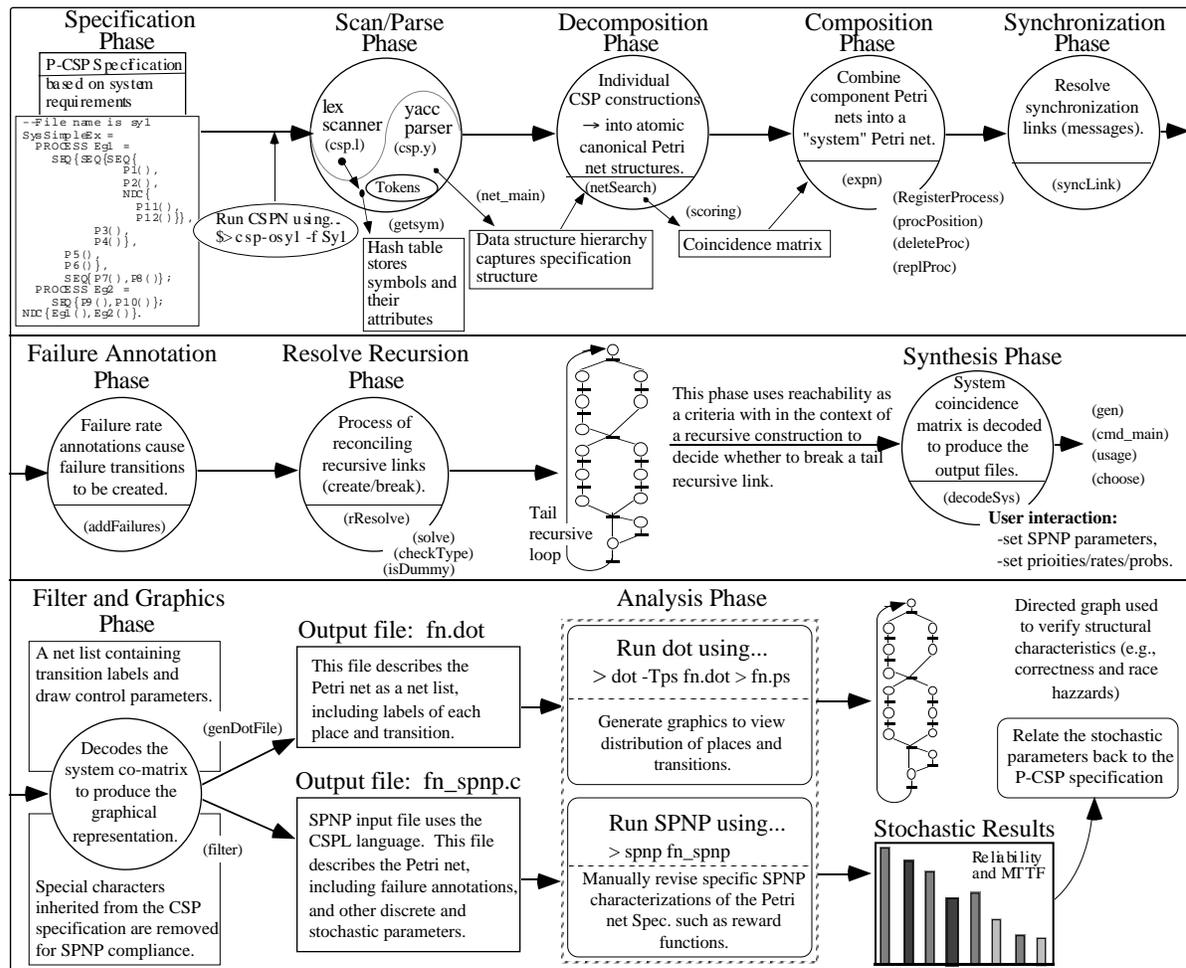


Figure 27. Context diagram and translation phases of the CSPN tool.

²This option causes CSPN to generate a *fn.dot* file which is processed to provide the graphical representation of the Petri net (embedded postscript). Dot is a tool used to create the Petri net graphic. The CSPN version 1.0 does not automatically invoke the dot program to create the postscript file. To do so, the user must manually run dot using the following command: `>> dot -Tps filename.dot > filename.ps`.

³The SPNP specification file can be run for a simple analysis without manual intervention.

In viewing Figure 27, note that the following eight steps occur during the translation process: (1) *Scanning and Parsing* –action rules embedded in the parser enable CSPN to capture the structural semantics of the specification, (2) *Decomposition* –allocating or scoring a coincidence matrix for each CSP element and the recording of any annotated service rates and probabilities, (3) *Composition* –combining elemental coincidence matrices and building their requisite process lists, (4) *Synchronization* –resolution or combining of message links, (5) *Failure annotations* –if active, an appropriately annotated process is augmented with a failure transition, (6) *Resolving recursion*, (7) *Synthesis phase* –takes the system coincidence matrix and creates the SPNP Petri net specification file during an *interactive* session with the user, and (8) *Filter* –removes special characters inherited from the CSP specification that are not valid in an SPNP specification and *graphics* –creates a digraph specification net list that is later compiled using “dot” to produce an embedded postscript graphic. In general, Figure 27 shows the various translation phases and the use of SPNP as it applies to this approach. The names in parenthesis are the *C-function name(s)* and are associated with a given phase. The CSPN tool is used in the context of the seven steps listed in Table 3.

TABLE 3

GENERAL STEPS FOR USING THE CSPN TOOL

Step	Description of steps in the approach
1.	Abstract the critical elements of the requirement specification and formulate a CSP specification for the system under study.
2.	Translate between CSP and Stochastic Petri nets.
3.	Assign performance and reliability parameters among subsystem components.
4.	Analyze the Petri nets for stochastic properties [using SPNP] (validate performance and reliability goals using stochastic system models).
5.	Decide what features of the system should be changed to improve the system's reliability (and/or other stochastic properties, e.g., performance).
6.	Augmentation: relate stochastic properties back to top level (CSP) specifications (e.g., failure rates, service rates, error handling).
7.	Understand the effect these non-functional requirements have on <i>cost</i> .

4.3 Running the CSPN tool

Running CSPN (i.e., $\$ > csp <options> specification-file$) and using the various command line options described in Table 4 enables the numerous features and functionalities. For example, if the user is in the process of correcting the syntax of the CSP specification then it would not be necessary to specify any of these options, only the input file. Also, if the user just wants to understand how the CSP specification looks in terms of the structural characteristics (i.e., investigating inherent weaknesses in communications, race hazards etc.)

TABLE 4

LISTING OF THE CSPN COMMAND LINE OPTIONS.

Option	Description
-h	Used to generate a help screen which displays the contents of the table below: "csp -h"
-v	Used to set the verbose mode and is only valid when the "-o" option is specified. An interactive menu is invoked which allows the user to set SPNP run parameters.
-f	Used to generate failure transitions into the <i>filename_snp.c</i> file. This option enables detection of failure annotations and causes interactive inputs with the "-o" option specified.
-F	Set to invoke the filter which will replace the 3 special characters (?,!,:) in the <i>filename_snp.c</i> with SPNP compliant characters (_i_, _o_, and _ respectively). Otherwise, SPNP will not compile the input file. Valid only when the "-o" option is used.
-s	Use the default service rates for timed transitions. If no service rate is specified as an annotation then CSPN will use 0.1.
-o<name>	To generate the SPNP input specification file (<i>filename_snp.c</i>) this option <u>must</u> be specified ("name" is optional and the default used is the tool name "cspn").
-i<number>	Number of iterations used by SPNP (default is 2000).
-a<number>	Rate for return to initial marking from absorbing markings (default is 0.0).
-p<number>	Set floating point precision used by SPNP (default is 0.000001).
-P	Set to enable selection of priorities for individual transitions (the default is none).
-d	Set to generate a "dot" graphics file. Dot uses this digraph specification file to generate the graphical representation of the Petri net.
-n	Set to enable a network list file. This file shows how CSPN has interpreted the structural aspects of the CSP specification.
-t	Set to generate a symbol table file containing all the data recorded for each element (process names, constructions, variables, channels, ...) of the process specification.

then adding the "-d" option would enable only the production of the graph. The "-F" option invokes a filter and is necessary only when the user plans to run an SPNP analysis. The "-f" option is a nice feature because it enables the analyst to assume a failure free environment by simply ignoring any embedded fail annotations that may exist in the CSP specification (without "-f" CSPN *ignores* failure annotations). Omitting failure annotations from the P-CSP specification has the same affect. The option "-s" streamlines the process of generating the SPNP input specification by assigning default service rates to timed transitions without querying the user to provide such. As mentioned above, the "-o" option generates a file for SPNP analysis. It is best if a file name be given with this option (i.e., "-ofilename"). This settles the problem of overwriting previous files generated using the default name that is assigned by CSPN if no name is provided. The "-i", "-a" and "-p" options are used to parameterize the SPNP run by setting the iteration number, absorbing rate (for recycling back to the initial marking), and precision for floating point operations respectively. The "-P" option is only valid when "-o" is used and enables the user to assign priorities to any of the transitions. The "-d", "-n" and "-t" options are useful when something unexpected happens after running CSPN such as a run time error. The user may wish to rerun the translation and view the internal data structures that are generated during the translation process.

4.4 CSPN data structures

Internally, there are four basic data structures employed by CSPN: (1) *Symbol table* which maintains attributes assigned to all system elements (actions, processes, communications and constructions), (2) *Process lists* which consist of all the names of the associated actions/processes involved in a particular construction, (3) A *network* of linked lists which capture the structure of the specification (adjacency and nesting), and (4) The bipartite digraph which defines the structural character of the Petri net is represented as a *coincidence matrix*. The coincidence matrix (or co-matrix) maintains the distributions of places, transitions and their connectivity.

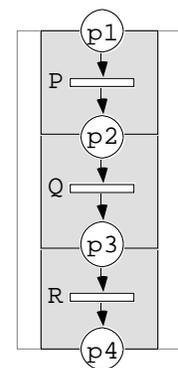
CSP: ... P;Q;R

P-CSP: ... SEQ{P,Q,R};

	p1	p2	p3	p4
T				
R	-	+		
A		-	+	
N			-	+
S				

- indicates an arc from p3 is input to transition R.

+ indicates an arc output from transition R to place p4.

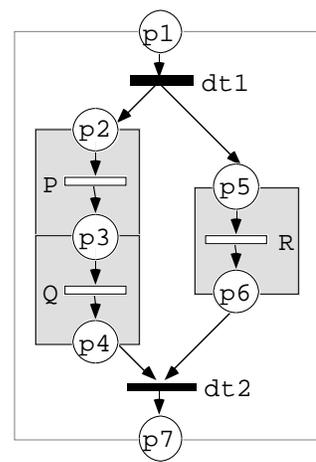


Note: each box in the Petri net (excluding transitions) represents a process.

CSP: ... P;Q||R

P-CSP: ... PAR{SEQ{P, Q}, R};

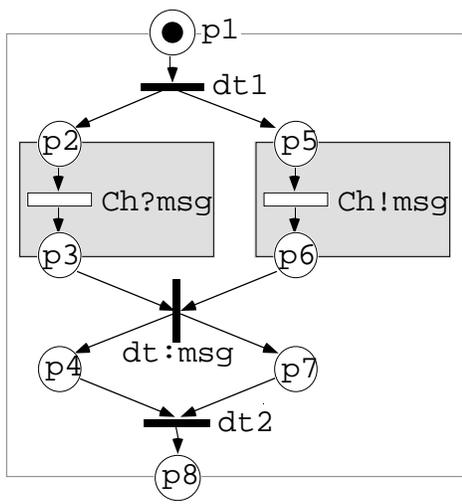
	p1	p2	p3	p4	p5	p6	p7
T	dt1	-	+		+		
R	P		-	+			
A	Q			-	+		
N	R					-	+
S	dt2			-		-	+



Note: each box in the Petri net (excluding transitions) represents a process.

P-CSP: ... PAR{{Ch?msg}, {Ch!msg} (msg)};

	p1	p2	p3	p4	p5	p6	p7	p8
T	dt1	-	+		+			
R	Ch?msg		-	+				
A	dt:msg			-	+	-	+	
N	dt2					-	-	+
S	Ch!msg					-	+	



The coincidence matrix provides a uniform data structure that can be used to manipulate (combine and reduce the Petri nets).

Note: each box in the Petri net (excluding transitions) represents a process.

Figure 28. P-CSP constructions with co-matrix and Petri net representations.

The construction of the n-by-m co-matrix is defined in terms of the transitions (CSP-process names become transition names). Transitions are associated with rows (from top to bottom).

combining the co-matrices of the component Petri nets to obtain a new co-matrix for the combined Petri net. Combining all of the sub-component co-matrices produces a complete *system* Petri net. The combining process expands one co-matrix by another. Figure 29 highlights the basics that involve expanding a co-matrix A by another co-matrix B. Thus, depending on the locality of co-matrix B one of three possible expansion methods is used.

Expand A (3x4) with B (5x6) into C (7x8).

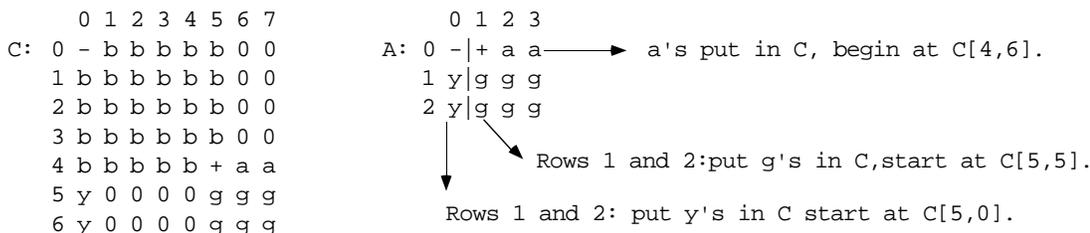


Figure 30. Diagram of expansion method one.

The Method 1 algorithm is pictured in Figure 30. The C matrix dimensions C[x,y] are determined as follows: $x = x_a + x_b - 1$ and $y = y_a + y_b - 2$ (where $[x_a, y_a]$, and $[x_b, y_b]$ are the dimensions of the A and B matrices). In the C matrix diagram, 0's are constant (i.e., not assigned from A or from B to C). Also, the "-" and "+" shown in A are now separated diagonally as shown in C.

Case 1: Expand A (5x5) with B (4x4) into C (8x7).

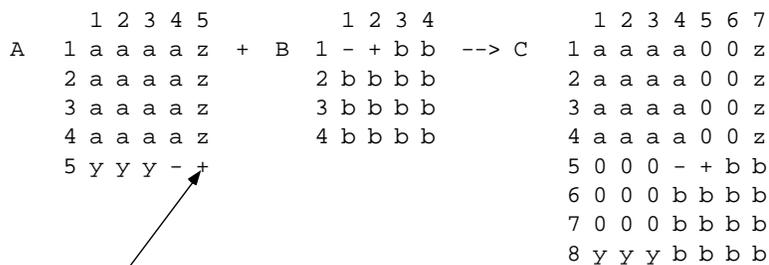


Figure 31. Diagram of expansion method two.

Method 2 is described in Figure 31. In case 1, the resultant C matrix is 8x7. Case 2 is a variation which occurs when "-" is discovered in the last column and row. This will occur

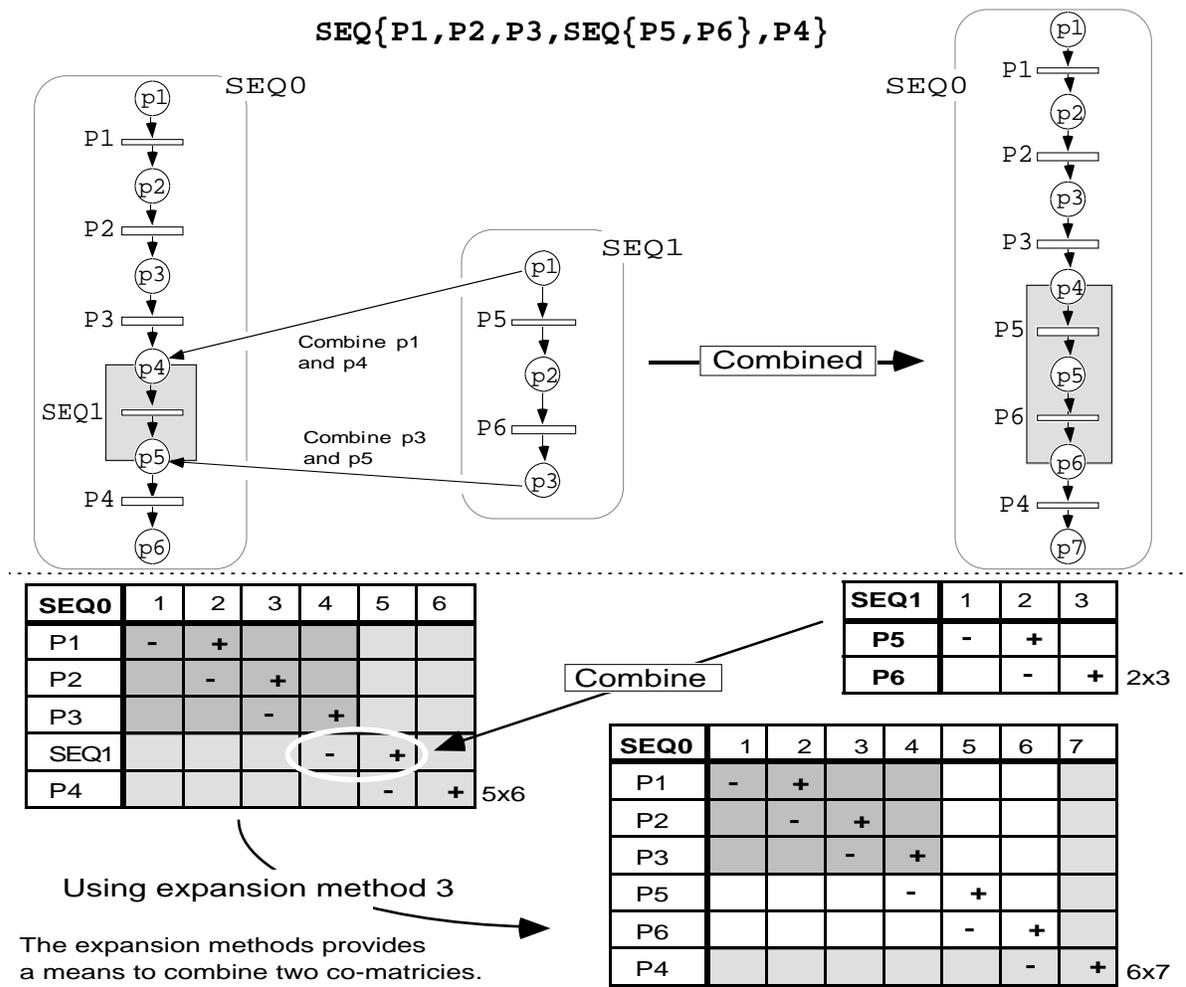


Figure 32. Diagram of expansion method three (shows Petri nets and co-matrices).

when a recursive construct is used in the P-CSP specification. In such a case the last column is dropped and the C matrix is 8x6. Also, for case 1 (where $A[m,n] = C[5,5] = "+"$), if a "z" in A is "+" then it will be moved to the last column (same row).⁴ Similarly, in either case 1 or 2, a "y" in A is "+" then it is moved to the last row in C (same column). The Method 3 expansion is too detailed to describe in the same terms as was done for Method 1 and 2 (refer to the Appendix C for the code on Method 3). The basic idea is given in Figure 32 which shows how the SEQ1 co-matrix (analogous to co-matrix B in Figure 29) is inserted into the

⁴Method 2a exception: catch all the +'s in last column which are to be moved to the new last column. These +'s are outputs from transitions to the last place in A so now they must be connected to the new last place in C. Only consider rows above rowMark which is the row being expanded (with the "- +" pair in the diagram).

SEQ0 co-matrix (analogous to co-matrix A in Figure 29). The expansion replaces the transition SEQ1 by the two process names P5 and P6. The final *combined* result retains the SEQ0 name. Note, the term SEQ is a key word (for sequential composition of processes), it may itself be considered a process. CSPN treats each occurrence of this type as a unique process by appending a unique number to the name (0 is appended to the first occurrence of SEQ to give SEQ0 and the next occurrence of SEQ will have "1" appended). This strategy allows the program to track each occurrence of a given keyword type. The keywords subjected to numbering include SEQ, PAR, NDC, DC, STOP and SKIP.⁵

```

Expanding Sys[0], Net: TrainXing
Searching links of net[1], symbol: PAR1
ProcList0: 1-dt1 2-Train 3-Gate 4-dt2
1. Symbol: dt1, Type: 21
2. Symbol: Train, Type: 10

Merge processes PAR1 <- Train
Expansion includes the following:
A: PAR1
ProcL1: 1-dt1 2-Train 3-Gate 4-dt2
PR Mtr: 1 2 3 4 5 6
[ 1]: - + 0 + 0 0
[ 2]: 0 [-+] 0 0 0
[ 3]: 0 0 0 - + 0
[ 4]: 0 0 - 0 - +

B: Train
ProcL2: 1-InTransit 2-Togate!Arrive
3-dt!Arrive 4-AtIntersection
5-Togate!Depart 6-dt!Depart
PR Mtr: 1 2 3 4 5 6 7
[ 1]: - + 0 0 0 0 0
[ 2]: 0 - + 0 0 0 0
[ 3]: 0 0 - + 0 0 0
[ 4]: 0 0 0 - + 0 0
[ 5]: 0 0 0 0 - + 0
[ 6]: 0 0 0 0 0 - +

C: A<-B is a new (9x11) Matrix
Running Method 3:
ProcL3: 1-dt1 2-InTransit 3-Togate!Arrive
4-dt!Arrive 5-AtIntersection
6-Togate!Depart 7-dt!Depart
8-Gate 9-dt2
PR Mtr: 1 2 3 4 5 6 7 8 9 0 1
[ 1]: - + 0 0 0 0 0 0 0 0 0 0
[ 2]: 0 [-+] + 0 0 0 0 0 0 0 0 0 0
[ 3]: 0 0 - + 0 0 0 0 0 0 0 0
[ 4]: 0 0 0 - + 0 0 0 0 0 0 0
[ 5]: 0 0 0 0 - + 0 0 0 0 0 0
[ 6]: 0 0 0 0 0 - + 0 0 0 0 0
[ 7]: 0 0 0 0 0 0 0 - + 0 0 0 0
[ 8]: 0 0 0 0 0 0 0 0 - + 0
[ 9]: 0 0 0 0 0 0 0 0 - 0 - +

Completed expansion!

```

Replace with Train symbol.

Train inserted in the PAR1

Figure 33. CSPN run shows before and after combining coincidence matrices.

In Figure 33 a more complex expansion is depicted where the "Train" symbol is located within the process list of the PAR1 symbol. CSPN expands PAR1's coincidence matrix (matrix A) by inserting the coincidence matrix of the Train (matrix B) into matrix A at the a_{ij} location (at $i=2$ and $j=2$). Because the Train symbol is of type 10 (indicating a compound

⁵Incidentally, the first four words listed give rise to P-nodes which constitute composition constructs which can themselves contain other P-nodes or L-nodes. Lnodes are nodes which can be 'listed' inside of a P-nodes (e.g., a channel!output or channel!input) which themselves are atomic. Not mentioned are stmlist, MU.identifier, and SystemID which are other possible Pnodes. These distinctions are made for the purpose of capturing structural characteristics of the specification.

sub-Petri net that can be embedded into other Petri nets), it can be replaced by its expanded coincidence matrix (including the replacement of the Train symbol in the PAR1 process list with the Train's process list). The resultant C matrix has 9 rows and 11 columns.

The combining of the sub-Petri net co-matrices is constrained to preserve the process algebraic structure in three dimensions (1) adjacency of terms within a process, (2) adjacency among declarations of processes and (3) nesting. Figure 34 shows an instance of the data structure which is used to capture all three structural dimensions. Adjacency refers to the sequential ordering of terms in the algebra while the word nesting is used in the normal algebraic sense. The first type of adjacency is illustrated by the sequence of process components: PAR1, dt1, Train, Gate, Arrive, Depart, dt2. In the case of nested structures, each new level of nesting requires a new NET[i+1] be appended to the tail of the declared process pointed to by SYS[i]. Each of the *two* lists is anchored by a pointer contained in an array of pointers. The two arrays SYS[] and NET[] are shown in Figure 34 as anchoring the lists of either adjacent or nested structures. The second type of adjacency (among declared

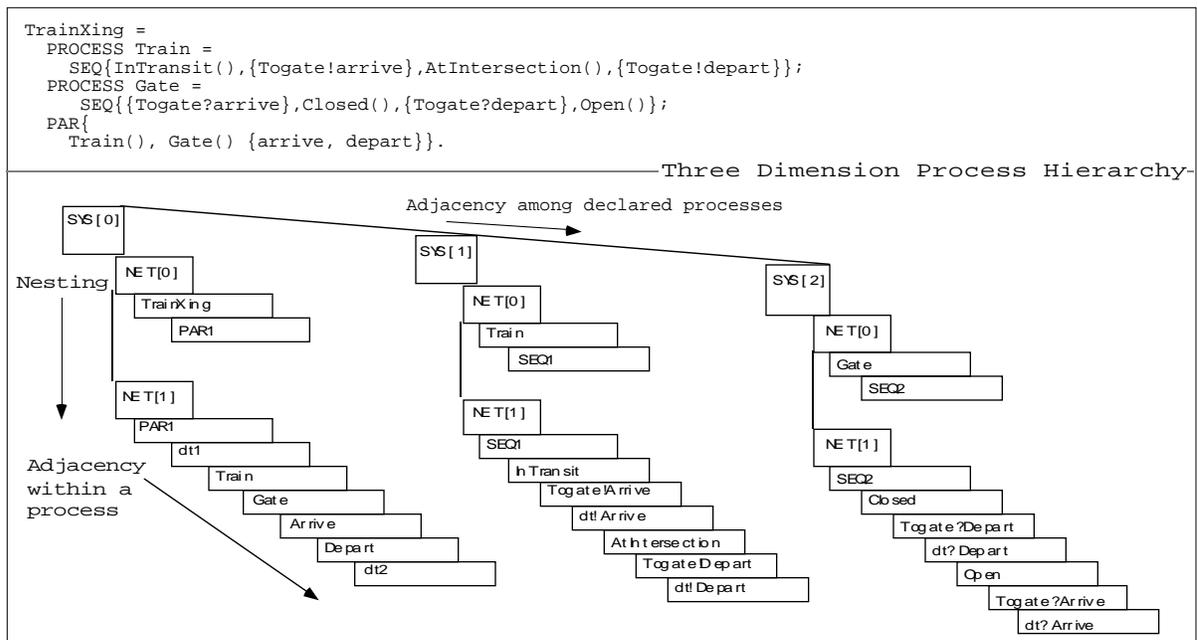


Figure 34. Data structure for nesting and adjacency detected in the specification.

processes) is recorded sequentially as follows SYS[0], SYS[1], ... SYS[n]. The SYS[0] pointer always gives the system identifier (the actual symbol itself is pointed-to by NET[0] [see Figure 37 to verify this example]) and the body (or *main* part) of the system composition. Each new SYS[i] pointer is a new "PROCESS" declaration. Each new NET[i+1] is a new level of nesting. The list attached to a given NET[i] contains the components within a given process constructor (so-called a p-node using the nomenclature of Figure 36).

Figure 35 gives another example of the linking associated with the process hierarchy for the specification named "SysSimpleEx." In this example, the nesting is overstated. Thus, the leg of SYS[1] runs from NET[0] to NET[5]. The first element of each list is the name of the process node (p-node for short, which caused a new NET[i] pointer to be generated).⁶ The p-nodes of the SYS[1] leg are as follows: Eg1, SEQ1, SEQ2, SEQ3, PAR1 and SEQ4. The depth is 6 but the level of nesting is not depth 6 (the deepest level of nesting is actually 4). To translate the nesting and the adjacency out of this leg into a Petri net, we must traverse the tree as shown in Figure 35 from left to right and from the bottom up. Actually, we start from the bottom of SYS[1] and move right to the end and then finish with SYS[0]. Let us consider the SYS[1] leg starting at NET[5]. Moving up the leg past NET[4] to NET[3] we encounter a p-node "PAR1" which must be expanded. By virtue of the syntactical correctness, we are guaranteed that the this p-node has been fully expanded. Thus, by accessing the symbol table entry for "PAR1" we find the list of sub-components (which includes dt1, P11, P12, dt2), and simply replace PAR1 in SEQ3's list (i.e., at position NET[3]) with the PAR1 list of sub-components. Actually the list is known as a *process list* (i.e., contains the sub-component symbols, each separated by a comma) and individual elements of the list are known as p-nodes. The new process list for SEQ3 that results is the following P1, P2, dt1, P11, P12, dt2. This same kind of replacement (expansion) mechanism continues until the top of the leg is

⁶See Figure 36 for a definition of P-nodes and L-nodes.

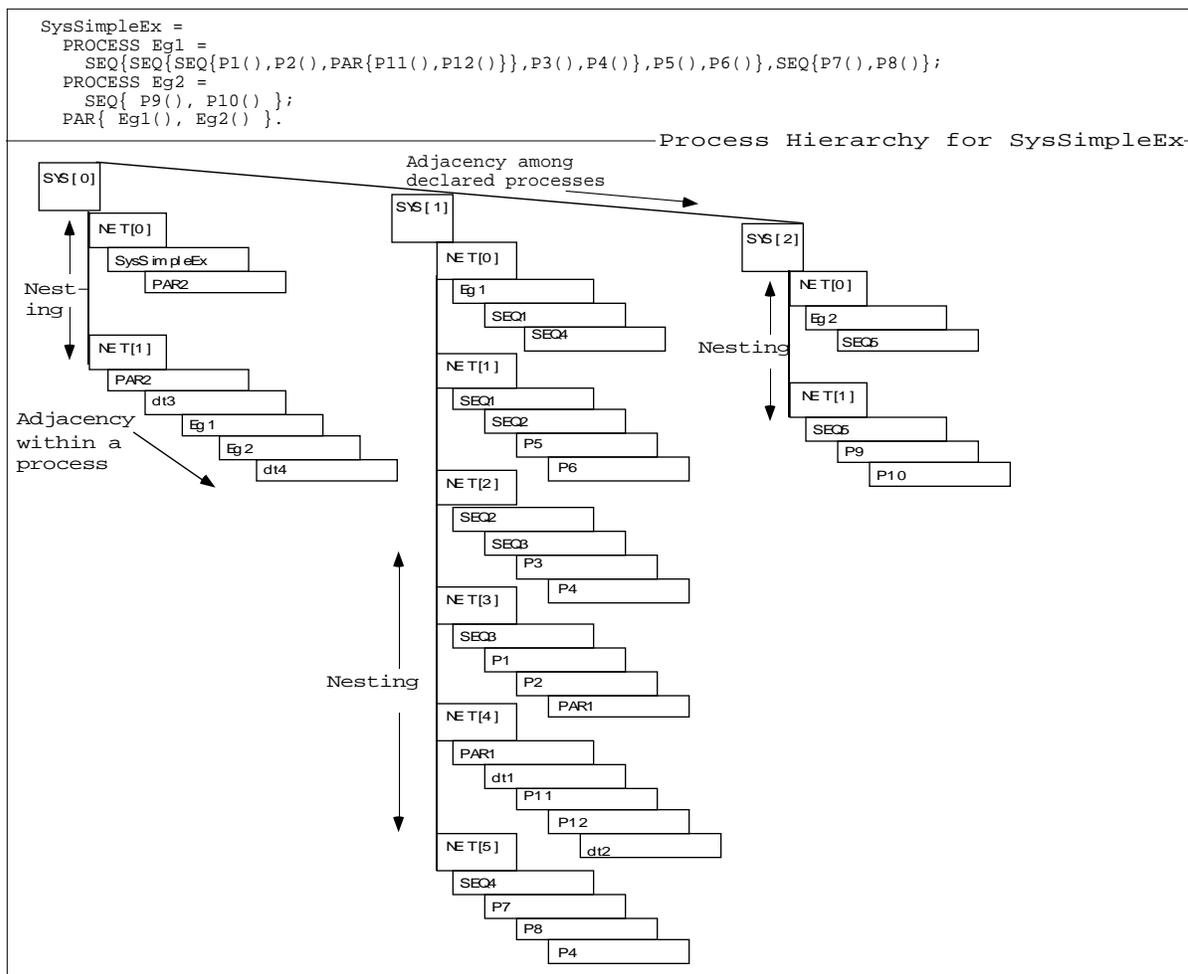


Figure 35. Process hierarchy for system "SysSimpleEx" with exaggerated nesting.

visited (i.e., at NET[0] = "Eg1"). The process is then repeated for both SEQ1 and for SEQ4. Thus, to recompose the whole process algebraic system in terms of a Petri net from the combined "SYS[] x NET[]" structure CSPN *expands* each of the p-node component and records the results in the symbol table entry recursively. Refer to Figure 36 for a relational diagram of the network (or process hierarchy) data structures and to Figure 37 for an exact definition of the (1) symbol table entry, (2) the net_node and (3) the node data structures used in recording the process hierarchical structure.

4.6 P-CSP semantics as it relates to the data structures

The structural characteristics of a P-CSP specification necessitate the framework of P-nodes and L-nodes defined in Figure 36. Table 5 enumerates the various symbol names assigned to the P-CSP components during the translation (parsing). The P-nodes are anchored by the "SYS[]" array. This is an array of NET_NODE pointers. The L-nodes are anchored by an array of NODE pointers called "NET[]." Each NET_NODE contains a NET[] array to capture both the nesting and adjacency defined within a P-node.

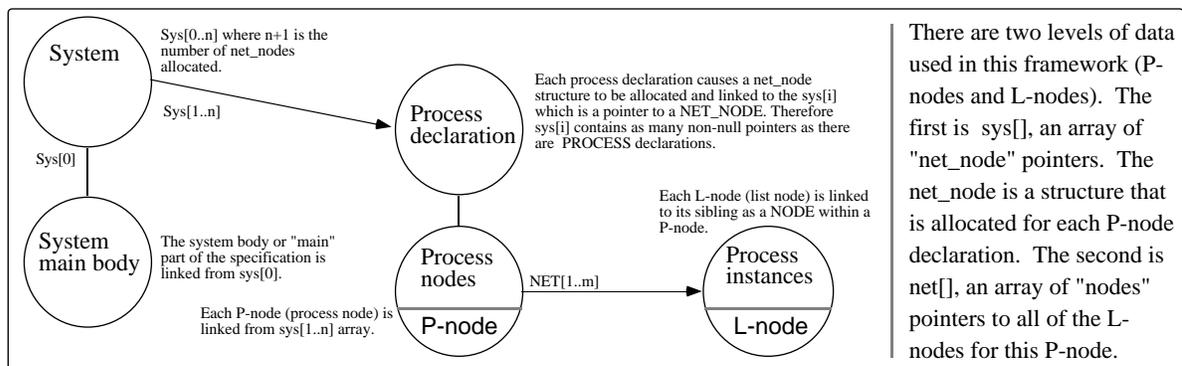


Figure 36. Relational diagram for the network (or process hierarchy) data structures.

The P-CSP grammar distinguishes 3 categories of primitive elements. The P-nodes are the composition statements used to express the semantics of the system description. The list elements (or L-nodes) are instances of pre-declared processes, variables or channels. The final category is other elements which consist of all other elements not included by the previous two categories (e.g., connectives, grouping symbols or punctuation). Each element is assigned a type number according to Table 5.

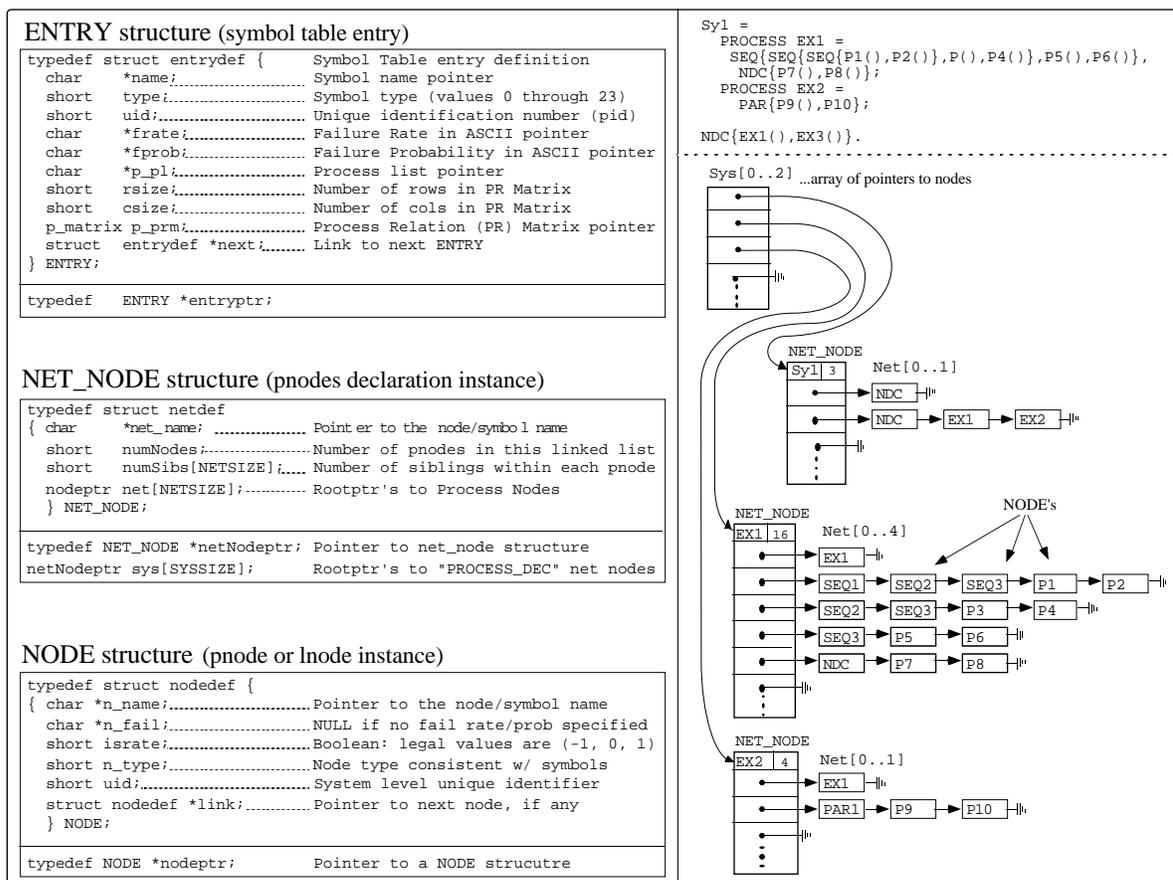


Figure 37. Definitions of the symbol table and process hierarchy data.

TABLE 5

CONSTRUCTS USED IN P-CSP AND THEIR TYPE VALUES

Process nodes and artifacts		Channels and variables	
NULL_TYPE	0	BOOL_VAR	15 l-node
SYSTEM_ID	1 p-node	VAR	16 l-node
STMT_LIST	2 p-node	EXPRESSION	17 l-node
STOP_PROC	3 l-node	RIGHTBRACE	18 not defined
SKIP_PROC	4 l-node	LEFTBRACE	18 not defined
PAR_PROC	5 p-node	BRACE	18 not defined
SEQ_PROC	6 p-node	SEMICOLON	19 not defined
NDC_PROC	7 p-node	SEMIC	19 not defined
DC_PROC	8 p-node	DOT	20 not defined
MU_PROC	9 p-node	DUMMY	21 not defined
PROC_CALL	10 l-node	SYNCH_MSG	22 l-node
PROCESS_DEC	11 special	GUARD1	23 not defined
CHAN_PROC	12 l-node (contains 13-14)	GUARD2	24 not defined
INPUT	13 l-node	RECURSE	25 not defined
OUTPUT	14 l-node	RECUR_TOP	26 not defined
		SDT	27 not defined
		TYPES	28 not defined

There are twenty three different types.

The symbol table contains all of the identifiers used in the specification (i.e., names of declared processes, channel variables, simple variables and system defined names) and is the primary source of information about the system. A hash function enables an efficient means of accessing the associated data shown in Figure 37. Any symbol used or defined within the specification is accessible.

4.7 P-CSP's usage of failure and service rate annotations

When the "-f" option flag is set on the command line, CSPN will incorporate any *legal* failure annotations into the SPNP file. Naturally, the "-o<fn>" option must also be specified, otherwise CSPN will not produce the fn_snp.c file. Legal annotations are specified as either a probability ": FAIL(p=x.xx)" or rate ": FAIL(r=x.xx)" of failure.

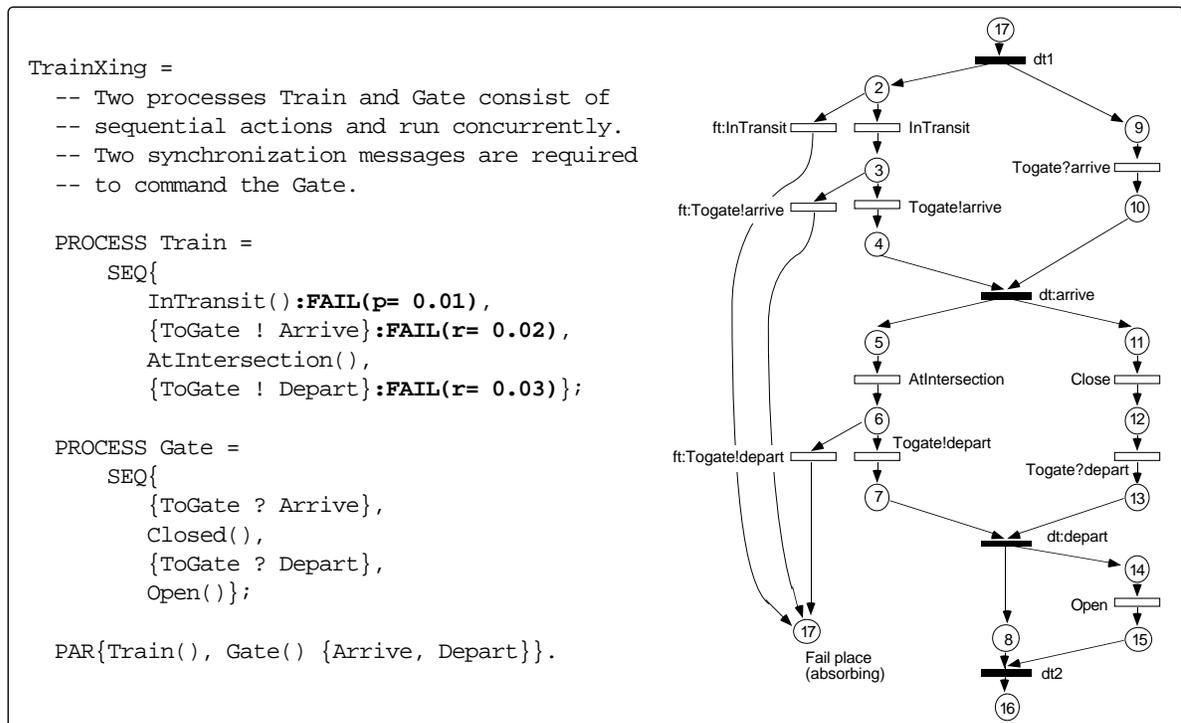


Figure 38. Specifying failure annotations in P-CSP and the resulting Petri net.

This is illustrated in Figure 38. A failure annotation can be related into the specification at any level. However, only the values that are associated with a *non-expandable element* (one

which may not be further decomposed) will actually be translated into the SPNP file. Thus, if a rate were attached to the process call: "Train():FAIL(r=x.xx)" in Figure 38 (composed inside a PAR construction) then the value would not be translated into the SPNP file. Thus, annotations associated with composite processes are not incorporated into the fn_snp.c file but can be maintained as a record of the results of any current or subsequent runs (e.g., failure probability of a group of components).⁷ Note, that service rates can also be annotated in a similar fashion with the same caveat that in order to be utilized in the SPNP file it must be attached to an non-expandable element. The notation is ":SERV(r=x.xx)".

4.8 Linking synchronization primitives

<pre>In synclink for symbol: TrainXing ... ProcL0: 1-dt1 2-InTransit 3-Togate!Arrive 4-dt!Arrive 5-AtIntersection 6-Togate!Depart 7-dt!Depart 8-Togate?Arrive 9-dt?Arrive 10-Closed 11-Togate?Depart 12-dt?Depart 13-Open 14-dt2 Sync message: Arrive, Find transition: dt!Arrive, at pos: 4. Matching trans: dt?Arrive, with pos: 9 Sync message: Depart, Find transition: dt!Depart, at pos: 7. Matching trans is: dt?Depart, with pos: 11 PR Mtr: 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 [1]: - + 0 0 0 0 0 0 + 0 0 0 0 0 0 0 0 [2]: 0 - + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [3]: 0 0 - + 0 0 0 0 0 0 0 0 0 0 0 0 [4]: 0 0 0 - + 0 0 0 0 - + 0 0 0 0 0 0 [5]: 0 0 0 0 - + 0 0 0 0 0 0 0 0 0 0 [6]: 0 0 0 0 0 - + 0 0 0 0 0 0 0 0 0 0 [7]: 0 0 0 0 0 0 - + 0 0 0 - + 0 0 0 0 [8]: 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [9]: 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [10]: 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [11]: 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [12]: 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [13]: 0 0 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [14]: 0 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - +</pre>	<pre>Release: Row: 9 Row: 11 ProcL0: 1-dt1 2-InTransit 3-Togate!Arrive 4-dt:Arrive 5-AtIntersection 6-Togate!Depart 7-dt:Depart 8-Togate?Arrive 9-Closed 10-Togate?Depart 11-Open 12-dt2 PR Mtr: 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 [1]: - + 0 0 0 0 0 0 + 0 0 0 0 0 0 0 0 [2]: 0 - + 0 0 0 0 0 0 0 0 0 0 0 0 0 0 [3]: 0 0 - + 0 0 0 0 0 0 0 0 0 0 0 0 [4]: 0 0 0 - + 0 0 0 0 - + 0 0 0 0 0 0 [5]: 0 0 0 0 - + 0 0 0 0 0 0 0 0 0 0 [6]: 0 0 0 0 0 - + 0 0 0 0 0 0 0 0 0 0 [7]: 0 0 0 0 0 0 - + 0 0 0 - + 0 0 0 0 [8]: 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [9]: 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [10]: 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [11]: 0 0 0 0 0 0 0 0 0 0 0 0 0 - + 0 0 0 0 0 0 [12]: 0 0 0 0 0 0 0 0 - 0 0 0 0 0 0 0 - +</pre>
---	---

Figure 39. Resolving synchronization links.

The process of linking the synchronization primitives occurs after all expansions have completed (except adding failure annotations). In Figure 39 rows 9 and 11 are removed in merging the output message transition with the matching input message transition.

⁷Sensitivity analysis is an examination of the effect of small variations in system parameters on the output measures can be studied by computing the derivatives of the output measures with respect to the parameter [Mainkar93]. Sensitivity analysis is useful to estimate how the output measures of a system model are affected by variations of its input parameters (as well as for system optimization and bottleneck analysis).

4.9 CSPN file descriptions

There are twelve files that make up the CSPN tool (not including the C files generated by lex and yacc and two small header files used in the lex and yacc specification files). These files are named here and are briefly described with respect to their contents (and in some cases multiple function capabilities): (1) `cmd_line.c`, (2) `csp.l`, (3) `csp.y`, (4) `expn_cspy.c`, (5) `itoa.c`, (6) `net.c`, (7) `petri_cspy.c`, (8) `prlist.c`, (9) `prmatrix.c`, (10) `scoring.c`, (11) `symbol_cspy.c`, (12) `symbol_cspy.h`.

4.9.1 `Cmd_line.c` description

Command line checks for command line arguments. If there are none it uses the defaults. Otherwise it allows the user to change certain options available from SPNP (SPNP Reference). There are three other noteworthy functions. The *do_file* is a function that displays the command line defaults for each run, *usage* displays the help screen, *gen* sets the defaults for the parameters part of the SPNP.c file, and *choose* is an interactive routine that is invoked by the command line verbose mode option flag "-v." This routine allows the user to choose from any of the available options in the parameters part of the SPNP.c file. *Pic1* and *Pic2* are functions associated with *choose*.

4.9.2 The `csp.l` and `csp.y` descriptions

Lex and yacc are tools designed for writers of compilers and interpreters (i.e., any application that looks for patterns in its input, or has an input or command language). They help one write programs that transform structured input. Lex takes a set of descriptions of possible tokens and produces a C routine (called the lexical analyzer or lexer or scanner). The set of descriptions given to lex is called a *lex specification*. The lex specification for the P-CSP language is found in Appendix B and is found in the `csp.l` file.

The token descriptions that lex uses are known as regular expressions. As the input is divided into tokens, the CSPN tool must establish the relationship among tokens. CSPN needs to find expressions, statements, declarations, blocks, and processes in the specification

program. This task is known as parsing and the list of rules that define the relationships that CSPN understands is the grammar (also called the yacc specification and for the P-CSP language is found in Appendix B). Yacc takes a concise description of the grammar (basically in BNF notation and is found in the *csp.y* file) and produces a C routine that can parse the grammar, called the parser. The parser detects when a sequence of input tokens matches one of the rules in the grammar and also detects syntax errors whenever the input doesn't match any of the rules.

4.9.3 **Symbol_cspy.h and symbol_cspy.c description**

Symbol_cspy.h is the primary header file included in the *csp.y* file. This file contains included C library files, global variable declarations and prototype declarations. *Symbol_cspy.c* manages updates to the symbol table as each new symbol token arrives to the parser from the scanner via a call to the *getsym* function. Table 6 lists all the functions associated with managing the symbol table structure.

TABLE 6

SYMBOL TABLE UTILITY FUNCTIONS

Function name	Description
look	Takes a pointer to a symbol name and returns a pointer to the entry if it exists (including found=1 => true). Otherwise, it returns a ptr to the free entry where it could be inserted.
getsym	Used in the parser to pick up the symbols and to "insert" and verify insertion into the symbol table.
insert	Takes a pointer to symbol and returns a ptr to the symbol table entry. The duplicate_sym pointer passes back: -1 is if a duplicate symbol exists (a failed operation), 0 is if the symbol was inserted successfully.
init_table	Initializes the symbol table.
print_table	Print_table has 2 loops to print (1) index the table, (2) index the linked list while traversing the links for collided symbols.
dumptable	Dumptable prints the contents of the symbol table in a stylized fashion.

4.9.4 Net.c description

This file contains two prime functions: (1) *net_main* and (2) *search_net*. *Net_main* is the driver function that invokes 12 other utility functions used to build a net hierarchy to capture the P-CSP specification structure. Once the net hierarchy is completed *search_net* traverses the net hierarchy in the process of constructing the process lists and co-matrices for each individual component (i.e., p-node) in the specification. The sub-functions *push*, *pop*, *peak* and *printStack* are used to manage the stack which is used to track the nesting of process compositions. The other functions are responsible for allocating and linking up new nodes that are generated for every new term in the process algebra. When *search_net* has completed, the specification is decomposed. The utility functions are listed in Table 7.

TABLE 7

NET UTILITY FUNCTIONS

Function name	Description
push	Puts integers on Stack[STACKSIZE].
pop	Returns the integer on top of stack.
peak	Non-destructive pop.
printStack	Prints the stack contents top to bottom.
linkToSiblings	If cur_pnode has sib relation link to the sib.
append	Given net[root pointer] append a node to end list.
allocate_net	Allocate a NET_NODE for a PROCESS declared symbol.
allocate	Allocate a NODE for a symbol w/in a PROCESS definition.
linkup	Link a NET_NODE to a net[root pointer].
<i>searchNet</i>	Traverse the net[i]'s to build atomic co-matrices.
updateNet	Traverses the net[i]'s to transfer any failure annotations in the symbol table to the net data structure (in the *n_fail field of NODE).
printNet	Given a netNodeptr print the contents of a NET_NODE.
net_init	Initializes net_main's global variables.
<i>net_main</i>	A (large) switch on sym_type to decide structure of the net.

Each invocation of the searchNet function requires a pointer to a NODE structure. These NODE structure pointers are contained in the array sys[] (each i in sys[i] is a PROCESS declaration). Each PROCESS declaration is represented by a NET_NODE which contains a net[i] pointing to individual p-nodes (see Figure 36) nested within the process declaration. The net[i] array contains pointers to related p-nodes (i.e., when they are used in a sys[i] PROCESS declaration). Each p-node instance is represented by a NODE with a name field for its name (process information is kept in the symbol table referenced by the n_name field).

4.9.5 Prlist.c description

In the P-CSP specification, process names are identified during translation and included in a process list according to their contextual relation in the specification. This file contains numerous utility functions which are defined in terms of a *process list* structure. The process list is a string of symbol *names* contained in the symbol table, each separated by a comma and terminated by an eos (end-of-string character). These routines can check if a process name is in the list (and its position), put a name in the list, replace a name with a new name or new list (called insertion), delete a name, count the occurrences of a name, remove by replacing a name with "*"s", destroy the list (and deallocate the memory), and display the list. In essence, the process list defines the transition names of the Petri net which are ordered row-wise in the co-matrix of each Petri net.⁸

4.9.6 Scoring.c description

This file contains two major functions: *scoring* and *AddFailures*. *Scoring* updates an integer array with "-1" indicating an input to the current row[i] (a transition) from the current column[j] (a place). A "+1" is used to indicate an output from a transition to a place. Given the number of rows (processes in the process list for this symbol), it returns the number of columns. *Scoring* knows what each P-CSP construct should look like in terms of the Petri net (i.e., it scores coincidence matrix using the canonical translation rules) by marking the n-

⁸These routines were developed with help from David Sheely (at The University of Texas at Arlington).

by-m co-matrix appropriately (i.e., n transitions and m places).

AddFailures is called from the main line code in the parser (*csp.y*) routine if the "-f" option was specified on the command line meaning that the *ignoreFailures* flag is not set. The routine parses through the process list of the system co-matrix (which is named as *prmatrix*), looks up each process in the symbol table and checks if a failure annotation is stored there. If so it will append a failure transition to the co-matrix and update the process list for the system symbol.

4.9.7 **Expn_cspy.c description**

The *expn* function combines two co-matrices using the following steps: (1) looks up the symbol name in the symbol table, (2) gets the size (m-by-n) of the co-matrix, (3) recalculates the mxn for the new (combined) co-matrix, (4) reallocates a new data structure, (5) combines the two co-matrices into the new one using one of three methods, (6) links up the result back in the symbol for that particular symbol name. See Appendix C for a complete description of these expansion algorithms.

Synclink matches transitions in the process list that look like *dt!msgX* with *dt?msgX* by the following algorithm:

- (1) Locate *dt!msgX* and rewrite ! <- ::;
- (2) Locate *dt?msgX* and remember its location;
- (3) Remove *dt?magX* transition from a duplicate process list;
- (4) Until all messages in the *synclist[]* array are located;
- (5) Now sort the remembered locations in descending order;
- (6) Delete the co-matrix row corresponding to the locations starting from the bottom up (descending order).

4.9.8 **Petri_cspy.c description**

This file contains the function *decodeSys* which uses the system (i.e., *sys[0]*) process list and co-matrix (these two items are the final product of the composition and clean-up phases in CSPN) to generate the *net()* part of the *fn_snp.c* file. The *net()* function gives the CSPL (i.e., the SPNP language) specification for the stochastic Petri net.

4.9.9 Miscellaneous file descriptions

Two additional files are the *prmatrix.c*, and *itoa.c* files. Given the number of rows and columns, *prmatrix* returns a pointer to an empty (zeroed) n-by-m process relation table (i.e., the coincidence matrix) used to specify a component Petri net. There is also a print routine which is designed to print the matrix in an easy to read format. The *itoa* function returns the ASCII (i.e., string representation) value of an integer.

4.9.10 Intermediate output files used for debugging

Numerous intermediate files are created by CSPN. All files are prefixed with the input file name dot "xxx" where xxx distinguishes the type of file. For example, if the input file (a P-CSP specification) were named "train" then the output file that contains all of the tokens generated during the translation of a train specification would be named "train.tok." Table 8 contains a list of the intermediate files and their contents.

TABLE 8

DESCRIPTION OF INTERMEDIATE TRANSLATION FILES

File name	Description
fn.tok	Lists the tokens passed from the scanner to the parser
fn.dec	Output from the searchNet routine (in net.c). Lists the symbols that were found in searching the net hierarchy, their co-matrix (Petri net representation) and failure annotation (if any).
fn.dsd1	Snapshot of the symbol table after the decomposition phase completes.
fn.epn	Lists all intermediate steps taken during expansion of the component Petri nets into the one system Petri net (i.e., combining co-matrices). This includes the steps associated with resolving synchronization links (a reduction process) and including failure annotations.
fn.net	A key file which lists all of the net hierarchy in a staggered format that shows the nesting and adjacency relationship in 2 dimensions.
fn.dsd2	Snap shot of the symbol table after the expansion process has completed.
fn_snp.c	This is the CSPL specified Petri net file on which the stochastic analysis may commence using the SPNP tool.

CHAPTER 5

ILLUSTRATION OF THE USEFULNESS OF THE CSPN TOOL

Some men see things the way they are and say, 'Why?'. I dream things that never were and say, 'Why not?'

–Robert F. Kennedy

5.1 Combining functional and performance analysis

A simple example showing a translation from the CSP specification into the stochastic Petri nets (SPNs) is provided to illustrate how performance and reliability analyses may be obtained. In this way, the merits of a powerful modeling technique using SPNs can be combined with a well defined formal specification language. The railroad crossing example was first formulated as a benchmark problem used to compare different formal methods for specifying, designing and analyzing real-time systems. Although it is both simple and easy to understand, it is complex enough to illustrate a number of aspects of the modeling and verification of timed systems. Basically, it concerns a point at which road vehicles attempt to cross over railroad tracks unless prevented by the gate which closes when a train is passing. The requirements are described in the next section.

5.2 Requirement specification for the railroad crossing

As modeled, the system combines a single train, a draw gate and a communications link. The system continuously handles one train at a time by closing the gate when the train is approaching [Heitmeyer94]. There are two basic properties the system must satisfy.⁹ (1) *Safety property* – the gate is down during all occupancy intervals (when the Train is at the intersection), and (2) *Utility property* – the gate is open when no train is in the crossing. The

⁹This model encompasses the environment which includes the train(s) and the gate, as well as the interface between them. Thus, the gate closes when a train arrives at the intersection and remains closed until the train completely passes by the intersection.

solution in general terms proceeds as follows:

- **Train** sends an "arrive" message to the Gate as it nears the intersection and proceeds towards the intersection.
- **Gate**, upon receiving the message, closes the gate and remains closed until the train departs.
- **Train** sends a "depart" signal after leaving the intersection.
- **Gate**, upon receiving the signal opens the gate and remains open.

In order to simplify this example we represent multiple interactions between these two processes, instead of multiple trains interacting with the gate.

5.3 The CSP for the railroad crossing

At the intersection, the gate closes for arriving trains and remains closed until the train has completely passed. The problem can be extended to handle multiple trains (see Appendix D which incorporates a monitor program), but only one train is specified here in Figure 40.

```

Train = (InTransit);
        (Togate ! arrive → AtIntersection);
        (Togate ! depart → Train)
Gate = (Togate ? arrive → Close);
        (Togate ? depart → Open → Gate)
TrainXing = Train ||{arrive,depart} Gate

```

Figure 40. Pure CSP specification of the railroad crossing problem.

Two concurrent processes, the **Train** and the **Gate**, communicate by sending and receiving messages. The Train outputs "arrive" on channel Togate to inform the Gate that it will soon arrive at the intersection. Upon passing through the intersection, the train sends a "depart" message to the Gate. The Gate process receives the "arrive" message and closes the gate. Once closed, the Gate waits for the "depart" message before causing the gate itself to open. Note how easy it is to identify the sender and receiver connected by the channel.¹⁰

¹⁰However, there are some drawbacks associated with using CSP. First, CSP as defined by Hoare has no concept of time. Recent extensions to CSP permit the association of time with actions [see Davies 94 and see 1, 2, 3 TBD and the references therein]. Second, since CSP uses point-to-point communication it is awkward to describe the case where the Gate process accepts inputs from multiple Train processes.

5.4 The P-CSP for the railroad crossing

In Figure 41 the train and gate processes are specified using the CSP-based language P-CSP along side the CSPN derived Petri net.

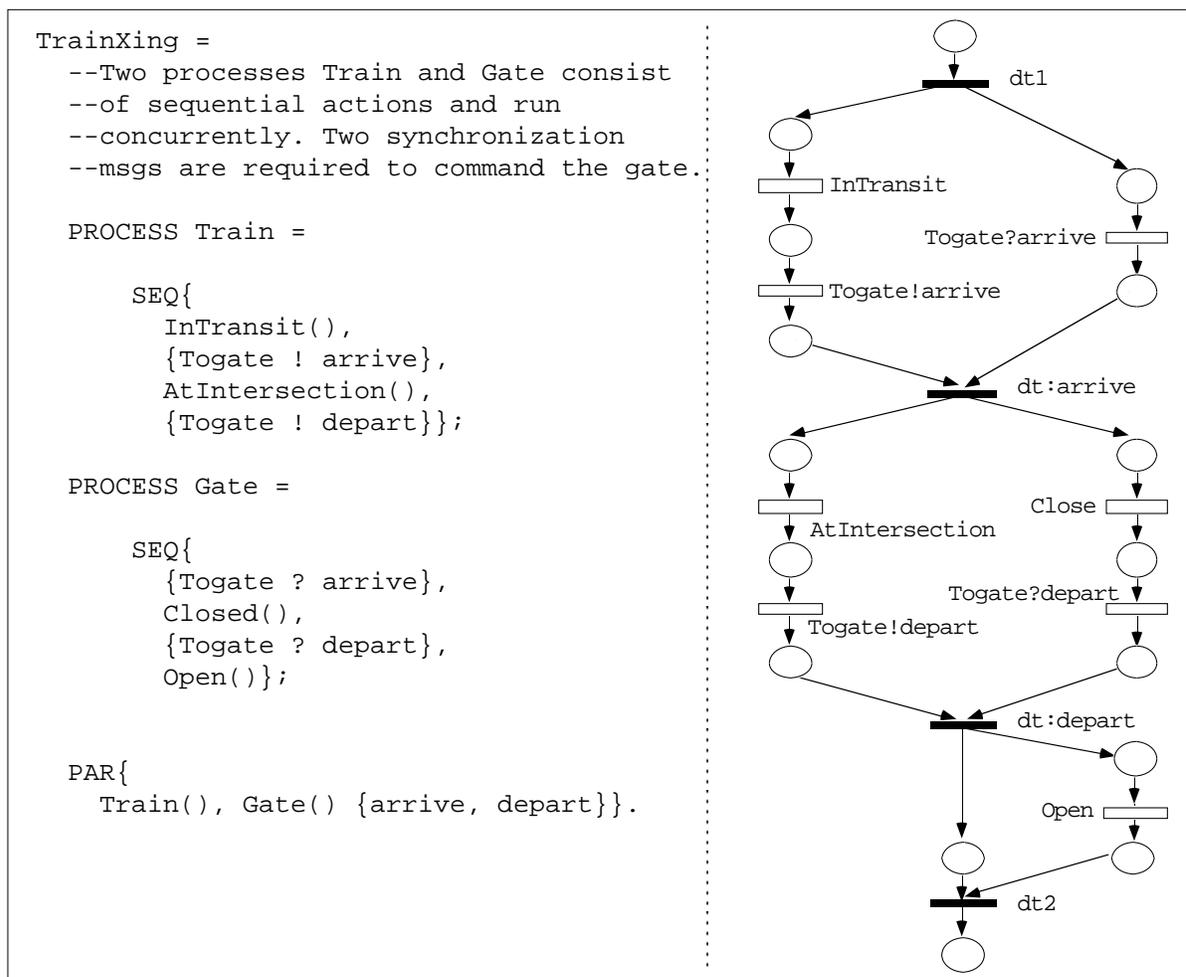


Figure 41. P-CSP specification for parallel composition of the railroad crossing.

The original CSP specification in Figure 37 provides that both processes repeat their internal activities continuously. However, given the P-CSP specification of Figure 38, the resultant Petri net graphically reveals the absence of iteration to provide for the handling of a continuous stream of trains. To provide iteration, an additional composition is added: namely $\text{Mu.X}\{\text{PAR}\{\text{Train}(), \text{Gate}() \text{ (arrive, depart)}\} \rightarrow \text{X}\}$. In this case, X is a recursive process

that provides the link between the dummy transitions dt1 and dt2 shown in Figure 41. The new net which incorporates iteration is shown in Figure 42.

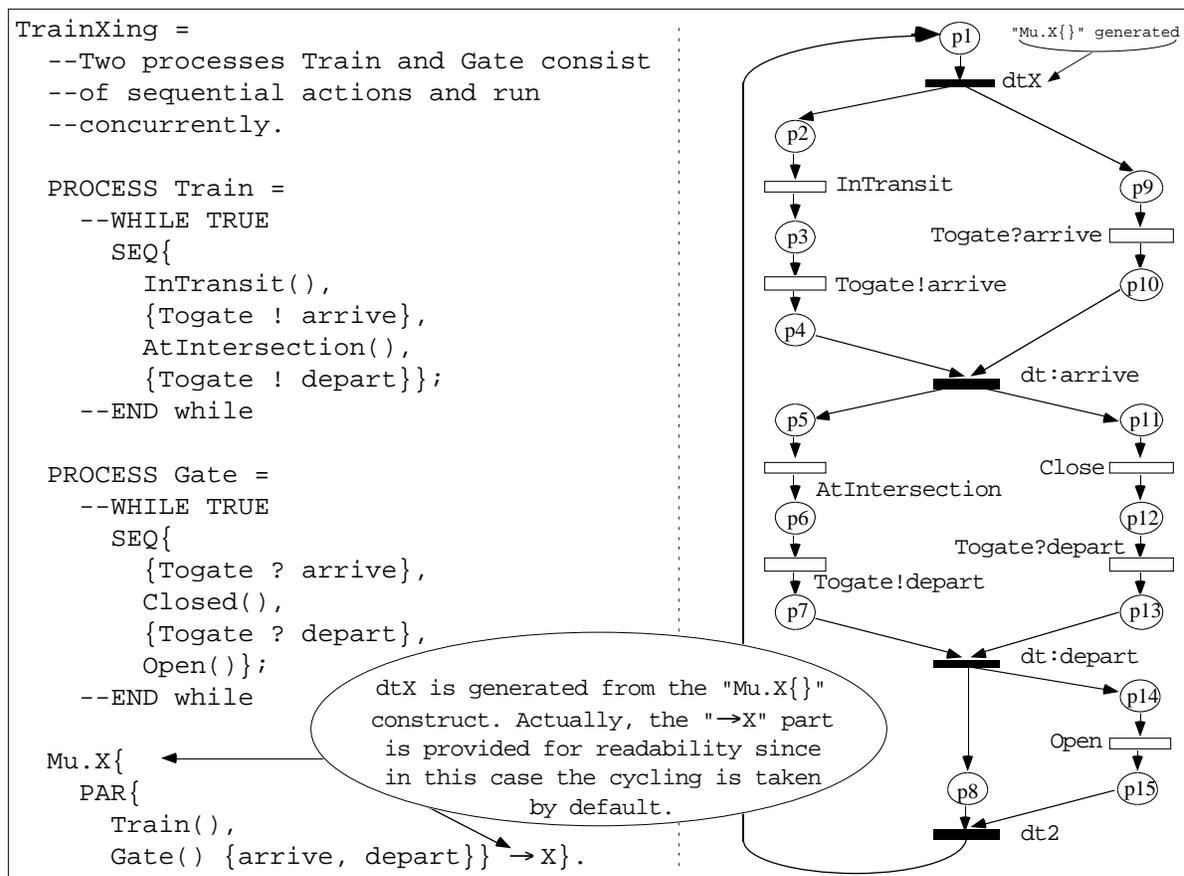


Figure 42. P-CSP specification for the (tail type) recursive composition.

5.5 Semantics of the Petri net for the railroad crossing

The train and gate operate concurrently and independently. However, for the system to meet its functional requirements both components must synchronize. To accomplish their missions (i.e., passing through the intersection and holding traffic to permit the train to pass safely) they use the channel "Togate" to synchronize. The synchronization described by the CSP may not readily reveal the potential race hazard that is more detectable in the Petri net.

The Train process could arrive to AtIntersection before the gate closes!¹¹ To avoid this unsafe state an extra "ok" gate closed synchronization message is used. In Figure 43 the messages are represented by transitions dt:arrive, dt:ok and dt:depart. The prefix "dt:" denotes a "dummy transition" that fires with probability one (i.e., an immediate transition).

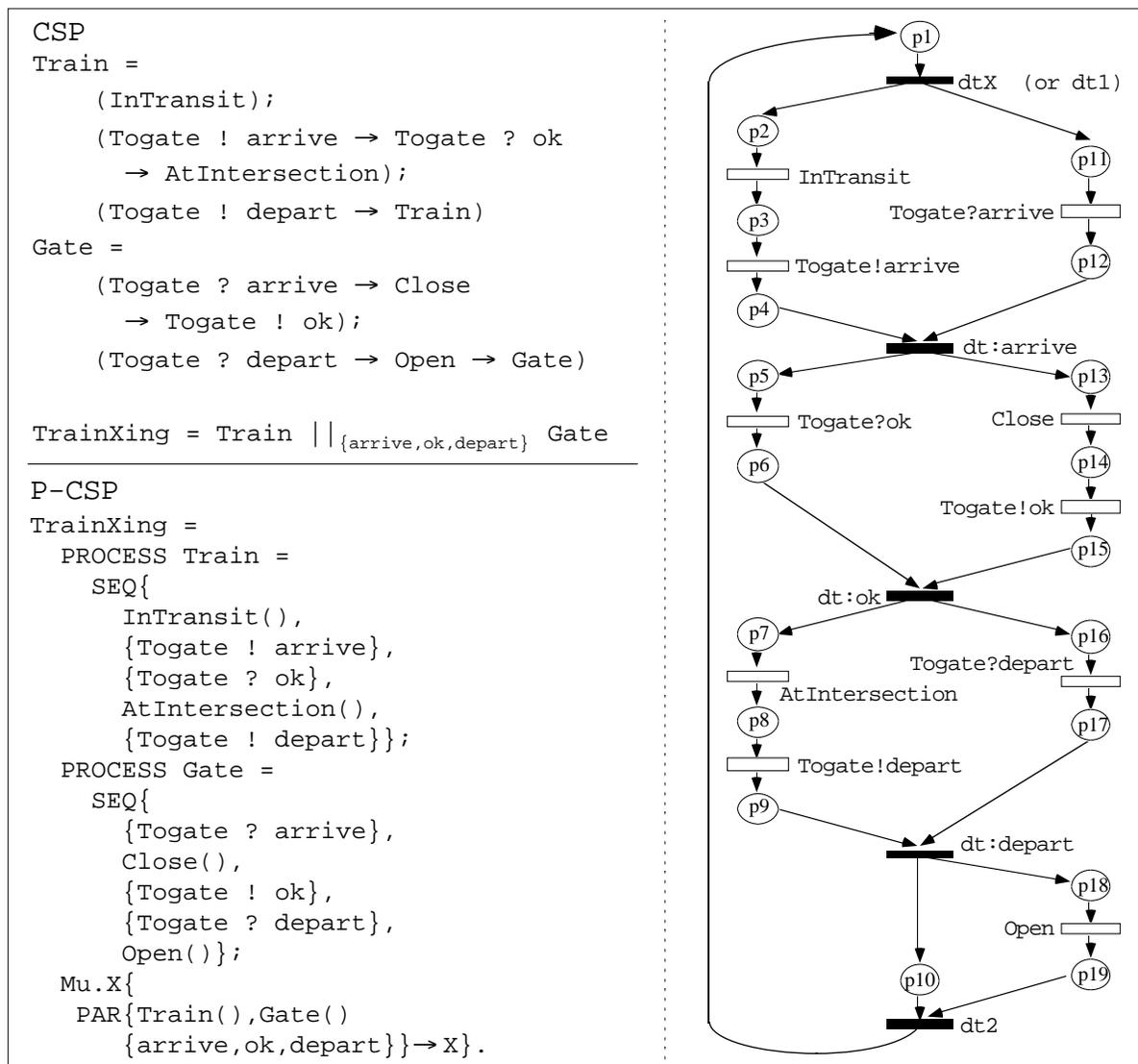


Figure 43. CSP and P-CSP specifications which address race hazard.

¹¹This is possible because after the synchronization on the "Togate" channel occurs (i.e., the "arrive" signal is received), the "AtIntersection" transition may fire before the "Close" transition denoting the case where the train arrived sooner than the time needed for the gate to close.

The gate will not begin to close until it receives the "arrive" message. First, the train must fire the transition "InTransit," and then send the "arrive" message by firing "Togate!arrive." In turn, the gate must be ready to receive the message by firing the transition "Togate?arrive." After all these actions have occurred, the gate may receive the command to close. The close command may occur (i.e., fires at some definite rate) when a token is on place "p13." This will occur immediately after the synchronizing "dt:arrive" transition has been enabled (tokens on "p4" and "p12") since this transition is immediate (consumes no resources). The marking with one token each on places p14 and p12 which enable the "dt:arrive" transition to fire.

In following the logical flow of feasible markings, we see that it is impossible for the train to proceed to the "AtIntersection" transition until the gate is closed and has fired off a message to the train: "ok" its safe to proceed. We can also notice that the same applies for the gate opening process by virtue of the transition "dt2" which essentially forces the two processes to synchronize. We could re-label the transition as "Motorist-Proceed" (perhaps).

In review, the semantics of synchronization provided by the revised CSP specification forces the train to wait until the gate closes to preserve the safety property.¹² Moreover, the "dt:ok" transition is needed because, after firing the "dt:arrive" transition (i.e., which enables the Togate?ok and Close transitions), the train may reach the intersection faster than the gate could close (e.g., "AtIntersection" fires sooner than the transition "Close").¹³ Consequently, with regard to this approach, we must ask what other possible failures are there that may cause a violation of the safety property.

5.5.1 Enumerating all possible failure transitions

In the Petri net of Figure 44, all of the possible failures are identified with respect to the activities described in the CSP specification. Transitions labeled with "ft:process-name" are

¹²We have studied the case where multiple trains may arrive at the intersection. In such cases, it becomes necessary to have a monitor arbitrate (see Appendix E for a brief look at the solution to such a case).

¹³If we assume the gate always opens and closes sooner than the time it takes the train to reach the crossing, the PN can be viewed as hazard free (except for the possibility of the gate having mechanical failure [unsafe]).

failure transitions. Dummy transitions are assigned a probability one and do not have any associated failure transitions. It is interesting to note, that instead of transitioning to place "p20" as shown in Figure 44, it would be possible to separate distinct failure types into different "absorbing" places so that the MTTF values (or the failure rate) associated with each type of failure mode can be separately denoted and computed.

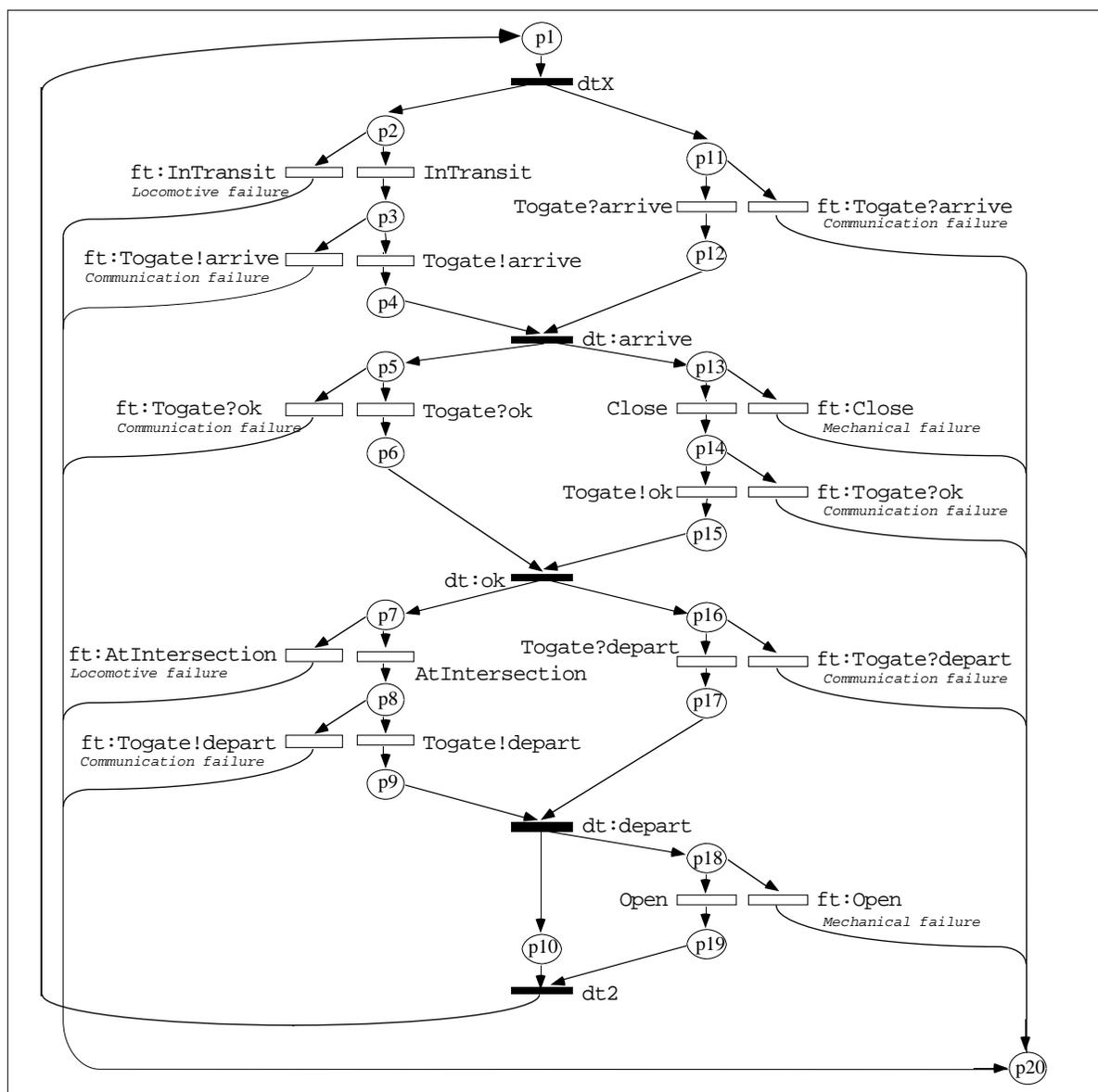


Figure 44. Railroad crossing Petri net showing all possible failure transitions.

For example, we could distinguish three types of failures based on the Petri net of Figure 44: (1) mechanical failures where the gate may fail to close or to open properly, (2) communication failures –the sending or receiving of signals could be lost, and (3) timing related failures where the train takes less time than the time taken for the gate to close. We can then distinguish three separate failure places, each to be associated with one of the failure types. The distribution of tokens in the various places of the Petri net defines the markings of the Petri net. As described in Section 3.3, we consider a transition enabled if each of its input places contains at least one token. An enabled transition may fire removing a token from each of its input places and depositing a token in each of its output places. In stochastic analysis actions are associated with an exponentially distributed times to indicate the amount of time needed for that action to complete. This firing time is the time that elapses from the point at which the transition becomes enabled to the point at which the transition actually fires. The firing of a transition causes the redistribution of the tokens in the stochastic Petri net resulting in a new marking.¹⁴

The set of all such markings together with the transitions among them is called the reachability graph. The states in the reachability graph are isomorphic to the states in a continuous (discrete) time Markov chain. We may identify unique markings that may lead to a failure and those failure transitions are then associated with an absorbing state in the Markov state diagram. Different markings potentially lead to different types of failures (e.g., a mechanical failure or some other such failure).

5.5.2 Enumerating safety critical failure transitions

We discussed the groupings of failures based on the similarity of their failure mechanism. Here we are now concerned with the manifestation (or impact) that a given failure has on the system (i.e., whether the failure may have catastrophic consequences or not). This categorization is important for determining for instance the cost or the risk that a

¹⁴For example, the time to failure of *ft:close* is known to be exponentially distributed with rate λ_1 (lets say). This is modeled in the stochastic Petri net by associating a firing time with each of the transitions.

given failure presents to its users (and/or developers). In this section the discussion will be based on the railroad crossing that is discussed above which has a race hazard resulting from a "runaway" train. The states in Figure 45 (which are based on the Petri net pictured at the right) demonstrate that there are two unique manifestations of failures (i.e., critical and non-(safety)-critical). In considering the criticality of timing, we see that the slow firing of transition *Close* makes it possible for the train to enter the intersection before the gate has properly (or completely) closed. Similarly transition *Open* makes it possible for the train to have departed and still, the gate is not open.

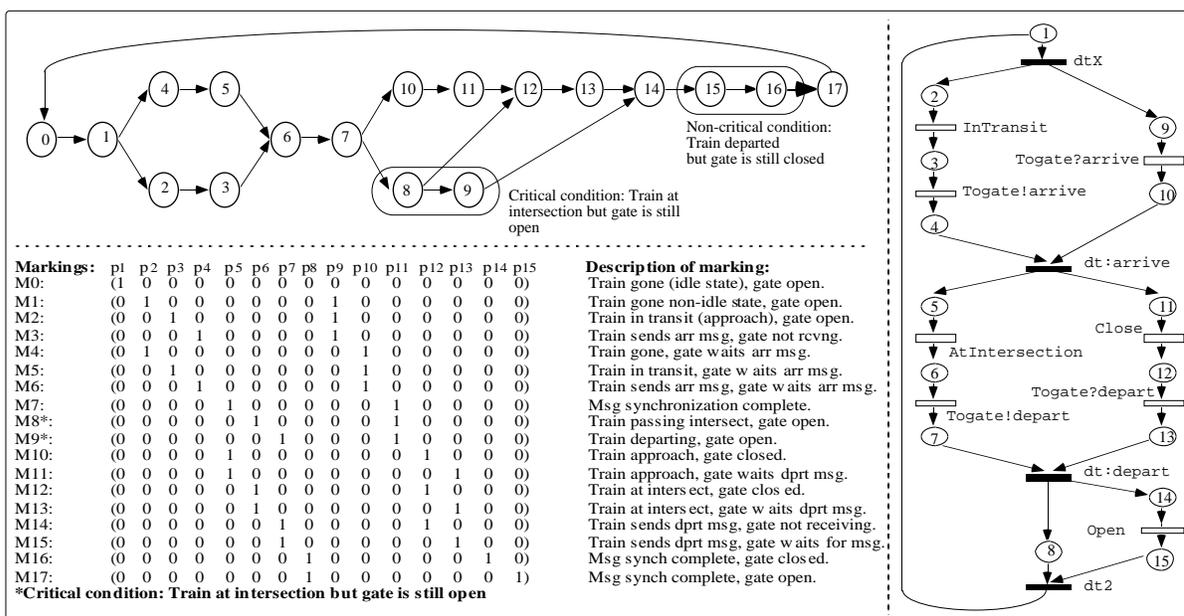


Figure 45. Markings and requisite Markov state transition diagram.

Missing from the Petri net of Figure 45 are transitions to reflect physical, communication related or mechanical failures. In our analysis, we do assume the existence of such failure transitions (and corresponding places) as discussed in the previous section (5.5.1). The CSP specification (and the corresponding Petri net) can be augmented to show how such failures should be handled. For example, the communication failures can be handled using time-out and re-transmit techniques. But still, should the gate fail to close, the question becomes

what can be done to possibly avoid a catastrophe. Perhaps an audible and visual alarm would alert unsuspecting pedestrians and traffic. Such fault-tolerant and fault-handling actions can be specified both with the CSP and Petri net models. However, they become more obvious by examining and analyzing the stochastic Petri net. The cost of providing fault-tolerance should be traded-off with the required level of reliability.

5.6 Parametric Sensitivity Analysis

Using conventional techniques such as those used by stochastic Petri net tools (e.g., SPNP), discrete and continuous analyses can be performed.¹⁵ For the purpose of this presentation, we have computed reliability of the train crossing with different failure rates (or probabilities) and service rates (e.g., speed of the train, rate at which the gate mechanism operates). The values used in this paper (and hence the results of the analysis) are only for illustrating the approach. It is not our intention to attach significance to the failure rates, MTTFs obtained, or the probability of detected and undetected failures. These analyses are useful in exploring different fault-handling mechanisms and the cost-benefit of providing fault tolerance. The following subsections outline the discrete and continuous analyses.

5.6.1 Discrete Analysis

Table 9 presents the probability assignments for our test runs of the train crossing ignoring deadline related failures (i.e., $P_{tf} = 0$). Four different trials were run with differing failure probabilities where P_c = communication failure, P_m = mechanical failure (either in opening or closing the gate). In all runs $P_m > P_c$, and in order to reduce the probability of critical failure in runs 2 - 4, we set $P_m(\text{close}) < P_m(\text{open})$ by the factors of 100, 3 and 5 respectively. Using fault-tolerant methods such reliability improvements are possible. Consequently, the probability of critical failures (P_{cf}) are reduced by the factors of 17.573, 1.975 and 2.974 respectively. Such analyses showing the magnitude of improvement

¹⁵The classic steady-state solution method for stochastic models that maps GSPN models to CTMCs is compared with a method based on DTMCs in [Ciardo89]. The DTMC method is shown to perform better.

TABLE 9
DISCRETE ANALYSIS ($P_{ff} = 0$)

Desc.	Run 1	Run 2	Run 3	Run 4	Desc.	Run 1	Run 2	Run 3	Run 4
Pc	.0001	.00001	.0001	.00001	Pcf	0.5026	0.0286	0.2544	0.1690
Pm(clo)	.01	.00001	.01	.001	Pncf	0.4974	0.9714	0.7456	0.8310
Pm(op)	.01	.001	.03	.005	MTTF	490.26	9524.07	248.19	1656.21

associated with a given design improvement can be useful in deciding what level of fault tolerance is appropriate. Note, P_{ncf} is the non-critical failure probability and the MTTF is given in the number of discrete steps (or time units).

5.6.2 Continuous Analysis

The results of the continuous analysis are shown in Figure 46. These results are based on the CTMC shown in Figure 45. The mechanical (λ_m), communication (λ_c) and timing (τ) failure rates are shown associated with their transition arcs. The trade-off between the rate of train arrivals (μ_1), speed of the train (μ_3), service rate of the gate mechanism (μ_6, μ_9) and the failure rates were investigated.

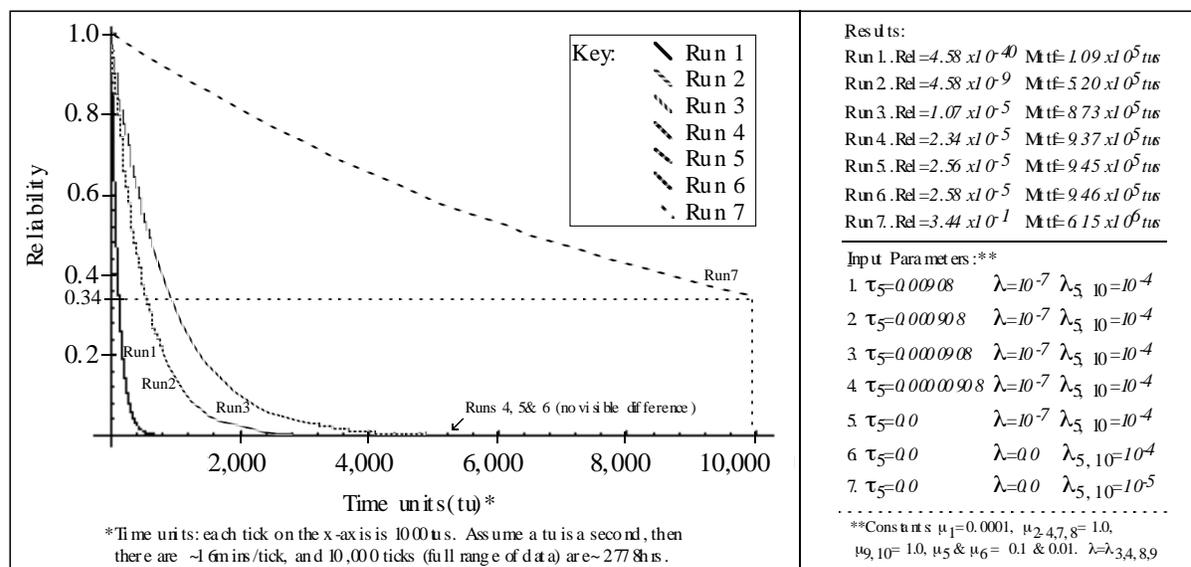


Figure 46. Results of the continuous analysis.

The unreliability of communications do not significantly impact the MTTFs because we have set those failure rates much lower than the rates associated with the gate's open/close mechanism by a factor of 1,000 (i.e., $\lambda_m = 0.0001 > \lambda_c = 0.0000001$). Mechanical failures and the possibility of the gate not closing (opening) in time (before the train arrives at the intersection) are assumed to be greater. In Figure 46 an interesting relation is evident. We observe that, if the train's speed tends to bring it to the intersection sooner than the gate can close, then an improvement in the gate's mechanical reliability doesn't really help! To improve the overall system's reliability it is more important to provide the additional synchronization between the train and gate processes as described in Section 5.5 (and Figure 43), so as to avoid the possibility of having the gate miss its deadline (τ_5). Alternatively, the train may signal "arrive" much sooner, allowing ample time for the gate to close.

In general, it is important to see how much the least reliable entity impacts the overall system reliability. In Figure 46, there are incremental improvements seen in the reliability of the system at 10,000 time units from 10^{-40} to 10^{-5} for various values of τ_5 (which reflects the probability that the train arrives before the gate closes). The next most significant gain in system reliability comes when the gate's mechanical failure rate is improved by a factor of ten (note the difference between run 6 and 7 in the graph). In this case, the MTTF improves by 6 times while the corresponding system reliability improves significantly from $\sim 2.6 \times 10^{-5}$ to $\sim 3.4 \times 10^{-1}$.

CHAPTER 6

CONCLUSIONS

Things which matter most must never be at the mercy of things which matter least.

– *Goethe*

6.1 Conclusion

The objective of this work was to show how CSP specifications can be translated into SPNs for the purpose of reliability and performance analyses. This objective was met with the construction of the CSPN tool. Such translations can give (1) insight into the feasibility of meeting non-functional requirements, (2) help to identify the best candidate design, (3) help to identify failure modes, and (4) to provide a means for describing how fault handling mechanisms can be incorporated as a part of the CSP specification. This approach enables the stochastic properties of the system specification to be ascertained while allowing the parameters used in the analysis to be formally captured in the P-CSP design specification. Subsequent analyses can then be run without having to rewrite all of the pertinent values. Only those parameters that are identified as critical in terms of their impact to the integrity of the overall system (i.e., sensitivity analysis) need be perturbed. The parameters (e.g., timing delays, probabilities, and rates) which are selected for sensitivity analysis are then considered in terms of their impact on system reliability and performance. In addition, these same parameters can be correlated to cost as is shown in [Sheldon95]. In general, this approach provides the designer with an analysis tool that facilitates judicious cost-benefit trade-offs in terms of how structural changes in the design specification will satisfy system's requirements (e.g., providing fault-avoidance and fault-tolerance).

A textual language for CSP specifications was designed. A software tool was

implemented for translating the CSP specifications into stochastic Petri nets. The Petri nets are coded in the form of a coincidence matrix. The graphical representation of the resulting Petri net can be viewed using the *dot* tool [a Unix filter for drawing directed graphs].¹⁶ The system coincidence matrix is converted into a file format needed for analysis using SPNP.

The tool has been tested using a diversity of process compositions and nesting of compositions. Some validation testing has been employed with the goal of determining how similar the resultant Petri nets are to those which motivated the CSP specification [Trivedi93]. Thus, some well known example Petri nets were first manually coded into P-CSP specifications and then translated back into Petri nets using the CSPN tool. The original Petri net was then compared to the translated Petri net. Except for additional dummy transitions and places which are the artifacts of the canonical translation rules, the Petri nets which were generated by the CSPN tool were equivalent to the original Petri nets.

6.2 Future plans

This work can be extended to incorporate a broader scope of translations and the characterization of properties other than structural that are useful for error avoidance, fault tolerance, detection of deadlocks and unsafe behaviors, and timeliness. Other issues include (1) ease of use (e.g., GUI) including mechanisms for detecting characteristics of the Petri net that can be used in automatically¹⁷ parameterizing the SPNP formatted file, (2) relating the analysis results back to the original specifications in a more rigorous and formal way, (3) expanding the language to incorporate some of the ideas of real-time CSP and others, (4) developing some state reduction techniques for the CSPN (e.g., combining dummy transitions with tangible transitions) and (5) validating our approach by applying the method to larger examples and/or a real system.

¹⁶See Drawing graphs with dot by Eleftherios Koutsofios and Stephen C. North at AT&T Bell Laboratories.

¹⁷Currently, CSPN uses a hard coded set of defaults that define the *parameters* part of the SPNP output file. Those defaults can be changed interactively using the "-v" verbose mode flag on the command line. For instance, in nets which generate absorbing states it only makes sense to run a transient analysis.