

APPENDIX A

CSP-TO-PETRI NET CANONICAL TRANSLATION DIAGRAMS

In this appendix a complete collection of standard translations from classic CSP and P-CSP to Petri nets is provided. The CSP primitives include STOP, SKIP (not included in CSP), recursion, parallel, deterministic and nondeterministic choice, hiding and sequential compositions. The arrow (\rightarrow) is also shown in various compositions.

Figure A-1 shows STOP which performs no action and never terminates (like deadlock) and SKIP which performs no action and terminates are shown at the top. In the center of Figure A-1 simple recursion is presented (note that P-CSP incurs an extra dummy transition which is an immediate non-timed transition). In the bottom, a parallel composition is shown and P-CSP uses two dummy transitions.

Figure A-2 shows DC (deterministic choice) where P-CSP employs three dummy transitions. In the center NDC (nondeterministic choice) is shown which also uses three dummy transitions. Note that the sdt1 and sdt2 dummy transitions are given as such because associated with each is a (by definition) probability. In the bottom of this figure, a sequential composition using the arrow is shown. The CSP translation for hiding is also shown (there is no P-CSP equivalent at this time).

Figure A-3 shows Mu.X (recursion where "X" can be any character). Compare the various configurations and notice that the translations are comparable to those of Figure 13 which defines the way CSPN translates P-CSP. Figure A-3 provides equivalent but reduced translations. The top half shows tail recursion and the bottom show a variation of such which cuts the tail recursion. Recursion using the CSP prefix notation is desirable because it describes the entire behavior of a process that eventually stops. For example, it would be tedious to write the full behavior of some systems which cycle over and over (e.g., a train crossing or vending machine). Recursion is useful for describing repetitive behavior patterns using much shorter notations. Such systems should not require a prior decision on the length of life of an object in order to permit the description of objects that continue to act and interact with their environment indefinitely.

Figure A.4 shows two varieties of synchronization. The first (top half) is blocking send and receive. This forces synchronization to occur while preventing either participant from moving forward until the other catches up. The CSPN tool has adopted this method because the interpretation of `chn!msg` combined with `chn?msg` was more natural (i.e., closer to the

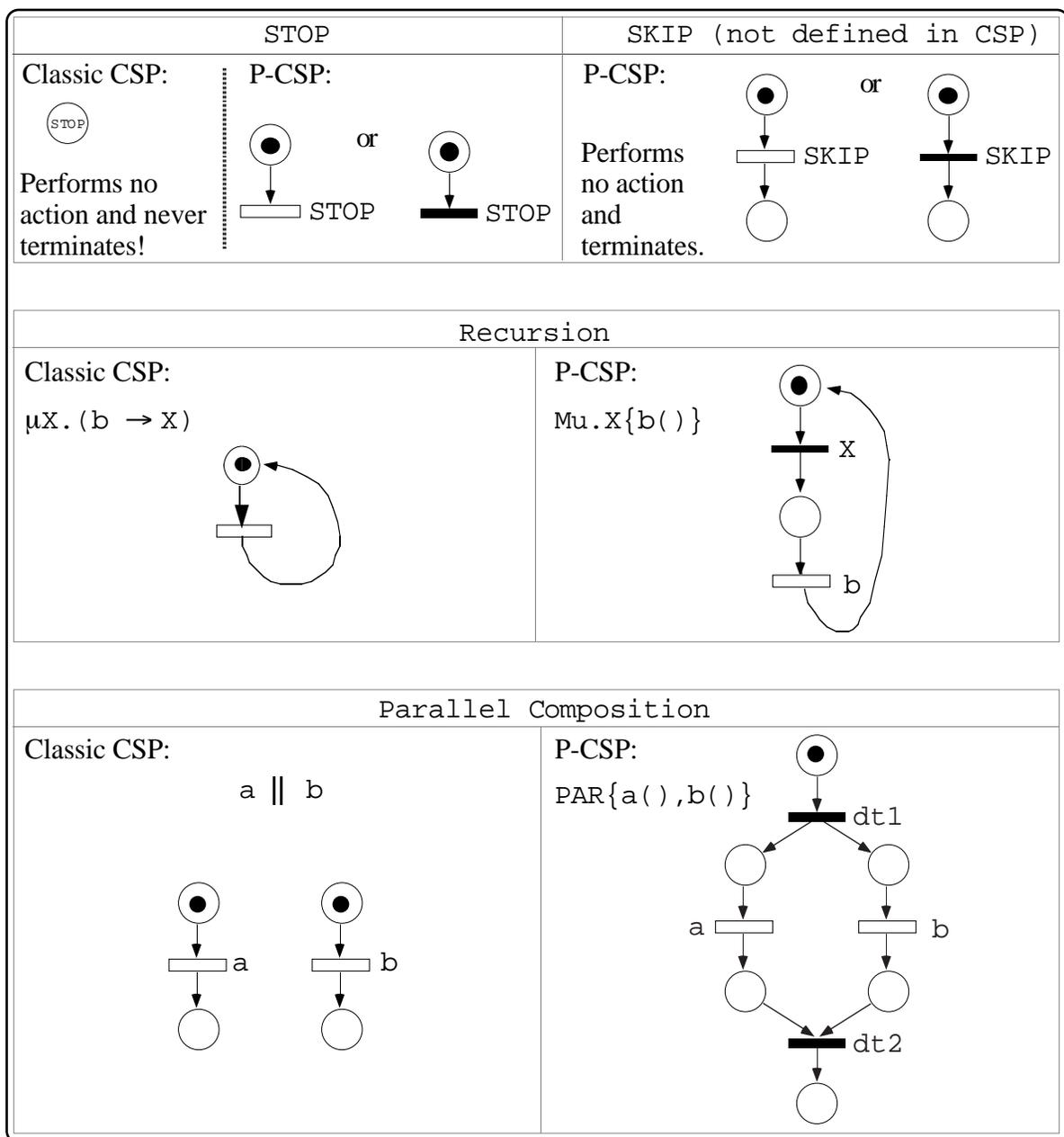


Figure A.1 Translations for (top) STOP / SKIP, (center) recursion and (bottom) PAR.

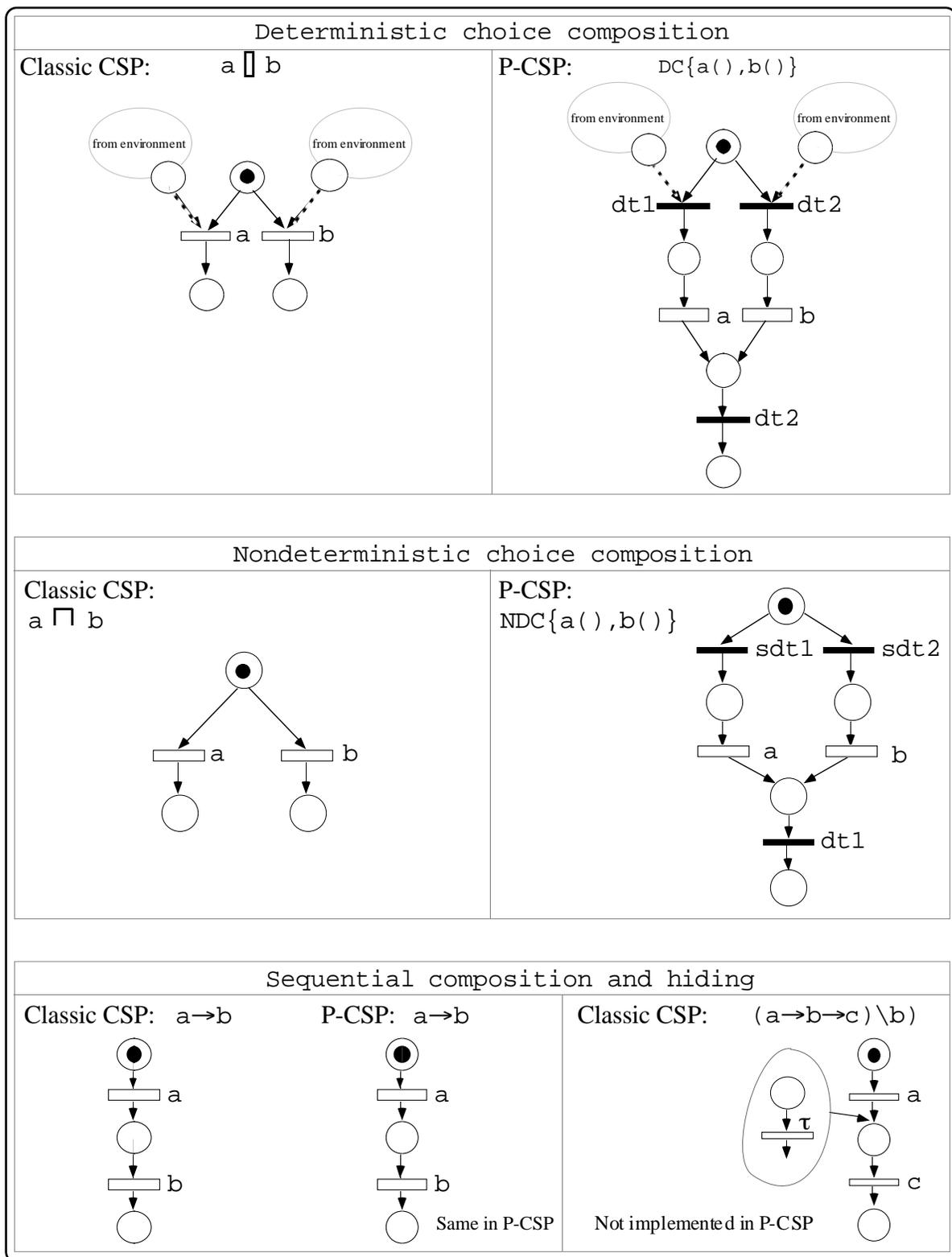


Figure A.2 Translations for (top) DC, (center) NDC and (bottom) arrow and hiding.

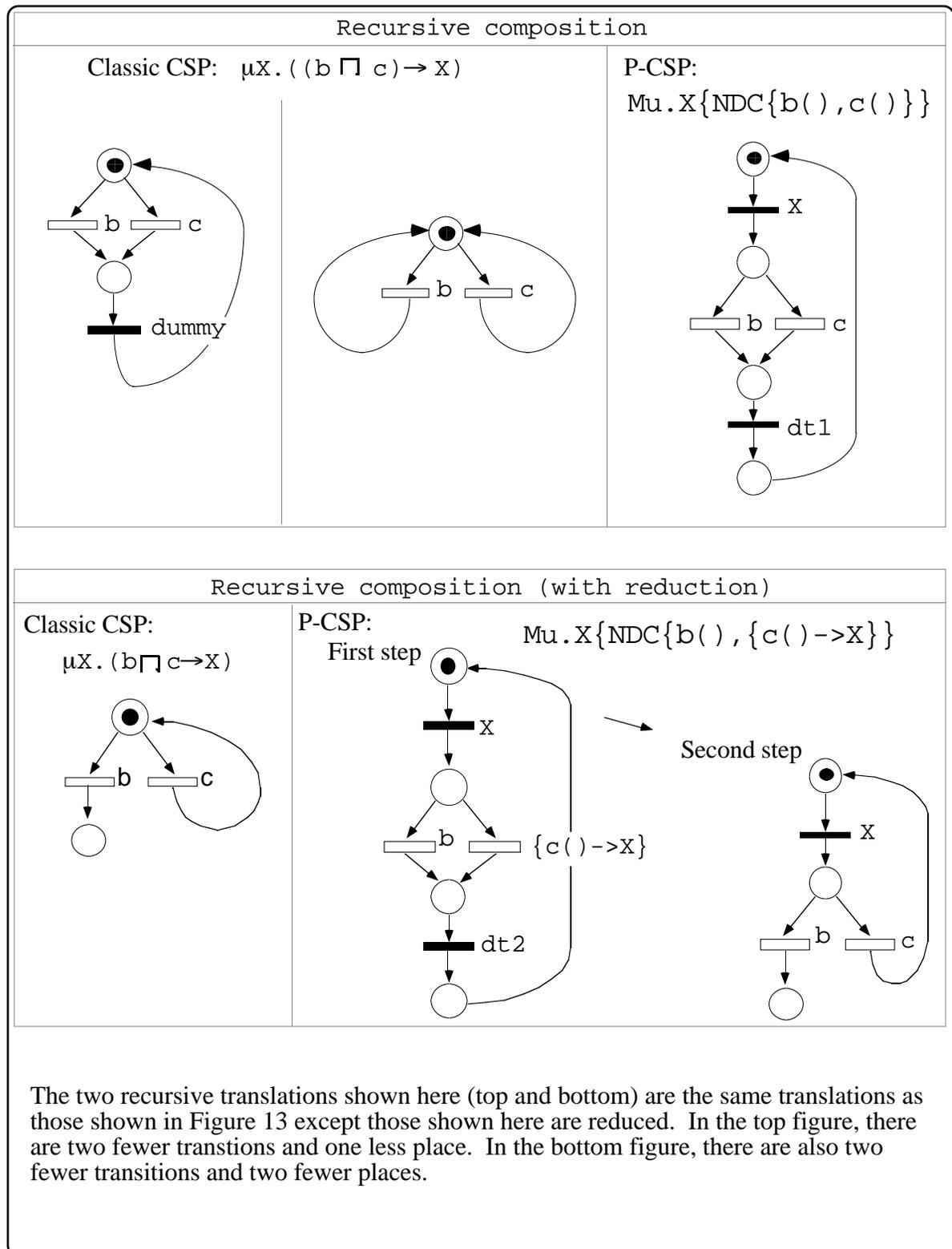


Figure A.3 Translation of recursive compositions in a reduced format.

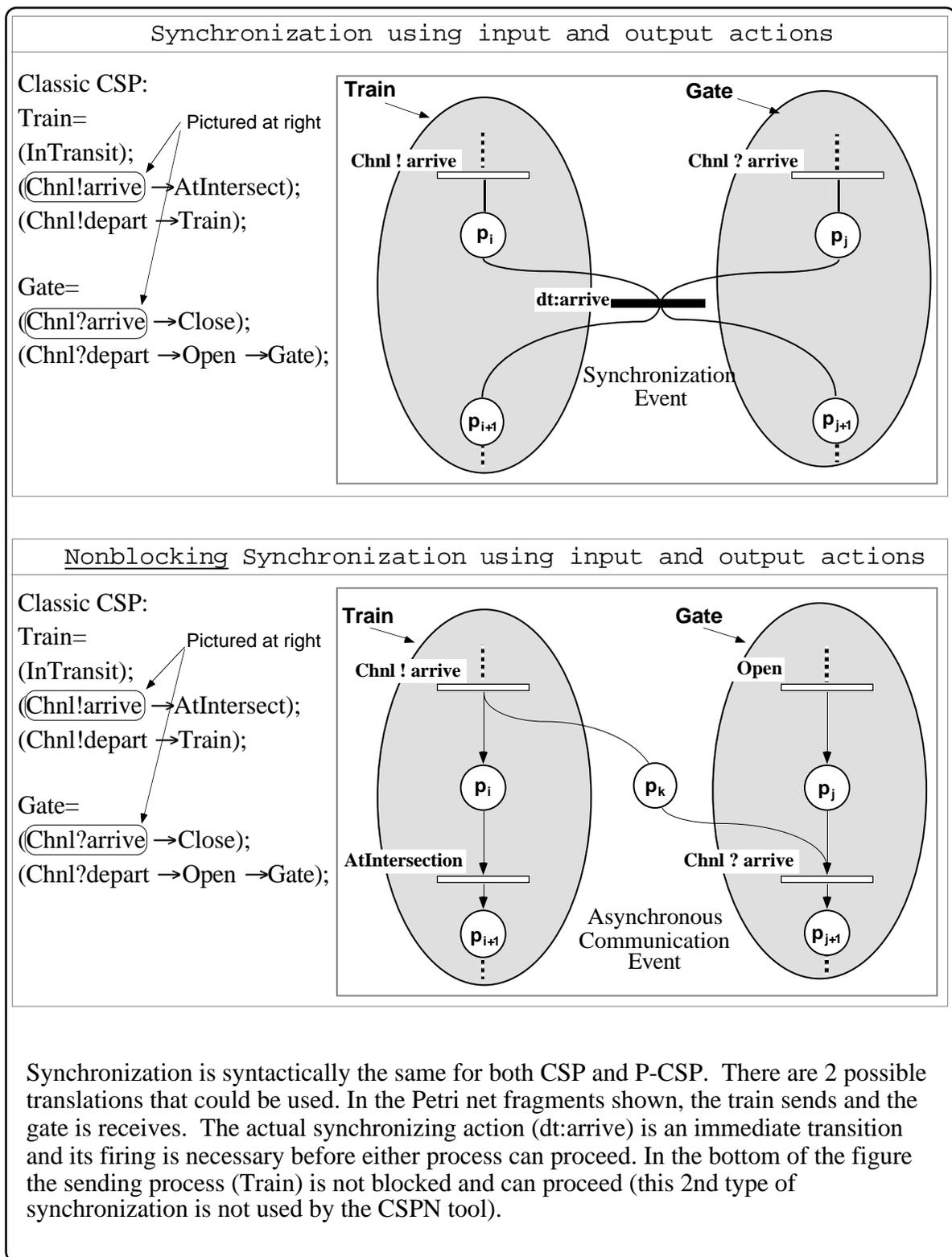


Figure A.4 Translations showing blocked and non-blocked send synchronization.

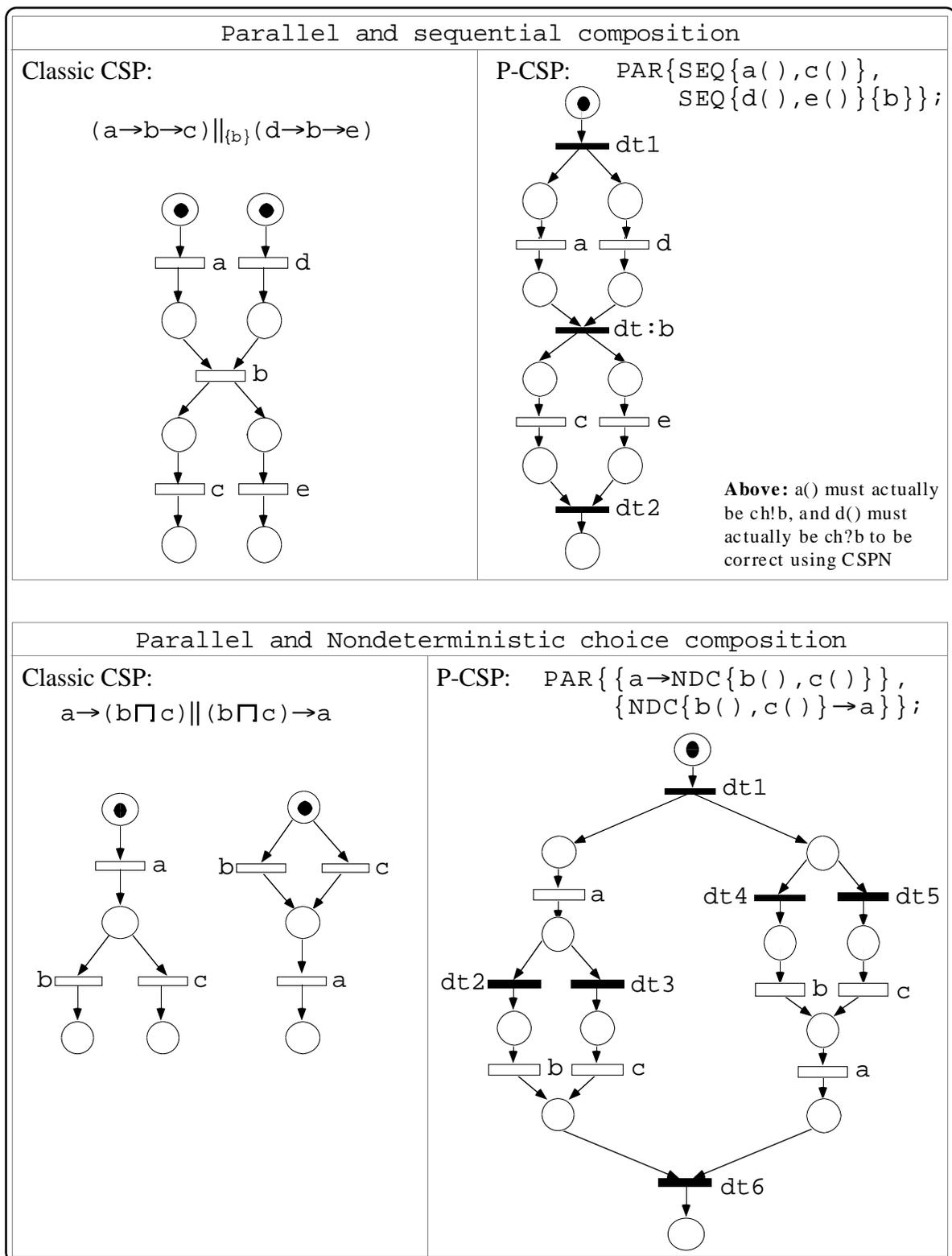


Figure A.5 Combined translations for parallel, sequential and nondeterministic choice.

inherently synchronous semantics of CSP) and more readable. Also, using the notion of hiding in CSP, both actions (input and output) can be replaced by tau (like "\b" in Figure A.2 bottom). In the bottom half (of Figure A-4) a message is output (on channel "Chnl") while processing continues (a token is distributed to place p_k) for the sending process independent of whether the message is received. On the receiving end, the transition that models the activity of message input (on the channel "Chnl" is this case) fires only after both places p_k and p_j have tokens. The interpretation of this type of communication is that the receiver must wait for the message from the sending process (the Train in this case). This is known as a blocking receive.

Finally, in Figure A-5 a number of larger compositions are collected to illustrate a combined parallel and sequential composition that has synchronization (blocking send and receive). The CSP translation uses 5 transitions and 8 places while the P-CSP translation uses 7 transitions and 10 places. In the bottom half of Figure A-5 two nondeterministic choice constructs are composed in parallel with an action "a" prefixed to the one and an action "a" suffixed to the other. Notice that the direct CSP translation only uses 6 transitions and seven places while the P-CSP translation uses 12 transitions and 12 places!

APPENDIX B

THE LEX AND YACC SPECIFICATION OF THE PARSEABLE CSP (GRAMMAR GIVEN IN BACKUS NORMAL FORM)

B.1 Lex regular expressions

```

delimiter      [ \t\n]
white_space    {delimiter}+
letter         [A-Za-z_+\-%@]
digit          [0-9]
identifier     {letter}({letter}|{digit})*
integer        {digit}+
comment        "--".*$

```

B.2 Yacc grammar specification

1. System production (start symbol = "system").


```

system: Identifier Equals processdeclist processlist1 Dot;

```
2. Processdec used to declare process names.


```

processdec: PROCESS Identifier Equals processlist1 Semicolon;

```
3. Processdeclist for listing multiple declarations under system.


```

processdeclist: EmptyList | processdeclist processdec;

```
4. Process definitions


```

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 describes the format of an annotation (rate or probability).


```

failable:
  FAIL LeftParen rEquals Real RightParen
  | FAIL LeftParen pEquals Real RightParen;

```
6. Probable describes the format of a probability annotation.


```

probable:
  PROB LeftParen pEquals Real RightParen

```
7. Servable describes the format of a service rate annotation.


```

servable:
  SERV LeftParen rEquals Real RightParen

```
8. Biprocess distinguishes an annotated process and permit such on any process.


```

biprocess:
  process | process Colon failable
  | process Colon probable
  | process Colon servable

```
9. Processlist1 permits one or more processes in a list.


```

processlist1: biprocess | processlist1 Comma biprocess;

```

10. Processlist2 permits no less than two processes in a list.

```
processlist2:
  biprocess Comma biprocess | processlist2 Comma biprocess;
```
11. Processlist3 permits no less than two processes in a list and specialized for NDC.

```
processlist3:
  biprocess Comma biprocess | processlist3 Comma biprocess;
```
12. Synclist used with PAR to indicate synchronization messages.

```
synclist: EmptyList | LeftParen anyvarlist RightParen;
```
13. Anyvar used to permit concise grammar of the rule for lists.

```
anyvar: booleanvar | variable;
```
14. Anyvarlist specifies an arbitrary number of anyvar in a list.

```
anyvarlist: anyvar | anyvarlist Comma anyvar;
```
15. Statement list allows an arbitrary number of statements to be listed.

```
stmtlist: stmt | stmtlist Comma stmt;
```
16. Statements can compose a process.

```
stmt:
  implication
  | expression
  | input
  | output
  | SKIP;
```
17. Implication (a statement event \rightarrow action [for $P \rightarrow Q$ use $SEQ\{P(),Q()\}$]).

```
implication:
  stmt Arrow consequent | variable Arrow consequent | biprocess;
```
18. Consequent belongs to the right hand side of an arrow.

```
consequent: variable | biprocess;
```
19. Processcall is an instance of a declared PROCESS and is simply set to Identifier().

```
processcall: Identifier LeftParen RightParen;
```
20. Assignment is covered by expression in integer
21. Input

```
input: channel InSym variable;
```
22. Output (note an operand is an integer or boolean expression).

```
output: channel OutSym operand;
```
23. Guarded process is defined for use in the guarded process list.

```
guardedprocess: guard biprocess;
```
24. Guarded process list

```
guardedproclst:
  guardedprocess | guardedproclst Comma guardedprocess;
```

25. Guard us used to provide for choosing an alternate in a deterministic choice (DC).

```
guard: input
  | booleanexpr AND input
  | booleanexpr AND SKIP;
```

26. Recursive definition is defined in the definition of processes (see Mu).

27. Channel is matched by paring a input message with an output message.

```
channel: Identifier;
```

28. Variable

```
variable: Identifier;
```

29. Boolean variable (AtSym to distinguish a variable from a boolean variable).

```
booleanvar: AtSym Identifier;
```

30. Expression

```
expression: integerexpr | booleanexpr | relationalexpr;
```

31. Boolean expression.

```
booleanexpr:
  booleanvar
  | TRUE
  | FALSE
  | booleanexpr AND booleanexpr
  | booleanexpr OR booleanexpr
  | NOT booleanexpr
  | booleanvar VarAsgn booleanexpr;
```

32. Relational expression.

```
relationalexpr:
  operand LESym operand
  | operand LTSym operand
  | operand EQSym operand
  | operand NESym operand
  | operand GESym operand
  | operand GTSym operand;
```

33. Integer expression.

```
integerexpr:
  operand Plus operand
  | operand Minus operand
  | operand Star operand
  | operand Slash operand
  | operand VarAsgn operand
  | Minus operand;
```

34. Operand.

```
operand:
  Integer
  | variable
  | integerexpr
  | relationalexpr;
```

35. Monadic operand (never used).
36. Dyadic operand (never used).
37. Integer is defined in lexer.
38. Digits are defined in lexer.
39. Digit is defined in lexer.
40. Declaration (never used).
41. Type (never used).
42. Selection (never used).
43. Conditional (never used).
44. Option (never used).
45. Loop (never used).
46. Relational operator (never used).
47. Timer (never used).
48. Hide (never used).

APPENDIX C
CO-MATRIX EXPANSION ALGORITHMS

C.1 Algorithm for choosing the correct expansion method

```

k = (nnptr ->numNodes); /* k is now 1 more than needed */
while (((p=(nnptr ->net[--k])) != NULL) && (k > -1)) {
    s0 = look (p ->n_name, &symthere);
    if (symthere < 1) {
        fprintf(epn, "\nExpn: Symbol %s not found!", p->n_name); exit(1);
    }
    fprintf(epn, "\n\nSearching links of net[%d], symbol: %s", k, s0 ->name);
    DisplayProcessList(epn, 0, s0->p_pl);
    cur_p = p; /* Skip over the head node (a pnode) */

    if ((p=p->link) == NULL) fprintf(epn, "\nExpn: No sibs for this pnode!");
    else {
        curLnkIdx=0;
        for (q = p; q != NULL; q = q -> link) {
            curLnkIdx++;
            typ = q->n_type;
            fprintf(epn, "\n%d. Symbol: %s, Type: %d", curLnkIdx, q->n_name, typ );

            /* There are three cases where an expansion is appropriate:
            * (1) Node type is 5-9 (PAR, SEQ, NDC, DC, MU)
            * (2) Node type is 10 and cpi=0 (cpi is current process index)
            *     Node type 10 indicates an instance of a previously defined
            *         process known as a process call.
            * (3) Node type 11 is really type 10 except it was 1st encountered
            *     in "PROCESS symbol =", thus was marked as type 11.
            */
            if (((typ > SKIP_PROC) && (typ < PROC_CALL)) || (typ == STMT_LIST) ||
                ((typ == PROC_CALL) && (cpi == 0)) || (typ == PROCESS_DEC)) {

                s1 = look (q ->n_name, &symthere);
                if (symthere < 1) {
                    fprintf(epn, "\nSymb %s not found in symbol table!", q ->n_name);
                }
                else {
                    /* * * * * *
                    * This logic determines the size of the matrices involved.
                    */
                    rA = s0 ->rsize; /* Row size of A */
                    cA = s0 ->csize; /* Col size of A */
                    orA= rA; /* Save the original Rsize of A */
                    oCA= cA; /* Save the original Csize of A */
                    rB = s1 ->rsize; /* Row size of B */
                    cB = s1 ->csize; /* Col size or B */
                    /* * * * * *
                    * Check if the B Matrix is null and if so abort the expansion.
                    */
                    if ((rB == 0) || (cB == 0) || (s1->p_prm == NULL)) {
                        fprintf(stderr, "\nIn expn[B]: %s has null matrix!", q->n_name);
                        fprintf(stderr, "\n%s may have not been declared!", q->n_name);
                        fprintf(stderr, " Expansion must be aborted ...\n\n");
                        exit(1);
                    }
                }
                /* * * * * *
                * Rem: B is inserted into A (A<-B (or A is expanded by B)
                * Thus the following logic sets the stage for an expansion:
                */
            }
        }
    }
}

```



```

/* * * * <<< Method IIa Exception >>> * * * * * * * * * * *
* Catch all the ones (+)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 (test cases are
* t2, t10 and wgood). Only consider rows above rowMark.
* -----
*/
if (!thinA) {
  i = rowMark;
  for (i=rowMark-1; i>=0; i--) {
    if (A[i].p_row[cubA] > 0) {
      C[i].p_row[cubA] = A[i].p_row[cubA];
      C[i].p_row[cubA] = 0;
      fprintf(epn, "\nMethod IIa expn (linked last place)!");
    }
  }
}
/* * * * <<< Method IIb Exception >>> * * * * * * * * * * *
* Moving the y's form the marked row if they are "+".
* -----
*/
for (j=clbA; j < clb; j++) {
  if (A[rlb].p_row[j] > 0) {
    C[rub].p_row[j] = A[rlb].p_row[j];
    C[rlb].p_row[j] = 0;
    fprintf(epn, "\nMethod IIb expn (linked recursive loop)!");
  }
}

```

C.4 Algorithm for expansion method 3 (centrally located)

```

/* Determine indexes for B whose size is rB x cB.
* rlb = row lower bound, rub = row upper bound
* clb = col lower bound, cub = col upper bound
* Remember: A<- expanded by <-B
*/
rlbA = 0;
rubA = rowMark -1;
clbA = 0;
rlb = rowMark;
rub = rlb + rB -1;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Find point in A for expanding B (1st '-' in marked row)
*/
j = 0;
while (A[rowMark].p_row[j] >= 0) j++;
clb = j;
cubA = clb + ONE;
cub = clb + cB -1;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Mark the RHS of A to be pushed right past B
* Mark the LHS of A to be replacement starting point.
*/
colMarkRht = cubA;
colMarkLft = clb;

```

```

/*-----*
 * Finish all columns in marked row up to clb (x's in comment
 * below).
 */
for (j = 0; j < clb; j++) C[rowMark].p_row[j] = A[rowMark].p_row[j];

/*-----*
/* Finish the middle box...
 *
 * C:  a a a a g g g g g g
 *      a a a a g g g g g g
 *      x x b b b b b b a a
 *      a a b b b b b b a a
 *      a a b b b b b b a a
 *      a a b b b b b b a a
 *      a a b b b b b b a a
 *      a a a a a a a a a a
 *      a a a a a a a a a a
 *
 * At this point, a's come from A and b's are put from
 * the B matrix (x's have been put but a's have not).
 * g's are then added in the next segment of code.
 */
(void)expand(epn, ZERO);

/*-----*
 * Finish update for the upper part of A which (g's in
 * the above comment) goes in the URH corner of C.
 */
clbA = colMarkRht +1;
cubA = ocA -1;
rlbA = 0;
rubA = rowMark -1;
clb  = cA - (cubA - clbA) -1;
cub  = cA - 1;
rlb  = 0;
rub  = rowMark -1;

e = rlb;
for (i = rlbA; i <= rubA; i++) {
    f = clb;
    for (j = clbA; j <= cubA; j++) {
        C[e].p_row[f] = A[i].p_row[j];
        if (!(f>cub)){f++;}
        else {
            fprintf(stderr, "\n1-Method III error(clb)!\n");
            exit(1);
        }
    }
    if (!(e>rub)){e++;}
    else {
        fprintf(stderr, "\n2-Method III error(rlb)!\n");
        exit(1);
    }
}
}

```

```

/*-----*
* colMarkRht is the clb + 1 replacement point
* Finish update for the lower part of A (LHS or x's)
*
* A:  d d d d d      "-" is replaced by B
*     d - d d d
*     x y y y y      Put x's ---> C
*     x y y y y
*/
e = rowMark + rB;
for (i = rowMark + 1; i < orA; i++) {
    for (j = 0; j < colMarkLft; j++)
        C[e].p_row[j] = A[i].p_row[j];
    e++;
}
/*-----*
* colMarkRht is the clb+1 replacement point (delta =cA-ocA)
* Finish update for the lower part of A (RHS or y's)
* colMarkLft is the col 2 in fig below where the minus is
* (which is the same as the clb).
*
*     1 2 3 4 5
* A: 1 t t t t t      t's & e's are put using above code
*     2 e - d d d      "-" is replaced by B (d's are handled
*     3 x y y y y      as exceptions below).
*     4 x y y y y      Put y's ---> C
*/
e = rowMark + rB;
for (i = rowMark + 1; i < orA; i++) {
    f = cA - 1;
    for (j = ocA - 1; j >= colMarkLft; j--)
        C[e].p_row[f--] = A[i].p_row[j];
    e++;
}
/*-----* <<< Method III Exceptions >>> *-----*
* -----*
* The output from the transition being expanded must go to
* the same place it was before. Check the rest of the row
* right of the intersection of A[rowMark][colLftMark] for
* pluses (+1). Place them in the last row of the B matrix
* inside of C. The same distance from the last col in C
* as they are in from the last col in A.
*
*     1 2 3 4 5
* A: 1 t t t t t      t's, e's, x's are put using above code
*     2 e - d d d      "-" is replaced by B (d's are handled
*     3 x y y y y      as exceptions below).
*     4 x y y y y      Put d's ---> C
*
* The idea is to connect the output from the transition being expanded
* to the same place as it originally was connected to in A (a place
* basically). Note the following code assumes that the last row of B has
* a plus (i.e., that its actually connected as it was in the higher
* level abstraction to another place.
*

```

```

* A case where this is not true: SEQ{P1(), P2(), STOP}. Until you know
* exactly what's in the transition being expanded you cannot decide to
* eliminate the connection. Here the STOP doesn't have an O/P place!
* This case is assumed not to occur. First print some diagnostics:
*/
if (s0->type == NDC_PROC){
    fprintf(epn, "\nMethod III exception!");
    prtExpn(epn);
    print_prm(epn, C, rA, cA); fprintf(epn, "\n");
    zeroGapEnds = colMarkRht + (cA - ocA);

    /* Zero out the columns starting with the column ColMarkRht
    * making sure to stay above the rowMark
    */
    for (i = 0; i < rowMark; i++)
        for (j = colMarkRht; j < zeroGapEnds; j++)
            C[i].p_row[j] = 0;

    /* C <- A for the values on the right of the zeroGap
    * column(s) and above the rowMark. Rem... the colMarkRht
    * defines the boundary in A (not C) where the expansion
    * occurs (just one col to the left of the colMarkRht column).
    */
    zeroGap = cA - ocA;
    for (i = 0; i < rowMark; i++)
        for (j = colMarkRht; j < ocA; j++)
            C[i].p_row[zeroGap + j] = A[i].p_row[j];
    for (j=colMarkRht; j< ocA; j++)
        C[rowMark+rB-1].p_row[cA-(ocA-j)] = A[rowMark].p_row[j];
}

```

C.5 Expand algorithm for combining co-matrices

```

/* Expand copies the old matrices (A, B) to the new one (C). */
void expand(FILE *epn, int rm) {
    int i, j, e, f, m, n;          /* Miscellaneous indices */
    e = rm; m = 0;
    for (i = 0; i < rA; i++) {
        f = 0; n = 0;
        for (j = 0; j < cA; j++) {
            if ((i>=rlb) && (i<=rub) && (j>=clb) && (j<=cub)) {
                C[i].p_row[j] = B[m].p_row[n++];
                Bflag=TRUE_;
            }
            else {
                if ((i>=rlbA) && (i<=rubA) && (j>=clbA) && (j<=cubA)) {
                    C[i].p_row[j] = A[e].p_row[f++];
                    Aflag=TRUE_;
                }
            }
        }
        /*rof*/
        if (Bflag) {Bflag=FALSE_; m++;}
        else
            if (Aflag) {Aflag=FALSE_; e++;}
    } /*rof*/
}

```

C.6 User defined data types

```

/* Integer array of pointers to the rows in the matrix called the Process
 * Relation Table (prm) which is dynamically allocated (2-D array matrix).
 */
typedef struct int_array
{
    int *p_row;
} IArray;
typedef IArray *p_matrix;

typedef struct entrydef          /* Symbol Table entry definition */
{
    char      *name;            /* Symbol name */
    short     type;            /* Sym type (assume < 32,767 impl dpndt) */
    short     uid;             /* Unique id number (process id or pid) */
    char      *frate;          /* Failure Rate in ASCII */
    char      *fprob;          /* Failure probability in ASCII */
    char      *sprob;          /* Service Probability in ASCII */
    char      *srate;          /* Service Rate in ASCII */
    char      *p_pl;           /* Process list ptr (can be diff types) */
    short     rsize;           /* Number of rows in PR Matrix */
    short     csize;           /* Number of cols in PR Matrix */
    p_matrix  p_prm;           /* Process Relation (PR) Matrix */
    struct    entrydef *next;   /* Link to next ENTRY */
} ENTRY;
typedef      ENTRY *entryptr;

typedef struct nodedef
{
    char *n_name;                /* Ptr to the node/symbol name */
    char *n_fail;                /* NULL if no fail rate/prob spec'd */
    short israte;                /* Boolean: legal vals (-1, 0, 1) */
    short n_type;                /* Node type consistent w/ symbols */
    short uid;                   /* System level unique identifier */
    struct nodedef *link;        /* Ptr to next node, if any */
} NODE;
typedef NODE *nodeptr;          /* Ptr to a NODE structure */

```

APPENDIX D

RAILROAD CROSSING USING A MONITOR

D.1 Overview of the multiple train / monitor problem

This appendix describes a solution to: (1) the race (safety) hazard (described in ¶5.5) and, (2) controlling passage of multiple trains using a monitor to arbitrate the trains and the gate. Figure E.1 shows the monitor's finite state machine. We assume that trains cannot arrive simultaneously but that they do arrive in close enough succession that it would be dangerous for the gate to be opened if another train is pending. The Petri net of Figure E.2 is a translation of the CSP in Figure E.2. Table E.1 describes the markings and failure states.

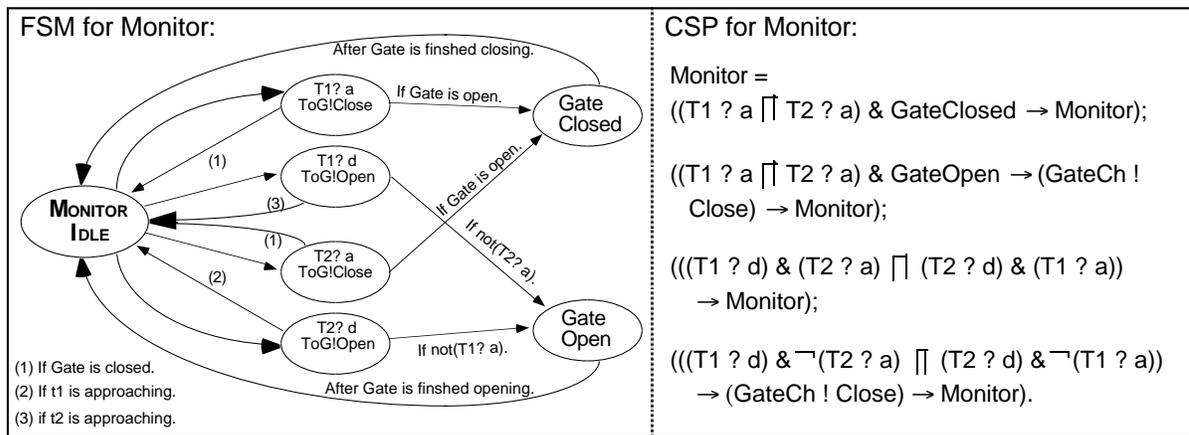


Figure D.1 Finite state machine and CSP for the monitor.

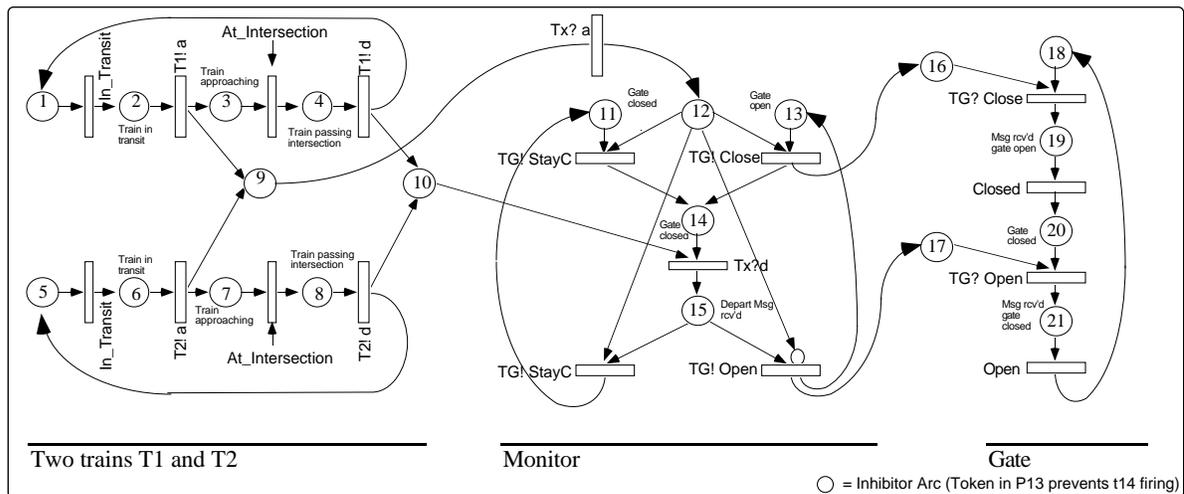


Figure D.2 Petri Net for the monitor (controller) to handle multiples trains.

Improving the system's performability is accomplished using more "slack" time for the Gate process to finish its task. Requiring the Train to send the arriving "a" signal sooner effectively increases the slack. Thus we have analyzed the Performability of the system by changing the slack time. The Stochastic Petri net of Figure D.2 is analyzed for reliability of the system under various failure modes. In this case, the Petri net elucidated the need for additional synchronization (so as to avoid a safety-critical failure). Accordingly, this is facilitated by translating CSP specifications into Stochastic Petri nets.

TABLE D.1
FAILURE MODES AND MARKINGS FOR THE RR-MONITOR

Mrkng	Monitor	Trains	Gate	Possible Failure Type
M1	Status = open	Both in transit	Open	Assume failure is not possible
M2	Status = open	TxCh ! a	Open	Critical communication failure
M3	TxCh ? a	Tx approaching	Open	Critical communication failure
M4	Status=pending train & GateCh!close	Tx approaching	Open	Critical communication failure
M5	Status = wait	Tx approaching	GateCh?close	Critical communication failure
M6	Status = wait	Tx approaching	Closing	Critical mechanical failure
M7	Status = closed	Tx at crossing	Closed	Assume failure not possible
M8	Tx ? a	Tx at crossing	Closed	Critical communication failure
M9	Status = closed	TxCh ! d	Closed	Non-critical communication failure
M10	Status= pending train and TxCh ? d	Tx approaching + one in transit	Closed	Non-critical communication or critical system failure (of monitor) possible.
M11	Status= not pending train and closed	One at crossing, one in transit	Closed	Assume failure is not possible
M12	TxCh ? d	Both in transit	Closed	Non-critical communication failure
M13	GateCh ! open	Both in transit	Closed	Non-critical communication failure
M14	Status = wait	Both in transit	GateCh?open	Non-critical communication failure
M15	Status = wait	Both in transit	Opening	Non-critical mechanical failure
FM16	Mcf and Mncf			Communication failures
FM17	Mcf and Mnfc			Mechanical failure (of gate)
FM18	Mcsf			System failure (of monitor)
FM19	Mtf			Timing failure (of train/gate)

Communication failures possible (Key: a → approaching, d → departing):

- 1) Failure when train sends message.
- 2) Failure when monitor receives message.
- 3) Failure when monitor sends message.
- 4) Failure when gate receives message.

In the Petri net of Figure D.2, we assume that all transitions can fail. The failure modes associated with transitions can be translated into failure modes of their corresponding CSP actions. When interpreting the failures of these actions, the user should identify critical failures. Improbable failures are easily identified in the Petri net (i.e., some transitions may not realistically fail or can be reasonably tolerated). Such evaluations can lead to an augmentation of the system model such as that of the multi-train/monitor system shown in Figure D.2. The markings in Table D.1 are based on the feasible states that trace the natural (and familiar) process: (M1) an idle state, (M2-5) communication transactions between the train, monitor, gate and *status = pending train*, (M6) gate begins to close, (M7-9) process of a new train arriving while the current train is passing, (M10) monitor has to decide not to open the gate when the current train departs since there is a pending train, (M_{Csf}) safety critical failure of the monitor, (M11) the current train starts the departing process and no trains are pending, and (M12-15) involve the actions necessary to restore the system to the idle state.

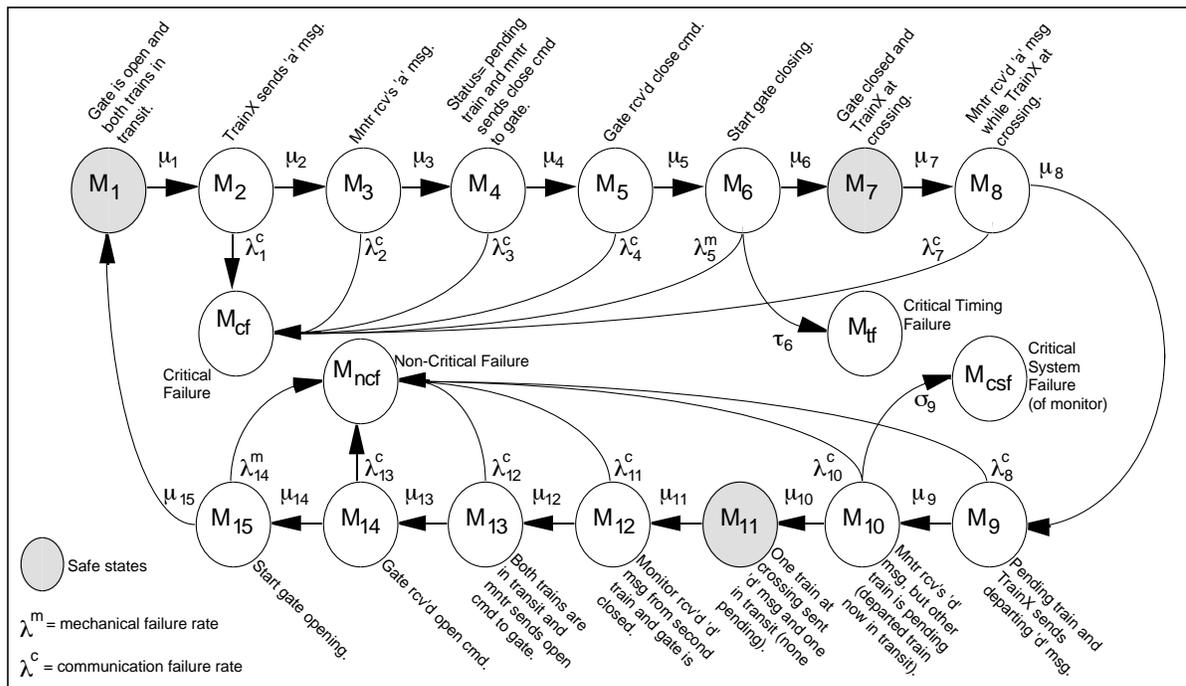


Figure D.3 State transitions [CTMC] for the trains-monitor-gate.

Figure D.3 shows the formalized flow of events and actions (i.e., CTMC) which include two failure states: (M_{cf}) safety critical failures involving gate closure, and (M_{ncf}) non-critical failures involving gate opening. Markings FM16-19 enumerate all failure categories. Realistically, one should account for the transitions which take the system from anywhere trains are being received (or are passing by) to new arrivals without having to visit the idle state. Admittedly this diagram is simplified, yet it incorporates all states necessary for receiving subsequent trains (assuming arrivals are not simultaneous).

Markings M6 and M7 are (safety) critical markings because the slow firing transitions ($TG?close [t_5]$) and ($Closed [t_6]$) make it possible for the train to enter the intersection before the gate has properly (or completely) closed. Similarly, non-critical conditions occur when the train departs the intersection but the gate stays closed resulting from the slow firing of transitions ($TG?open [t_7]$) and ($Open [t_8]$).¹

The CSP specification (and the corresponding Petri net) can be refined or augmented to state how such hazards could be avoided or handled. For example, communication failures can be handled using time-out and re-transmit techniques. Gate closing failures can be handled by sounding an alarm. Tolerance to time-related failures can be improved by requiring more slack time. In Figure D.3 the only critical deadline, is the one that requires the gate to close before the train arrives (i.e., gate closure must complete in a time less than:

$$\frac{\text{(distance to the gate when "arriving" signal was sent)}}{\text{(the speed of the train)}}$$

A failure mode resulting from incorrect (both logical and timing) operation of the monitor is modeled. The monitor must track all approaching trains, and command the safe operation of the gate. In controlling the gate, the monitor prevents the gate from opening when a train departs if another is too close down the line that opening the gate would endanger other traffic since the next train could arrive before the gate could again be closed.

¹Note: Waiting in M7 is assumed so that the gate has time to close (the end of the delay is the event that allows the next state transition to occur. Considering M11 we see that no waiting is necessary since the gate is already closed (i.e., a pervious train just passed trough).

D.2 Stochastic analysis

Using conventional techniques (i.e., SPNP's Markov solvers), discrete and/or continuous analyses can be performed. *Mathematica*[®] was used to compute the reliability of the railroad crossing system with different failure rates (or probabilities) and service rates (e.g., speed of the train, gate closing/opening rates etc.). The sensitivity of the system to variations in train speed (μ_7) and the gate closure rates (μ_6) were evaluated. The system's performability was studied to determine how reliably the gate closes before the train arrives with and without hardware and communication failures (i.e., mechanical gate failures [λ_5 , λ_{13} superscript 'm'] and communication failures [$\lambda_{1,2,3,4,7}$, and $\lambda_{8,10,11,12,13}$ superscript 'c']). *The values used (and hence the results of the analysis) are only for illustrating the approach (i.e., do not attach empirical significance to the failure rates or MTTFs obtained.* This type analysis is useful in exploring different fault-handling mechanisms and the cost of providing fault-tolerance.² The discrete analysis was not performed.

D.2.1 Continuous analysis

The results shown in Figures D.4 through D.7 predict reliability over the same operational life: up to 10,000 time units (tus) on the x-axis (each unit is further divided into 1000 sub-tus). The sensitivity of the a system to different transition rates (i.e., μ_6 and μ_7 for the various train speeds and the speed of the gate closing) are presented in Figure D.4. Note, the "rel" stands for reliability and is the instantaneous reliability of the data point at 10,000 tus. However, since the reliability was so close to zero the plotter stopped at the position indicated by the arrow head. The predicted mean time to failure is also provided (MTTF). In Figure D.5 the effect of varying the timing failure rate, in the presence of timing failures [including σ_9 failures caused by software or hardware or timing problems]) is shown.

²More elaborate fault-handling and fault-recovery mechanisms should be used to tolerate or prevent safety critical failures, while less attention may be paid to non-safety critical failures. Failure to open the gate may anger people waiting at the crossing but such failures can be handled inexpensively by providing a mechanism to manually open the gate. On the other hand, failure to close the gate is more severe, so traffic at the crossing should be alerted reliably and automatically.

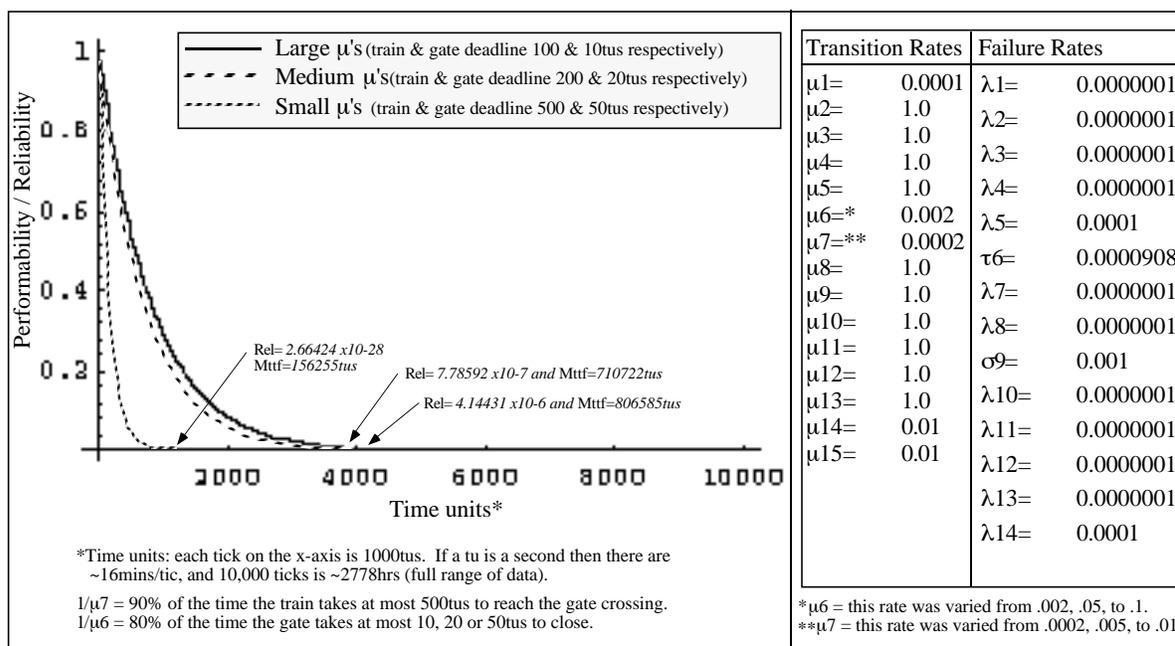


Figure D.4 Performability for different train and gate speeds (based on CTMC).

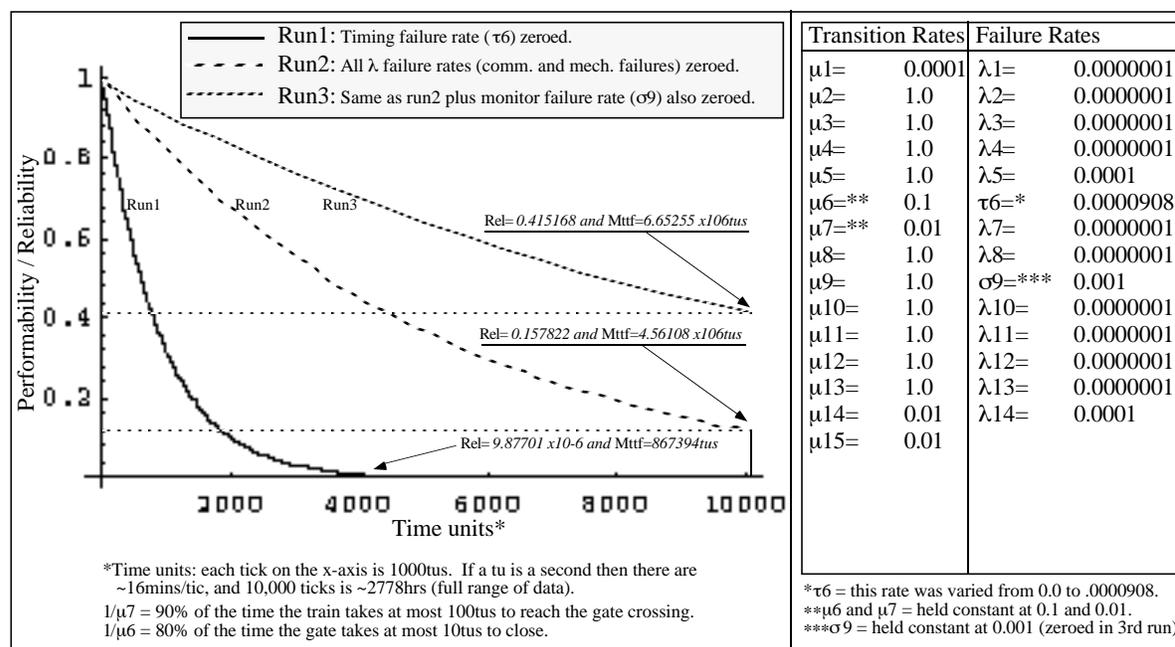


Figure D.5 Performability for different timing failure and monitor failure rates.

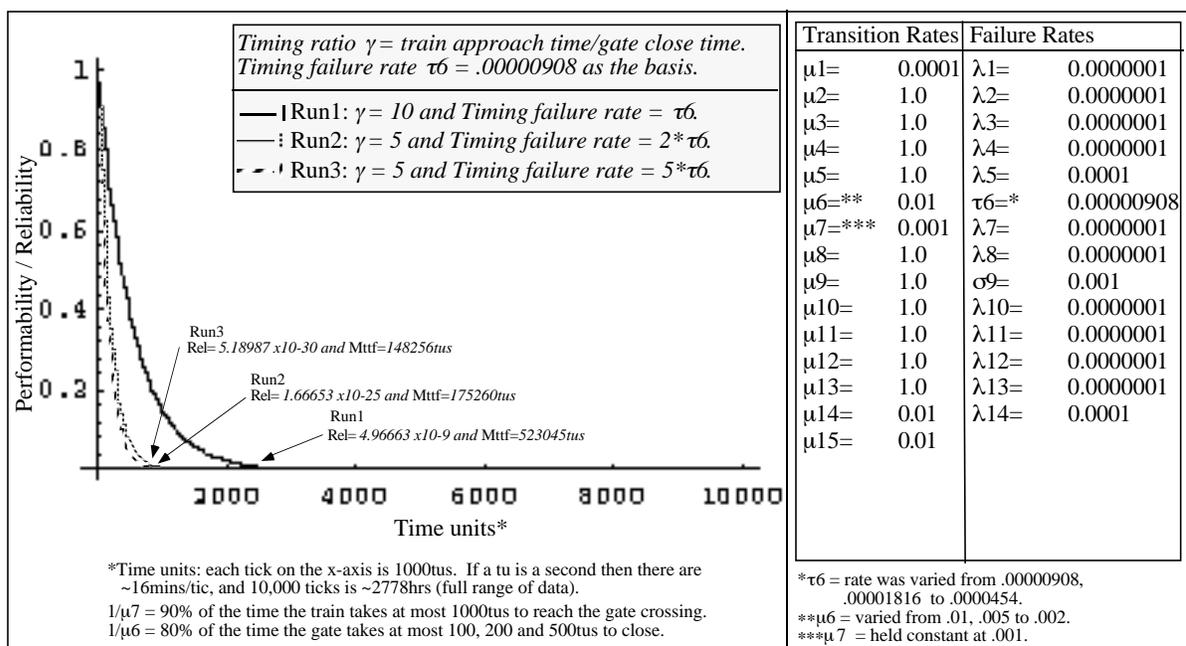


Figure D.6 Performability for different train speeds and gate closing speeds.

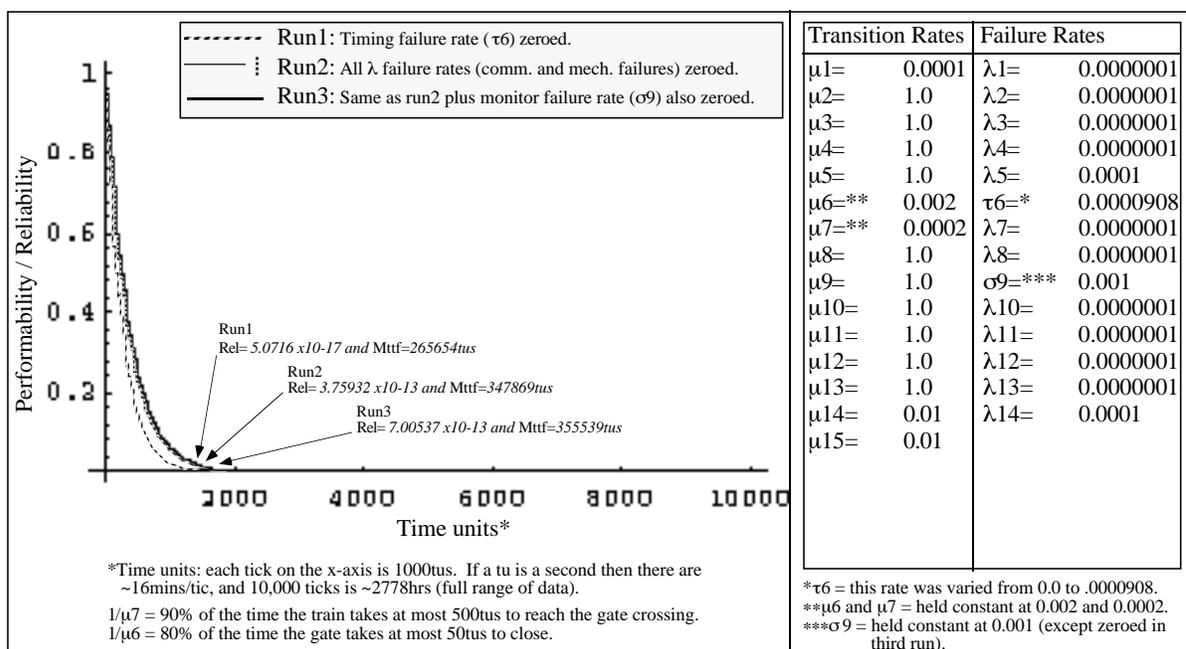


Figure D.7 Performability for different train speeds and gate closing speeds.

Figure D.6 shows the relation between the time needed for the train to reach the intersection ($1/\mu_7$), the time needed for the gate to close ($1/\mu_6$), and the timing failure rate (τ_6). These parameters are negatively correlated (i.e., as the slack time $[1/\mu_7 - 1/\mu_6]$ gets smaller τ_6 increases). The differences between rates associated with the train and the gate transitions were taken as a factor of 10, 5 and 2 for runs 1 - 3 while the τ_6 timing failure rate varied from 0.00000908 by a factor of 2 and 5 for runs 1 - 3 respectively. As can be seen from the graphs, the performability of the system decreases dramatically as the slack time decreases.

In order to study the effect of the timing critical transition rates on the predefined failure rates Figure D.7 is included. Compared this figure to Figure D.5. All of the parameters are the same except that instead of assuming large transition rates for μ_6 and μ_7 (i.e., 0.1 and 0.01 respectively) smaller rates were assumed (i.e., 0.002 and 0.0002).

D.3 Summary

The results show that the model is fairly sensitive to small changes in the rate assignments. There is less of an impact to the performability caused by the inherent failure rates of the subsystems when the transition rates are small. For example, comparing the difference between the best and the worst MTTF in each of the three runs of Figure D.5, we find a difference of a factor of 10, whereas that same comparison in Figure D.7 yields only a difference factor of 0.5 (approximately). Once again, do not attach any significance to the actual numbers. These numbers only illustrate the usefulness of these analyses in designing real-time systems with sufficient slack times and fault-tolerance to achieve a desired level of performability.