SPECIFICATION AND ANALYSIS OF STOCHASTIC PROPERTIES FOR CONCURRENT SYSTEMS EXPRESSED USING CSP

The members of the Committee approve the doctoral dissertation of Frederick T. Sheldon

Krishna M. Kavi  Supervising Professor

Bill D. Carroll

Roger S. Walker

Bob P. Weems

Hee Yong Youn

Kishor S. Trivedi

Dean of the Graduate School
DEDICATION

To my parents Donald and Monica and youngest brother Bruce whom we miss very much and dearly love. Also, to my sister Mary and her three daughters Josie, Danny and Alicia. Finally, to my brother Larry who I love and ask that God have mercy on us both.
SPECIFICATION AND ANALYSIS OF STOCHASTIC PROPERTIES FOR
CONCURRENT SYSTEMS EXPRESSED USING CSP

by

FREDERICK T. SHELDON

Presented to the Faculty of the Graduate School of
The University of Texas at Arlington in Partial Fulfillment
of the Requirements
for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 1996
ACKNOWLEDGEMENTS

In 1992, I received a fellowship from the Systems Integration Branch of NASA Langley Research Center (supervised by Steve D. Young, Carl Elks, Dave E. Eckhardt, and Wayne Bryant et al.). I am very thankful for the opportunities and experiences NASA thus provided and for the help that Seung-Min Yang provided in winning the contract. Bill D. Carroll maintained the fellowship and contributed remarkably to my experiences in both research and teaching as did Bob P. Weems. I greatly respect them both and appreciate their attention. Roger S. Walker inspired this work into the realm of stochastic analysis. Hee-Yong Youn provided insight into the significance of performability as a stochastic property.

Krishna M. Kavi is my major professor and who from I have received extraordinarily precious insights throughout my doctoral program. I am very thankful for his dedicated teaching. We also received assistance from Professor Kishor S. Trivedi, who kindly extended his time, laboratory facilities, tools and the help of his graduate students.

I am grateful to my friends at Lockheed Martin, my former employer, and the USAF whom, in the early years, provided research assignments, letters of support, financial support, flexible hours and encouragement. Trina Bornejko of the USAF helped me compete for the NASA GSRP Fellowship. I would also like to thank Chris Turbeville, David Sheely, Farhad Kamangar, Terry Sergeant, Sherman Reed and Marco Bernardo (Università` di Bologna) whom helped with analysis techniques, C-code and tools. The conscientious support from L. Peterson, P. Hsia, R. Elmasri, J. Fitzer, K. Ecker, S. Greiner, and K.H. Kim (UC Irvine) was very much appreciated. Finally, my most sincere thanks goes to Bill and Patty Howell and their son Thomas. They opened their home and let me be a part of their family while I was TDY at NASA Langley.

December 1, 1995
ABSTRACT

SPECIFICATION AND ANALYSIS OF STOCHASTIC PROPERTIES FOR CONCURRENT SYSTEMS EXPRESSED USING CSP

Publication No. __________

Frederick T. Sheldon, Ph.D.
The University of Texas at Arlington, 1995

Supervising Professor: Krishna M. Kavi

This work offers an innovative approach to predicting system behavior (in terms of reliability and performance) based primarily on the structural characteristics of a formal functional specification. This work extends parts of the work by E-R. Olderog, by developing a CSP-based grammar and canonical CSP-to-Petri net translation rules for process composition and decomposition. The mechanism for process composition is codified in the CSP-to-Stochastic Petri net (CSPN) tool and consists of expanding the process description represented as a series of small Petri nets into larger and larger nets while preserving structural relationships and functional nomenclature. In the last phase, the tool reconciles synchronization points (for communicating processes), stochastic annotations and generates an executable "spnp.c" file used for stochastic analysis. Numerous command line options provide a high degree of versatility and control to the user including the ability to generate and view the Petri net graph. CSPN supports systematic specification, automatic translation and subsequent augmentation (e.g., failure rates, service rates, and transition probabilities) of the resultant Petri nets for assessing stochastic properties of different
candidate implementations and relating those properties back to the specification level.

The CSPN tool and methodology is based on the sound formalism of CSP. The approach abstracts the critical information necessary for performance analysis and translates it to a Petri net for exploring feasible and critical markings and subsequent analysis of the Markov state space. The CSP-based language, P-CSP, is used for system specification. The CSPN tool parses the P-CSP specification and, using the set of canonical translation rules, produces equivalent Petri nets represented as coincidence matrices.

In the design cycle, it is important to systematically and iteratively incorporate capabilities (enhancements) such as fault-tolerance, and then re-evaluate their impacts to optimize design parameters in terms of their stochastic properties. Thus, the approach advanced in this work (1) takes the results of the stochastic analysis and provides a formal and automated mechanism for annotating those results (and their parameterization) back into the original specification and, (2) those results are then automatically incorporated into the computation of subsequent refinements.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ........................................................................................................ iv

ABSTRACT ........................................................................................................................ v

LIST OF FIGURES ........................................................................................................... xi

LIST OF TABLES ............................................................................................................. xiii

CHAPTER 1

INTRODUCTION ............................................................................................................... 1

1.1 Problem definition and goal ..................................................................................... 1

1.2 Motivation ............................................................................................................... 1

1.2.1 Predicting the reliability of formal specifications ....................................... 2

1.3 Organization ........................................................................................................... 4

CHAPTER 2

SURVEY OF RELATED WORK .................................................................................... 5

2.1 In the beginning ...................................................................................................... 5

2.2 Integrating function, performance and structural modeling ................................ 6

2.3 Process algebras provide functional semantics ............................................... 6

2.4 Process algebras provide a notion of program equivalence ............................... 7

2.5 Stochastic process algebras add performance semantics ................................. 8

2.6 Petri nets add structural semantics in a distributed setting ............................... 9

2.7 Other related work ............................................................................................... 11

2.8 Communicating Sequential Processes ............................................................... 12

2.8 The CSP-based language (P-CSP) primitives .................................................. 14

2.9 Stochastic Petri nets ............................................................................................ 16
2.10 Introducing SPNP’s C-based Stochastic Petri net Language (CSPL) .......... 18
2.11 The original contribution of this work .................................................. 20
2.12 Where does this work fit ........................................................................ 21

CHAPTER 3
MAPPING CSP TO PETRI NETS ......................................................... 22
3.1 Mapping CSP to Petri nets ....................................................................... 22
3.2 CSP translation incongruencies .................................................................. 23
3.3 CSP translation mechanism and commentary ............................................. 24
3.4 Specification of failures and failure Handling in CSP ............................... 30
3.5 Preserving the trace behavior of CSP specifications .................................. 33

CHAPTER 4
CSPN TOOL OVERVIEW AND IMPLEMENTATION DETAILS .......... 38
4.1 CSPN tool overview ................................................................................. 38
4.2 Translation phases of the CSPN tool ....................................................... 39
4.3 Running the CSPN tool ............................................................................ 42
4.4 CSPN data structures ............................................................................... 43
4.5 Petri net compositions ............................................................................. 45
4.6 P-CSP semantics as it relates to the data structures .................................... 52
4.7 P-CSP’s usage of failure and service rate annotations .............................. 54
4.8 Linking synchronization primitives ............................................................ 55
4.9 CSPN file descriptions ............................................................................. 56
  4.9.1 Cmd_line.c description .................................................................... 56
  4.9.2 The csp.l and csp.y descriptions ..................................................... 56
  4.9.3 Symbol_cspy.h and symbol_cspy.c description ............................... 57
  4.9.4 Net.c description ............................................................................ 58
  4.9.5 Prlist.c description ......................................................................... 59
CO-MATRIX EXPANSION ALGORITHMS ................................................................. 89

APPENDIX D

RAILROAD CROSSING USING A MONITOR ......................................................... 101

BIBLIOGRAPHY ............................................................................................................. 110
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Linking failure behavior to specification characteristics</td>
<td>3</td>
</tr>
<tr>
<td>2.</td>
<td>Trace equivalence versus bisimilarity</td>
<td>7</td>
</tr>
<tr>
<td>3.</td>
<td>CSP-to-Petri net example of trace equivalence versus bisimilarity</td>
<td>8</td>
</tr>
<tr>
<td>4.</td>
<td>An integrated approach of stochastic analysis [Bernardo95]</td>
<td>10</td>
</tr>
<tr>
<td>5.</td>
<td>SPNP input file structure</td>
<td>19</td>
</tr>
<tr>
<td>6.</td>
<td>Refinement of system models</td>
<td>21</td>
</tr>
<tr>
<td>7.</td>
<td>Example CSP to Petri net translation rules (P-CSP shown in lower portion)</td>
<td>22</td>
</tr>
<tr>
<td>8.</td>
<td>CSP to Petri net rules for Actions and the STOP and SKIP processes</td>
<td>24</td>
</tr>
<tr>
<td>9.</td>
<td>CSP to Petri net rules for Combining actions and processes</td>
<td>25</td>
</tr>
<tr>
<td>10.</td>
<td>CSP to Petri net example of combining actions to form one process</td>
<td>26</td>
</tr>
<tr>
<td>11.</td>
<td>CSP to Petri net rules for Deterministic and nondeterministic choice</td>
<td>27</td>
</tr>
<tr>
<td>12.</td>
<td>CSP to Petri net rules for Parallel and synchronized parallel composition</td>
<td>28</td>
</tr>
<tr>
<td>13.</td>
<td>Comparing CSP and P-CSP to Petri net rules for recursion</td>
<td>29</td>
</tr>
<tr>
<td>14.</td>
<td>CSP to Petri net rules for recursion</td>
<td>30</td>
</tr>
<tr>
<td>15.</td>
<td>VMC specification for CSP to Petri net with failure annotations</td>
<td>32</td>
</tr>
<tr>
<td>16.</td>
<td>CSPN translation of the VMC example</td>
<td>34</td>
</tr>
<tr>
<td>17.</td>
<td>Structure of the feasible sequences of VMC Marking</td>
<td>36</td>
</tr>
<tr>
<td>18.</td>
<td>Partial VMC translation showing deterministic choice</td>
<td>36</td>
</tr>
<tr>
<td>19.</td>
<td>Activities associated with the translation phases of the CSPN tool</td>
<td>39</td>
</tr>
<tr>
<td>20.</td>
<td>Context diagram and translation phases of the CSPN tool</td>
<td>40</td>
</tr>
<tr>
<td>21.</td>
<td>P-CSP constructions with co-matrix and Petri net representations</td>
<td>44</td>
</tr>
<tr>
<td>22.</td>
<td>Choosing a combining method for expansion that depends on locality</td>
<td>45</td>
</tr>
</tbody>
</table>
30. Diagram of expansion method one ................................................................. 46
31. Diagram of expansion method two ............................................................... 46
32. Diagram of expansion method three (shows Petri nets and co-matrices) ........ 47
33. CSPN run shows before and after combining coincidence matrices ............... 48
34. Data structure for nesting and adjacency detected in the specification .......... 49
35. Process hierarchy for system “SysSimpleEx” with exaggerated nesting ........... 51
36. Relational diagram for the network (or process hierarchy) data structures ...... 52
37. Definitions of the symbol table and process hierarchy data ............................ 53
38. Specifying failure annotations in P-CSP and the resulting Petri net ............... 54
39. Resolving synchronization links .................................................................... 55
40. Pure CSP specification of the railroad crossing problem ............................... 63
41. P-CSP specification for parallel composition of the railroad crossing .......... 64
42. P-CSP specification for the (tail type) recursive composition ....................... 65
43. CSP and P-CSP specifications which address race hazard ............................ 66
44. Railroad crossing Petri net showing all possible failure transitions ............... 68
45. Markings and requisite Markov state transition diagrams ............................ 70
46. Results of the continuous analysis ............................................................... 72
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. THEORETICAL FOUNDATIONS OF CSP</td>
<td>13</td>
</tr>
<tr>
<td>2. MARKINGS FOR VMC PETRI NET</td>
<td>35</td>
</tr>
<tr>
<td>3. GENERAL STEPS FOR USING THE CSPN TOOL</td>
<td>41</td>
</tr>
<tr>
<td>4. LISTING OF THE CSPN COMMAND LINE OPTIONS</td>
<td>42</td>
</tr>
<tr>
<td>5. CONSTRUCTS USED IN P-CSP AND THEIR TYPE VALUES</td>
<td>53</td>
</tr>
<tr>
<td>6. SYMBOL TABLE UTILITY FUNCTIONS</td>
<td>57</td>
</tr>
<tr>
<td>7. NET UTILITY FUNCTIONS</td>
<td>58</td>
</tr>
<tr>
<td>8. DESCRIPTION OF INTERMEDIATE TRANSLATION FILES</td>
<td>61</td>
</tr>
<tr>
<td>9. DISCRETE ANALYSIS (Ptf = 0)</td>
<td>72</td>
</tr>
</tbody>
</table>
CHAPTER 1
INTRODUCTION

I have six honest serving men, they taught me all I knew. Their names are where and why and when, and how and what and who.

—Rudyard Kipling

1.1 Problem definition and goal

The main interests in this research involve dependability and fault-tolerance of computing systems in devising techniques to prevent, detect and compensate for anomalies. An experimental tool and modeling approach has been developed to explore the specification and analysis of stochastic properties for concurrent systems expressed using CSP. The idea is to translate the formal system description into the information needed to predict its behavior as a function of observable parameters (topology, timeliness, communications and failure categories). The modeling approach uses a theory based on proven translations between CSP (communicating sequential processes) and Petri nets. In particular, the tool translates the design specification, written in a CSP dialect called P-CSP, into stochastic Petri nets for analysis based on the structural and stochastic properties of the specification. The grammar and CSP-to-Petri net (CSPN) tool enable service and failure rate annotations to be related back to the original CSP specification. The annotations are then incorporated in the next round of translations and stochastic analysis. The tool automates the analysis and iterative refinement of the design specification process. Within this setting, we can investigate whether functional and non-functional requirements have been satisfied.

1.2 Motivation

Today's computing systems are large and complex [Basili91]. Therefore, informal and intuitive specifications are too vague and imprecise to capture the complete semantics of a
system's requirements [Gomaa94, Hall90]. A formal specification language is founded on mathematical principles and is used to describe system properties precisely and to provide a systematic approach to avoid ambiguity, incompleteness and inconsistency [Collins87, Abadi93, Alur90, Dahbura90, Delisle90, Deng92, 91, 90, Dillon92, Garlan90, Genrich92, Gerhart90, Heitmeyer91, Hird91, Hooman90, Van Leeuwen90, Wang93, Wing90, Wood90]. Formal specifications provide good support for designing a functionally correct system, however they are weak at incorporating non-functional performance requirements (like reliability) [Enand89, Palumba92]. Current systems must also have high performance and reliability. Techniques which utilize stochastic Petri nets (SPNs) are good for evaluating the performance and reliability for a system, but they may be too abstract and cumbersome from the stand point of specifying and evaluating functional behavior [Balbo95, 94, 92a,b, Ibe89, Choi93, 92]. Therefore, one major objective of this research is to provide an integrated approach to assist the user in specifying both functionality (qualitative: mutual exclusion and synchronization) and performance requirements (quantitative: reliability and execution deadlines). In this way, the merits of a powerful modeling technique for performability analysis (using SPNs) can be combined with a well-defined formal specification language. In doing so, we can come closer to providing a formal approach to designing a functionally correct system that meets reliability and performance goals [Wang94, 95].

1.2.1 Predicting the reliability of formal specifications

Our approach is based on the notion that formal, mathematically precise methods should be used to design complex, safety critical systems [Butler86, 88a,b 89, 93, Jahanian86, 87, Ostroff91, 92a,b]. Thus, given a formalized functional specification of a system and its external constraints (e.g., failure rates, communication delays, synchronization dependencies, deadlines), what mechanisms are available for avoiding or tolerating faults/errors and how do they impact the performance and reliability (i.e., performability) of the system [Meyer80a,b, 89a,b, Kavi92a,b, 94a,b, 95, Sheldon94]? The approach can be visualized from Figure 1.
As specifications are refined into detailed designs, the reliability and performance requirements can also be refined to reveal trade-offs in design alternatives such as deciding—what are the critical system elements;—what features of the system should be changed to improve the system's reliability;—or validating performance and reliability goals using stochastic system models. To address these design issues, in the our approach, the critical components of the requirements specification are abstracted. A system is specified using the P-CSP language providing a design specification. The CSP-based grammar does not restrict us from considering correctness properties; however, we are interested only that the structural properties be preserved. Once the specification has been translated, we enumerate modeling assumptions, estimate model parameters, and solve the model for specific values of the parameters using Markov analysis [Johnson89, Ciardo87, 89, 91, 92b, Sahner95]. At this point it is easy to introduce timing constraints among feasible markings of the net and to employ any of the numerous tools developed for stochastic Petri net analysis (e.g., GreatSPN, SPNP, GSPN) [Chiola93a,b, Ciardo92a, 93b, Marsan84, 90]. Thus, having converted the design specification into Petri nets allows the system model to be analyzed against non-
functional requirements using any of the various Petri net tools available to predict its behavior [Johnson88, Lloret92, Shatz90, 88]. SPNP was chosen for our purpose [Ciardo93b, 92a, 89, Mainkar93].

1.3 Organization

The following chapters provide a survey of the related work, mapping of the CSP based language P-CSP to Petri nets, CSPN translation tool overview and its implementation details, as well as an illustration of the usefulness of the tool and the conclusions. There are four brief appendices which cover the complete set of CSP-to-Petri net canonical translation rules, P-CSP grammar, coincidence matrix expansion algorithms and finally another example showing a more complex version of the railroad crossing using a monitor to arbitrate multiple trains arriving in tight succession.
CHAPTER 2

SURVEY OF RELATED WORK

The longer I live, the more I realize the impact of attitude on life. Attitude, to me, is more important than facts.

*Charles Swindoll*

2.1 In the beginning

Stochastic Petri nets and stochastic process algebras have somewhat of a common evolution. Indeed, their original definitions (standard Petri nets and pure process algebra) did not include any temporal information and thus were only used for qualitative (ordered sequences of actions/events) analysis of concurrent systems [Ribaudo95, Donatelli95]. Extending these basic formalisms with a notion of time has allowed the study of quantitative properties of systems. In this work the notion of *time* is studied in relation to the structural properties of the process algebraic (CSP) specification and is introduced after translating the system description into Petri nets. Some of the original ideas used in this work came from a tutorial by K.M. Kavi and B.P. Buckles, *Formal Methods for the Specification and Analysis of Concurrent Processes*, Tutorial Notes, 1993 Int'l Conf. on Parallel Processing, Lake Charles, IL, August 20, 1993. This tutorial examined and extended a Petri net semantics for a subset of CCSP (the union of Milner's CCS and Hoare's CSP) based on [Olderog87]. The semantics are provided by operationally defining for each process term in the subset, a labeled place/transition net. Olderog's definitions uses predicate/transition style translation rules that are mainly concerned with concurrency, nondeterminism and recursion. This dissertation has extended and formalized parts of the work by Olderog, (described in “Operational Petri Net Semantics for CCSP,” LNCS-266, pp. 196-223, 1987) and that of Kavi and Buckles.
2.2 Integrating function, performance and structural modeling

Most research in formal methods has been in the development of theories, methods, and tools for the design and analysis of functional and logical (correctness) aspects of computing systems [Burns91, Carreno93, 92, Camilleri90, Cleveland93, Cook91, Craigen93a,b, Firth87, Hall90, Harel92, 90, Kavi92, Kemmerer90, Mokkedem95, Reisig92]. On the other hand, the analysis of performance is concerned with the statistical aspects of such systems [Covington89, Molloy82, 87]. Some have investigated simulation based approach to integrated performance and reliability modeling [Cutright93, 91, Bagrodia91a,b, 90, Geist93, 90, 89, Goyal87, Heidelberger92, 88, Haverkort93, Nicol93, Nicola92, Rubinstein89, Shahabuddin88]. These two research communities (functional vs. performance) have largely proceeded independently.¹ Stochastic Petri nets are well established in the field of performance analysis [Ciardo94, 93a,c, 92b, 91a,b, 90, 87, Lindemann93, ICASE93]. More recently, a growing interest in stochastic and probabilistic process algebraic techniques has emerged [Gilmore95, 94, Gotz93, Milner92a,b, Nicollin91a,b, Priami95]. Given the technological means and the financial basis (i.e., cost benefit), there is a clear need to treat quantitative performance parameters as non-functional requirements in functional specifications. This merging of functionality and performance is especially attractive and calls for the integration of qualitative and quantitative approaches to design and realization [Pomello92].

2.3 Process algebras provide functional semantics

Typically, process algebraic laws allow the rewriting of a system description into another, while preserving the notion of correctness that is captured by the equivalence used in the underlying semantic model [Donatelli95, 94]. Their inherent support of compositional reasoning enables the construction of complex systems as the combination of conceptually simpler systems [Buchholz94].

¹However, a number of efforts have put forth formal models of system behavior into the world of performance (and dependability analysis) [Priami and Bernardo articles].
2.4 Process algebras provide a notion of program equivalence

In the concurrent process algebra CCS, two programs are considered the same if they are bisimilar [Nielsen86]. Many researchers have demonstrated that the theory of bisimulation is mathematically appealing and useful in practice [Bloom95]. In CSP, the distinction between two processes can be understood by observing completed traces (sequences of visible actions performed by a process). The meaning of a process is determined according to a synchronization tree, which is a rooted unordered tree whose edges are labeled with symbols that denote basic actions or events (typically specified by a structured operational semantics). Two trees are trace equivalent iff they have the same set of traces.\(^2\) For example, two processes P and Q are distinguished iff there is some CSP context C[X] and string \(s\) such that only one of C[P] and C[Q] has \(s\) as a trace.

![Diagram](image)

A. Trace equivalent but not trace congruent.
B. CSP trace congruent but not bisimilar.
C. The machines are different but their transition behaviour is identical.

Figure 2. Trace equivalence versus bisimilarity.

In CCS two processes are different according to an interactive game-like protocol called bisimulation. Indistinguishable CCS processes are bisimilar. An example (assuming atomic actions a, b, c and d) of this relation are the two trees a(b + c) and (ab + ac) in Figure 2A, which are trace equivalent but not CSP trace congruent (i.e., in both CSP and CCS they are distinct processes). The trees (abc + abd) and a(bc + bd) in part B, are CSP trace congruent.

\(^2\)In contrast, given any set of operations on trees, trace congruence is defined to be the coarsest congruence with respect to the operations that refines the trace equivalence.
but not bisimilar (i.e., equal in CSP but considered distinct in CCS). Thus, we cannot simulate the behavior of the first machine with the second and visa versa. In part C, both machines are bisimilar because one can simulate the other and visa versa [Olderog86].

Figure 3 shows that a recursive loop allowing action "a" to be repeated indefinitely can be structured such that it provides CSP trace equivalence. However, the two Petri nets (PN\textsubscript{l} & PN\textsubscript{r} for left & right) are not bisimilar because the rightmost Petri net produces multiple instances of the action "b" (i.e., the structural properties of the two graphs are distinct). Thus, even though the visible actions are trace equivalent they are not bisimilar because PN\textsubscript{r} can distinguish the specific firings of individual "b" transitions while PN\textsubscript{l} can not.

**Figure 3.** CSP-to-Petri net example of trace equivalence versus bisimilarity.

### 2.5 Stochastic process algebras add performance semantics

*Stochastic* process algebras (SPAs) appeared only recently as a solution to an important problem of process algebras: their inability to express performance aspects of concurrent systems [Buchholz94, Brinksma95]. Like classical process algebras, they are abstract languages used to represent concurrent systems in a compositional way. Such algebras provide the specifier with a small set of powerful operators whereby it is possible to construct
process terms (compositional algebraic formulas) from simpler ones, without the graphical complexity of nets and making the task of detecting and modifying subsystems easy. SPAs extend the expressiveness of their predecessors by assigning each action a random variable determining its duration and thus producing algebraic descriptions of concurrent systems amenable to both functional and performance analysis.

Some early SPAs include PEPA [Hilston93a,b, 94], TIPP [Gotz93], MTIPP [Hermanns94], and EMPA [Bernardo95a,b,c 94, Herzog94]. Take for example EMPA, Extended Markovian Process Algebra, which comes equipped with an interleaving semantics, a Markovian semantics and a net semantics. The main drawback is related to state space explosion which is due to the interleaving representation of concurrency. This problem manifests itself in both the state space of the LTS (labeled transition system) underlying the process term and the reachability graph of the net semantics for the term. One idea that researchers have used is the notion of equivalence as a rewriting mechanism for reducing the state space of the LTS. The rewriting system is useful to analyze terms without generating the underlying state space and also to obtain equivalent terms whose state space is smaller.

In general, the reductions (at least those that are not a congruence) are based on simplifying assumptions and thus lead to approximate solutions. Clearly, the interleaving semantics of a parallel composition will lead to an exponential set of states since for instance, if we combine n processes each with m states, we can end up with as many as \( m^n \) states.

2.6 **Petri nets add structural semantics in a distributed setting**

Consider modeling and analysis of concurrent systems based on SPAs and SPNs as presented in Figure 4. Process algebraic laws enable rewriting one description of a system into another while preserving the notion of correctness. The transformation laws can be used to model the application of actual design principles in a strategy of stepwise refinement to obtain concrete descriptions of implementations from abstract system specifications. A key

---

3Such compositions of stochastic/probabilistic specifications can lead to complex analysis and approximate solutions.
feature of SPAs is compositionality. Compositionality concerns both the syntactical and semantic level of the language. Syntactical compositionality is related to system modeling using a small set of operators that make it possible to construct process terms (formulas) from simpler ones, without the graphical complexity of nets and making the task of recognizing or modifying components of the system easier. On the other hand, semantic compositionality is related to system analysis which enables the study of separate system components (provided an appropriate notion of equivalence over process terms is developed). This is accomplished by decomposing the system so that a given property of the composition can be recognized.

As an alternative to SPAs (and an approach similar to ours), a two phased approach can be envisioned. In the first phase, components of the concurrent system are represented as a term of the SPA which, in rich environments like EPOCA [Donatelli94] EMPA [Bernardo95], are equipped with an interleaving semantics accounting for both the qualitative (i.e., functional) and quantitative (i.e., performance) part of the system behavior. Thus, the

![Diagram](image)

Figure 4. An integrated approach of stochastic analysis [Bernardo95].

---

4 An interleaving semantics for a concurrent language maps programs to interleaving models. In these models, every parallel execution is simulated by means of the set of the alternative sequential executions obtained by just interleaving the activities occurring in the parallel execution itself. For example, consider terms \( a \parallel b \) and \( a.b + b.a \). From the interleaving point of view, these two terms are equivalent because each of them can perform action \( a \) followed by action \( b \), or action \( b \) followed by action \( a \). Classical interleaving models are labeled transition systems. Classical non-interleaving models are Petri nets, because the net semantics of the two terms above are quite different [Bernardo95].
interleaving model of the process algebraic representation can be projected onto both a functional and performance model (top half of Figure 4).

Phase 2 consists of automatically obtaining from the algebraic representation of the system an equivalent distributed representation (i.e., Petri nets or labeled transition systems). A suitable distributed model would be stochastic Petri nets (naturally, numerous tools are available to support performance evaluation within such a context GSPN, SPNP, SHARPE, ASSIST) [Johnson91, 88, Sahner95]. The net representation of the concurrent system is derived from the algebraic one without intervention of the designer and is useful when a less abstract form is needed to highlight dependencies and conflicts among activities, or to support establishing some properties (e.g., partial deadlock, race hazards). These cases can be easily checked only in a distributed setting. Yet, there are limitations to this approach because of the need to make simplifying assumptions which lead to approximate solutions.

2.7 Other related work

Wang presents a procedure (which could be automated) for transforming an Estelle specification into a Stochastic Reward Net (SRN) formalism.⁵ The objective of transforming Estelle into an SRN is to have a system designer specify a system using Estelle and then the specification is automatically transformed into an SRN to carry out the performance and reliability analysis [Wang94].

Davies and Schneider ['94] describe the language of real-time CSP used to specify reactive systems in terms of their communicating behavior (also see Reed91). Each system component is represented as a process that shows where communication takes place. By combining processes, a description of the system in terms of its components is produced. Moore ['90] shows the specification and verified decomposition of system requirements using CSP for an abstract voice transmitter. Peleska ['91] gives a formal method based on CSP to design fault tolerant systems combining algebraic and assertional techniques to formally

---

⁵Estelle is an ISO standard formal specification language and SRN is a well-developed modeling technique that is used to carry out performance and reliability analysis.
verify correctness properties. Liu and Joseph ['92] give a method for transformation of
programs constructed for a fault-free system into fault-tolerant programs suitable for
execution on a system susceptible to failures. Lee ['94a,b] gives a formal language GCSR
(Graphical Communicating Shared Resources) for the specification, refinement, and analysis
of (resource-bound) real-time systems. The semantics are defined through a precise
translation to ACSR, a timed process algebra. Execution of a GCSR specification is
supported through a precise correspondence between GCSR and ACSR and the operational
semantics of ACSR (e.g., requirements and design) [Gerber90, 88, Ben-Abdallah95, 94,
Choi95]. Priami ['95] gives a technique for integrating behavioral and performance analysis
with topology information using Stochastic pi-calculus. Van Glabbeek ['90] gives a
structural operational semantics of PCCS as a set of inference rules which constitute a
semantic mapping from the set of process expressions to a particular domain of probabilistic
labeled transition systems. Moller ['90] gives a temporal calculus of communicating systems.

2.8 Communicating Sequential Processes

CSP is a classic process algebra (like CCS [Milner80], and ACP [Bergstra84]). The CSP
model was developed by Hoare in the late 70's to early 80's and later, in 1986 extended by
Olderog [Olderog86, 87]. Table 1 gives five of the theoretical foundations that are supported
by CSP.6 The basic idea is that systems can readily be decomposed into subsystems common

---

6The theoretical foundations of CSP can be found in [Hoare85]. There, processes are presented as certain
mathematical elements (or structures) that can be manipulated algebraically, combined by various operators to
fork other processes, and proved or disproved to satisfy formally stated specifications. Fridge ['88] has
implemented a working version of the CSP model in LISP. Kourie ['87] has written a working version of the
CSP model in Prolog (without CSP's input/output notation for data transfer between processes). Delisle and
Schwartz ['87] have created a CSP programming environment where programs can be subjected to experiments
and animated on the screen. This model of CSP is written in Scheme. Finally, Olszewski ['93] has developed a
CSP laboratory for students of parallel programming which provides tools and facilities to experiment with, test
and analyze CSP descriptions/prototypes of parallel systems. The analysis includes automatic detection of
deadlocks and unsafe behaviors of CSP processes. Visualization facilities are planned with regard to
components of parallel systems and the communication between them.
TABLE 1
THEORETICAL FOUNDATIONS OF CSP [Hoare85]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mathematical abstraction of process interactions (communication, concurrency, recursion, etc.).</td>
</tr>
<tr>
<td>2</td>
<td>Rules to help in the implementation of processes (laws used to prove a specification is satisfied).</td>
</tr>
<tr>
<td>3</td>
<td>How processes can be composed together into systems where components interact internally and with their environment.</td>
</tr>
<tr>
<td>4</td>
<td>Definition of a mathematical theory for deterministic and nondeterministic processes.</td>
</tr>
<tr>
<td>5</td>
<td>Algebraic laws which describe the essential properties of the various operations that are useful in expressing new problems, solutions and proofs.</td>
</tr>
</tbody>
</table>

environment (e.g., typical real-time). Parallel composition of such systems is as simple as sequential composition using traditional languages (e.g., Pascal). Major benefits from using CSP include its simplicity, generic nature of the algebraic operations, and the mathematical foundation on which it is based [Sanders90].

A CSP program consists of $n > 1$ communicating processes; this is normally represented using the parallel composition operator ($\parallel$), which is associative: $P = \{ P_1 \parallel P_2 \parallel \ldots \parallel P_n \}$. Processes are assumed to have a disjoint set of variables (visible actions, trace alphabet). Processes communicate synchronously by sending and receiving messages: the sending and receiving actions (or events) are indicated using the input (?) and output (!) actions. $P_i ? x$ is the action of receiving a value sent by process $P_i$ (or received on a channel $P_i$ based on the notation of occam) into variable $x$. $P_j ! <expression>$ describes the action of sending the value of the expression to $P_j$ (or sending on a channel $P_j$). Synchronization uses complementary input and output commands by two communicating processes (i.e., using the same channel). Communication can be made selective by providing guards, where one of the alternative communication actions with a satisfied guard is selected. A guarded command has the general syntax of the form $<guard> \rightarrow <command list>$. A command list is a set of commands defining a sequence of actions, alternative actions based on either deterministic or non-deterministic choice, recursive actions, or a STOP action. STOP terminates (or...
deadlocks) a process. The following summarizes CSP syntax (| means 'choice'):

\[ P ::= \text{STOP} | (a \rightarrow P) | (P \mid b) | (P \mid Q) | (P \parallel b Q) | (P; Q) | (\mu X \bullet P). \]

Notationally, in CSP, capitalized names are process names, and lower case characters denote visible actions. Here, \((a \rightarrow P)\) means, action 'a' followed by process P, \((P \mid b)\) is the same as P except action b is hidden\(^7\), \((P \mid Q)\) represents a non-deterministic choice between P and Q, \((P \parallel Q)\) represents a deterministic choice between P and Q, \((P \parallel b Q)\) shows concurrent processes P and Q that synchronize on action b, \((P; Q)\) a sequence between P and Q, \((\mu X \bullet P)\) is used for recursion.

2.8 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 \(\parallel\)), SEQ (or ;), NDC (or \(\mid\)), DC (or \(\square\)), and Mu.X\{\} (or \(\mu X \bullet P\)) are the constructors that can be used for this purpose. The CSP-based grammar is provided formally as a yacc specification in Appendix B [Barrett90, Roscoe86, Jones87, INMOS88].\(^8\)

An example construction would be: PROCESS My_example = SEQ\{P, Q, R\}; where each process is performed in succession. In our language, a process need not be declared, but declared processes must subsequently be used as a "process call." 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

\(^7\)In describing the internal behavior of a mechanism, we often need to consider events representing internal transitions of that mechanism (interactions and communications internal to that mechanism). After construction of the mechanism, we may conceal the structure of its components; and also wish to conceal all occurrences of actions internal to the mechanism. Such actions can occur automatically and instantaneously without being observed or controlled by the environment of the process. Thus, if b is a finite set of events to be concealed in this way, then \(P \mid b\) is a process that behaves like P, except that each occurrence of any event in b is hidden and not visible to be observed.

\(^8\)In P-CSP, process and channel names are capitalized (at least the first letter) while other elements (i.e., actions or messages) use only lower case. These are style guidelines and are not enforced by the CSPN tool.
process to engage in an action (e.g., a $\rightarrow P$). This process is defined as an implication. Input and output 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).

The first symbol encountered is the start symbol which is always be taken as the system symbol. The general structure of a P-CSP specification is similar to that of Ada except that package specifications are process declarations composed of internal activities. Process declarations must come before the main body of the system specification.9 As shown below, the main body begins after the last semicolon. The system specification ends with a period (or dot "."): 

```
System =
    Global declarations would be located here.
    PROCESS = declaration;
    PROCESS = declaration;
    PROCESS = declaration;
    Process constructor {main body of system}.
```

The use of indentation helps to show subordinate relationships (i.e., activities or processes that are contained within a given constructor like SEQ are indented). An important syntactic rule is enforced for messages during the translation. Each message variable specified in a synchronized PAR must have matching input and output (i.e., `channel!messageX-x` must match `channel?messageX-x`). See Paragraph 4.7 - 8 for the syntax and usage of failure and service rate annotations.

---

9C allows this if you ignore the "=" signs and consider the system symbol as the "main" part of a C program.
2.9 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) [Peterson81]. In our 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 our actions / processes). However, in our approach, modeling is based on the notion in CSP of event-action pairings. The conditions are the events that cause actions (transitions) to take place. For example, a coin inserted in a vending machine causes a candy to be dispensed, the event is the coin insertion (token on an input-place) while dispensing a candy is an action which causes a one-input-place transition firing as a result of the coin insertion (token on an output-place).

A transition is enabled if all its inputs contain at least one token. When a transition is enabled, it can fire (asynchronously), leading the Petri net into a different arrangement of tokens. A marking represents a configuration of tokens in the places of the Petri net, and denotes the state of the Petri net. A marking is reachable if, starting in an initial marking, it is obtained by a sequence of firings. The reachability graph is the set of all reachable markings connected by arcs representing the transition firings. In a stochastic Petri net, each transition has an associated firing time, which can be zero (immediate shown as dark bars) or exponentially distributed random variable (timed shown as light bars).

Completion of the action defined by a transition causes a token to be assigned to each of its output places. When a place is the input to several transitions, only one of the transitions is enabled non-deterministically. As transitions are enabled, the state of the Petri net moves

---

10CSP processes perform the systems actions, while the events that trigger such actions are characterized by the completion of an action (i.e., process) or the occurrence of conditions that enable the actions (or processes).
11Murata, describes a slightly different abstraction that defines conditions and events. Murata uses places to represent conditions, and transitions to represent events. A transition has a certain number of input places and output places representing the preconditions and post-conditions of an event (see [Murata89] page 542).
12Coincidentally, if several conflicting immediate transitions are enabled in a marking, a firing probability must be defined. If at least one immediate transition is enabled, the marking is said to be a vanishing marking (otherwise, if only timed transitions are enabled [or no transitions are enabled] it is a tangible marking).
from marking to marking. An *inhibitor* arc prevents a transition's firing when its corresponding input place contains tokens.

A Stochastic Petri net (SPN) is simply a Petri net which has been extended in several ways. These extensions embed the model into a stochastic environment by associating a random time with each of the transitions in the net. The most general extensions allow the usage of random variables for times (rates) and probabilities.\textsuperscript{13} 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) [Marsan84] provided that exponential distributions are used for transition firing rates. Since a SPN permits a probability distribution to be associated with arcs (or transitions) they are very suitable for modeling system performance and reliability. Thus, each transition is associated with a random variable that expresses the delay from the enabling to the firing of the transition. When multiple transitions are enabled, the transition with a minimum delay fires first. The transition rate from state $M_i$ to $M_j = q_{ij}$ is given by $q_{ij} = \lambda_{i1} + \lambda_{i2} + \ldots + \lambda_{im}$ where $\lambda_{ik}$ 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 an especially clear discussion of SPN models in chapter 7 of [Sahner95].

Markov and performability models are covered in the same book (chapters 4, and 6 respectively). Examples of these types of models are available in part two (chapters 9, 10 and 12). Also refer to [Ciardo89, Murata89, Kavi93, Balbo95, Laprie95, Levenson87, Lewis88, Sahner93] for more details on Petri nets and SPNs, as well as Markov processes and Markov Reward processes (an extension of Markov processes).

Traditional performance analysis, which assumes a fault free system, is separate from dependability analysis which is carried out to study system behavior in the presence of faults. Dependability analysis generally disregards the different performance levels that may be

\textsuperscript{13}When there are multiple transitions enabled by one token, a probability is associated with each of the involved transitions. Such a transition is immediate and its firing is instantaneous (no time is consumed).
associated with differing configurations [Arlat90, 93, Clark92, 93, 94, Dahlberg93, Goswami92, Goyal92, Iyer89, 95, Yount95]. By combining performance and dependability, the different types of interactions and their corresponding trade-offs can be assessed (this is called performability analysis) [Sanders87, 88, 89, 91, 93, Muppala91, 94a,b]. Most of the work on this combined evaluation is based on Markov reward processes (known as SRNs) where a reward (or weight) is attached to each state of the Markov process (usually by defining a C function). Markov reward processes can potentially reflect concurrency, contention, fault-tolerance, and degradable performance [Anderson85, Beli91, 90, Clark93, Dugan94, 93a,b, 89, 87, Eckhardt85, 91, Elks91, Geist90, 83, Kim92]. They are used to obtain not only program and system performance and system reliability (or availability) measures, but also the combined measure of performability. Though Markov reward models posses the power to solve dependability, performance and performability problems, there is still one major drawback which is the largeness of their state space [Aupperle91, 89, Bobbio86, 90, Smotherman86, Sorensen93]. SPNP was designed to address this problem. The SRN model is used to generate the underlying Markov reward model automatically starting from a concise description written using the language for SPNP.

2.10 Introducing SPNP's 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. The language used for describing stochastic Petri nets for the Stochastic Petri Net Package (SPNP) is CSPL. CSPL is a super set of the C language and thus 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 single CSPL file.

The function parameters allows the user to customize how the package will perform the
```plaintext
/* Definition of rates */
trans("ft_Slot_i_a1p");
trans("ft_Slot_i_a2p");
trans("dt3");
trans("dt2");
trans("dt_o_large");
trans("Tray_o_large");
trans("dt_i_a1p");
trans("sdt6");
trans("dt_o_small");
trans("Tray_o_small");
trans("sdt5");
trans("dt_i_a1p");
trans("Slot_i_a1p");
trans("sdt4");
trans("dt1");
trans("dt_o_a1p");
trans("Tray_o_a1p");
trans("dt_o_small");
trans("Tray_o_small");
trans("sdt3");
trans("dt_o_large");
trans("Tray_o_large");
trans("sdt2");
trans("dt_i_a2p");
trans("Slot_i_a2p");
trans("sdt1");
trans("dt_MuX");
/* Definition of transitions */
/* Definition of places */
place("p21");
place("p22");
place("p23");
place("p18");
place("p19");
place("p20");
place("p15");
place("p16");
place("p17");
place("p13");
place("p14");
place("p15");
place("p10");
place("p11");
place("p12");
place("p7");
place("p8");
place("p9");
place("p4");
place("p5");
place("p6");
place("p1");
place("p2");
place("p3");
place("p0");
init("p0", 1);
/* Definition of transitions */
trans("dt_MuX");
trans("Slot_o_a1p");
trans("sdt2");
trans("dt_o_large");
trans("Tray_o_large");
trans("dt_o_small");
trans("Tray_o_small");
trans("sdt3");
trans("dt_o_large");
trans("Tray_o_large");
trans("sdt2");
trans("dt_i_a2p");
trans("Slot_i_a2p");
trans("sdt1");
trans("dt_MuX");
/* Definition of rates */
rateval("ft_Slot_i_a1p", 0.0045);
rateval("ft_Slot_i_a2p", 0.0055);
probval("dt3", 1.0);
probval("dt2", 1.0);
probval("dt_o_large", 1.0);
rateval("Tray_o_large", 0.1);
probval("dt_i_a1p", 1.0);
rateval("Slot_i_a1p", 0.1);
rateval("sdt6", 0.50000000);
probval("dt_o_small", 1.0);
rateval("Tray_o_small", 0.1);
probval("dt_o_small", 1.0);
rateval("sdt6", 0.50000000);
trans("Slot_o_a1p");
trans("sdt2");
trans("dt_o_large");
trans("Tray_o_large");
trans("sdt3");
trans("dt_o_large");
rateval("Tray_o_large", 0.0055);
rateval("Tray_o_large", 0.0045);
```

Figure 5. SPNP input file structure.
analysis. Several parameters establishing a specific behavior can be selected (a complete
description of parameters are available in [Ciardo94]). 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(), trans(), iarc(), oarc(), and init() which defines the
initial marking. The CSPL input file has the basic structure shown in Figure 5.

More advanced functions include harc() for making inhibitor arcs while the functions
miarc(), moarc(), and mharc() define multiple input, output and inhibitor arcs (these more
advanced functions are not synthesized during the translation process). Guards which are
logical conditioning functions associated with a transition(s) and priorities can be specified
using guard() and priority(). Probabilistic behavior may be specified using probval(), the
timing of events can be specified by assigning rates to the transitions in rateval() and variable
cardinality arc can also be specified for input, output and inhibitor arcs. Marking dependence
is specifiable using the mark() and enabled() functions.

2.11 The original contribution of this work

Our approach predicts system behavior (in terms of reliability and performance) based
primarily on the structural characteristics of a formal functional specification. The core
augmentation to existing approaches is provided by our CSP-based grammar and canonical
CSP-to-Petri net translation rules for process composition/decomposition. The mechanism
for process composition is codified in the CSPN tool and consists of expanding the process
description represented as sub-Petri nets into larger and larger nets. In the last phase the tool
reconciles synchronization points, failure annotations and generates an executable spnp.c file
(at various levels of user controllable interaction). In essence the contribution 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
implementations; relating stochastic parameters back to the specification level; and analyzing
the stochastic Petri nets using the SPNP tool [Ciardo87, 89, 90, 91, 92, 93a, b, Trivedi93].
2.12 Where does this work fit

The CSPN tool and methodology is based on a sound formalization of CSP which provides process constructors, including primitives for parallel and sequential composition, nondeterministic choice, and recursion. To support top-down development, the grammar and CSPN tool provide a notion of refinement (see Figure 6 below) that allows a designer to describe a system at an appropriate abstract level. At this level, a designer may estimate the values of non-functional requirements (so called budgeting). Later, the designer may add more details by showing the internal structure of a component, explicitly presenting local communications, and modifying the budget. It is important to facilitate systematic refinements and then re-evaluate their impacts to optimize design parameters. Figure 6 shows that the approach involves abstraction from the requirements specification into a design specification and subsequent evaluation based on the stochastic analysis of the system models. Automatic translation of the design specification into a stochastic Petri net representation enables the use of a good number of sophisticated design and analysis tools.

![Diagram of system models refinement](image)

Figure 6. Refinement of system models.
CHAPTER 3

MAPPING CSP TO PETRI NETS

The heart has its reasons which reason knows not of.

—Pascal

3.1 Mapping CSP to Petri nets

An initial set of rules for translating CSP specifications into Petri nets (Petri nets) is defined in [Kavi93]. The translations between CSP and Petri nets are based on the CSP premise that processes execute actions which in turn enable other actions (in this way, CSP processes move from one action to another). Activities that enable a process can be viewed as conditions (or events) which are represented by places, while the actions themselves are viewed as transitions. Some example translations are given in Figure 7. Note that the P-CSP

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>Nondeterministic choice to proc a or b</td>
</tr>
<tr>
<td>B.</td>
<td>Nondeterministic choice w/ recursion</td>
</tr>
<tr>
<td>C.</td>
<td>Parallel actions are transitions</td>
</tr>
<tr>
<td>D.</td>
<td>Deterministic choice</td>
</tr>
<tr>
<td>E.</td>
<td>Non- and deterministic choices run in parallel</td>
</tr>
<tr>
<td>F.</td>
<td>Parallel actions synchronize on b</td>
</tr>
</tbody>
</table>

Figure 7. Example CSP to Petri net translation rules (P-CSP shown in lower portion).
(our textual language for CSP) equivalents constructions are shown below the graphs. The P-CSP grammar is described in Appendix B.

The CSP to Petri net translations were designed to facilitate automatic decomposition of the CSP constructs into Petri net sub-components and subsequent 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, the composition of subnets requires that we introduce dummy places and transitions to maintain the Petri net's bipartite nature. Thus, the CSP to Petri net translations are not isomorphic because of the introduction of dummy transitions and places which are necessary to facilitate the automatic composition of the subnets. However, for our purpose, we do not require isomorphism. 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 a smaller model 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.

3.2 CSP translation incongruencies

Petri nets are inherently non-deterministic and asynchronous while CSP is inherently deterministic and synchronous (though an explicit definition of non-deterministic choice exists for CSP). Since our purpose is stochastic analysis, we depend on the non-deterministic nature of the Petri nets. This may appear that the determinism of CSP (and the nondeterministic construct) are translated to the same PN representation. The translation of the CSP structural properties is standardized (i.e., based on canonical rules of translation).

---

14Intuitively, it is possible to reduce different Petri net equivalents into a canonical form. A set of canonical translation rules are applied to derive each component's Petri net equivalent. Refer to Appendix A for the complete set of canonical translation rules.

15A task which is left to the Petri net tool (i.e., SPNP).

16However, the nondeterministic choice composition operator of CSP (i.e., NDC in P-CSP) is treated as a selection between two or more transitions which is made on the basis of assigning a (discrete) probability to each.
The goal is to demonstrate the feasibility of translating from CSP and Petri nets by decomposing a CSP specification into its component parts (processes, actions, and synchronizing actions etc.). This is done by choosing one standard (canonical) translation path from among potentially numerous equivalents.

### 3.3 CSP translation mechanism and commentary

This section provides the conventions and mechanisms that have been defined for the canonical translations [see Olderog87]. All CSP composition constructs (e.g., PAR, SEQ, NDC, etc.) have an input place and this input maybe connected to the output of a different process. All processes have an initial marking. Synchronization actions (a transition common to communicating processes) do not consume time or resources. The solid black bar represents such a case. The open light bar does consume resources (so called timed transition) [Muppala94, Sahner95]. Solid bar transitions are used (primarily) to represent dummy transitions (transitions added to maintain the bipartite structure of Petri nets). This type of transition is known as **immediate**. In addition, some transitions are needed to indicate synchronization with the environment (Figure 7D and E). Such transitions are represented with solid bars even though there may be delays associated with this type of synchronization.

![CSP to Petri net rules for Actions and the STOP and SKIP processes.](image)

Figure 8. CSP to Petri net rules for Actions and the STOP and SKIP processes.

An action is represented with one (or more) input places, a transition and one (or more) output places as illustrated in Figure 8. However, the STOP action has no output place

---

17Synchronization is represented by a synchronized action and is shown as a transition between two processes (i.e., Channel-X ? message-X [for input] and Channel-X ! message-X [for output]).
because it never terminates. Actions comprising a process must be composed to form the process. In order to achieve this, input places and output places must be overlapped (is combined) whenever an action follows another as illustrated in Figure 9B. The same rule is observed when one process "P" is followed by another process Q as shown in Figure 9C. In general, the process "P," as shown in Figure 9A, has an input place and an output place on the boarder of a box. The box symbolizes a complete process. The box encompasses the process transition(s) and any of the requisite places which may be necessary to compose other transitions. A box can contain other boxes just as a process can be made up of other processes. It should be clear that, within this framework, all Petri nets start with an input place which is used to represent an initial marking (or initial state). During the compositions, these input places are combined with output places of other processes. The complete specification has exactly one input place, and hence one initial marking.

Figure 9. CSP to Petri net rules for Combining actions and processes.

In Figure 10, the action "a" triggers the nondeterministic composition of actions b and c. Part A of this figure sufficiently represents the meaning of the composition: \( a \rightarrow \text{NDC}\{b, c\} \). When transition "a" fires, this enables a choice between \([dt1 \rightarrow b]\) and \([dt2 \rightarrow c]\). The whole composition consumes one token and produces one token on either place "\(p_b\)" or place
"p_c." Thus, part A has a certain compact and sufficient semantics, but lacks in its ability to combine cleanly with other constructs. For example, consider \([a \rightarrow \text{NDC}\{b, c\}]\) combined using "\(\rightarrow\)" with \([\text{PAR}\{P, Q\}]\). The composition rewritten is: \(a \rightarrow \text{NDC}\{b, c\} \rightarrow \text{PAR}\{P, Q\}\). This construction represents an action "a" that triggers the choice between b and c which in turn triggers both processes "P" and "Q" to proceed in parallel. However, there is no single (unambiguous) place for connecting the succeeding PAR construct to the NDC predecessor. In Figure 10A, would the PAR be connected to place "p_b" or place "p_c"? The answer is both. The solution however, is shown in Figure 10B as the place "p_dt3." The connection is made as described for figure 9B and C, by overlapping (or snapping together) the output place of one with the input place of the other.

**Figure 10. CSP to Petri net example of combining actions to form one process.**

It is necessary to create additional places (and transitions) when the choice of an action(s) is made in a deterministic sense (conditionally) or in a non-deterministic sense as illustrated in Figure 11A and B respectively. The same rule is observed for processes. In Figure 11A the additional darkened places and dummy transitions represent the possible states or values of guards. When these guards are combined with the initial place of the
given process, this represents the deterministic enabling (or choosing) of one action (or process) over another. Thus, if msg1 occurs prior to msg2 the dummy transition dt1 is enabled and thus fires to enable the process "P" to become active.

Figure 11. CSP to Petri net rules for Deterministic and nondeterministic choice.

In Figure 11B, there are no environmental conditions that provide for the enabling of one path versus the other. Instead the choice is random (i.e., arbitrary): a path is chosen on the basis of probability and the sum of the probabilities across all choices is one. The special nature of the Non-Deterministic Choice (NDC) construct requires that the initial dummy transitions be distinguished from other normal dummy transitions. Thus, normal dummy transitions can be assigned a firing probability of one (e.g., dt3 in Figure 10B), whereas the "sdtx" (x = 1, 2, ...) or special dummy transitions are assigned a probability.

In accordance with the 1-input place rule, the parallel composition of Figure 12A shows an additional input place (and the transition "dt1" is added to keep the bipartite structure). The immediate firing of "dt1" enables both processes P and Q to proceed independently. However, before a token can be deposited in the "p5" place to conclude the complete sequence of feasible markings, both P and Q must finish. This subsequent joining is an
artifact for observing the one input place one output place per process composition rule (or the combining rule) described above. This rule implements the combining of sub-Petri nets.

**Figure 12.** CSP to Petri net rules for Parallel and synchronized parallel composition.

In Figure 12B, two CSP processes synchronize using a channel. The channel and message by necessity, are common to both processes. The two processes that are participating in the synchronization are without names. As shown, they are simply the input and output statements composed using a synchronized parallel composition. Both processes must first complete their part of the synchronizing action. Once complete, they cooperate in a joint synchronization transition (shown as a solid "immediate" transition since it is an artifact of the translation). This representation is more appropriate since the individual processes must execute their respective actions and these actions consume time (and other resources), while the rendezvous action is an event that consumes no time (or resources). This type of synchronization causes both the sender (Ch ! msg) and the receiver (Ch ? msg) to be blocked until the transition labeled "dt:msg" has fired.
In Figure 13, a number of recursive compositions are shown. In the top half on the left are two equivalent nets that represent a recursive non-deterministic choice. The use of the "dummy" provides a way to combine $\mu X.((P \sqcap Q) \rightarrow X)$ with another composition. The adjacent (top middle) Petri net shows an equivalent solution (without the dummy transition). Since dummy transitions are immediate (consume no time or resource) and fire with probability one, they can be eliminated without a loss of generality. The rightmost Petri net was created by the CSPN tool. The four dummy transitions can be reduced to either of the
other two Petri nets in the top half of Figure 13. This Petri net exemplifies tail recursion.

In the lower half of Figure 13 there is an example which is not a tail recursion. The first Petri net on the left is the smallest possible translation (most reduced). The other two Petri nets on the right (bottom half) illustrate a two step process where the tail recursion is cut and the result is thus equivalent to the small one on the left. Actually, CSPN, in structuring a recursive composition assumes tail recursion. Thus, CSPN begins the translation by attaching a recursive link back to the initial place. After doing so, CSPN checks all of the elements within the scope of the link to see if there is any cause (such as $Q() \rightarrow X$) to cut the tail recursive link and re-attach it in some other fashion, as shown in the example of Figure 13 (bottom half). Figure 14 shows how these process compositions are viewed when combining (or nesting) a non-deterministic choice inside a recursive construction. The larger clear shaded box defines the recursive process. The smaller shaded boxes define the component processes. See the Appendix for further examples.

![Figure 14. CSP to Petri net rules for recursion.](image)

3.4 Specification of failures and failure handling in CSP

A failure can be specified for any action in CSP (or any process). For example, consider
the term "a→P" which can be extended to include failure specification as "(a → P) \[ (a →
Failure → Repair)." The translation into a Petri net from this composition would look like
that of Figure 11B except the (a → P) term would replace P and the (a → Failure → Repair)
term would replace Q. The Q transition is the failure (and repair) transition which could be
further expanded to account for the repair action. This type of non-deterministic construction
can be extended to other types of CSP constructions.

Consider the vending machine (VMC) example specified in Figure 15. This figure
shows how a failure annotation can cause a structural change in the Petri net. A simple VMC
may include only two kinds events: (1) insertion of a coin in the VMC slot(s), and (2)
dispensing of a candy. A more complex VMC may offer a choice of slots for inserting a 2p
(i.e., two penny coin) or a 1p coin (thus, the customer may make the choice external to the
machine which slot). Also, in the complex VMC case, there can be the possibility of
receiving a small or large candy. In Figure 15 the particular choice of a 2p or a 1p is made
non-deterministically. Examine the body of the VMC specification which begins with
construct "Mu.X{ ... " and concludes with "}." which includes the ending period. This is a
recursive construction because once a VMC transaction is complete the system can return to
a state which allows it to engage in another transaction. Inside of the recursion, we see that
the choice is either one of two SEQ constructs that represent the 2p or the 1p input actions.
When the 2p path is chosen, the VMC will either dispense a small candy (and 1p change) or a
large candy. If the 1p path is chosen then either another 1p will cause a large candy to be
produced or, with out the additional 1p coin, only a small candy can be dispensed.

The failure annotations are attached to the OnePenny and TwoPenny process
descriptions. CSPN thus includes three additional failure transitions that all deposit their
tokens into place "p23." Assume the transition "Slot ? a2p" may fail with a failure rate of
0.0055 while, transition "Slot ? a1p" fails with a failure rate of 0.0045 (of which there are two
actual transitions). Only tangible (light colored) markings can fail.
Figure 15. VMC specification for CSP to Petri net with failure annotations.

Notice that in the Figure 15 Petri net there are three (light colored) transitions of which two are labeled "ft:slot?a2p" and one is labeled as "ft:slot?a1p." These three transitions are
generated by CSPN automatically when the failure annotations are encountered and represent the structural changes mentioned above (in this section).

3.5 Preserving the trace behavior of CSP specifications

Let’s consider a sequence from the set of possible traces given by a particular CSP composition structure (refer to Section 3.1.2 on program equivalence). A CSP action takes place when an enabling predicate is true. The enabling predicate is defined either by some external system events or by the completion of some other actions (remember, the completion of an action may be viewed as an event). A *trace specification* of a CSP process is the set of all possible traces. A trace is a sequence of events drawn from the process alphabet (which defines the set of all visible events).

Now, consider the structure of a Petri net in relation to a trace specification. The Petri net that results from a CSP specification can be viewed as follows. An action is represented by a transition, and the completion of the action is represented by token(s) in its output place(s). A sequence of enabling transitions can easily be mapped to a feasible trace. The sequence of transitions produce a sequence of Petri net markings. Accordingly, for each trace, it is possible to find a sequence of Petri net markings. It is necessary to prefix all sequences of Petri net markings with an *initial marking* based on the definition of a Petri net. Also, the asynchronous nature of a Petri net may lead to more markings in the sequence than a CSP trace (unless time is associated with the CSP actions and Petri net transitions). If time is used as a criterion to constrain the set of feasible markings, then it may be possible to show that the traces and the feasible markings coincide.

Using the translation / composition rules that are codified in CSPN, the Petri net of Figure 16 was created. This Petri net is the same net as the one shown in Figure 15 except the failure annotations have been removed and thus it has three fewer transitions. Let us examine, the various characteristics of the translations in greater detail so as to understand how is it possible to preserve the trace behavior of CSP specifications. The solid colored
VMC = (Slt?2p→(Tr!Lrg→VMC)) \(\cap\) \\
(Tr!Sm→Tr!1p→VMC)) \\
\(\cap\) \\
(Slt?1p→((Tr!Sm→VMC) \(\cap\) \\
(Slt?1p→(Tr!Lrg→VMC))))

VMC =

PROCESS DispLg = {Tray ! large};
PROCESS DispSm = {Tray ! small};
PROCESS DispSm1p = SEQ{
    DispSm(),
    {Tray ! a1p}
};
PROCESS OnePenny = {Slot ? a1p};
PROCESS TwoPenny = {Slot ? a2p};

\(\text{Mu.X}\{
    \text{NDC(}
        \text{SEQ(}
            \text{TwoPenny(),}
            \text{NDC(}
                \text{DispLg()},
                \text{DispSm1p()}
            }
        \}
    \}
)

\(\text{SEQ(}
    \text{OnePenny(),}
    \text{NDC(}
        \text{DispSm()},
        \text{SEQ(}
            \text{OnePenny(),}
            \text{DispLg()}
        }
    \}
)

\}
).

Figure 16. CSPN translation of the VMC example.

bars are of three types (1) dtx, (2) sdtx or (3) dt<:Mu/?/!>, where x is an integer. A label prefixed with "dt" is a dummy transition. Dummy transitions are further broken into those associated with the composition constructs (e.g., PAR, SEQ, etc.), and those associated with
synchronization. Dummy transitions with "?" or "!" and an identifier (e.g., large, small, etc.) are manufactured for the purpose of creating a synchronization (or rendezvous) point.\textsuperscript{18} A label prefixed with "sdtx" is a special dummy transition which has a probability associated with its firing. Also shown in Figure 16 is the specification (top one uses CSP bottom uses P-CSP) used by CSPN to generate the associated Petri net.

**TABLE 2**

MARKINGS FOR VMC PETRI NET

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M4</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M6</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M7</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M8</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M9</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M10</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M11</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M12</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M13</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M14</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M15</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M16</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M17</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M18</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M19</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M20</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M21</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M22</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\textsuperscript{18}When processes rendezvous, CSPN matches an input with an output (i.e., same message and channel identifiers). The result is to combining the two synchronizing transitions into one. The one is re-labeled \texttt{"dt:msg\_name."} In the VMC construction there are no rendezvous as such (just input of change and output of candy). See more details on the notion of rendezvous (that cause a combining of two into one) in the Appendix.
Figure 17. Structure of the feasible sequences of VMC Marking transitions.

In the VMC construction of Figure 16, there are no rendezvous (just input of change and output of candy). The path choice is made nondeterministically. The complete deterministic

Figure 18. Partial VMC translation showing deterministic choice.
construction is envisioned based on Figure 18. Each dummy transition (e.g., dt1 and dt4) has
two different places which must contain a token before a choice (firing) can be possible. If
the customer chooses to deposit a 2-penny coin in the 2penny slot the place labeled pe1 gains
a token (or pe2 for the 1penny slot). This is an external action.

The place "p01" has a token while the VMC is in an idle (waiting state) until inputs from
the environment (customer selections) cause a choice to be made. The environmental input
in the case of the VMC example represents user selections. If the NDC compositions are
replaced with DC compositions, then the complete solution would use six additional places.

Table 2 shows the markings of the VMC Petri net. Figure 17 gives the marking
transitions. The dummy markings (in shaded ovals) can be combined into a single marking.
Likewise, the markings that are due to synchronization (in the clear rounded rectangles) can
be combined if the synchronization action is instantaneous. The CSP specification in Figure
16 has the following possible traces:

1. \(? a2p, \! \text{large}\)
2. \(? a2p, \! \text{small}, \! \text{alp}\)
3. \(? \text{alp}, \! \text{small}\)
4. \(? \text{alp}, \? \text{alp}, \! \text{large}\)

The corresponding transitions of Markings that are possible from the Petri net are:

1. \text{M0 M1 M2 M3 M4 M5 M6 M11 M22}
2. \text{M0 M1 M2 M3 M4 M5 M7 M8 M9 M10 M11 M22}
3. \text{M0 M1 M12 M13 M14 M15 M16 M21 M22}
4. \text{M0 M1 M12 M13 M14 M17 M18 M19 M20 M21 M22}

The bold face markings are the essential markings while the others are an artifact of the
translation rules. Note that there is a one to one correspondence between the set of CSP
traces and the bold face marking (assume that the M0 marking is removed).
CHAPTER 4

CSPN TOOL OVERVIEW AND IMPLEMENTATION DETAILS

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

—Mike Rutheford

4.1 CSPN tool overview

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

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

\(^1\)Version 1.0 of CSPN does not automatically invoke the dot program to create the postscript graphic file. To do so use the command: `>> dot -Tps filename.dot > filename.ps`. Dot is a available from AT&T Bell Laboratories.
appropriate (timed) transition in the resultant SPNP specification. All of the values assigned from annotations are subject to change if the user so chooses during an interactive CSPN run.

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

### 4.2 Translation phases of the CSPN tool

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

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

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

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

---

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

3 The SPNP specification file can be run for a simple analysis without manual intervention.
In viewing Figure 27, note that the following eight steps occur during the translation process: (1) *Scanning and Parsing* – action rules embedded in the parser enable CSPN to capture the structural semantics of the specification, (2) *Decomposition* – allocating or scoring a coincidence matrix for each CSP element and the recording of any annotated service rates and probabilities, (3) *Composition* – combining elemental coincidence matrices and building their requisite process lists, (4) *Synchronization* – resolution or combining of message links, (5) *Failure annotations* – if active, an appropriately annotated process is augmented with a failure transition, (6) *Resolving recursion*, (7) *Synthesis phase* – takes the system coincidence matrix and creates the SPNP Petri net specification file during an *interactive session* with the user, and (8) *Filter* – removes special characters inherited from the CSP specification that are not valid in an SPNP specification and *graphics* – creates a digraph specification net list that is later compiled using “dot” to produce an embedded postscript graphic. In general, Figure 27 shows the various translation phases and the use of SPNP as it applies to this approach. The names in parenthesis are the *C-function name(s)* and are associated with a given phase. The CSPN tool is used in the context of the seven steps listed in Table 3.

### TABLE 3

**GENERAL STEPS FOR USING THE CSPN TOOL**

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

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

TABLE 4

LISTING OF THE CSPN COMMAND LINE OPTIONS.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h</td>
<td>Used to generate a help screen which displays the contents of the table below: “csp -h”</td>
</tr>
<tr>
<td>-v</td>
<td>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.</td>
</tr>
<tr>
<td>-f</td>
<td>Used to generate failure transitions into the <code>filename_spnp.c</code> file. This option enables detection of failure annotations and causes interactive inputs with the ”-o” option specified.</td>
</tr>
<tr>
<td>-F</td>
<td>Set to invoke the filter which will replace the 3 special characters (?,!,:) in the <code>filename_spnp.c</code> with SPNP compliant characters (<em>i</em>, <em>o</em>, and _ respectively). Otherwise, SPNP will not compile the input file. Valid only when the ”-o” option is used.</td>
</tr>
<tr>
<td>-s</td>
<td>Use the default service rates for timed transitions. If no service rate is specified as an annotation then CSPN will use 0.1.</td>
</tr>
<tr>
<td>-o&lt;name&gt;</td>
<td>To generate the SPNP input specification file (<code>filename_spnp.c</code>) this option must be specified (”name” is optional and the default used is the tool name ”cspn”).</td>
</tr>
<tr>
<td>-i&lt;number&gt;</td>
<td>Number of iterations used by SPNP (default is 2000).</td>
</tr>
<tr>
<td>-a&lt;number&gt;</td>
<td>Rate for return to initial marking from absorbing markings (default is 0.0).</td>
</tr>
<tr>
<td>-p&lt;number&gt;</td>
<td>Set floating point precision used by SPNP (default is 0.000001).</td>
</tr>
<tr>
<td>-P</td>
<td>Set to enable selection of priorities for individual transitions (the default is none).</td>
</tr>
<tr>
<td>-d</td>
<td>Set to generate a ”dot” graphics file. Dot uses this digraph specification file to generate the graphical representation of the Petri net.</td>
</tr>
<tr>
<td>-n</td>
<td>Set to enable a network list file. This file shows how CSPN has interpreted the structural aspects of the CSP specification.</td>
</tr>
<tr>
<td>-t</td>
<td>Set to generate a symbol table file containing all the data recorded for each element (process names, constructions, variables, channels, ...) of the process specification.</td>
</tr>
</tbody>
</table>
then adding the "-d" option would enable only the production of the graph. The "-F" option invokes a filter and is necessary only when the user plans to run an SPNP analysis. The "-f" option is a nice feature because it enables the analyst to assume a failure free environment by simply ignoring any embedded fail annotations that may exist in the CSP specification (without "-f" CSPN ignores failure annotations). Omitting failure annotations from the P-CSP specification has the same affect. The option "-s" streamlines the process of generating the SPNP input specification by assigning default service rates to timed transitions without querying the user to provide such. As mentioned above, the "-o" option generates a file for SPNP analysis. It is best if a file name be given with this option (i.e., "-ofilename"). This settles the problem of overwriting previous files generated using the default name that is assigned by CSPN if no name is provided. The "-i", "-a" and "-p" options are used to parameterize the SPNP run by setting the iteration number, absorbing rate (for recycling back to the initial marking), and precision for floating point operations respectively. The "-P" option is only valid when "-o" is used and enables the user to assign priorities to any of the transitions. The "-d", "-n" and "-t" options are useful when something unexpected happens after running CSPN such as a run time error. The user may wish to rerun the translation and view the internal data structures that are generated during the translation process.

### 4.4 CSPN data structures

Internally, there are four basic data structures employed by CSPN: (1) Symbol table which maintains attributes assigned to all system elements (actions, processes, communications and constructions), (2) Process lists which consist of all the names of the associated actions/processes involved in a particular construction, (3) A network of linked lists which capture the structure of the specification (adjacency and nesting), and (4) The bipartite digraph which defines the structural character of the Petri net is represented as a coincidence matrix. The coincidence matrix (or co-matrix) maintains the distributions of places, transitions and their connectivity.
Figure 28. P-CSP constructions with co-matrix and Petri net representations.

The construction of the n-by-m co-matrix is defined in terms of the transitions (CSP-process names become transition names). Transitions are associated with rows (from top to bottom).
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 or a place to a transition. 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 the order of their appearance in the CSP specification. The naming of places is ordered (e.g., p1, p2, ..., pn), and the meaning associated with each is defined in terms of the transitions with which they are connected. Each element and each expanded composition from the CSP specification has a coincidence matrix (or co-matrix) maintained in the symbol table.

Figure 28 gives examples of P-CSP compositions. During the parsing phase, each construct (e.g., PAR, SEQ, etc.) is separated into its component elements (process names, channels, variables) and represented as a sub-Petri net. The sequential (i.e., SEQ) construct, shown in the top portion of Figure 28, illustrates scoring of the co-matrix (marked with "-" or "+") to denote input to and output from the given transition (e.g., P, Q, R). The middle part of Figure 28 shows a similar translation for the parallel (i.e., PAR) construct and the last part shows a synchronized parallel construction.

4.5 Petri net compositions

Expand Co-matrix A using B

Figure 29. Choosing a combining method for expansion that depends on locality.

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

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

\[
\begin{align*}
C: & \quad \begin{array}{cccccccc}
0 & - & b & b & b & b & b & 0 \\
1 & b & b & b & b & b & b & 0 \\
2 & b & b & b & b & b & b & 0 \\
3 & b & b & b & b & b & b & 0 \\
4 & b & b & b & b & + & a & a \\
5 & y & 0 & 0 & 0 & 0 & g & g \\
6 & y & 0 & 0 & 0 & 0 & g & g
\end{array} & \quad A: & \quad \begin{array}{cccc}
0 & - & + & a & a \\
1 & y & g & g \\
2 & y & g & g \\
\end{array} \\
\end{align*}
\]

Rows 1 and 2: put g's in C, start at C[5,5].
Rows 1 and 2: put y's in C start at C[5,0].

Figure 30. Diagram of expansion method one.

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

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

\[
\begin{align*}
A: & \quad \begin{array}{cccc}
1 & a & a & a & a & a & a & z \\
2 & a & a & a & a & a & a & a \\
3 & a & a & a & a & a & a & a \\
4 & a & a & a & a & a & a & a \\
5 & y & y & y & - & + \\
\end{array} & \quad B: & \quad \begin{array}{cccc}
1 & - & + & b & b \\
2 & b & b & b & b \\
3 & b & b & b & b \\
4 & b & b & b & b \\
5 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & b & b & b \\
7 & 0 & 0 & 0 & b & b & b & b \\
8 & y & y & y & b & b & b & b
\end{array} \\
\end{align*}
\]

Case 2: If \( A[5,5]="-" \) \( \Rightarrow \) C(8x6).

Figure 31. Diagram of expansion method two.

Method 2 is described in Figure 31. In case 1, the resultant C matrix is 8x7. Case 2 is a variation which occurs when "-" is discovered in the last column and row. This will occur
The expansion methods provides a means to combine two co-matrices. When a recursive construct is used in the P-CSP specification, in such a case the last column is dropped and the C matrix is 8x6. Also, for case 1 (where $A[m,n] = C[5,5] = +$), if a "z" in A is "+" then it will be moved to the last column (same row). Similarly, in either case 1 or 2, a "y" in A is "+" then it is moved to the last row in C (same column). The Method 3 expansion is too detailed to describe in the same terms as was done for Method 1 and 2 (refer to the Appendix C for the code on Method 3). The basic idea is given in Figure 32 which shows how the SEQ1 co-matrix (analogous to co-matrix B in Figure 29) is inserted into the

---

4Method 2a exception: catch all the +'s in last column which are to be moved to the new last column. These +'s are outputs from transitions to the last place in A so now they must be connected to the new last place in C. Only consider rows above rowMark which is the row being expanded (with the "- +" pair in the diagram).
SEQ0 co-matrix (analogous to co-matrix A in Figure 29). The expansion replaces the
transition SEQ1 by the two process names P5 and P6. The final combined result retains the
SEQ0 name. Note, the term SEQ is a key word (for sequential composition of processes), it
may itself be considered a process. CSPN treats each occurrence of this type as a unique
process by appending a unique number to the name (0 is appended to the first occurrence of
SEQ to give SEQ0 and the next occurrence of SEQ will have "1" appended). This strategy
allows the program to track each occurrence of a given keyword type. The keywords
subjected to numbering include SEQ, PAR, NDC, DC, STOP and SKIP.5

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

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

5Incidentally, the first four words listed give rise to P-nodes which constitute composition constructs which can
take themselves contain other P-nodes or L-nodes. Lnodes are nodes which can be 'listed' inside of a P-nodes (e.g.,
a channel!output or channel?input) which themselves are atomic. Not mentioned are stmtlist, MU.identifier, and
SystemID which are other possible Pnodes. These distinctions are made for the purpose of capturing structural
characteristics of the specification.
sub-Petri net that can be embedded into other Petri nets), it can be replaced by its expanded coincidence matrix (including the replacement of the Train symbol in the PAR1 process list with the Train's process list). The resultant C matrix has 9 rows and 11 columns.

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

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

Figure 34. Data structure for nesting and adjacency detected in the specification.
processes) is recorded sequentially as follows SYS[0], SYS[1], ... SYS[n]. The SYS[0] pointer always gives the system identifier (the actual symbol itself is pointed-to by NET[0] [see Figure 37 to verify this example]) and the body (or main part) of the system composition. Each new SYS[i] pointer is a new "PROCESS" declaration. Each new NET[i+1] is a new level of nesting. The list attached to a given NET[i] contains the components within a given process constructor (so-called a p-node using the nomenclature of Figure 36).

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

This same kind of replacement (expansion) mechanism continues until the top of the leg is

---

6See Figure 36 for a definition of P-nodes and L-nodes.
Figure 35. Process hierarchy for system "SysSimpleEx" with exaggerated nesting.

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

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

Each process declaration causes a net_node structure to be allocated and linked to the sys[i] which is a pointer to a NET_NODE. Therefore sys[i] contains as many non-null pointers as there are PROCESS declarations.

Each L-node (list node) is linked to its sibling as a NODE within a P-node.

The system body or "main" part of the specification is linked from sys[0]. Sys [0..n] where n+1 is the number of net_nodes allocated.

There are two levels of data used in this framework (P-nodes and L-nodes). The first is sys[], an array of "net_node" pointers. The net_node is a structure that is allocated for each P-node declaration. The second is net[], an array of "nodes" pointers to all of the L-nodes for this P-node.

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

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

```c
typedef struct entrydef {
    char *name;  // Symbol name pointer
    short type;  // Symbol type (values 0 through 23)
    short uid;   // Unique identification number (pid)
    short numNodes;  // Number of nodes in this linked list
    short numSibs[NETSIZE];  // Number of siblings within each node
    nodeptr net[NETSIZE];  // Rootptr's to Process Nodes
} ENTRY;
```

NET_NODE structure (pnodes declaration instance)

```c
typedef struct netdef {
    char *name;  // Pointer to the node/symbol name
    short numNodes;  // Number of pnodes in this linked list
    short numSibs[NETSIZE];  // Number of siblings within each pnode
    nodeptr net[NETSIZE];  // Rootptr's to Process Nodes
} NET_NODE;
```

NODE structure (pnode or lnode instance)

```c
typedef struct nodedef {
    char *name;  // Pointer to the node/symbol name
    short type;  // Node type consistent w/ symbols
    short uid;   // System level unique identifier
    struct nodedef *link;  // Link to next node, if any
} NODE;
```

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

<table>
<thead>
<tr>
<th>Process nodes and artifacts</th>
<th>Channels and variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL_TYPE</td>
<td>BOOL_VAR</td>
</tr>
<tr>
<td>SYSTEM_ID</td>
<td>VAR</td>
</tr>
<tr>
<td>STMT_LIST</td>
<td>EXPRESSION</td>
</tr>
<tr>
<td>STOP_PROC</td>
<td>RIGHTBRACE</td>
</tr>
<tr>
<td>SKIP_PROC</td>
<td>LEFTBRACE</td>
</tr>
<tr>
<td>PAR_PROC</td>
<td>BRACE</td>
</tr>
<tr>
<td>SEQ_PROC</td>
<td>SEMICOLON</td>
</tr>
<tr>
<td>NDC_PROC</td>
<td>SEMICOLON</td>
</tr>
<tr>
<td>DC_PROC</td>
<td>DOT</td>
</tr>
<tr>
<td>MU_PROC</td>
<td>DUMMY</td>
</tr>
<tr>
<td>PROC_CALL</td>
<td>SYNCH_MSG</td>
</tr>
<tr>
<td>PROCESS_DEC</td>
<td>GUARD1</td>
</tr>
<tr>
<td>CHANNEL</td>
<td>GUARD2</td>
</tr>
<tr>
<td>INPUT</td>
<td>RECURSE</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>RECUR_TOP</td>
</tr>
<tr>
<td></td>
<td>SDT</td>
</tr>
<tr>
<td></td>
<td>TYPES</td>
</tr>
</tbody>
</table>

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

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

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

```
TrainXing =
-- Two processes Train and Gate consist of
-- sequential actions and run concurrently.
-- Two synchronization messages are required
-- to command the Gate.

PROCESS Train =
SEQ{
    InTransit():FAIL(p= 0.01),
    (ToGate ! Arrive):FAIL(r= 0.02),
    AtIntersection(),
    (ToGate ! Depart):FAIL(r= 0.03)};

PROCESS Gate =
SEQ{
    (ToGate ? Arrive),
    Closed(),
    (ToGate ? Depart),
    Open();

PAR{Train(), Gate() {Arrive, Depart}}.
```

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

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

### 4.8 Linking synchronization primitives

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

---

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

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

4.9.1  Cmd_line.c description

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

4.9.2  The csp.l and csp.y descriptions

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

The token descriptions that lex uses are known as regular expressions. As the input is divided into tokens, the CSPN tool must establish the relationship among tokens. CSPN needs to find expressions, statements, declarations, blocks, and processes in the specification
program. This task is known as parsing and the list of rules that define the relationships that CSPN understands is the grammar (also called the yacc specification and for the P-CSP language is found in Appendix B). Yacc takes a concise description of the grammar (basically in BNF notation and is found in the csp.y file) and produces a C routine that can parse the grammar, called the parser. The parser detects when a sequence of input tokens matches one of the rules in the grammar and also detects syntax errors whenever the input doesn't match any of the rules.

4.9.3 Symbol_cspy.h and symbol_cspy.c description

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

TABLE 6

SYMBOL TABLE UTILITY FUNCTIONS

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

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

TABLE 7
NET UTILITY FUNCTIONS

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Puts integers on Stack[STACKSIZE].</td>
</tr>
<tr>
<td>pop</td>
<td>Returns the integer on top of stack.</td>
</tr>
<tr>
<td>peak</td>
<td>Non-destructive pop.</td>
</tr>
<tr>
<td>printStack</td>
<td>Prints the stack contents top to bottom.</td>
</tr>
<tr>
<td>linkToSiblings</td>
<td>If cur_pnode has sib relation link to the sib.</td>
</tr>
<tr>
<td>append</td>
<td>Given net[root pointer] append a node to end list.</td>
</tr>
<tr>
<td>allocate_net</td>
<td>Allocate a NET_NODE for a PROCESS declared symbol.</td>
</tr>
<tr>
<td>allocate</td>
<td>Allocate a NODE for a symbol w/in a PROCESS definition.</td>
</tr>
<tr>
<td>linkup</td>
<td>Link a NET_NODE to a net[root pointer].</td>
</tr>
<tr>
<td>searchNet</td>
<td>Traverse the net[i]'s to build atomic co-matrices.</td>
</tr>
<tr>
<td>updateNet</td>
<td>Traverses the net[i]'s to transfer any failure annotations in the symbol table to the net data structure (in the *n_fail field of NODE).</td>
</tr>
<tr>
<td>printNet</td>
<td>Given a netNodeptr print the contents of a NET_NODE.</td>
</tr>
<tr>
<td>net_init</td>
<td>Initializes net_main's global variables.</td>
</tr>
<tr>
<td>net_main</td>
<td>A (large) switch on sym_type to decide structure of the net.</td>
</tr>
</tbody>
</table>
Each invocation of the searchNet function requires a pointer to a NODE structure. These
NODE structure pointers are contained in the array sys[] (each i in sys[i] is a PROCESS
declaration). Each PROCESS declaration is represented by a NET_NODE which contains a
net[i] pointing to individual p-nodes (see Figure 36) nested within the process declaration.
The net[i] array contains pointers to related p-nodes (i.e., when they are used in a sys[i]
PROCESS declaration). Each p-node instance is represented by a NODE with a name field
for its name (process information is kept in the symbol table referenced by the n_name field).

4.9.5 Prlist.c description

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

4.9.6 Scoring.c description

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

---

8These routines were developed with help from David Sheely (at The University of Texas at Arlington).
AddFailures is called from the main line code in the parser (csp.y) routine if the "-f" option was specified on the command line meaning that the ignoreFailures flag is not set. The routine parses through the process list of the system co-matrix (which is named as prmatrix), looks up each process in the symbol table and checks if a failure annotation is stored there. If so it will append a failure transition to the co-matrix and update the process list for the system symbol.

4.9.7 Exn_csp.y.c description

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

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

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

4.9.8 Petri_csp.y.c description

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

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

4.9.10 **Intermediate output files used for debugging**

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

**TABLE 8**

**DESCRIPTION OF INTERMEDIATE TRANSLATION FILES**

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fn.tok</td>
<td>Lists the tokens passed from the scanner to the parser</td>
</tr>
<tr>
<td>fn.dec</td>
<td>Output from the searchNet routine (in net.c). Lists the symbols that were found in searching the net hierarchy, their co-matrix (Petri net representation) and failure annotation (if any).</td>
</tr>
<tr>
<td>fn.dsd1</td>
<td>Snapshot of the symbol table after the decomposition phase completes.</td>
</tr>
<tr>
<td>fn.epn</td>
<td>Lists all intermediate steps taken during expansion of the component Petri nets into the one system Petri net (i.e., combining co-matrices). This includes the steps associated with resolving synchronization links (a reduction process) and including failure annotations.</td>
</tr>
<tr>
<td>fn.net</td>
<td>A key file which lists all of the net hierarchy in a staggered format that shows the nesting and adjacency relationship in 2 dimensions.</td>
</tr>
<tr>
<td>fn.dsd2</td>
<td>Snap shot of the symbol table after the expansion process has completed.</td>
</tr>
<tr>
<td>fn_spnp.c</td>
<td>This is the CSPL specified Petri net file on which the stochastic analysis may commence using the SPNP tool.</td>
</tr>
</tbody>
</table>
CHAPTER 5

ILLUSTRATION OF THE USEFULNESS OF THE CSPN TOOL

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

—Robert F. Kennedy

5.1 Combining functional and performance analysis

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

5.2 Requirement specification for the railroad crossing

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

---

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

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

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

### 5.3 The CSP for the railroad crossing

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

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

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

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

---

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

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

```
TrainXing =
  --Two processes Train and Gate consist
  --of sequential actions and run
  --concurrently. Two synchronization
  --msgs are required to command the gate.

PROCESS Train =
  SEQ{
    InTransit(),
    {Togate ! arrive},
    AtIntersection(),
    {Togate ! depart}};

PROCESS Gate =
  SEQ{
    {Togate ? arrive},
    Closed(),
    {Togate ? depart},
    Open()};

PAR{
  Train(), Gate() {arrive, depart}}.
```

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

The original CSP specification in Figure 37 provides that both processes repeat their internal activities continuously. However, given the P-CSP specification of Figure 38, the resultant Petri net graphically reveals the absence of iteration to provide for the handling of a continuous stream of trains. To provide iteration, an additional composition is added: namely Mu.X{PAR{Train(), Gate() (arrive, depart)} → X}. In this case, X is a recursive process
that provides the link between the dummy transitions dt1 and dt2 shown in Figure 41. The new net which incorporates iteration is shown in Figure 42.

TrainXing =
--Two processes Train and Gate consist
--of sequential actions and run
--concurrently.
PROCESS Train =
  --WHILE TRUE
  SEQ{
    InTransit(),
    (Togate ! arrive),
    AtIntersection(),
    (Togate ! depart)};
--END while
PROCESS Gate =
  --WHILE TRUE
  SEQ{
    (Togate ? arrive),
    Closed(),
    (Togate ? depart),
    Open());
--END while
Mu.X{
  PAR{
    Train(),
    Gate() {arrive, depart}} \rightarrow X}.

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

5.5 Semantics of the Petri net for the railroad crossing

The train and gate operate concurrently and independently. However, for the system to meet its functional requirements both components must synchronize. To accomplish their missions (i.e., passing through the intersection and holding traffic to permit the train to pass safely) they use the channel "Togate" to synchronize. The synchronization described by the CSP may not readily reveal the potential race hazard that is more detectable in the Petri net.
The Train process could arrive to AtIntersection before the gate closes! To avoid this unsafe state an extra "ok" gate closed synchronization message is used. In Figure 43 the messages are represented by transitions dt:arrive, dt:ok and dt:depart. The prefix "dt:" denotes a "dummy transition" that fires with probability one (i.e., an immediate transition).

\[
\text{CSP} \\
\text{Train} = \\
\quad \text{(InTransit);} \\
\quad \text{(Togate ! arrive} \rightarrow \text{Togate ? ok} \\
\quad \quad \rightarrow \text{AtIntersection);} \\
\quad \text{(Togate ! depart} \rightarrow \text{Train)} \\
\text{Gate} = \\
\quad \text{(Togate ? arrive} \rightarrow \text{Close} \\
\quad \quad \rightarrow \text{Togate ! ok);} \\
\quad \text{(Togate ? depart} \rightarrow \text{Open} \rightarrow \text{Gate)} \\
\text{TrainXing = Train || (arrive,ok,depart) Gate} \\
\]

\[
\text{P-CSP} \\
\text{TrainXing =} \\
\quad \text{PROCESS Train =} \\
\quad \quad \text{SEQ(} \\
\quad \quad \quad \text{InTransit()}, \\
\quad \quad \quad \text{(Togate ! arrive),} \\
\quad \quad \quad \text{(Togate ? ok),} \\
\quad \quad \quad \text{AtIntersection()}, \\
\quad \quad \quad \text{(Togate ! depart));} \\
\quad \text{PROCESS Gate =} \\
\quad \quad \text{SEQ(} \\
\quad \quad \quad \text{(Togate ? arrive),} \\
\quad \quad \quad \text{Close()}, \\
\quad \quad \quad \text{(Togate ! ok),} \\
\quad \quad \quad \text{(Togate ? depart),} \\
\quad \quad \quad \text{Open());} \\
\quad \text{Mu.X(} \\
\quad \quad \quad \text{PAR(Train(),Gate())} \\
\quad \quad \quad \text{(arrive,ok,depart)} \rightarrow X). \\
\]

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

\[11\text{This is possible because after the synchronization on the "Togate" channel occurs (i.e., the "arrive" signal is received), the "AtIntersection" transition may fire before the "Close" transition denoting the case where the train arrived sooner than the time needed for the gate to close.}\]
The gate will not begin to close until it receives the "arrive" message. First, the train must fire the transition "InTransit," and then send the "arrive" message by firing "Togate!arrive." In turn, the gate must be ready to receive the message by firing the transition "Togate?arrive." After all these actions have occurred, the gate may receive the command to close. The close command may occur (i.e., fires at some definite rate) when a token is on place "p13." This will occur immediately after the synchronizing "dt:arrive" transition has been enabled (tokens on "p4" and "p12") since this transition is immediate (consumes no resources). The marking with one token each on places p14 and p12 which enable the "dt:arrive" transition to fire.

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

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

5.5.1 Enumerating all possible failure transitions

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

---

12We have studied the case where multiple trains may arrive at the intersection. In such cases, it becomes necessary to have a monitor arbitrate (see Appendix E for a brief look at the solution to such a case).
13If we assume the gate always opens and closes sooner than the time it takes the train to reach the crossing, the PN can be viewed as hazard free (except for the possibility of the gate having mechanical failure [unsafe]).
failure transitions. Dummy transitions are assigned a probability one and do not have any associated failure transitions. It is interesting to note, that instead of transitioning to place "p20" as shown in Figure 44, it would be possible to separate distinct failure types into different "absorbing" places so that the MTTF values (or the failure rate) associated with each type of failure mode can be separately denoted and computed.

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

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

5.5.2 Enumerating safety critical failure transitions

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

\(^{14}\)For example, the time to failure of \(ft:close\) is known to be exponentially distributed with rate \(\lambda_1\) (lets say). This is modeled in the stochastic Petri net by associating a firing time with each of the transitions.
given failure presents to its users (and/or developers). In this section the discussion will be based on the railroad crossing that is discussed above which has a race hazard resulting from a "runaway" train. The states in Figure 45 (which are based on the Petri net pictured at the right) demonstrate that there are two unique manifestations of failures (i.e., critical and non-(safety)-critical). In considering the criticality of timing, we see that the slow firing of transition Close makes it possible for the train to enter the intersection before the gate has properly (or completely) closed. Similarly transition Open makes it possible for the train to have departed and still, the gate is not open.

Figure 45. Markings and requisite Markov state transition diagram.

Missing from the Petri net of Figure 45 are transitions to reflect physical, communication related or mechanical failures. In our analysis, we do assume the existence of such failure transitions (and corresponding places) as discussed in the previous section (5.5.1). The CSP specification (and the corresponding Petri net) can be augmented to show how such failures should be handled. For example, the communication failures can be handled using time-out and re-transmit techniques. But still, should the gate fail to close, the question becomes
what can be done to possibly avoid a catastrophe. Perhaps an audible and visual alarm would alert unsuspecting pedestrians and traffic. Such fault-tolerant and fault-handling actions can be specified both with the CSP and Petri net models. However, they become more obvious by examining and analyzing the stochastic Petri net. The cost of providing fault-tolerance should be traded-off with the required level of reliability.

5.6 Parametric Sensitivity Analysis

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

5.6.1 Discrete Analysis

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

\[\text{15The classic steady-state solution method for stochastic models that maps GSPN models to CTMCs is compared with a method based on DTMCs in [Ciaronso89]. The DTMC method is shown to perform better.}\]
TABLE 9
DISCRETE ANALYSIS (Pt = 0)

<table>
<thead>
<tr>
<th>Desc.</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
<th>Desc.</th>
<th>Run 1</th>
<th>Run 2</th>
<th>Run 3</th>
<th>Run 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>P_c</td>
<td>.0001</td>
<td>.0001</td>
<td>.0001</td>
<td>.0001</td>
<td>P_c</td>
<td>.0526</td>
<td>.0286</td>
<td>.2544</td>
<td>.1690</td>
</tr>
<tr>
<td>P_m(clo)</td>
<td>.01</td>
<td>.0001</td>
<td>.01</td>
<td>.001</td>
<td>P_m(clo)</td>
<td>.4974</td>
<td>.9714</td>
<td>.7456</td>
<td>.8310</td>
</tr>
<tr>
<td>P_m(op)</td>
<td>.01</td>
<td>.001</td>
<td>.03</td>
<td>.005</td>
<td>MTTF</td>
<td>490.26</td>
<td>9254.07</td>
<td>248.19</td>
<td>1656.21</td>
</tr>
</tbody>
</table>

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

5.6.2 Continuous Analysis

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

Figure 46. Results of the continuous analysis.
The unreliability of communications do not significantly impact the MTTFs because we have set those failure rates much lower than the rates associated with the gate's open/close mechanism by a factor of 1,000 (i.e., $\lambda_m = 0.0001 > \lambda_c = 0.0000001$). Mechanical failures and the possibility of the gate not closing (opening) in time (before the train arrives at the intersection) are assumed to be greater. In Figure 46 an interesting relation is evident. We observe that, if the train's speed tends to bring it to the intersection sooner than the gate can close, then an improvement in the gate's mechanical reliability doesn't really help! To improve the overall system's reliability it is more important to provide the additional synchronization between the train and gate processes as described in Section 5.5 (and Figure 43), so as to avoid the possibility of having the gate miss its deadline ($\tau_5$). Alternatively, the train may signal "arrive" much sooner, allowing ample time for the gate to close.

In general, it is important to see how much the least reliable entity impacts the overall system reliability. In Figure 46, there are incremental improvements seen in the reliability of the system at 10,000 time units from $10^{-40}$ to $10^{-5}$ for various values of $\tau_5$ (which reflects the probability that the train arrives before the gate closes). The next most significant gain in system reliability comes when the gate's mechanical failure rate is improved by a factor of ten (note the difference between run 6 and 7 in the graph). In this case, the MTTF improves by 6 times while the corresponding system reliability improves significantly from $\sim2.6\times10^{-5}$ to $\sim3.4\times10^{-1}$.
CHAPTER 6

CONCLUSIONS

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

—Goethe

6.1 Conclusion

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

A textual language for CSP specifications was designed. A software tool was
implemented for translating the CSP specifications into stochastic Petri nets. The Petri nets are coded in the form of a coincidence matrix. The graphical representation of the resulting Petri net can be viewed using the *dot* tool [a Unix filter for drawing directed graphs].\(^\text{16}\) The system coincidence matrix is converted into a file format needed for analysis using SPNP.

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

### 6.2 Future plans

This work can be extended to incorporate a broader scope of translations and the characterization of properties other than structural that are useful for error avoidance, fault tolerance, detection of deadlocks and unsafe behaviors, and timeliness. Other issues include (1) ease of use (e.g., GUI) including mechanisms for detecting characteristics of the Petri net that can be used in automatically\(^\text{17}\) parameterizing the SPNP formatted file, (2) relating the analysis results back to the original specifications in a more rigorous and formal way, (3) expanding the language to incorporate some of the ideas of real-time CSP and others, (4) developing some state reduction techniques for the CSPN (e.g., combining dummy transitions with tangible transitions) and (5) validating our approach by applying the method to larger examples and/or a real system.

---

\(^{16}\)See Drawing graphs with dot by Eleftherios Koutsofios and Stephen C. North at AT&T Bell Laboratories.

\(^{17}\)Currently, CSPN uses a hard coded set of defaults that define the *parameters* part of the SPNP output file. Those defaults can be changed interactively using the "-v" verbose mode flag on the command line. For instance, in nets which generate absorbing states it only makes sense to run a transient analysis.
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 (→) 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 chnl!msg combined with chnl?msg was more natural (i.e., closer to the

<table>
<thead>
<tr>
<th>STOP</th>
<th>SKIP (not defined in CSP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic CSP:</td>
<td>P-CSP:</td>
</tr>
<tr>
<td>STOP</td>
<td>or</td>
</tr>
<tr>
<td>Performs no</td>
<td>STOP</td>
</tr>
<tr>
<td>action and</td>
<td>or</td>
</tr>
<tr>
<td>never</td>
<td>terminal</td>
</tr>
<tr>
<td>terminates!</td>
<td></td>
</tr>
</tbody>
</table>

Recursion

<table>
<thead>
<tr>
<th>Classic CSP:</th>
<th>P-CSP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>μX. (b → X)</td>
<td>Mu.X{b()}</td>
</tr>
</tbody>
</table>

Parallel Composition

<table>
<thead>
<tr>
<th>Classic CSP:</th>
<th>P-CSP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

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

**Classic CSP:**\( a \parallel b \)  
**P-CSP:**\( DC\{a() , b() \} \)

![Diagram of Deterministic choice composition](image)

### Nondeterministic choice composition

**Classic CSP:**\( a \sqcap b \)  
**P-CSP:**\( NDC\{a() , b() \} \)

![Diagram of Nondeterministic choice composition](image)

### Sequential composition and hiding

**Classic CSP:**\( a \rightarrow b \)  
**P-CSP:**\( a \rightarrow b \)  
**P-CSP:**\( (a \rightarrow b \rightarrow c) \setminus b \)

![Diagram of Sequential composition and hiding](image)

Figure A.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 transitions and one less place. In the bottom figure, there are also two fewer transitions and two fewer places.

Figure A.3 Translation of recursive compositions in a reduced format.
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 A.4 Translations showing blocked and non-blocked send synchronization.
Parallel and sequential composition

Classic CSP:

\[(a \rightarrow b \rightarrow c) \parallel_{(b)} (d \rightarrow b \rightarrow e)\]

P-CSP:

\[\text{PAR}\{\text{SEQ}\{a(), c()\}, \text{SEQ}\{d(), e()\}\{b\}\};\]

Parallel and Nondeterministic choice composition

Classic CSP:

\[a \rightarrow (b \sqcap c) \parallel (b \sqcap c) \rightarrow a\]

P-CSP:

\[\text{PAR}\{\{a \rightarrow \text{NDC}\{b(), c()\}\}, \{\text{NDC}\{b(), c()\} \rightarrow a\}\};\]

Above: a() must actually be ch!b, and d() must actually be ch?b to be correct using CSPN

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 guardedproclist 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 -> action [for P->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 an 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
This appendix presents five algorithms. The first chooses one of three methods of expansion (Section C.1), the 3 expansion methods (Section C.2-4), and 'expand' which combines two matrices into one (Section C.5). First, the variable definitions are presented immediately below: Note, the user defined types are found at the end of Appendix C.

```c
/* Three matrices are involved: C = A <- B where "<-" means "is inserted into by." So A is the original matrix, B is the matrix which is poured and C is the new matrix that holds both combined. */
int orA, rA, /* orA x ocA is size of A (original)*/
ocA, cA, /* rA x cA is size of C (new) */
rB, cB, /* rB x cB is size of B (pored) */
rlb, rub, /* B row lower and upper bounds in C*/
clb, cub, /* B column lower and upper bounds in C*/
rlbA, rubA, /* A row lower and upper bounds in C*/
clbA, cubA; /* A column lower and upper bounds in C*/
curLnkIndx = 0, /* Current link index */
rowMark = 0,
colMarkRht = 0,
colMarkLft = 0,
Bflag = FALSE_,
Aflag = FALSE_,
p_matrix A = NULL,
B = NULL,
C = NULL;
void expn(FILE *epn, int cpi, netNodeptr nnptr)
{
    int e,f,i,j,row,col,link,
k, /* k number of nodes in this list */
typ =0,
thinA =FALSE_,
symthere =0; /* Boolean: Is symbol there? */
nodeptr p =NULL,
cur_p =NULL,
q =NULL;
entryptr s0 =NULL,
s1 =NULL;
char *call =NULL;
```
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 {
        curLnkIndx=0;
        for (q = p; q != NULL; q = q -> link) {
            curLnkIndx++;
            fprintf(epn,"\n%d. Symbol: %s, Type: %d",curLnkIndx, 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,"\nExpansion must be aborted ...
\n\n");
                exit(1);
            }
        }
    } else {
        /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * Rem: B is inserted into A (A<-B (or A is expanded by B)
        * Thus the following logic sets the stage for an expansion:
        */
fprintf(epn,"\n\nExpansion includes the following:");

A = s0 ->p_prm;
fprintf(epn,"\n\nA: %s",s0->name);
DisplayProcessList(epn, 0, s0->p_pl);
print_prm(epn,A,s0->rsize,s0->csize);

B = s1 ->p_prm;
fprintf(epn,"\n\nB: %s",q->n_name);
DisplayProcessList(epn, 0, s1->p_pl);
print_prm(epn,B,s1->rsize,s1->csize);

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* Calc size of the new C matrix (unless B matrix is null) *
*/
thinA = FALSE_;
rA = rA+(rB-1);
if (A[orA-1].p_row[ocA-1] > 0)
cA = cA+(cB-2);
else
  if (A[orA-1].p_row[ocA-1] < 0) {
    cA = cA+(cB-2);
    thinA = TRUE_;
  } else
    fprintf(stderr,"\nA[orA][ocA] = 0! Aborting expansion ...\n");
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* Create the new matrix C that A and B will be combined into. *
* ----------------------------------------------------------- *
*/
C = prmatrix(rA,cA);
fprintf(epn,"\n\nC: A<-B is a new (%dx%d) Matrix",rA,cA);
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* Determine the rowMark where expansion begins. The rowMark *
* tracks the place in A that will be replaced. s0->p_pl is the *
* process list of A, q->name is the process who will be replaced. *
* Determine where that is in the co-matrix (0 =1st position). *
*/
rowMark=procPosition(s0->p_pl, q->n_name);
if (rowMark == -1) {
  fprintf(stderr,"\nrowMark undetermined!\n");
  exit(1);
}
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
* Use Method I if B goes into Upper Left corner of C. *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * 
if (rowMark == 0) {
  fprintf(epn,"\n\nRunning Method 1:
-----------------\n" );
SEE SECTION C.2 FOR METHOD 1 LOGIC

/*fi use Method 1 */
else {
    /* - - - - - - - - - - - - - -<<2>>- - - - - - - - - - - - */
    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    /* Use Method II if B goes into Lower Right corner of C. */
    /* A goes into Upper left corner of C. */
    */
    if (rowMark == (orA-1)) {
        fprintf(epn,"

Running Method 2:
-----------------");
        SEE SECTION C.3 FOR METHOD 2 LOGIC
    }      /*fi use Method II */
    else {
        /* - - - - - - - - - - - - - -<<3>>- - - - - - - - - - - - -
        * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
        * Method III: all other cases B goes in center of C. */
        */
        fprintf(epn,"

Running Method 3:
-----------------");
        SEE SECTION C.4 FOR METHOD 3 LOGIC
    } /*esle in all other cases */
} /*esle */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* CLEAN UP: Free the old prmatrix to conserve memory. */
/*
    for (i = 0; i < s0 ->rsize; i++) {
        free(A[i].p_row);
        free(A);
    }
    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    /* Update the symbol table entry for this pnode */
    */
    s0 ->rsize = rA; s0 ->csize = cA; s0 ->p_prm = C;
    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    /* replProc takes a process list and replaces a process name */
    /* known to exist in the list by another process list of */
    /* one or more process names. */
    */
    call= replProc (&(s0 ->p_pl), s1 ->name , &(s1 ->p_pl));
    if (call == NULL) {
        fprintf(stderr,"\nExn: ReplProc failed update %s",s0 ->name);
        exit(1);
    }
    /* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
    /* Adjust the link index to comply with the prior expansion */
    */
    curLnkIndx=curLnkIndx+rB-1;
} /*esle*/
} /*fi (typ...*/
} /*rof*/
} /*esle*/
} /*elihw*/
fprintf(epn,"\nCompleted expansion!");
}/*npxe*/
C.2 Algorithm for expansion method 1 (upper LH corner)

```c
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Determine indexes for B whose size is rB x cB (C <- B).
* rl = row lower bound, rub = row upper bound
* clb = col lower bound, cub = col upper bound
*/
rlb = curLnkIndx - 1;  rub = rl + rB - 1;
clb = 0;               cub = clb + cB - 1;

/*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
* Expand A (3x4) with B (5x6) into C (7x8).
*
* 0 1 2 3 4 5 6 7             0 1 2 3
* A: 0 -|+ a a <-a's put in C,
* 1 b b b b b 0 0           1 y g g g   begin at C[4,6].
* 2 b b b b b 0 0           2 y g g g
* 3 b b b b b 0 0 /          / Rows 1 and 2:
* 4 b b b b b + a a /       / put g's in C,
* 5 y 0 0 0 0 0 g g g       / start at C[5,5].
* 6 y 0 0 0 0 0 g g g /     / Rows 1 and 2: put y's
*                             / in C start at C[5,0].
*                                                                         
* In C above, 0's are constant (i.e., not x-fered from A or B into C). Also note that the "-" and "+" paired in A are now separated as shown in C. 
*                                                                         
* Determine indexes for A whose size is orA x ocA (C <- A).
*/
rlbA = rA - (orA -1);  rubA = rA - rl - 1;
clbA = cA - (ocA -1);  cubA = cA - clb - 1;
(void)expand(epn, ZERO);
```

C.3 Algorithm for expansion method 2 (lower RH corner)

```c
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
* Determine indexes for B whose size is rB x cB (C <- B).
* rl = row lower bound, rub = row upper bound
* clb = col lower bound, cub = col upper bound
*/
rlb = rowMark;
rub = rl + rB - 1;
```
if (!thinA) clb = cA - cB;
else clb = cA - cB + 1;
cub = cA - 1;
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 * Determine indexes for A whose size is orA x ocA (C <- A).
 * ---------------------------------------------------------
 */
rlbA = 0;
rubA = orA - 2;
clbA = 0;
cubA = ocA - 1;
/*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
* Case: 1
* (orA x ocA) (rB x cB) (rA x cA)
* 1 2 3 4 5 1 2 3 4 1 2 3 4 5 6 7
*A: 1 a a a a z + B: 1 - + b b --> C: 1 a a a a 0 0 z
* 2 a a a a z 2 b b b b 2 a a a a 0 0 z
* 3 a a a a z 3 b b b b 3 a a a a 0 0 z
* 4 a a a a z 4 b b b b 4 a a a a 0 0 z
* 5 y y y + 5 0 0 0 - + b b
* / 6 0 0 0 b b b b
* Case 2:
* If A[5,5]="-" => C(8x6). 8 y y y b b b b
*
* Here (in case 1), A(5x5) + B(4x4) -> C(8x7).
* There is one variation (Case 2) occurs when the "-" is
* in the last column (e.g., occurs with Mu recursion). In
* this case, A(5x5) + B(4x4) -> C(8x6).
*
* For e.g., (remember rowMark=row to replace [exactly]):
* *
* Case 1: C(8x7) Case 2: C(8x8)
* *
* rlb = 4 (counting from 0) 4
* rub = 7 = 4 + 4 - 1 7
* clb = 3 = 7 - 4 3 = 6 - 4 + 1 (test)
* cub = 6 = 7 - 1 5 = 6 - 1
* *
* rlbA = 0 0
* rubA = 3 = 5 - 2 3
* clbA = 0 0
* cubA = 4 = 5 - 1 4
* *
* For case 1 (where C[5,5] = "+") the z's in A are moved
* to the last col of C ONLY if "+" otherwise, they stay
* put (this is handled in the method 2a exceptions below).
* Similarly, in either case 1 or case 2, the y's in A are
* moved to the last row in C ONLY if "+" otherwise they
* stay put (this is handled in the method IIB exceptions).
* *
* If a "z" or a "y" is moved it must be replaced by a "0".
* ---------------------------------------------------------
* */
(void)expand(epn,ZERO);
/* * * * <<< 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] = 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[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;
clb = j;
cubA = clb + ONE;
cub = clb + cB -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 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++) {
  if (!(f>cub)) {f++;}
  else {
    fprintf(stderr, "\n1-Method III error(clb)!
");
    exit(1);
  }
}
if (!(e>rub)) {e++;}
else {
  fprintf(stderr, "\n2-Method III error(rlb)!
");
  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++)
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: */

```c
if (s0->type == NDC_PROC) {
  fprintf(epn, "Method III exception!\n");
  prtExpn(epn);
  printprm(epn, C, rA, cA); fprintf(epn, "\n");
  zeroGapStarts = colMarkLft + (cA - ocA);

  /* Zero out the columns starting with the column ColMarkLft
  * making sure to stay above the rowMark
  */
  for (i = 0; i < rowMark; i++)
  for (j = colMarkLft; j < zeroGapStarts; 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 colMarkLft
  * defines the boundary in A (not C) where the expansion
  * occurs (just one col to the left of the colMarkLft column).
  */
  zeroGap = cA - ocA;
  for (i = 0; i < rowMark; i++)
  for (j = colMarkLft; j < ocA; j++)
  for (j=colMarkLft; 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

```c
/* 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)) {
          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 dpndnt) */
   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 /* Node definition */
{
   char  *n_name;            /* NULL if no fail rate/prob spec'd */
   char  *n_fail;            /* Boolean: legal vals (-1, 0, 1) */
   short isrte;             /* Node type consistent w/ symbols */
   short n_type;            /* System level unique identifier */
   short uid;               /* Ptr to next node, if any */
   struct nodedef *link;
} 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.

![FSM for Monitor:](image1)

**CSP for Monitor:**

\[
\text{Monitor} = \left( (T1 \ ? a \land T2 \ ? a) \land \text{GateClosed} \rightarrow \text{Monitor} \right);
\]

\[
\left( (T1 \ ? a \land T2 \ ? a) \land \text{GateOpen} \rightarrow (\text{GateCh ! Close}) \rightarrow \text{Monitor} \right);
\]

\[
\left( ((T1 \ ? d) \land (T2 \ ? a) \land (T2 \ ? d) \land (T1 \ ? a)) \rightarrow \text{Monitor} \right);
\]

\[
\left( ((T1 \ ? d) \land (T2 \ ? a) \land (T2 \ ? d) \land \neg (T1 \ ? a)) \rightarrow (\text{GateCh ! Close}) \rightarrow \text{Monitor} \right).
\]

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

![Petri Net for the monitor (controller) to handle multiples trains](image2)

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**

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

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, (Mtf) timing failure if train arrives before the gate is closed, (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, (Mcsf) 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: \((M_{\text{cf}})\) safety critical failures involving gate closure, and \((M_{\text{ncf}})\) 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 \((\text{TG?close} \ [t_5])\) and \((\text{Closed} \ [t_6])\) 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 \((\text{TG?open} \ [t_7])\) and \((\text{Open} \ [t_8])\).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:

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

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 ($\mu_7$) and the gate closure rates ($\mu_6$) were evaluated. The system's performability was studies to determine how reliably the gate closes before the train arrives with and without hardware and communication failures (i.e., mechanical gate failures [$\lambda_5$, $\lambda_{13}$ 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., $\mu_6$ and $\mu_7$ 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 $\sigma_9$ 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.
*Time units: each tick on the x-axis is 1000tus. If a tu is a second then there are ~16mins/tic, and 10,000 ticks is ~2778hrs (full range of data).

<table>
<thead>
<tr>
<th>Transition Rates</th>
<th>Failure Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu_1$ = 0.0001</td>
<td>$\lambda_1$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_2$ = 1.0</td>
<td>$\lambda_2$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_3$ = 1.0</td>
<td>$\lambda_3$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_4$ = 1.0</td>
<td>$\lambda_4$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_5$ = 1.0</td>
<td>$\lambda_5$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_6$ = 0.002</td>
<td>$\lambda_6$ = 0.0001</td>
</tr>
<tr>
<td>$\mu_7$ = 0.0002</td>
<td>$\lambda_7$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_8$ = 1.0</td>
<td>$\lambda_8$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_9$ = 1.0</td>
<td>$\lambda_9$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_{10}$ = 1.0</td>
<td>$\lambda_{10}$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_{11}$ = 1.0</td>
<td>$\lambda_{11}$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_{12}$ = 1.0</td>
<td>$\lambda_{12}$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_{13}$ = 1.0</td>
<td>$\lambda_{13}$ = 0.0000001</td>
</tr>
<tr>
<td>$\mu_{14}$ = 0.01</td>
<td>$\lambda_{14}$ = 0.0001</td>
</tr>
<tr>
<td>$\mu_{15}$ = 0.01</td>
<td>$\lambda_{15}$ = 0.0001</td>
</tr>
</tbody>
</table>

* $\mu_6$ is this rate was varied from .002, .05 to .1.
** $\mu_7$ is this rate was varied from .0002, .005 to .01.

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

*Time units: each tick on the x-axis is 1000tus. If a tu is a second then there are ~16mins/tic, and 10,000 ticks is ~2778hrs (full range of data).

$1/\mu_7 = 90\%$ of the time the train takes at most 500tus to reach the gate crossing.

$1/\mu_6 = 80\%$ of the time the gate takes at most 10tus to close.

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

* $\mu_6$ is this rate was varied from 0.0 to .0000908.
** $\mu_6$ and $\mu_7$ is held constant at 1.0 and 0.01.
*** $\sigma_9$ is held constant at 0.001 (zeroed in 3rd run).
Time units: each tick on the x-axis is 1000tus. If a tu is a second then there are ~16 mins/tic, and 10,000 ticks is ~2778 hrs (full range of data).

Performability / Reliability

Transition Rates

<table>
<thead>
<tr>
<th>Transition Rates</th>
<th>Failure Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ1= 0.00001</td>
<td>λ1= 0.0000001</td>
</tr>
<tr>
<td>μ2= 1.0</td>
<td>λ2= 0.0000001</td>
</tr>
<tr>
<td>μ3= 1.0</td>
<td>λ3= 0.0000001</td>
</tr>
<tr>
<td>μ4= 1.0</td>
<td>λ4= 0.0000001</td>
</tr>
<tr>
<td>μ5= 1.0</td>
<td>λ5= 0.00001</td>
</tr>
<tr>
<td>μ6= 0.002</td>
<td>τ6= 0.0000908</td>
</tr>
<tr>
<td>μ7= 0.001</td>
<td>λ7= 0.0000001</td>
</tr>
<tr>
<td>μ8= 1.0</td>
<td>λ8= 0.0000001</td>
</tr>
<tr>
<td>μ9= 1.0</td>
<td>σ9= 0.001</td>
</tr>
<tr>
<td>μ10= 1.0</td>
<td>λ10= 0.0000001</td>
</tr>
<tr>
<td>μ11= 1.0</td>
<td>λ11= 0.0000001</td>
</tr>
<tr>
<td>μ12= 1.0</td>
<td>λ12= 0.0000001</td>
</tr>
<tr>
<td>μ13= 1.0</td>
<td>λ13= 0.0000001</td>
</tr>
<tr>
<td>μ14= 0.01</td>
<td>λ14= 0.0001</td>
</tr>
<tr>
<td>μ15= 0.01</td>
<td></td>
</tr>
</tbody>
</table>

*τ6 is rate varied from .00000908, .00001816 to .0000454.
**μ6 = varied from 0.01 to 0.002.
***μ7 = held constant at .001.

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

Timing ratio γ = train approach time / gate close time. Timing failure rate τ6 = .000000908 as the basis.

Run1: γ = 10 and Timing failure rate = τ6
Run2: γ = 5 and Timing failure rate = 2* τ6
Run3: γ = 5 and Timing failure rate = 5* τ6

Transition Rates

<table>
<thead>
<tr>
<th>Transition Rates</th>
<th>Failure Rates</th>
</tr>
</thead>
<tbody>
<tr>
<td>μ1= 0.0001</td>
<td>λ1= 0.0000001</td>
</tr>
<tr>
<td>μ2= 1.0</td>
<td>λ2= 0.0000001</td>
</tr>
<tr>
<td>μ3= 1.0</td>
<td>λ3= 0.0000001</td>
</tr>
<tr>
<td>μ4= 1.0</td>
<td>λ4= 0.0000001</td>
</tr>
<tr>
<td>μ5= 1.0</td>
<td>λ5= 0.00001</td>
</tr>
<tr>
<td>μ6= 0.002</td>
<td>τ6= 0.0000908</td>
</tr>
<tr>
<td>μ7= 0.001</td>
<td>λ7= 0.0000001</td>
</tr>
<tr>
<td>μ8= 1.0</td>
<td>λ8= 0.0000001</td>
</tr>
<tr>
<td>μ9= 1.0</td>
<td>σ9= 0.001</td>
</tr>
<tr>
<td>μ10= 1.0</td>
<td>λ10= 0.0000001</td>
</tr>
<tr>
<td>μ11= 1.0</td>
<td>λ11= 0.0000001</td>
</tr>
<tr>
<td>μ12= 1.0</td>
<td>λ12= 0.0000001</td>
</tr>
<tr>
<td>μ13= 1.0</td>
<td>λ13= 0.0000001</td>
</tr>
<tr>
<td>μ14= 0.01</td>
<td>λ14= 0.0001</td>
</tr>
<tr>
<td>μ15= 0.01</td>
<td></td>
</tr>
</tbody>
</table>

*τ6 = rate was varied from .00000908, .00001816 to .0000454.
**μ6 = varied from 0.01 to 0.002.
***μ7 = held constant at .001.

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

*Time units: each tick on the x-axis is 1000tus. If a tu is a second then there are ~16 mins/tic, and 10,000 ticks is ~2778 hrs (full range of data).

1/μ7 = 90% of the time the train takes at most 500tus to reach the gate crossing.
1/μ6 = 80% of the time the gate takes at most 50tus to close.
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 \(\tau_6\). These parameters are negatively correlated (i.e., as the slack time \([1/\mu_7 - 1/\mu_6]\) gets smaller \(\tau_6\) 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 \(\tau_6\) 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 \(\mu_6\) and \(\mu_7\) (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.
BIBLIOGRAPHY


