Table of Contents

1 Introduction....1
- 1.1 Intended Audience....1
- 1.2 Applicability....1
- 1.3 Purpose....1
- 1.4 Document Usage Description....1
- 1.5 Conventions....2
- 1.6 Problem Reporting Instructions....2

2 Using CSPN....2
- 2.1 CSPN Tool Overview....2
- 2.2 Translation Phases of the CSPN Tool....3
- 2.3 Running the CSPN tool....5
- 2.4 CSPN user interactive mode....6
- 2.5 CSPLTOPS....6
- 2.6 CSPN Data Structures....7

3 The CSP-based Language (P-CSP) Primitives....7
- 3.1 Stochastic Petri Nets....8
- 3.2 SPNP and the C-based Stochastic Petri net Language (CSPL)....9
- 3.3 Mapping CSP to Petri nets using canonical translation rules....10

4 Some CSPN Mechanics....11
- 4.1 Coincidence Matrix Expansion....11
- 4.2 CSP represented as a network structure....13

5 An Example: The Computer Bus Arbiter System....15
- 5.1 Formulate a CSP System Description....15
- 5.2 Translate the P-CSP specification to a stochastic Petri net....16
- 5.3 Assign performance and reliability parameters....16
- 5.4 Analyze the Petri net for stochastic properties....17
- 5.5 Perform reliability analysis....17
- 5.6 Augment the original P-CSP specification....17
- 5.7 Look at cost impacts of changes....17

References....19

APPENDIX A: Available Parameter Options....20

APPENDIX B: CSP-Based Grammar and Constructs....22

APPENDIX C: CSP to Petri Net Translations....24

APPENDIX D: Description of CSPN Program Files....29

1 Introduction

This manual is a reference specification intended to provide the necessary information to enable a user to understand how to run the CSPN software tool. There is a ReadMe, Source-lst, Makefile and two basic examples included with the source files. There are additional components intended for use with CSPN, which are the property of other researchers and/or organizations. See the Source-lst for third party software components that are needed to make the CSPN tool fully featured (most are freely distributed [except SPNP][1]. CSPN for all intensive purposes runs on Unix. At the time of this writing a PC version was being considered.

1.1 Intended Audience

The intended audience includes the beginner who has had introductory courses in software design/analysis and software engineering. This manual briefly covers some of the language and symbol conventions commonly used in the software design area. For more information, the reader is directed to the referenced documents in the bibliography.

1.2 Applicability

This software user's manual version 1.0 covers version 3.0 of the CSP-to-Stochastic-Petri-Net (CSPN) tool, which runs under Solaris 2.5.1.[2]

1.3 Purpose

The purpose of this software user's manual is to:

- Explain to beginners how to use the CSPN tool to convert specifications into Stochastic Petri-net (SPN),
- Provide information with respect to the prior version of CSPN in regards to enhancements,
- Provide background material with respect to stochastic modeling and analysis,
- Enumerate the current set of features (old and new) including menus and possible execution scenarios,
- Describe the basic P-CSP language primitives and formally define the P-CSP grammar (using Backus-Naur Form, BNF),[3]
- Exemplify the automated process which is supported by CSPN,
- Exemplify model creation using P-CSP.

1.4 Document Usage Description

This manual provides a tutorial on how to use CSPN. Chapter 2 provides a general

---

[1] SPNP is stochastic Petri Net Package and is available from Professor Kishor Trivedi (kst@ee.duke.edu).

[2] CSP stands for Communicating Sequential Processes [Hoare 85]. This tool is written in ANSII C, and will run on most versions of Unix (it was developed under Solaris 2.5.1, and has been run on DEC's OSF/1 Unix. CSPN is planned to run on the Wintel (in process) and Macintosh platforms.

[3] P-CSP is the Parsable CSP-based specification language used by CSPN for formal specification based stochastic analysis and design.

overview of how to run the CSPN tool. Chapter 3 goes into detail on the CSP-based language primitives and how to map from CSP to Petri nets using canonical translation rules. Chapter 4 covers some CSPN mechanics to familiarize the user with how the translation process works. An example run of the CSPN tool is covered in Chapter 5, to demonstrate some of the capabilities of CSPN. Following Chapter 5, we have listed some reference documents for those interested in further study in this area. Appendix A lists the available parameter options that the user can use to perform analysis and design of the specification.

1.5 Conventions

The symbols and conventions used in this manual are described in Appendix B, CSP-Based Grammar and Constructs, and Appendix C, CSP to Petri Net Translations.

1.6 Problem Reporting Instructions

Forward any comments or suggested changes to Dr. Frederick T. Sheldon via e-mail: sheldon@cs.uccs.edu or via regular mail: University of Colorado at Colorado Springs, Dept. of Computer Science, Attn: Dr. Frederick T. Sheldon, P.O. Box 7150, Colorado Springs, CO 80933-7150.

2 Using CSPN

In this section, we will cover

- An overview of the CSPN tool (Section 2.1),
- The translation phases of using the tool (Section 2.2),
- How to run the tool (Section 2.3), and
- The data structures that the tool uses (Section 2.4).

2.1 CSPN Tool Overview

The common steps used in the methodology for specification and analysis of system modeling are listed in Table 1. The CSPN tool augments this process by automating some of these steps.

TABLE 1. METHODOLOGY: STEPS FOR SPECIFICATION AND ANALYSIS

Step	Description of steps used 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 *cost*.

The core augmentation to existing approaches is provided by the CSP-based grammar and

canonical CSP-to-Petri net translation rules for process composition and decomposition. The translation rules are codified in the CSPN tool. In brief, the mechanism consists of decomposing individual CSP constructions into canonical Petri net structures (see Appendix C). The elemental structures are linked together in a hierarchical fashion according to their adjacency and nesting within the CSP specification (see Figures 4-7). 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 (see Figures 6-7). Also, as CSPN decomposes the CSP constructions, it identifies and records service and failure rate annotations, which are later incorporated in the SPNP specification file. When CSPN encounters failure annotations (and the "-f" command line option is set), it creates supplemental failure transitions. After the preliminary structure of the Petri net is complete, CSPN must reconcile synchronization points because CSP input/output actions must rendezvous at a particular point which is translated into a transition that is named by the message that is sent/received. There are two different synchronization translations possible (blocking and non-blocking, see Figure C-4). CSPN finally generates the Petri net graphic specification and an SPNP Petri net specification file named "<fn>_spnp.c" (fn is the name of the input file). All of this process occurs at various levels of user controllable interaction, as will be described later in this document. In essence, the approach provides for systematic and automatic translation and subsequent augmentation (e.g., failure rates, service rates, and deadlines) of the resultant Petri nets for assessing different candidate design specifications:

- Formally (as provided by the grammar) relating stochastic parameters back to the specification level;
- And analyzing the stochastic Petri nets using the SPNP tool [2-6, 18].

The CSPN tool is currently textual based, but future work will provide a graphical interface. The initial specification and parameterization work must be completed using a text editor. 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.

2.2 Translation Phases of the CSPN Tool

There are four basic activities (parts) involved in the context of Figure 1 (bottom half). The first part (1) involves the specification phase. The second part (2-7) involves running the CSPN tool which will invoke any of the set of available command line options defined in Table 2, and the de-/composition algorithms (translation codes). 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 Petri Net transitions) (see Appendix A). The fourth and last phase (11-12) concerns the structural and stochastic analysis of the Petri net. Structural analysis involves viewing the distribution of places and transitions of the graphical representation of the Petri net. The stochastic analysis involves running SPNP to derive dependability and performance results based on the prior phase of 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 the characteristics of the SPNP specification prior

to running the stochastic analysis. Once SPNP is run, the results can be considered in the process of conducting further analysis.

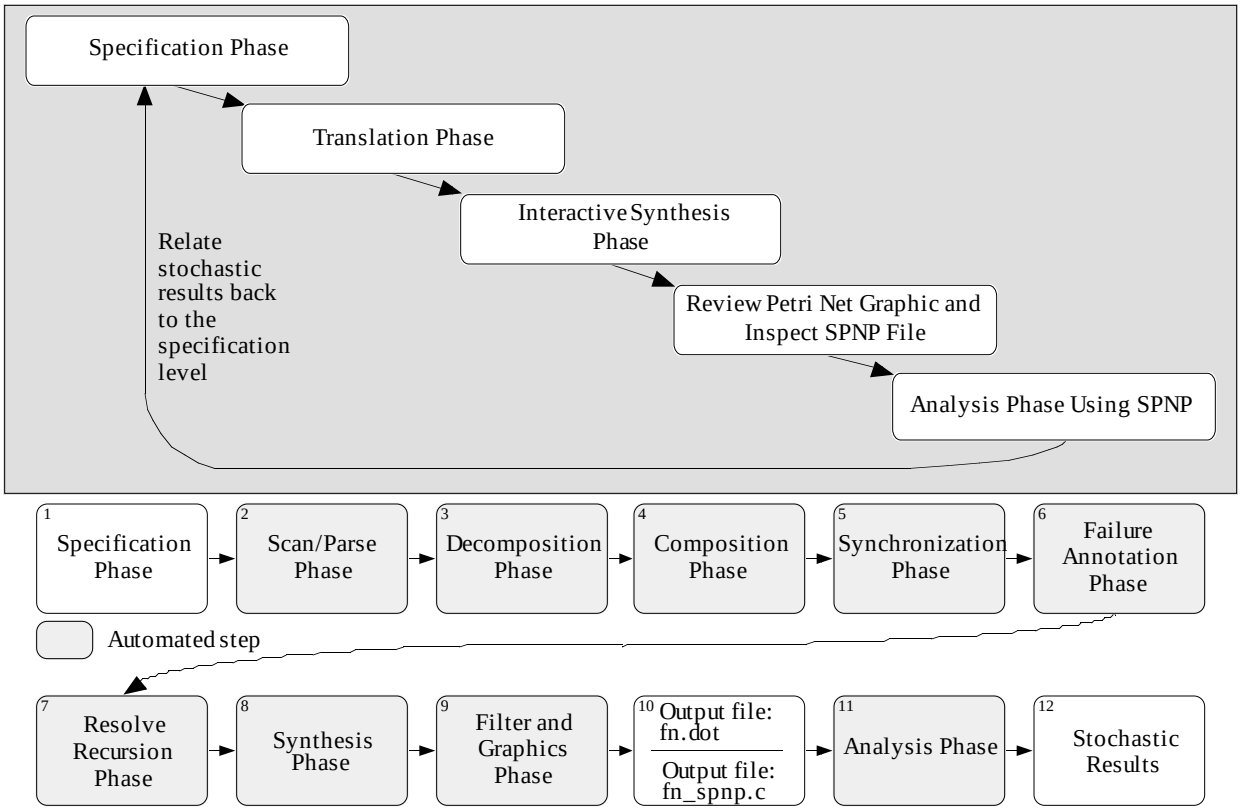


Figure 1. Translation phases of CSPN (top: general phases, bottom: detailed activities).

In viewing Figure 1 (bottom half), 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 semantics of the specification), (2) *Decomposition* (allocating/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* (assumes tail recursion and involves finding and linking loop-back arcs as well as breaking tail recursive loops), (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 1 shows the various translation phases and the use of SPNP as it applies to this approach.

TABLE 2. LISTING OF THE CSPN COMMAND LINE OPTIONS.

Option	Description
-a<number>	Rate for return to initial marking from absorbing markings (default is 0.000000).
-d	Set to generate a "dot" graphics file. Dot uses this digraph specification file to generate the graphical representation of the Petri net.
-f	Used to generate failure transitions into the *filename_spnp.c* 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 *filename spnp.c* with SPNP compliant characters (_i_, _o_, and _ respectively). Otherwise, SPNP will not compile the input file. Valid only when the "-o" option is used.
-j	Set to generate a "vgj" graphics file. VGJ uses this digraph specification file to generate the graphical representation of the Petri net.
-i<number>	Number of iterations used by SPNP (default is 2000).
-n	Set to enable a network list file. This file shows how CSPN has interpreted the structural aspects of the CSP specification.
-o<name>	To generate the SPNP input specification file (*filename_spnp.c*) this option <u>must</u> be specified ("name" is optional and the default used is the tool name "cspn").
-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).
-r	Use the default reward functions in the output (allow interactive choice).
-s	Use the default service rates for timed transitions. If no service rate is specified as an annotation, then CSPN will use 0.1.
-t	Set to generate a symbol table file. This file will contain all the data recorded for each element (process names, constructions, variables, channels, ...) of the process specification.
-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.

2.3 Running the CSPN tool

Running CSPN (i.e., *$> csp <options> specification-file*) and using the various command line options 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.) then adding the "-d" or "-j" 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 is given with this option (i.e., "-o *filename*"). This settles the problem of overwriting previous files generated using the default name that is assigned by CSPN when 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", "-j", "-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.

2.4 CSPN user interactive mode

The cspn tool allows the user to adjust CSPL file description values for File parameters, transition priority values, transition service rates and CSPL reward function from the default values used by the tool. The user interactive mode is invoked when the -o (output file generation) switch is used in conjunction with a any of the following switches: -v (verbose mode for changing CSPL parameter values) -e (allows the user to adjust the transition priority values) -s (allows the user to adjust the transition service rate values) -r (allows the user to adjust the CSPL reward functions generation) any one or a combination of any of these switches will invoke the interactive user function. When this function is invoked, a menu is presented to the user with 4 choices: 1) The user may run the tool with different values than the default ones, 2) the user may change default values and save them to a user defined value file, 3) the user may run the tool using a previously created user defined value file, or 4) the user may edit the tool values based on the values of a previously created user defined value file. The user is allowed to keep up to twenty user defined value files in the working directory. An over write process of files begins when file 21 is created (the user is warned of the impending overwrite and offered the option to exit the tool at this point) allowable file names are fn_def.0 through fn_def.999 where fn is the P-CSP input specification file name. Only the values for those command line switches that were used for the current tool execution are presented to the user for value adjustment. As far as how the values are changed, this is still the same as your original design with one addition, when in verbose mode (-v switch) I have added a help selection for each parameter in the different parameter choice selection menus. When selected (help that is) a message giving a short description of the parameter and the parameter value choices is displayed (these are the same as the parameter descriptions in the CSPL um appendices).

2.5 CSPLTOPS

The *cspltops* tool allows for a PN graph to be created based on a CSPL model description. This allows the user to compare the PN graph created by the CSPN tool with the CSPL file it also created graphically by the user. The cspltops tool has three command line switches: -v verbose mode - this allows the user to watch the net list created by the tool scroll up the screen while it is being generated, -h help - displays command line usage of the tool and a brief description of the command line switches, and -d display - allows the user to display the PN graph using ghost script, ghost view or page view through a user selection menu. When the -d switch is used the tool can only be exited by selecting the exit choice from the user display selection menu (or you could use the old ctrl-C).

The use of cspltops by the CSPN tool is invoked using the -x switch, it can only be used when the -o switch is also set on the command line (we need to create the CSPL file i.e. fn_spn.c before we can read it using cspltops). CSPN calls the cspltops tool using a command line switch -

d so that the user can view the resulting PN graph without having to exit the CSPN tool and invoking a graphical viewer on their own.

2.6 CSPN Data Structures

Internally, there are four basic data structures employed by CSPN: (1) a *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) a *coincidence matrix* which maintains the graphical distributions of places, transitions and their connectivity.

The coincidence matrix (or co-matrix) is the Petri net. Each element and composition from the CSP specification has a co-matrix maintained in the symbol table. The construction of the n-by-m co-matrix is defined in terms of the transitions (i.e., CSP-process names become transition names). Transitions are associated with rows (from top to bottom). Places are associated with columns (numbered from left to right starting from zero). A non-zero element in the matrix A represents an arc which links a transition to a place. Elements (a_{ij}) can have one of three values (zero, +1 or -1): $a_{ij} = +1$ indicates an arc *from* the transition of row i *to* the place of column j; $a_{ij} = -1$ indicates an arc *to* the transition of row i *from* the place of column j. The process list stores the transition names in their proper order. The numbering of places is ordered, but their semantics are defined in terms of the transitions.

3 The CSP-based Language (P-CSP) Primitives

Systems are built from processes. The simplest process is an action (an assignment, input or output). SKIP and STOP are two special processes: they both perform no action (i.e., engage in no event), but SKIP terminates while STOP does not terminate (engages in infinite internal actions) causing a deadlock. Larger processes are built by combining smaller processes. PAR (or ||), SEQ (or ;), NDC (or), DC (or), and MU.x{} (or $\mu X.$ P) are the constructors that can be used for this purpose. The complete CSP-based grammar is provided as a yacc (viz., BNF) specification in [20, 21].

An example declarative construction would be: PROCESS My_example = SEQ{P(), Q(), R()}; where the process "My_example" is declared as a sequential composition of three process calls (P, Q, R). A process call is one of the main elements used to denote processes. Processes can have internal structures which are defined in a PROCESS declaration and used later much like a function call in a typical programming language (e.g., My_example()). The simple rule to remember is that when a process call is made inside a declarative construction (i.e., between "PROCESS =" and ";") it need not be pre-declared (i.e., the process can defined on-the-fly rather than being pre-declared). However, when a process is pre-declared, it must subsequently be used as a "process call." In the example below, P1() and all the other elements in similar format, having "()" appended to their name, are process calls. Process calls in the main body of the specification (last line of this example) *must* be pre-declared:

```
Example_System =
                PROCESS My_example = SEQ{P1(), P2(), P3()};
                PROCESS Your_example = PAR{Q1(), Q2(), My_example()};
                PROCESS Monitor = NDC{SKIP, R1, R2()};
Mu.X{ NDC{ My_example(), Your_example()}}.  --Dashes are prefixed to comments
```

Pre-declaring a process is a way of abstracting away the internal details of the process function. Translation of a pre-declared process call expands all of its internal structure in creating a "sub-Petri net." In this way, larger processes are formed from the composition of smaller processes. A statement list is a sequential list of $n \geq 1$ statement(s). A statement can be an event (or trigger) which causes a process to engage in an action (e.g., $a \rightarrow P$). This process is defined as an implication. Input and output of messages require a channel. Channels provide unbuffered, unidirectional point-to-point communication of values between two concurrent processes (similar to Ada rendezvous). A guarded process combines one or more processes, each of which is conditional on an input, a Boolean expression, or both. An expression can be integer, Boolean or relational (Boolean expressions must consist of Boolean variables prefixed with "@"). Operands can be integers, variables, integer expressions, or relational expressions (distinct from Boolean). A partial BNF specification of the P-CSP grammar can be found in the Appendix.

3.1 Stochastic Petri Nets

The Petri net in its simplest form is a directed bipartite graph, where the two types of nodes are known as *places* (circles) and *transitions* (bars). In this approach, places represent *events*, while transitions represent actions. Other researchers have based their system models on conditions and events (where their events are similar to P-CSP's actions or processes). However, in this approach, modeling is based on the notion in CSP of event-action pairings.

A Stochastic Petri net (SPN) is simply a Petri net which has been extended to permit stochastic analysis. These extensions embed the model into a timed environment by associating a time to each of the transitions in the net. The most general extensions allow the usage of stochastic times (rates) and probabilities. The underlying stochastic process is captured by the "extended reachability graph" (ERG), a reachability graph with additional stochastic information on the arcs. The ERG has been shown to be reducible to a Continuous Time Markov Chain (CTMC), provided that the exponential distribution is associated with the Petri net transitions. Since SPNs permit a probability distribution to be associated with transitions (to express delays or failure rates), they are very suitable for modeling system performance and reliability. Stochastic Petri net markings correspond to the states of an equivalent stochastic process. The transition rate from state M_i to M_j (of the equivalent stochastic process) is given by $q_{ij} = \lambda_{i1} + \lambda_{i2} + \ldots + \lambda_{im}$ where λ_{ik} is the delay in firing a transition t_k which takes the Petri net from marking M_i to M_j (when several transitions enable the firing from M_i to M_j). See [1, 6, 9-10, 14-15, 22] for more details on Petri nets, SPNs, Markov, and Markov Reward processes.

3.2 SPNP and the C-based Stochastic Petri net Language (CSPL)

The SPNP package allows the user to perform steady state, transient, cumulative transient, and sensitivity analysis of SRNs [23]. The language used for describing stochastic Petri nets for SPNP (Stochastic Petri Net Package) is CSPL. It is a super set of the C language and provides the full expressive power of C. Predefined functions are available to define SPNP objects. A single CSPL file is sufficient to describe any legal SRN because the SPNP user can input (at run-time): the number of places and transitions, the arcs among them, and any other required parameter. The numerical parameters used in the specification of rates and probabilities are incorporated in the same CSPL file. An example of the CSPL file structure is shown in Figure 2.

The function *parameters* allows the user to customize how the package will perform the analysis by setting specific call parameters in the sub-functions iopt() and fopt(). Several parameters establishing a specific behavior can be selected [3]. The function *net* permits the user to completely define the structure and parameters of an SRN model. The basic functions that can be used inside the net include: place() for naming all the places, trans() for naming all the transitions, iarc() for defining a transition's input arc, oarc() for the output arcs, and init() which defines the initial marking. Probabilistic behavior may be specified using probval(); the timing of events can be specified by assigning rates to the transitions in rateval(). More advanced functions include harc() for making inhibitor arcs while the functions miarc(), moarc(), and mharc() define multiple cardinality input, output and inhibitor arcs (these more advanced functions are not synthesized by CSPN during the translation process).

```
parameters(){
        iopt(IOP_PR_MARK_ORDER, VAL_CANONIC);
        iopt(IOP_PR_MERG_MARK, VAL_YES);
        iopt(IOP_PR_FULL_MARK, VAL_NO);
        iopt(IOP_PR_RSET, VAL_NO);
        iopt(IOP_PR_RGRAPH, VAL_NO);
        iopt(IOP_PR_MC, VAL_NO);
        iopt(IOP_PR_MC_ORDER, VAL_FROMTO);
        iopt(IOP_PR_PROB, VAL_NO);
        iopt(IOP_MC, VAL_CTMC);
        iopt(IOP_OK_ABSMARK, VAL_NO);
        iopt(IOP_OK_VANLOOP, VAL_NO);
        iopt(IOP_OK_TRANS_M0, VAL_YES);
        iopt(IOP_METHOD, VAL_SSSOR);
        iopt(IOP_CUMULATIVE, VAL_YES);
        iopt(IOP_SENSITIVITY, VAL_NO);
        iopt(IOP_ITERATIONS, 2000);
        iopt(IOP_DEBUG, VAL_NO);
        iopt(IOP_USENAME, VAL_NO);
        fopt(FOP_ABS_RET_M0, 0.000000);
        fopt(FOP_PRECISION, 0.000001);
}
net(){
        /* Definition of places */
        place("p0");
        init ("p0",1);
        place("p1");
        place("p2");...

        /* Definition of transitions */
        trans("dt1");
        trans("InTransit");
        trans("Togate_o_arrive");
        trans("dt_arrive");..

        /* Definition of rates */
        probval("dt1",1.0);
        rateval("InTransit",20.00000000);
        rateval("Togate_o_arrive",19.00000000);
        probval("dt_arrive",1.0);
        rateval("AtIntersection",18.00000000);

        /* Definition of input arcs */
        iarc("dt1", "p0");
        iarc("InTransit", "p1");
        iarc("Togate_o_arrive", "p2");..

        /* Definition of output arcs */
        oarc("dt1", "p1");
        oarc("dt1", "p8");
        oarc("InTransit", "p2");..
}
assert() {
  return(RES_NOERR);
}
ac_init() {
  fprintf(stderr,"\n<<<Run title goes here>>>");
  fprintf(stderr,"\nGenerating SRN data ...\n\n");
  pr_net_info();
}
ac_reach() {
  fprintf(stderr,"\nThe reachability graph is being
  fprintf(stderr,"generated ...\n\n");
  pr_rg_info();
}
/* - reward_type definitions go here --------------
ac_final(){
        int i;
        time value( 0.1 );
        pr_mc_info();
        pr_std_average();
        pr_std_cum_average();
}
```

Figure 2. SPNP input file structure.

3.3 Mapping CSP to Petri nets using canonical translation rules

An initial set of rules for translating CSP specifications into Petri nets (Petri nets) is defined in [7, 10, and 16-17]. The translations between CSP and Petri nets are based on the fact that in CSP, processes execute actions, which in turn may enable other actions, and, in this way, CSP processes move from one action to another. Activities that enable a process to be activated can be viewed as conditions (or events) which are represented by places, while the actions themselves are viewed as transitions. Some example translations are shown in Figure 3 (under each of the Petri net constructions is the P-CSP specification).

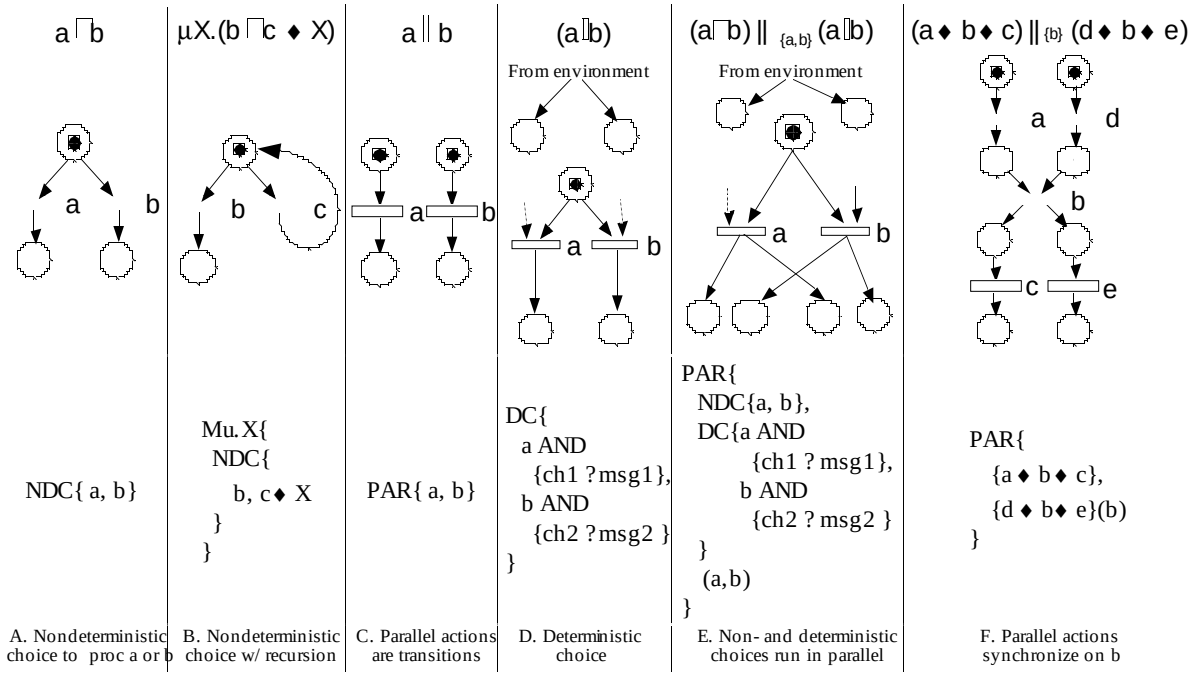


Figure 3. Example of some representative CSP → Petri net translation rules (see next figure).

The CSP to Petri net translations were designed to facilitate the automatic decomposition of the CSP constructs into Petri net sub-components and subsequent re-composition of the subnet components into a complete *system* Petri net. The Petri net translation from a given CSP construction (i.e., specification) need not be unique because ultimately, when we combine the subnets, we must introduce dummy places and dummy transitions to maintain the complete Petri net's bipartite nature. And, once the complete *system* net is obtained, the structure itself may be reduced (e.g., by combining adjacent dummy transitions or collapsing such places and transitions into their predecessor/successor transitions) to something that is trace equivalent to the CSP specification. This in itself is all that is necessary to define a complete set of markings and hence an equivalent Markov process.

Petri nets are inherently non-deterministic and asynchronous while CSP is inherently deterministic and synchronous (though an explicit definition of non-deterministic choice exists in CSP). Since the purpose is stochastic analysis, we depend on the non-deterministic nature of the Petri nets to conduct the stochastic analysis. This implies that the determinism of CSP is also

translated into the non-determinism of Petri nets. However, the resolution of structure within the translation is standardized and deterministic, while the probabilistic transitioning is only applied to the appropriate nondeterministic choice composition operator of CSP. In any event, the goal is to demonstrate the feasibility of translating from CSP and Petri nets by decomposing a CSP specification into its component parts (processes, channels, constructors, etc.), and this is done by choosing one standard (canonical) translation path from among equivalents. Section 4 provides some mechanisms and conventions, which have been defined, for the canonical translations.

4 Some CSPN Mechanics

This section provides some of the CSPN implementation details. The canonical translation rules are codified in CSPN. The set of details provided give the basic framework for CSPN.

4.1 Coincidence Matrix Expansion

When the P-CSP specification is parsed, each construct (e.g., PAR, SEQ, etc.) of the specification is separated into its component elements (process names, channels, variables) and represented as a sub-Petri net. The sequential construction shown in Figure 4 illustrates how the co-matrix is used to represent, in this case, a SEQ composition as a Petri net. Figure 5 shows the same translation for the parallel (i.e., PAR) construct.

Combining the component co-matrices to produce a complete system Petri net is a process of co-matrix expansion. The expansion is constrained in two dimensions to preserve the algebraic structure associated with (1) adjacency and (2) nesting. Figures 6 and 7 give an illustration of the expansion process.

The test shown in Figure 6 is designed to determine, based on the location of the transition to be expanded, which method of expansion to use. In Figure 7, an expansion is performed using method 3. Inserting SEQ1 expands the co-matrix SEQ0 (analogous to co-matrix A in Figure 6). SEQ1 consists of two processes, P1 and P2 (and is analogous to co-matrix B in Figure 6). The expansion must replace the transition SEQ1 by the two process names P1 and P2. The final *combined* result retains the SEQ0 name.

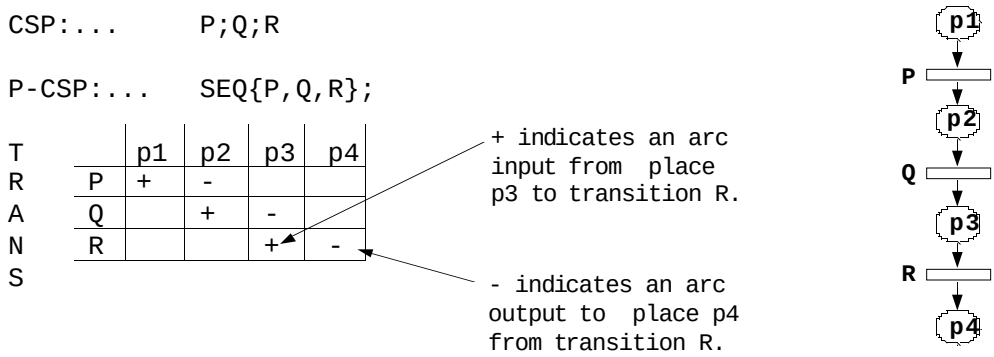


Figure 4. SEQ construct with co-matrix and Petri net representations

```
CSP:...      P;Q||R

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

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

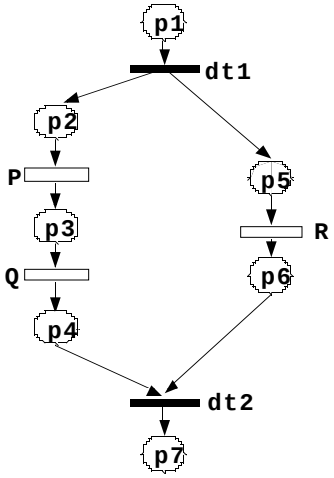


Figure 5. PAR and SEQ construct with co-matrix and Petri net representations.

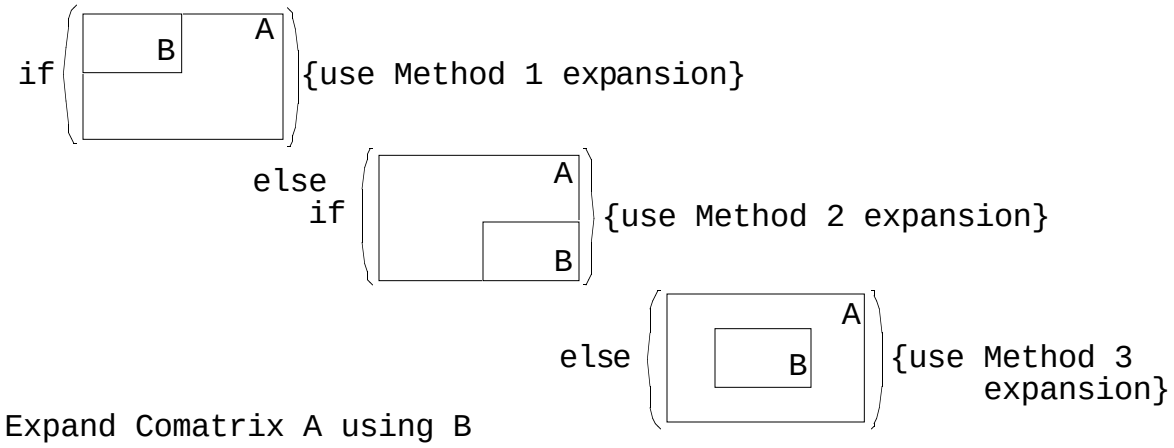


Figure 6. Choosing a combining method 1, 2 or 3 for expansion depends on locality.

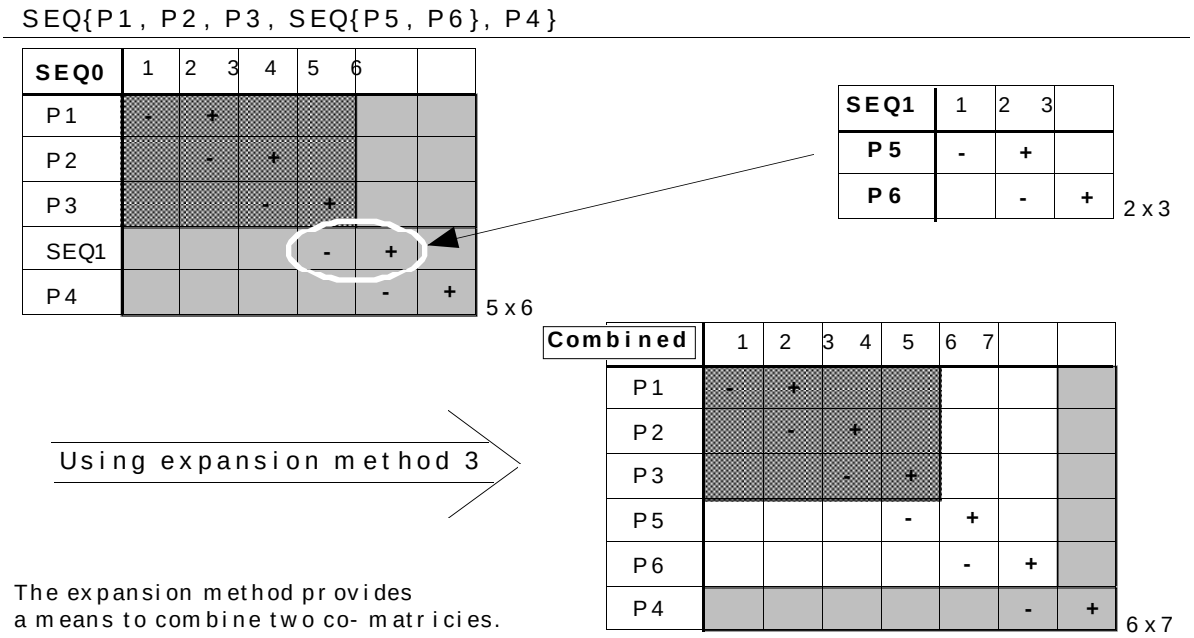


Figure 7. Identifying the transition to expand into a larger Petri net

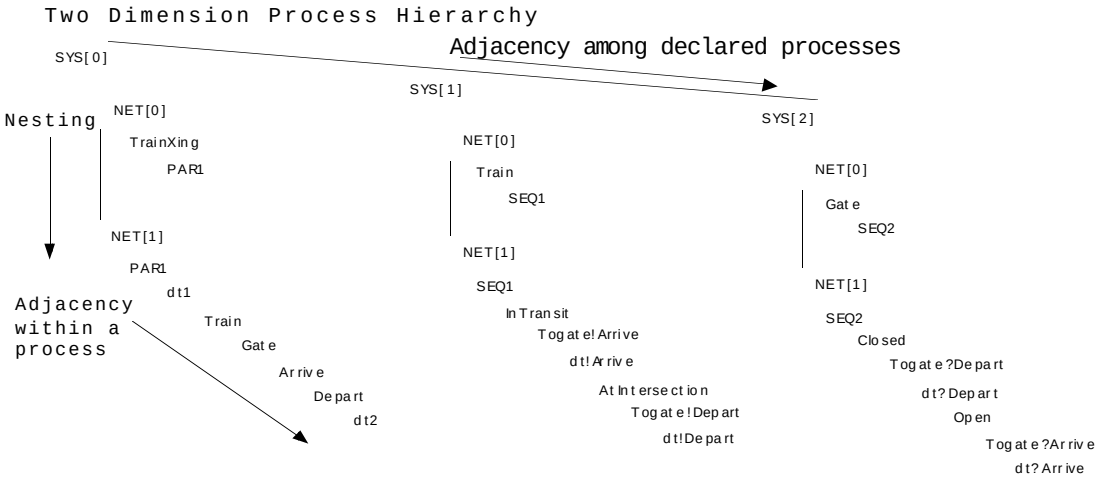
Note, the term SEQ is a key word (used for sequential composition of processes), and is considered itself to be a process. CSPN treats each occurrence of this type as a unique process by appending a unique value to the name (e.g., 0 is appended to the first occurrence of SEQ to give SEQ0, and the next occurrence of SEQ will have "1" appended). In this way, CSPN tracks each occurrence of a given type of keyword (i.e., SEQ, PAR, NDC, DC, STOP and SKIP).

4.2 CSP represented as a network structure

A network of linked lists is used to capture the algebraic structure in two dimensions (1) adjacency (among declared processes or within a process) and (2) nesting within processes. Two examples are provided in Figure 8. The first one emphasizes adjacency and the other emphasizes nesting.

There are two main network arrays used (1) SYS[] — each SYS[i] points to a process defined in a PROCESS declaration and (2) NET[] — each NET[i] points to a "process" element as defined by the P-CSP grammar (e.g., process call, constructor like PAR, NDC, etc.). Figure 9 gives some details of the three major C data structures employed by CSPN: (1) Symbol table entry (as described above) as well as (2) NET_NODE and (3) NODE structures, which together provide the building blocks for the network.

```
TrainXing =
  PROCESS Train =
    SEQ{InTransit(),{Togate!arrive},AtIntersection(),{Togate!depart}};
  PROCESS Gate =
     SEQ{{Togate?arrive},Closed(),{Togate?depart},Open()};
  PAR{
    Train(), Gate() {arrive, depart}}.
```




```
SysSimpleEx =
  PROCESS Eg1 =
    SEQ{SEQ{SEQ{P1(),P2(),PAR{P11(),P12()}},P3(),P4()},P5(),P6()},SEQ{P7(),P8()};
  PROCESS Eg2 =
    SEQ{ P9(), P10() };
  PAR{ Eg1(), Eg2() }.
```

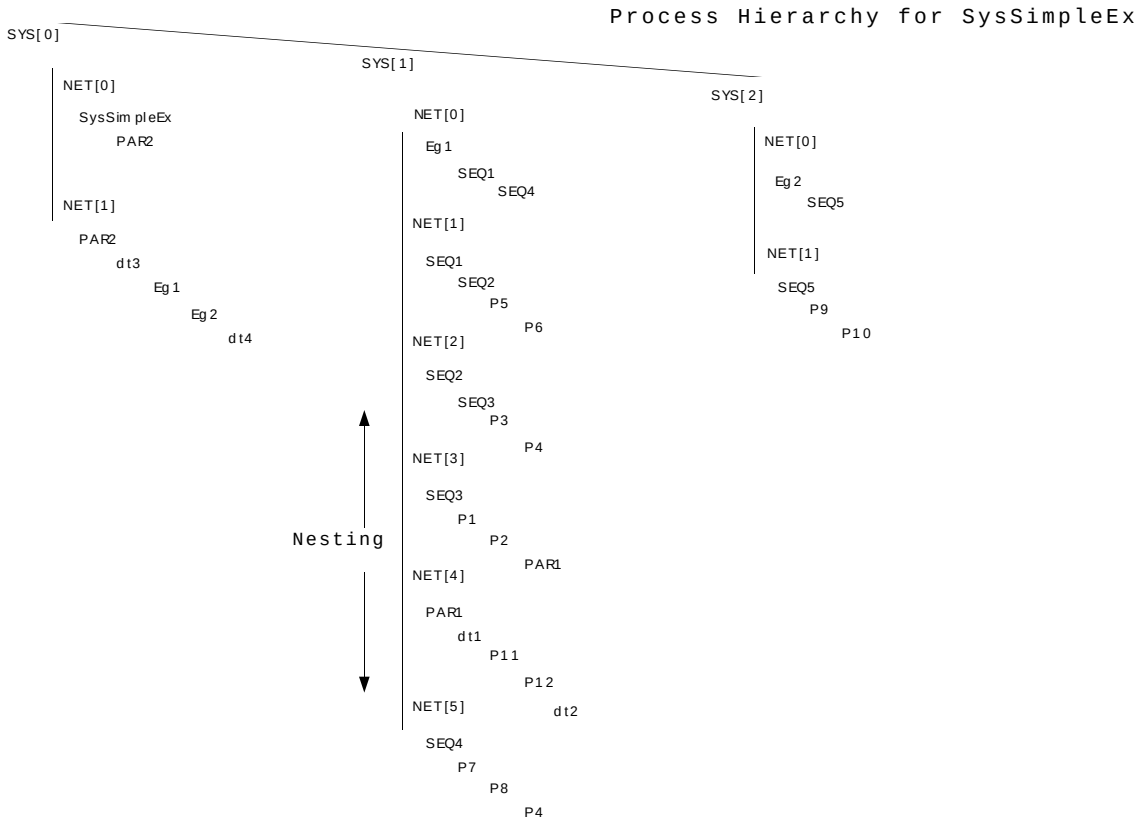


Figure 8. Process hierarchy for system "SysSimpleEx" with exaggerated nesting

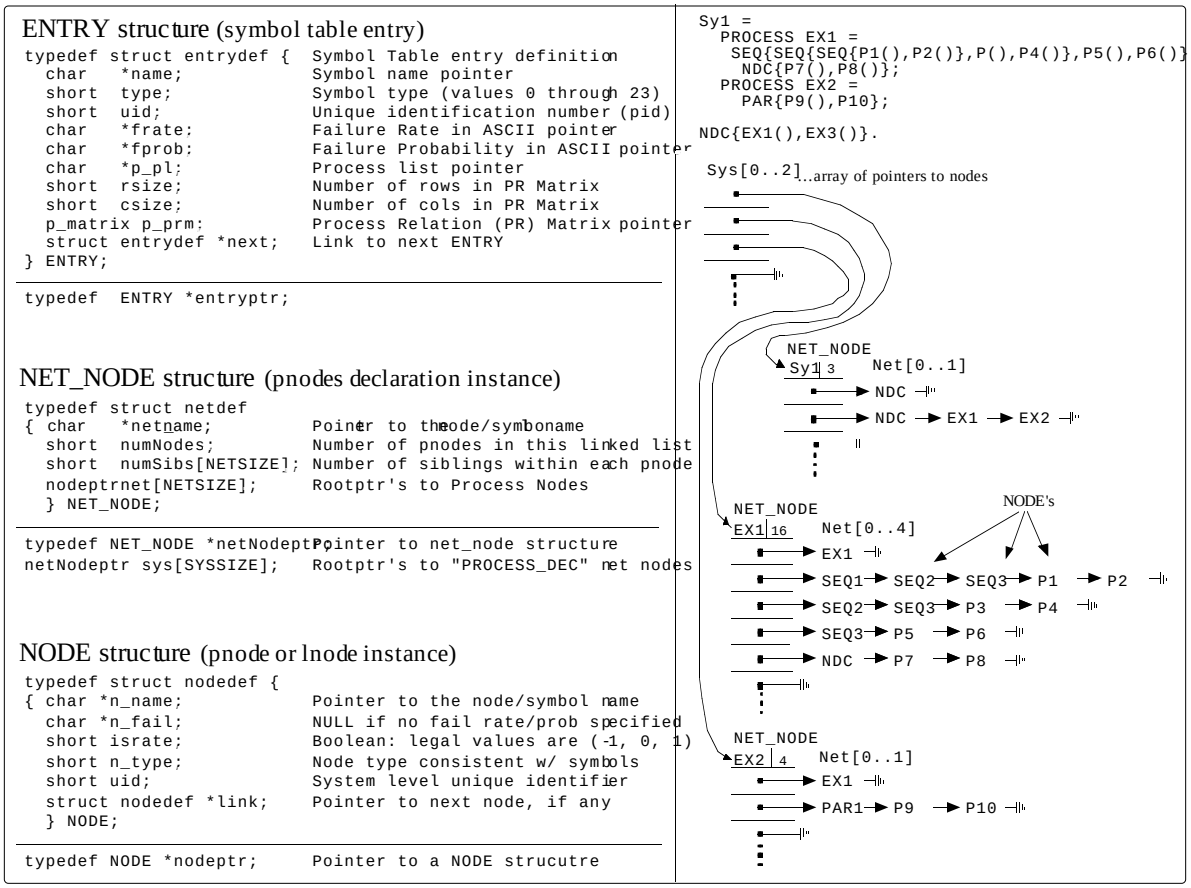


Figure 9. An example of the entry, net_node, and node data structures.

5 An Example: The Computer Bus Arbiter System.

In this example, we will take a P-CSP design specification for a computer bus arbiter and perform an analysis on it as described in Table 1, using the CSPN tool. The first step of the analysis results in the P-CSP specification in Figure 10.

5.1 Formulate a CSP System Description.

The system we are modeling has two processors and one arbiter using a shared bus. The processors are message synchronized with the arbiter for bus control. All three processes are running in parallel operation. In this scenario, the two processors are trying to perform a read or a write operation and must gain exclusive control of the bus to accomplish their task.

The message channels and operations used in the specification are as defined below:

- **Request** - request access channel uses messages uniquely identified to each processor.
- **Grant** - access granted channel uses messages uniquely identified to each processor.
- **Read** - read operation, each operation is uniquely identified to each processor.
- **Write** - write operation, each operation is uniquely identified to each processor.

- **Rel_Bus** - return bus control to arbiter, signal channel uses messages uniquely identified to each processor.
- **Update_PM** - update private memory operation, each operation is uniquely identified to each processor.

The processor processes P1 and P2 are implemented as follows: (1) the process starts by requesting bus access. (2) The process then waits until it receives a message from the arbiter granting access. (3) The process then performs a read or write operation, based on a non-deterministic choice. (4) When the read or write operation is complete, the process sends a message to the arbiter indicating that it is done and is ready to release the bus. (5) The process finishes by updating its private processor memory and returning to step 1.

The bus arbiter process is implemented as follows: (1) based on a deterministic choice, the process waits for a request for bus access from one of the processors. (2) The process grants access to that processor. (3) The process then waits for the processor to finishes and returns to step 1.

5.2 Translate the P-CSP specification to a stochastic Petri net.

Using a text editor, we create a design specification file like the one in Figure 10, a file, which we will name "arb" in this example. To use the CSPN tool on this file, we type "*csp arb*" at the prompt on the computer screen. CSPN generates a postscript file named "arb.ps", which is the Petri net shown in Figure 11. Note how the two processor process flows are rather symmetric, with the bus arbiter process in the middle negotiating between the two-processor processes through the use of messages.

5.3 Assign performance and reliability parameters.

We can now assign performance parameters, if necessary. To use user-defined values, we run the CSPN tool again but in the verbose mode, i.e. we run "*csp –o –v arb*". A menu will appear with the following options:

- Save user default values to arb_def.0
- Change default values without saving
- Use user defined defaults
- Edit and use user defined defaults

If we wanted to perform a detailed analysis, we can choose the last option and define our own defaults. The last option will save our values under the file name "*arb_def.x*", where the "*.x*" extension will be a value from 0 to 9. This allows the user to define several different sets of default values. After we have created a default-values file, we can choose to use one of these files by choosing the third option above. If we are just experimenting and do not want to save any of the values from our current run, then we would choose the second option above.

5.4 Analyze the Petri net for stochastic properties.

In this step, we see how its component parts affect the overall system performance. If one part is operating within tolerance but is operating at one extreme of its tolerance, does the overall system still work within its performance tolerance? Does the system even run to completion?

Looking at the Petri net in Figure 11, we can see the main bottleneck in this design is the common bus. Depending on the speed and synchronization requirements of the microprocessor processes P1 and P2, this system design may not meet its overall performance requirements. This design was used in early personal computer design. As performance requirements and capabilities increased, special buses like a dedicated graphics bus were required to deliver large quantities of data to peripherals without bringing down the performance of the system.

5.5 Perform reliability analysis.

From the last step, we can see that the single point of failure is the common bus. If the performance of the bus degrades or if the bus complete fails, the overall system reading and writing capability is greatly affected. We can look at the impact by changing the performance parameters of our model and analyze the effects. From that analysis, we can determine if the system performance requirements are met or if the system must be modified (e.g., add an additional bus).

Another critical area of system performance is the command messaging traffic. If the system loses the messages between the arbiter and microprocessor processes that grants access to and release the bus, the system comes to a halt. Again, we can analyze that using the CSPN tool. If required, we could at adding timeouts to the messaging system to avoid lost messages.

5.6 Augment the original P-CSP specification.

Based on results of our analysis, we can add the changes to the system specification listed in Figure 10. In this simple example, adding an additional bus requires only some minor text editing.

5.7 Look at cost impacts of changes.

The cost impacts of changing the specification and the system design can be significant, even in a minor design like our example. The earlier in the design process that the changes are made, usually the less expensive is the impact. Adding another data bus or changing the messaging scheme, when the system is in production or beyond, can require significant re-engineering, system testing, and documentation changes. Through the use of the CSPN tool, we can determine how the change will impact system performance and then compare that improvement against the cost to implement the change. The results from the CSPN analysis can help justify to management the need for the change and the additional cost.

```
Arbiter =

  PROCESS P1 =
    Mu.Z1 {
      SEQ  {
        {Request ! req1},
        {Grant ? grant1},
        NDC  {
          Read1( ),
          Write1( )
        },
      {Rel_Bus ! rel1},
      Update_PM1( )
      }
    };

  PROCESS P2 =
    Mu.Z2 {
      SEQ  {
        {Request ! req2},
        {Grant?grant2},
        NDC  {
          Read2( ),
          Write2( )
        },
      {Rel_Bus ! rel2},
      Update_PM2( )
      }
    };

  PROCESS bus =
    Mu.Y {
      DC   {
        Request ? req1  {
           Grant ! grant1,
           Rel_Bus ? rel1
        },
        Request ? req2  {
           Grant ! grant2,
           Rel_Bus ? rel2
        },
      }
    };

  Mu.X  {
    PAR  {
      P1( ),
      P2( ),
      bus( )    (req1, grant1, rel1,
                 req2, grant2, rel2)
    }
  }.
```

Figure 10. The Computer Bus Arbiter P-CSP design specification.

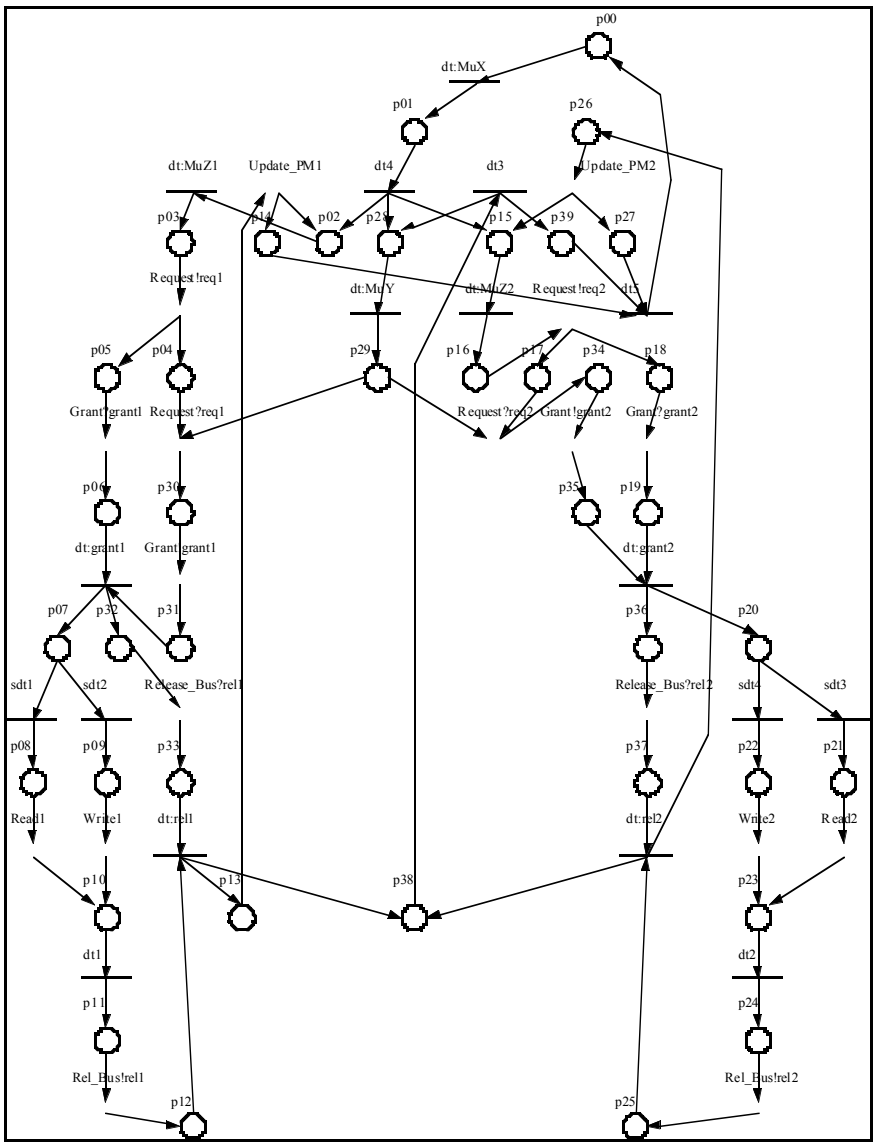


Figure 11. Petri Net for Computer Bus Arbiter.

References

1. Balbo, Gianfranco, "On the Success of Stochastic Petri Nets", IEEE Proceedings Petri Nets and Performance Modeling 95, Durham, NC, pp. 2-9, October 3-6, 1995.
2. Ciardo, Gianfranco and Trivedi, Kishor S., "A Decomposition Approach for Stochastic Petri Net Models", Proceedings of the Fourth International Workshop of Petri Nets and Performance Models, IEEE CS Press, Los Alamitos, CA, pp. 74-83, December 1991.
3. Ciardo, Gianfranco and Trivedi, Kishor S., "SPNP: The Stochastic Petri Net Package (Ver. 3.1)", MOSCOTS'93 Simulation Series, Vol. 25, No. 1, 1993, pp. 390-391, January 17-20.
4. Ciardo, Gianfranco, "Toward a Definition of Modeling Power for Stochastic Petri Net Models", International Workshop on Petri Nets and Performance Models, Madison, Wisconsin, pp. 54-62, August 24-26, 1997.
5. Ciardo, Gianfranco, Muppala, Jogesh and Trivedi, Kishor S., "On the Solution of GSPN Reward Models", Performance Evaluation, Vol. 12, pp. 237-253, December 1991.
6. Ciardo, Gianfranco, Muppala, Jogesh and Trivedi, Kishor S., "SPNP: Stochastic Petri Net Package", International Workshop on Petri Nets and Performance Models, Kyoto, Japan, pp. 142-151, Dec. 11-13, 1989.
7. Gerhart, Susan L., "Applications of Formal Methods: Developing Virtuoso Software", IEEE Software, pp. 7-10, September 1990.
8. Hoare, C.A.R, Communicating Sequential Processes, Prentice-Hall Int'l Series in Comp. Sci, 256 pages, 1985.
9. Johnson, Barry w., Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wesley Publishing Company, 584 pages, 1989.
10. Kavi, K.M. and Buckles, B.P., "Formal Methods for the Specification and Analysis of Concurrent Systems", Tutorial Notes, 1993 Int'l Conference on Parallel Processing, Lake Charles, IL. 75 pgs, Aug. 20, 1993.
11. Kavi, K.M., and Sheldon, F.T., "Specification of Stochastic Properties with CSP", IEEE Proc. Int'l Conf. On Parallel and Distributed Systems, Taiwan, pp. 288-293, Dec. 1994.
12. Kavi, K.M., Sheldon, F.T. and Reed, S.C., "Specification and Analysis of Real-Time Systems Using CSP and Petri Nets", Int'l Journal of Software Engineering and Knowledge Engineering – Special Issue on S/E Practices and Tools for Real-Time Systems, June 1996.
13. Kavi, Krishna M. and Sheldon, Frederick T., "Reliability Analysis of SCP Specification Using Petri Net and Markov Models", HICSS-28, January 1995.
14. Laprie, J.C., Kaaniche, M., and Kanoun, K., "Modeling Computer Systems Evolutions: Non-Stationary Processes and Stochastic Petri Nets –Application to Dependability Growth", IEEE PNPM'95, Durham, NC, pp. 221-230, October 1995.
15. Murata, Tadao, "Petri Nets: Properties, Analysis and Applications", Proceedings of the IEEE, Vol. 77, No. 4, pp. 541-580, April 1989.
16. Olderog, Ernst-Rudiger, "Operational Petri Net Semantics for CCSP", Lecture Notes in Computer Science, Springer-Verlag, Vol. 266, pp. 1996-223, 1987.
17. Olderog, Ernst-Rudiger, "TCSP: Theory of Communicating Sequential Processes", Lecture Notes in Computer Science, Springer-Verlag, Vol. 255, pp. 441-465, 1986.
18. Sahner, Robin A., and Trivedi, Kishor S., "A Software Tool for Learning About Stochastic Models", IEEE Transactions on Education, Vol. 36, No. 1, pp. 56-61, February 1993.
19. Sheldon, F.T., and Kavi, K.M., "Linking Software Failure Behavior to Specification Characteristics II", IEEE Proc. Fourth Int'l Workshop on Evaluation Techniques for Dependable Systems, San Antonio, TX, Oct. 1995.
20. Sheldon, F.T., Kavi, K.M., and Kamangar, F.A., "Reliability Analysis of CSP Specifications: A New Method Using Petri Nets", AIAA Proceedings Computing in Aerospace 10, pp. 317-326, March 1995.
21. Sheldon, F.T., "Specification and Analysis of Stochastic Properties for Concurrent Systems Expressed Using CSP", Ph.D. Dissertation, Comp. Sci. and Engrng. Dept., Univ. of TX at Arlington, May 1996.
22. Sorenson, Erling, Vagn, Nordahl, Jens and Hansen, Niels Herman, "From CSP Models to Markov Models", IEEE Trans of SE, Vol. 19, No. 6, 1, pp. 554-570, June 1993.
23. Wang, Chang-Yu, "Some Problems in the Specification and Analysis of Computers and Networks, Ph.D. Dissertation, Department of Computer Science, Duke University, 101 refs. 218 pages, 1995.

APPENDIX A: Available Parameter Options

Table A-1. Parameter Options Available for Generating SPNP files.

Parameter	Description & Choices
IOP_PR_MARK_ORDER	Specifies the order in which the markings are printed. VAL_CANONIC - markings are printed in the order they are found, in a breadth-first search starting from the initial markings, and increasing order of enabled transition indices. It is the most natural order and it is particularly helpful when debugging the SRN. VAL_LEXICAL - markings are printed in increasing order, where markings are compared as words in a vocabulary, the possible number of tokens being the alphabet, and the order of the "letter" in a "word" being given by the order of the non-empty places in the marking. This order may be useful when searching for a particular marking in a large ".rg" file, although an editor with search capabilities used with the VAL_CANONICAL order is usually adequate for the purpose. VAL_MATRIX - markings are printed in the same order as the states of the two Markov chains built internally: the DTMC corresponding to the vanishing markings, and the CTMC corresponding to the tangible markings. This corresponds to the following ordering: vanishing, tangible non-absorbing, and tangible absorbing, each of these group ordered in canonical order. Choices: **VAL_CANONICAL** (default); VAL_LEXICAL; VAL_MATRIX
IOP_PR_MERG_MARK	Specifies whether the tangible and vanishing markings should be printed together, or two separate lists should be printed. Choices: VAL_YES; **VAL_NO** (default)
IOP_PR_FULL_MARK	Specifies whether the markings are printed in long format (a full matrix indicating, for each marking, the number of tokens in each place, possibly zero), or short format (for each marking, a list of the number of tokens in the non-empty places). VAL_YES - looks good only when the SRN has a small number of places. Choices: **VAL_YES** (default); VAL_NO
IOP_PR_RSET	Specifies whether the reachability set is to be printed. VAL_TANGIBLE - specifies that only the tangible markings are to be printed; it cannot be used for IOP_PR_RGRAPH Choices: VAL_YES; **VAL_NO** (default); VAL_TANGIBLE
IOP_PR_RGRAPH	Specifies whether the reachability graph is to be printed. Choices: VAL_YES; **VAL_NO** (default)
IOP_PR_MC	Specifies whether the ".mc" file is generated or not. Choices: VAL_YES; **VAL_NO** (default)
IOP_PR_MC_ORDER	VAL_FROMTO - specifies that the transition rate matrix is printed in the ".mc" file. VAL_TOFROM - specifies that the transpose of the transition rate matrix is printed in the ".mc" file. Choices: **VAL_FROMTO** (default); VAL_TOFROM
IOP_PR_PROB	Specifies whether the ".prb" file is generated. Choices: VAL_YES; **VAL_NO** (default)
IOP_MC	Specifies the solution approach. VAL_CTMC - transforms the SRN into a CTMC. VAL_DTMC - uses an alternative approach, where vanishing markings are not eliminated and a DTMC is solved instead. Choices: **VAL_CTMC** (default); VAL_DTMC

Parameter	Description & Choices
IOP_OK_ABSMARK	Specifies whether absorbing markings are acceptable or not. VAL_YES - the program will signal such occurrences, but will continue to execute. VAL_NO - the program will stop if such an occurrence is encountered. Choices: VAL_YES; **VAL_NO** (default)
IOP_OK_VANLOOP	Specifies whether transient vanishing loops are acceptable or not. VAL_YES - the program will signal such occurrences, but will continue to execute. VAL_NO - the program will stop if such an occurrence is encountered. Choices: VAL_YES; **VAL_NO** (default)
IOP_OK_TRANS_M0	Specifies whether a transient initial marking is acceptable or not. VAL_YES - the program will signal such occurrences, but will continue to execute. VAL_NO - the program will stop if such an occurrence is encountered. Choices: VAL_YES; **VAL_NO** (default)
IOP_METHOD	Allows the user to set the numerical solution method for the Markov chain. VAL_SSSOR - Steady State SOR. VAL_GASEI - Steady State Gauss-Seidel. VAL_TSUNIF - Transient Solution using Uniformation. VAL_SSPOWER - power method for steady state solution. This is slow, but guaranteed to converge. Choices: **VAL_SSSOR** (default); VAL_GASEI; VAL_TSUNIF; VAL_FOXUNIF; VAL_POWER
IOP_CUMULATIVE	Specifies whether cumulative probabilities should be computed. Choices: **VAL_YES** (default); VAL_NO
IOP_SENSITIVITY	Specifies whether sensitivity analysis should be performed. Choices: **VAL_YES** (default); VAL_NO
IOP_ITERATIONS	Specifies the maximum number of iterations allowed for the numerical solution. Choices: non-negative int – **2000** (default)
IOP_DEBUG	Causes the output (on the "stderr" stream) of the markings as they are generated, and of the transitions enabled in them. It is extremely useful when debugging a SRN. Choices: VAL_YES; **VAL_NO** (default)
IOP_USENAME	Specifies whether the names must be used to indicate the places and transitions involved when printing the reachability set and graph instead of the index. Using names generates a large ".rg" file and prevents its subsequent parsing, but is useful when debugging a SRN. Choices: VAL_YES; **VAL_NO** (default)
FOP_ABS_RET_M0	Specifies the value of the rate from each absorbing marking back to the initial marking. If this rate is positive, these markings will not correspond to absorbing states in the CTMC. This is useful to model a situation that would otherwise require a large number of transitions to model this "restart". Of course the numerical results will depend on the value specified for this option. Choices: non-negative double – **0.000000** (default)
FOP_PRECISION	Specifies the minimum precision required for the numerical solution. The numerical solution will stop either if the precision is reached, or if the maximum number of iterations is reached. Both the reached precision and the actual number of iterations are always output in the ".prb" file, so you can (and should) check how well the numerical algorithm performed. **Choices:** non-negative double – **0.000001** (default)

APPENDIX B: CSP-Based Grammar and Constructs

Table B-1. CSP-Based Grammar (using the lex and yacc specification conventions).

Lex Regular Expressions:

```
delimiter        [ \t]
eol              [\n]
white_space      {delimiter}+
letter           [A-Za-z_]
digit            [0-9]
identifier       {letter}({letter}|{digit})*
integer          {digit}+
real number      {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
comment          "--".*$
```

Yacc Grammar Specification:

```
Start symbol = "system"
1. system: Identifier Equals
              processdeclist processlist1 Dot;
2. processdec: PROCESS Identifier Equals
                  processlist1 Semicolon;
3. processdeclist: | processdeclist processdec;
4. process:
    STOP
  | LeftBrace stmtlist RightBrace
  | PAR LeftBrace processlist2 synclist RightBrace
  | SEQ LeftBrace processlist1 RightBrace
  | NDC LeftBrace processlist3 RightBrace
  | DC LeftBrace guardedproclst RightBrace
  | MU Dot Identifier LeftBrace processlist1 RightBrace
  | processcall;
5. failable:
             FAIL LeftParen rEquals Real RightParen
           | FAIL LeftParen pEquals Real RightParen;
5.1 probable: PROB LeftParen pEquals
                 Real RightParen;
5.2 servable: SERV LeftParen rEquals
                 Real RightParen;
6. biprocess:
              process
            | process Colon failable
            | process Colon probable
            | process Colon servable;
7. processlist1:
              biprocess
            | processlist1 Comma biprocess;
8. processlist2:
             biprocess Comma biprocess
           | processlist2 Comma biprocess;
8.5. processlist3:
             biprocess Comma biprocess
           | processlist3 Comma biprocess;
9. synclist: | LeftParen anyvarlist RightParen;
10. anyvar: booleanvar | variable;
11. anyvarlist: anyvar | anyvarlist Comma anyvar;
12. stmtlist: stmt | stmtlist Comma stmt;
13. stmt:
              implication
            | expression
            | input **looks like {channel ? variable}
            | output **looks like {channel ! variable}
            | SKIP;
14. implication:
              stmt Arrow consequent
            | variable Arrow consequent
            | biprocess;
14.5 consequent:
              variable
            | biprocess;
15. processcall: Identifier LeftParen RightParen;
16. assignment is covered by expression in integer;
17. input: channel InSym variable;
18. output: channel OutSym operand;
19. guardedprocess: guard biprocess;
20. guardedproclst:
              guardedprocess
            | guardedproclst Comma guardedprocess;
21. guard:
              input
            | booleanexpr AND input
            | booleanexpr AND SKIP;
22. recursive definition (not needed but defined in
    process definition)
23. channel: Identifier;
24. variable: Identifier;
25. booleanvar: AtSym Identifier;
26. expression:
              integerexpr
            | booleanexpr
            | relationalexpr;
27. booleanexpr:
              booleanvar
            | TRUE
            | FALSE
            | booleanexpr AND booleanexpr
            | booleanexpr OR booleanexpr
            | NOT booleanexpr
            | booleanvar VarAsgn booleanexpr;
28. relationalexpr:
              operand LESym operand
            | operand LTSym operand
            | operand EQSym operand
            | operand NESym operand
            | operand GESym operand
            | operand GTSym operand;
```

Table B-1. CSP-based grammar (cont.)

29. integerexpr: operand Plus operand | operand Minus operand | operand Star operand | operand Slash operand | operand VarAsgn operand | Minus operand; 30. operand: Integer | variable | integerexpr | relationalexpr;	Under Construction (or otherwise): 31. monadic operand: (never used) 32. dyadic operand: (never used) 33. integer: (defined in lexer) 34. digits: (defined in lexer) 35. digit: (defined in lexer) 36. declaration: (currently undefined) 37. type: (currently undefined) 38. selection: (currently undefined) 39. conditional: (currently undefined) 40. option: (currently undefined) 41. loop: (currently undefined) 42. relational operator: (currently undefined) 43. timer: (currently undefined) 44. hide: (currently undefined)

Table B-2. CSP Constructs.

Placeholders are used to help the user know that an element must be provided.

"a"	variable place holder.
"P"	Process place holder.
"_"	Either variable or Process is permissible.

1. STOP	-- Never terminates (like deadlock).
2. SKIP	-- Does nothing then terminates.
3. PAR{_, _}	-- Two elements (at least) are supplied.
4. PAR{_, _ (messages)}	-- Two elements (at least) are supplied (and at least one message).
5. SEQ{_}	-- One element (at least) is supplied.
6. NDC{_, _}	-- Two elements (at least) are supplied.
7. DC{gP, gP}	-- gP = "guard process"
8. Mu.X{_}	-- One element (at least) is supplied (X is any identifier).
9. guard:	-- input | booleanexpr AND input | booleanexpr AND SKIP
10. statement (stmt):	-- Is a process: implication | expression | input | output | SKIP
11. Implication:	-- Is a process: stmt -> consequent | variable -> consequent
12. Consequent:	-- variable | process
13. Input:	-- Channel ? variable (variable is usually message, an identifier)
14. Output:	-- Channel ! operand (operand is usually a message, an identifier)
15. Operand:	-- variable | integerexpr | relationalexpr
16. variable:	-- an identifier (usually a lower case alpha).
17. process:	-- Must be declared (usually a higher case alphanumeric: "P()").
	-- "P()" is known as a process call based on a pre-defined process.
	-- Items 1 through 8 are all processes (by default).

APPENDIX C: CSP to Petri Net Translations

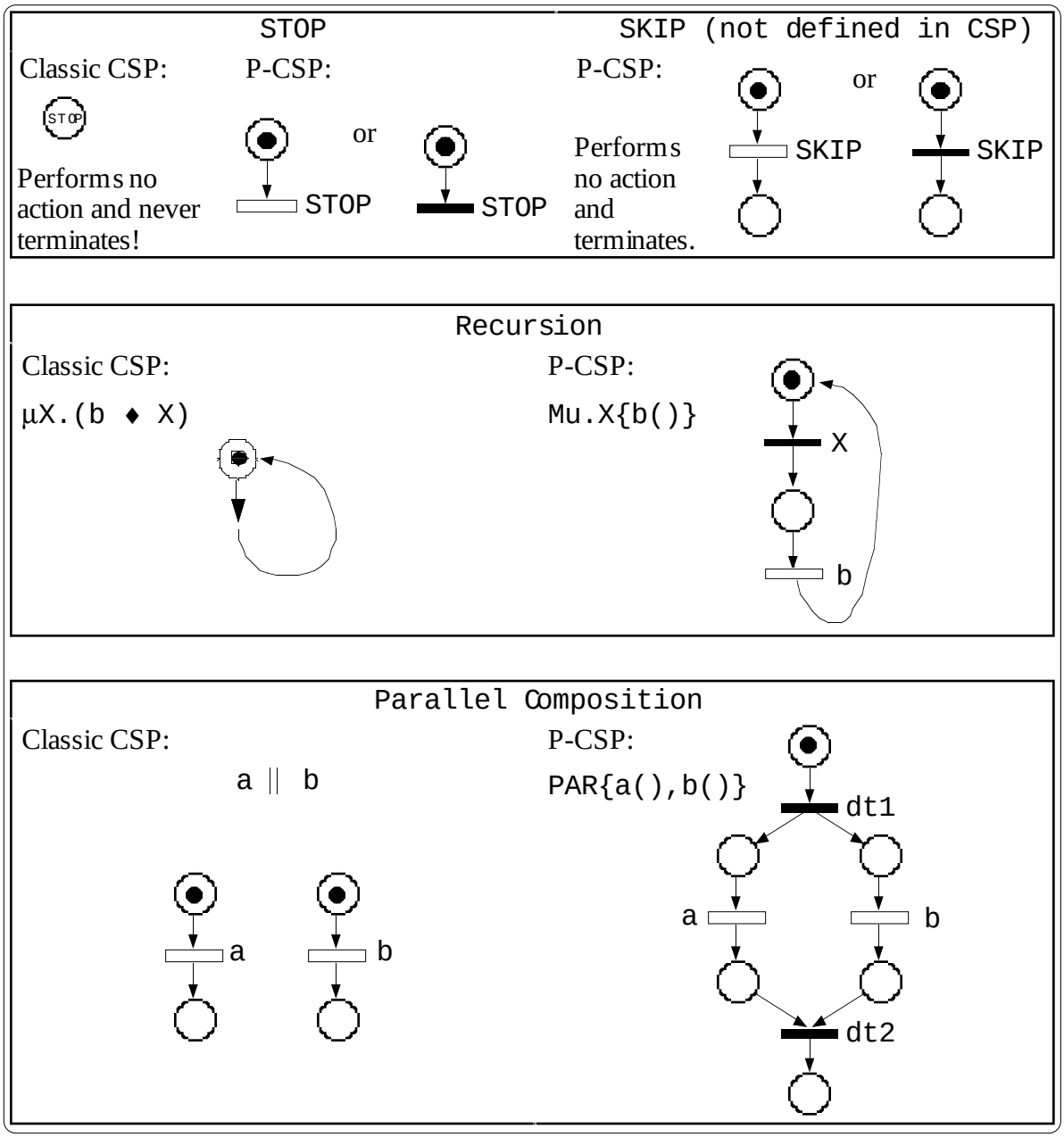


Figure C-1. Translations for (top) STOP / SKIP, (center) recursion and (bottom) PAR.

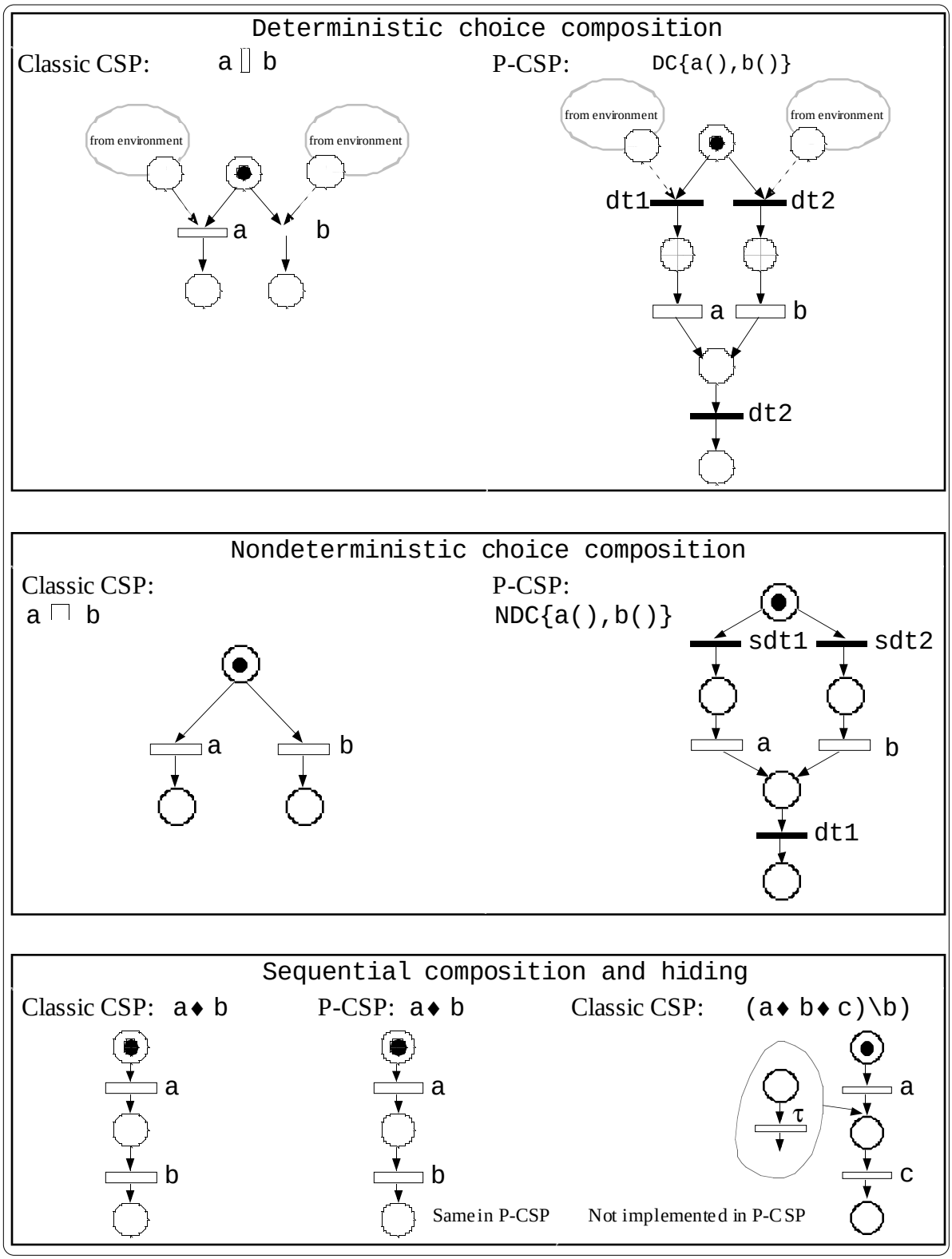
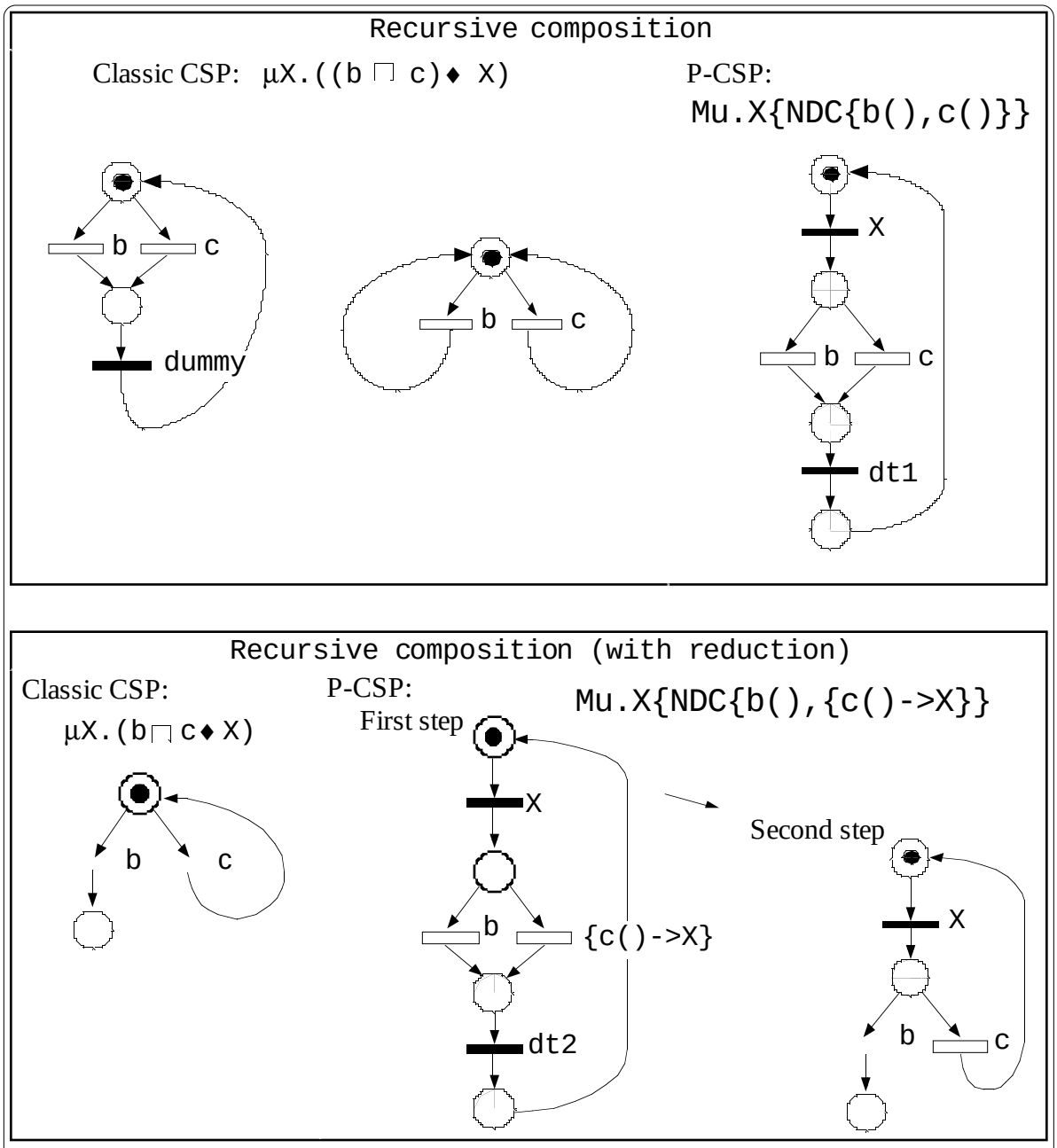


Figure C-2. Translations for (top) DC, (center) NDC and (bottom) arrow and hiding.



The two recursive translations shown here (top and bottom) are the same translations as those shown in Figure 13 except those shown here are reduced. In the top figure, there are two fewer transtions and one less place. In the bottom figure, there are also two fewer transitions and two fewer places.

Figure C-3. The translation of a recursive compositions in a reduced format.

Synchronization using input and output actions

Classic CSP:

Train=

(InTransit);

(Chnl!arrive ♦ AtIntersect);

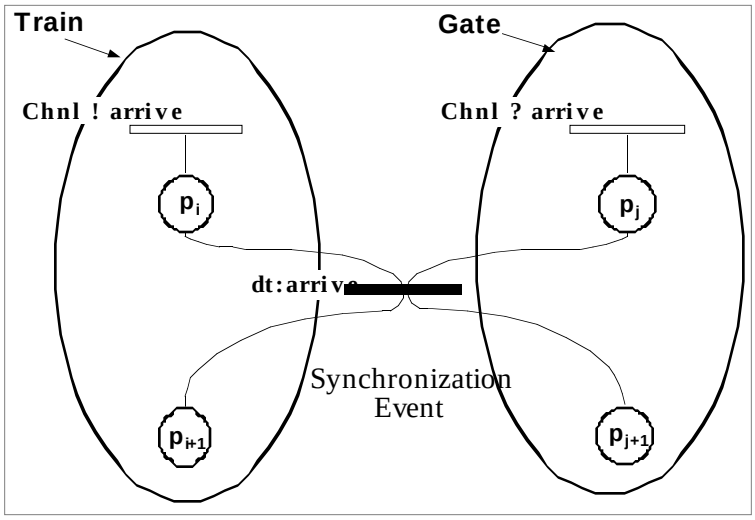
(Chnl!depart ♦ Train);

Gate=

(Chnl?arrive ♦ Close);

(Chnl?depart ♦ Open ♦ Gate);

Pictured at right



Nonblocking Synchronization using input and output actions

Classic CSP:

Train=

(InTransit);

(Chnl!arrive ♦ AtIntersect);

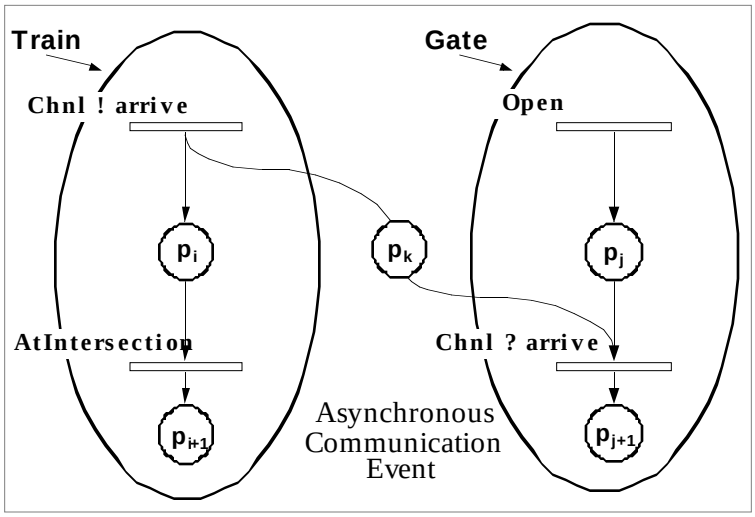
(Chnl!depart ♦ Train);

Gate=

(Chnl?arrive ♦ Close);

(Chnl?depart ♦ Open ♦ Gate);

Pictured at right



Synchronization is syntactically the same for both CSP and P-CSP. There are 2 possible translations that could be used. In the Petri net fragments shown, the train sends and the gate is receives. The actual synchronizing action (dt:arrive) is an immediate transition and its firing is necessary before either process can proceed. In the bottom of the figure the sending process (Train) is not blocked and can proceed (this 2nd type of synchronization is not used by the CSPN tool).

Figure C-4. The translations show both blocked and non-blocked send synchronization.

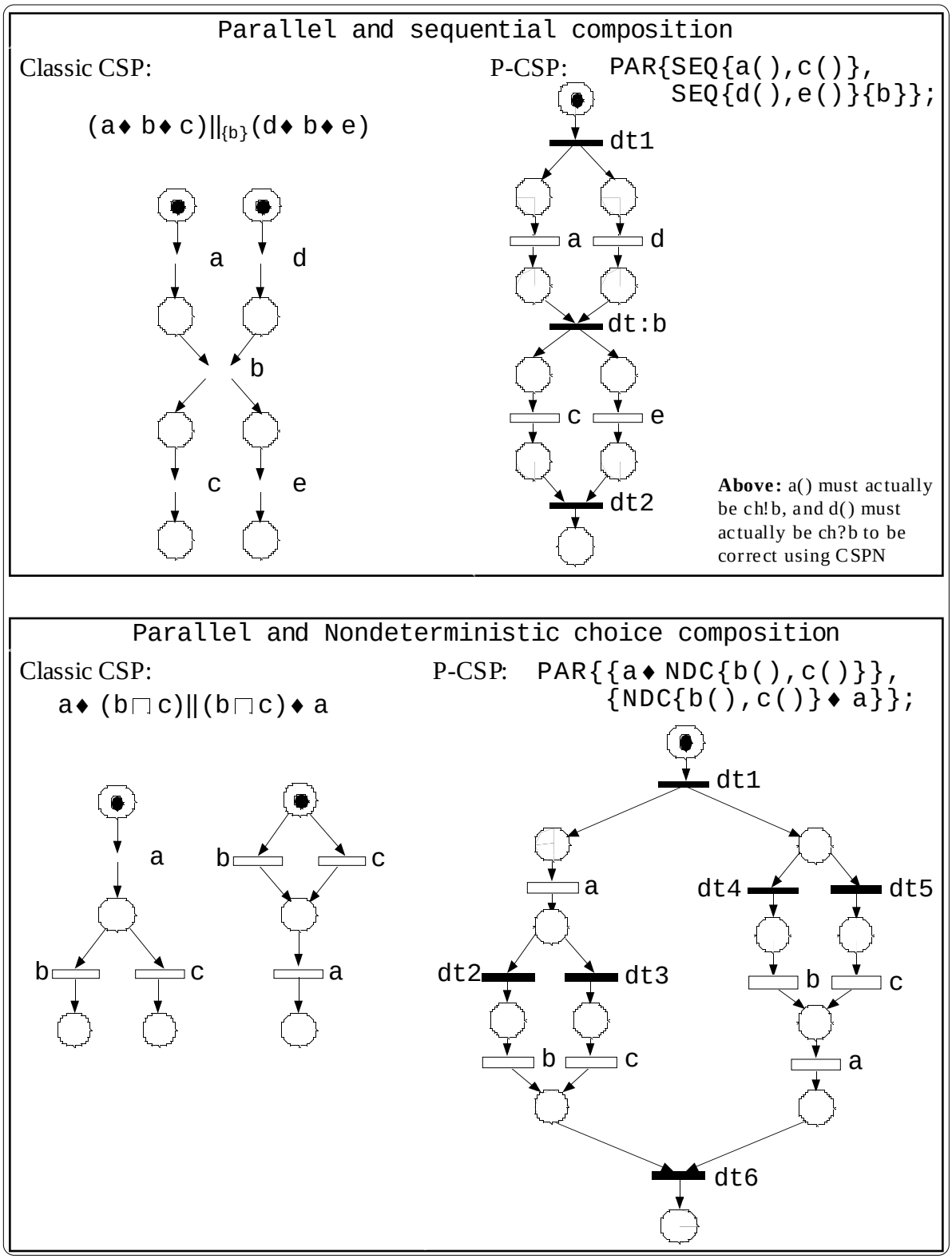


Figure C-5. Combined translations for parallel, sequential and nondeterministic choice.

APPENDIX D: Description of CSPN Program Files

File Name	Description
cmd_line.c	Establishes the command line interface and flags control.
csp.l	Scanner/lexer needed for lexographic analyzer used to define tokens. This file establishes what characters make up words, numbers, white space, etc.
csp.y	Parser which determines the language and grammar (also main, and most open/close of auxiliary files.
cspl.h	Header file needed specifically by the lexer.
cspltops.h	Header file needed by cspltops.c.
cspltops.c	Menuing and main program.
cspy.h	Header file needed specifically by the parser.
expn_cspy.c	When the "net" structure of the CSP is completed, a process of composition and expansion occurs to generate the complete Petri net structure.
genDot.h	Generates a dot program readable input file.
itoa.c	Small file containing a integer-to-ascii routine.
net.c	Algorithms used to construct, display and traverse an array structure of linked lists which define the structure (declarations, adjacency, and nesting) of CSP.
petri_cspy.c	Used to decode the process list and coincidence-matrix, to build the net() part of the fn_spnp.c output.
prlist.c	Contains code to define, display and manipulate the process lists, which are the names of CSP processes as defined in the input file as well as dummy names.
prmatrix.c	Contains code to define, display and manipulate the coincidence-matrices.
reduce_matrix.c	Algorithms to reduce the number of dummy transitions and places after all the smaller pieces are assembled.
saferMalloc.c	Very short file which devises a safer way to allocate memory which exits more gracefulls.
scoring.c	Routines that mark the atomic coincidence-matrices, which define the structure of each individual CSP construct.
spnp_end.c	File used to generate the CSPL file from the total "system" coincidence-matrix.
symbol_cspy.h	Header file needed by symbol_cspy.c.
symbol_cspy.c	Routines to create, display, manipulate and update the symbol table of CSP process names (including [dummy] transitions and places).