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 (Universita` 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

4.9.6 Scoring.c description 59
4.9.7 Expn_cspy.c description 60
4.9.8 Petri_cspy.c description 60
4.9.9 Miscellaneous file descriptions 60
4.9.10 Intermediate output files used for debugging 61

CHAPTER 5

ILLUSTRATION OF THE USEFULNESS OF THE CSPN TOOL 62

5.1 Combining functional and performance analysis 62
5.2 Requirement specification for the railroad crossing 62
5.3 The CSP for the railroad crossing 63
5.4 The P-CSP for the railroad crossing 64
5.5 Semantics of the Petri net for the railroad crossing 65
5.5.1 Enumerating all possible failure transitions 67
5.5.2 Enumerating safety critical failure transitions 69
5.6 Parametric Sensitivity Analysis 71
5.6.1 Discrete Analysis 71
5.6.2 Continuous Analysis 72

CHAPTER 6

CONCLUSIONS 74

6.1 Conclusion 74
6.2 Future plans 75

APPENDIX A

CSP-TO-PETRI NET CANONICAL TRANSLATION DIAGRAMS 76

APPENDIX B

THE LEX AND YACC SPECIFICATION OF THE PARSEABLE CSP 84

APPENDIX C

CO-MATRIX EXPANSION ALGORITHMS 89

APPENDIX D

RAILROAD CROSSING USING A MONITOR 101

BIBLIOGRAPHY 110

LIST OF FIGURES

Figure Page

1. Linking failure behavior to specification characteristics 3
2. Trace equivalence versus bisimilarity 7
3. CSP-to-Petri net example of trace equivalence versus bisimilarity 8
4. An integrated approach of stochastic analysis [Bernardo95] 10
5. SPNP input file structure 19
6. Refinement of system models 21
7. Example CSP to Petri net translation rules (P-CSP shown in lower portion) 22
8. CSP to Petri net rules for Actions and the STOP and SKIP processes 24
9. CSP to Petri net rules for Combining actions and processes 25
10. CSP to Petri net example of combining actions to form one process 26
11. CSP to Petri net rules for Deterministic and nondeterministic choice 27
12. CSP to Petri net rules for Parallel and synchronized parallel composition 28
13. Comparing CSP and P-CSP to Petri net rules for recursion 29
14. CSP to Petri net rules for recursion 30
15. VMC specification for CSP to Petri net with failure annotations 32
16. CSPN translation of the VMC example 34
17. Structure of the feasible sequences of VMC Marking transitions 36
18. Partial VMC translation showing deterministic choice 36
26. Activities associated with the translation phases of the CSPN tool 39
27. Context diagram and translation phases of the CSPN tool 40
28. P-CSP constructions with co-matrix and Petri net representations 44
29. Choosing a combining method for expansion that depends on locality 45

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 Page

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

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 <u>and</u> 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) <u>and</u> 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.

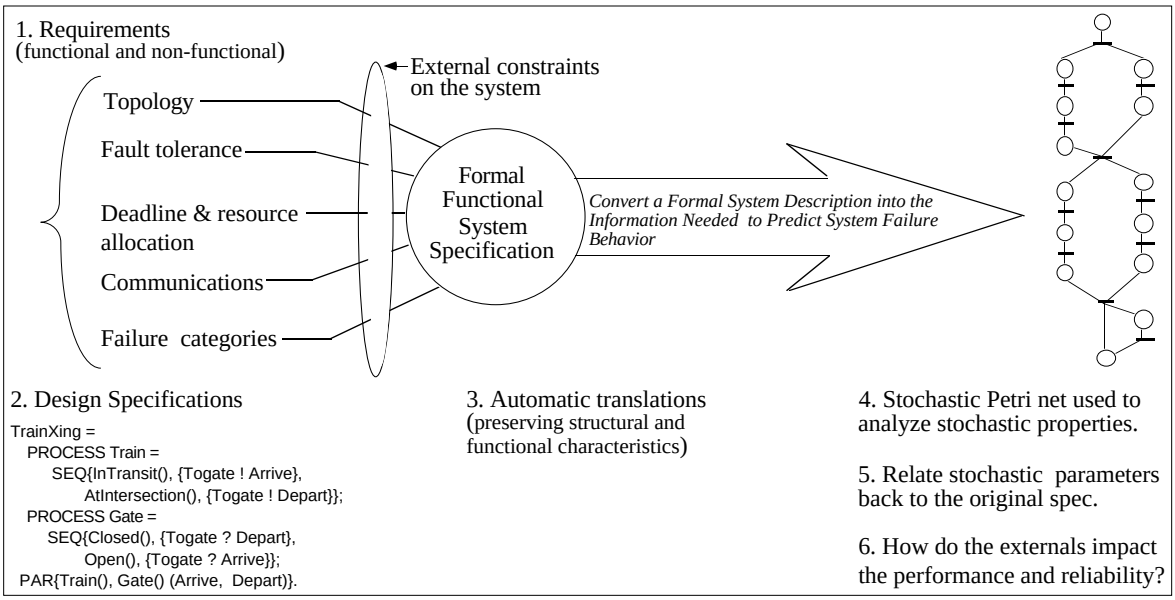


Figure 1. Linking failure behavior to specification characteristics.

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].

---

[1]However, 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.

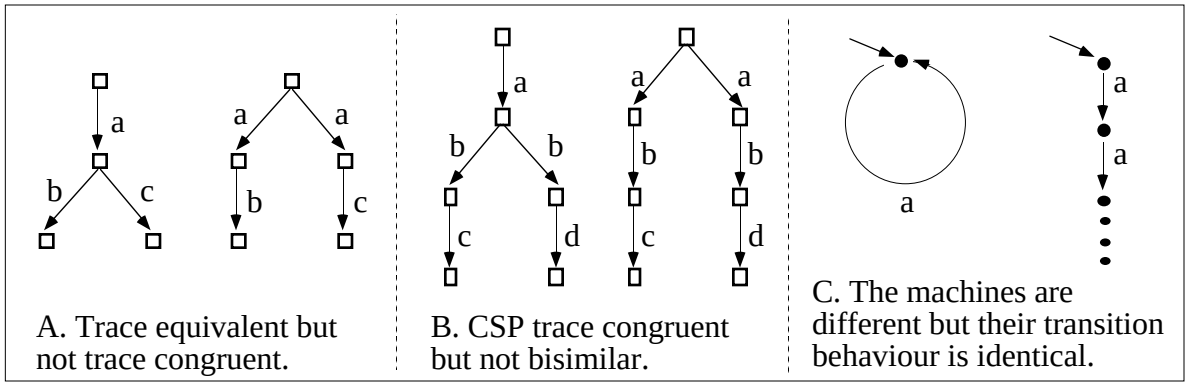


Figure 2. Trace equivalence versus bisimilarity.

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_l & PN_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_r can distinguish the specific firings of individual "b" transitions while PN_l can not.

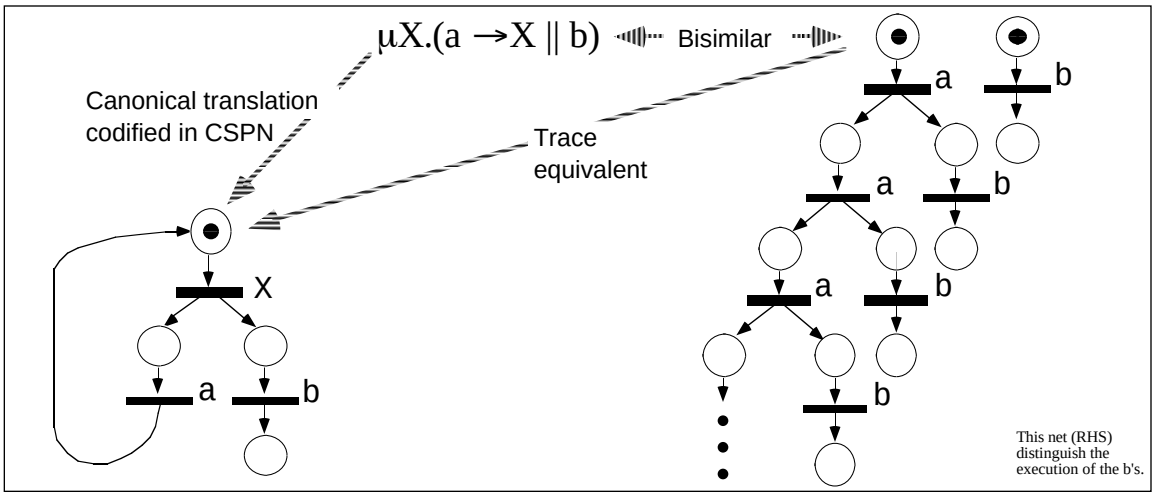


Figure 3. CSP-to-Petri net example of trace equivalence versus bisimilarity.

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 specfier 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*.[3] 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 <u>and</u> 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

[3]Such 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.[4] Thus, the

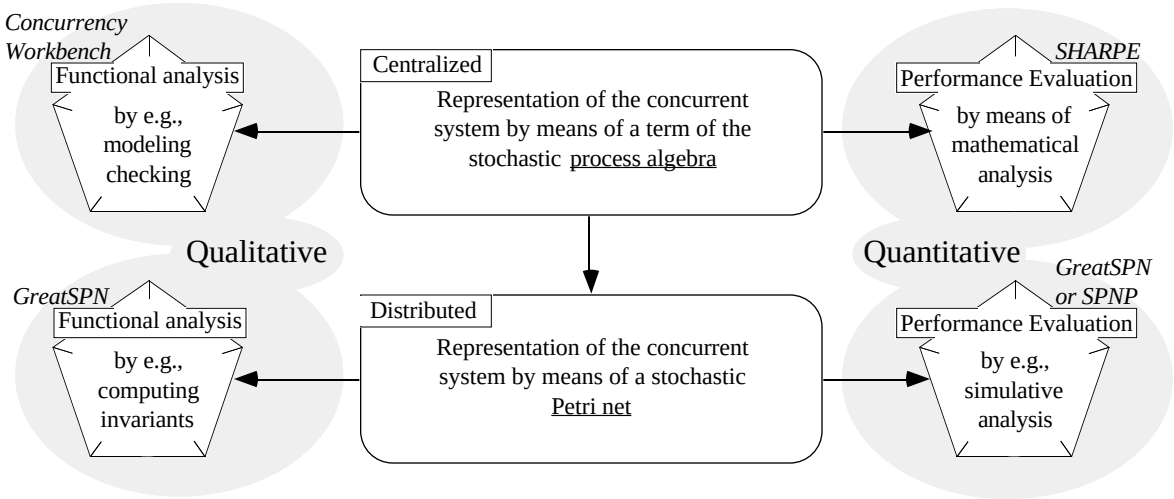


Figure 4. An integrated approach of stochastic analysis [Bernardo95].

[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 || 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

[5]Estelle 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

[6]The 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]

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

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 (||), which is associative: $P = \{P_1 \parallel P_2 \parallel ..... \parallel P_n\}$. 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 (<u>or</u> 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> → <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'):

$$P ::= STOP \mid (a \rightarrow P) \mid (P\backslash b) \mid (P \sqcap Q) \mid (P \,\Box\, Q) \mid (P \parallel_b Q) \mid (P; Q) \mid (\mu X \bullet P).$$

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, $(P\backslash b)$ is the same as P except action b is hidden[7], $(P \sqcap Q)$ represents a non-deterministic choice between P and Q, $(P \,\Box\, Q)$ 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 ||), SEQ (or ;), NDC (or $\sqcap$), DC (or $\Box$), 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\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 inforced by the CSPN tool.

process to engage in an action (e.g., a → 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.

[9]C 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.[10] Other researchers have based their system models on conditions and events (where their events are similar to our actions / processes).[11] 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.[12] As transitions are enabled, the state of the Petri net moves

---

[10]CSP processes perform the systems actions, while the events that trigger such actions are characterized by the completion of an action (i.e., process) <u>or</u> the occurrence of conditions that enable the actions (or processes).

[11]Murata, 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).

[12]Coincidentally, 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.[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 λ_{ik} is the delay in firing a transition t_k which takes the Petri net from marking M_i to M_j (when several transitions enable the firing from M_i to M_j). See 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

[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

```
/*******************************************************
 * SPNP File Name: vmc4_spnp.c
 * Run this file as follows: SPNP vmc4_spnp.c
 *
 * Mon Feb 12 12:59:41 1996
 *******************************************************/
parameters(){
          iopt(IOP_PR_MARK_ORDER, VAL_CANONIC);
          iopt(IOP_PR_MERG_MARK, VAL_YES);
          iopt(IOP_PR_FULL_MARK, VAL_NO);
          iopt(IOP_PR_RSET, VAL_NO);
          iopt(IOP_PR_RGRAPH, VAL_NO);
          iopt(IOP_PR_MC, VAL_NO);
          iopt(IOP_PR_MC_ORDER, VAL_FROMTO);
          iopt(IOP_PR_PROB, VAL_NO);
          iopt(IOP_MC, VAL_CTMC);
          iopt(IOP_OK_ABSMARK, VAL_NO);
          iopt(IOP_OK_VANLOOP, VAL_NO);
          iopt(IOP_OK_TRANS_M0, VAL_YES);
          iopt(IOP_METHOD, VAL_SSSOR);
          iopt(IOP_CUMULATIVE, VAL_YES);
          iopt(IOP_SENSITIVITY, VAL_NO);
          iopt(IOP_ITERATIONS, 2000);
          iopt(IOP_DEBUG, VAL_NO);
          iopt(IOP_USENAME, VAL_NO);
          fopt(FOP_ABS_RET_M0, 0.000000);
          fopt(FOP_PRECISION, 0.000001);
}
net(){
          /* Definition of places */
          place("p0");   init ("p0",1);
          place("p1");   place("p2");   place("p3");
          place("p4");   place("p5");   place("p6");
          place("p7");   place("p8");   place("p9");
          place("p10");  place("p11");  place("p12");
          place("p13");  place("p14");  place("p15");
          place("p15");  place("p16");  place("p17");
          place("p18");  place("p19");  place("p20");
          place("p21");  place("p22");  place("p23");
          /* Definition of transitions */
          trans("dt_MuX");
          trans("sdt1");
          trans("Slot_i_a2p");
          trans("dt_i_a2p");
          trans("sdt2");
          trans("Tray_o_large");
          trans("dt_o_large");
          trans("sdt3");
          trans("Tray_o_small");
          trans("dt_o_small");
          trans("Tray_o_a1p");
          trans("dt_o_a1p");
          trans("dt1");
          trans("sdt4");
          trans("Slot_i_a1p");
          trans("dt_i_a1p");
          trans("sdt5");
          trans("Tray_o_small");
          trans("dt_o_small");
          trans("sdt6");
          trans("Slot_i_a1p");
          trans("dt_i_a1p");
          trans("Tray_o_large");
          trans("dt_o_large");
          trans("dt2");
          trans("dt3");
          trans("ft_Slot_i_a2p");
          trans("ft_Slot_i_a1p");
          /* Definition of rates */
          probval("dt_MuX",1.0);
          rateval("sdt1",0.50000000);
          rateval("Slot_i_a2p",0.1);
          probval("dt_i_a2p",1.0);
          rateval("sdt2",0.50000000);
          rateval("Tray_o_large",0.1);
          probval("dt_o_large",1.0);
          rateval("sdt3",0.50000000);
          rateval("Tray_o_small",0.1);
          probval("dt_o_small",1.0);
          rateval("Tray_o_a1p",0.1);
          probval("dt_o_a1p",1.0);
          probval("dt1",1.0);
          rateval("sdt4",0.50000000);
          rateval("Slot_i_a1p",0.1);
          probval("dt_i_a1p",1.0);
          rateval("sdt5",0.50000000);
          rateval("Tray_o_small",0.1);
          probval("dt_o_small",1.0);
          rateval("sdt6",0.50000000);
          rateval("Slot_i_a1p",0.1);
          probval("dt_i_a1p",1.0);
          rateval("Tray_o_large",0.1);
          probval("dt_o_large",1.0);
          probval("dt2",1.0);
          probval("dt3",1.0);
          rateval("ft_Slot_i_a2p",0.0055);
          rateval("ft_Slot_i_a1p",0.0045);
          /* Definition of input arcs */
          iarc("dt_MuX", "p0");
          iarc("sdt1", "p1");
          iarc("Slot_i_a2p", "p2");
          iarc("dt_i_a2p", "p3");
          iarc("sdt2", "p4");
          iarc("Tray_o_large", "p5");
          iarc("dt_o_large", "p6");
          iarc("sdt3", "p4");
          iarc("Tray_o_small", "p7");
          iarc("dt_o_small", "p8");
          iarc("Tray_o_a1p", "p9");
          iarc("dt_o_a1p", "p10");
          iarc("dt1", "p11");
          iarc("sdt4", "p1");
          iarc("Slot_i_a1p", "p12");
          iarc("dt_i_a1p", "p13");
          iarc("sdt5", "p14");
          iarc("Tray_o_small", "p15");
          iarc("dt_o_small", "p16");
          iarc("sdt6", "p14");
          iarc("Slot_i_a1p", "p17");
          iarc("dt_i_a1p", "p18");
          iarc("Tray_o_large", "p19");
          iarc("dt_o_large", "p20");
          iarc("dt2", "p21");
          iarc("dt3", "p22");
          iarc("ft_Slot_i_a2p", "p2");
          iarc("ft_Slot_i_a1p", "p12");
          /* Definition of output arcs */
          oarc("dt_MuX", "p1");
          oarc("sdt1", "p2");
          oarc("Slot_i_a2p", "p3");
          oarc("dt_i_a2p", "p4");
          oarc("sdt2", "p5");
          oarc("Tray_o_large", "p6");
          oarc("dt_o_large", "p11");
          oarc("sdt3", "p7");
          oarc("Tray_o_small", "p8");
          oarc("dt_o_small", "p9");
          oarc("Tray_o_a1p", "p10");
          oarc("dt_o_a1p", "p11");
          oarc("dt1", "p22");
          oarc("sdt4", "p12");
          oarc("Slot_i_a1p", "p13");
          oarc("dt_i_a1p", "p14");
          oarc("sdt5", "p15");
          oarc("Tray_o_small", "p16");
          oarc("dt_o_small", "p21");
          oarc("sdt6", "p17");
          oarc("Slot_i_a1p", "p18");
          oarc("dt_i_a1p", "p19");
          oarc("Tray_o_large", "p20");
          oarc("dt_o_large", "p21");
          oarc("dt2", "p22");
          oarc("dt3", "p0");
          oarc("ft_Slot_i_a2p", "p23");
          oarc("ft_Slot_i_a1p", "p23");
}
assert() {
  return(RES_NOERR);
}
ac_init() {
  fprintf(stderr,"\n<<<Run title goes here>>>");
  fprintf(stderr,"\nGenerating SRN data ...\n\n");
  pr_net_info();
}
ac_reach() {
  fprintf(stderr,"\nThe reachability graph is being ");
  fprintf(stderr,"generated ...\n\n");
  pr_rg_info();
}
/* - reward_type definitions go here ----------------*/
ac_final(){
          int i;
          time value( 0.1 );
          pr_mc_info();
          pr_std_average();
          pr_std_cum_average();
}
```

Figure 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.

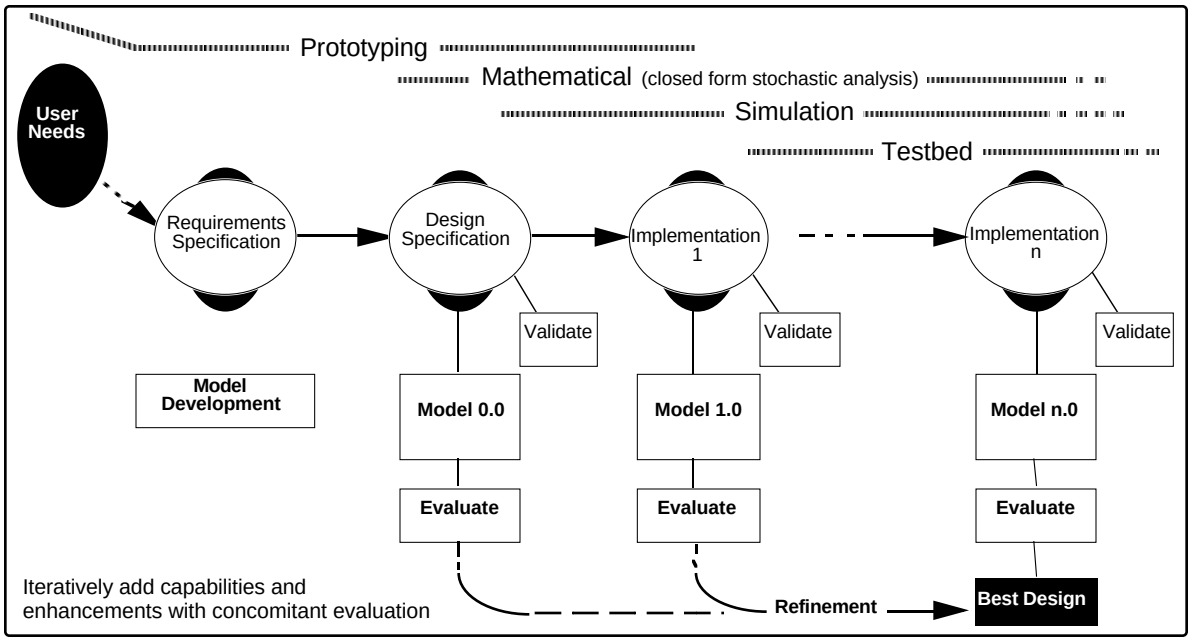


Figure 6. Refinement of system models.

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

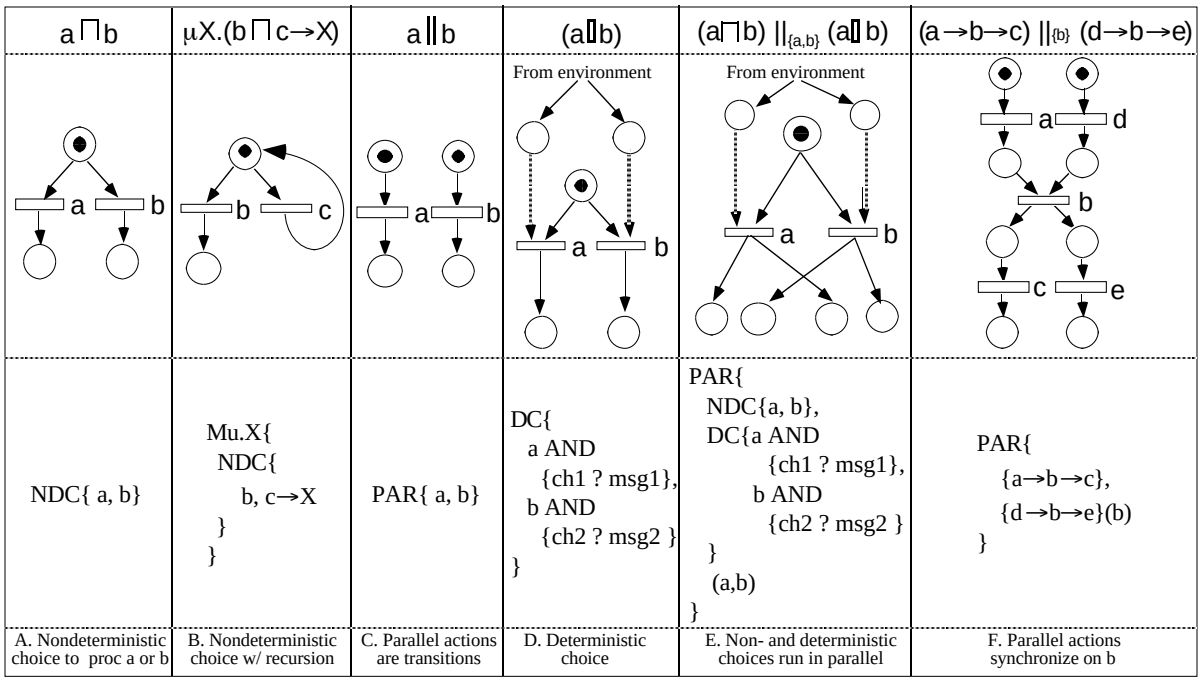


Figure 7. Example CSP to Petri net translation rules (P-CSP shown in lower portion).

(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.[14] 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.[15]

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).[16]

---

[14]Intuitively, 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.

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

[16]However, 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.[17] 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.

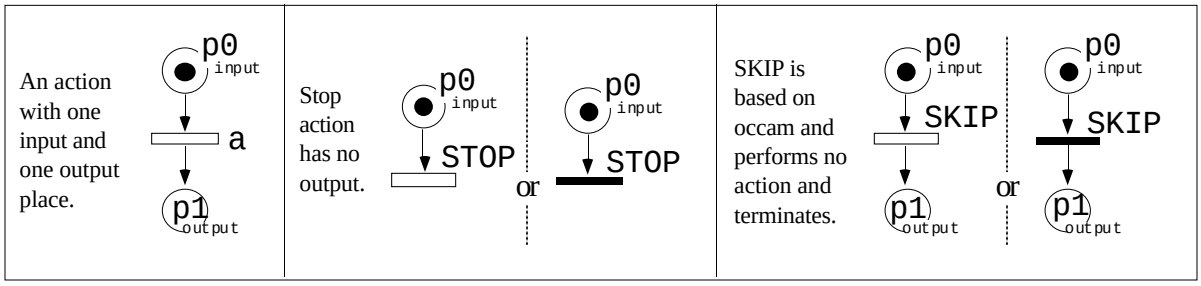


Figure 8. CSP to Petri net rules for Actions and the STOP and SKIP processes.

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

[17]Synchronization 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.

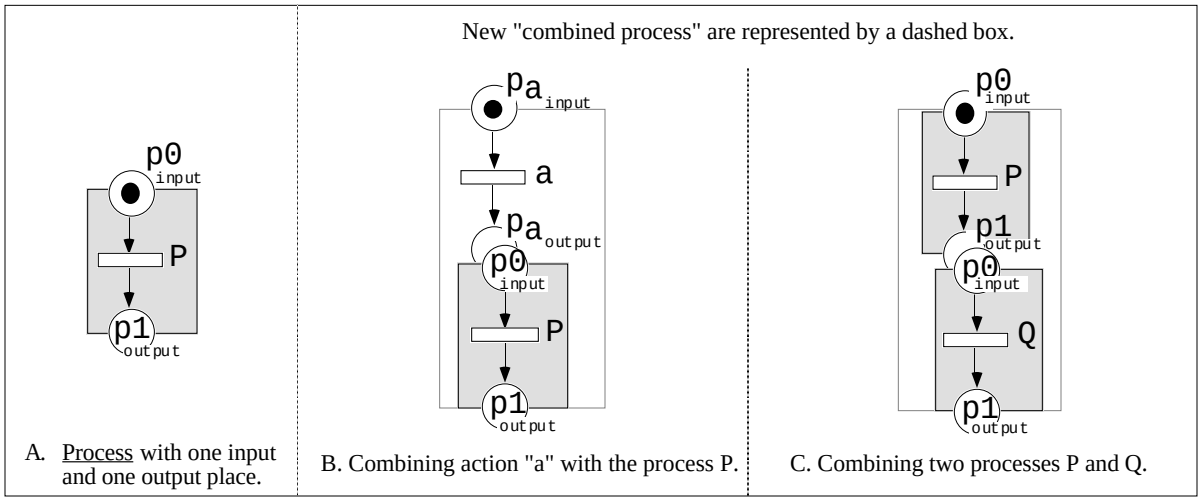
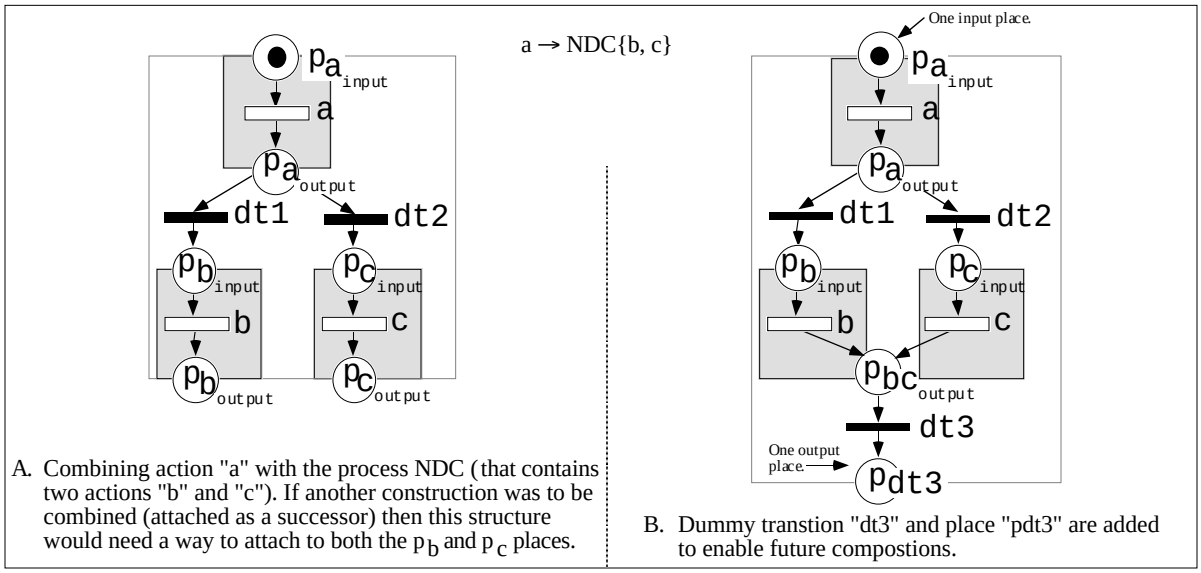


Figure 9. CSP to Petri net rules for Combining actions and processes.

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 → NDC{b, c}. When transition "a" fires, this enables a choice between [dt1 → b] and [dt2 → c]. The whole composition consumes <u>one</u> token and produces <u>one</u> 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 → NDC{b, c}] combined using "→" with [PAR{P, Q}]. The composition rewritten is: a → NDC{b, c} → 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.



A. Combining action "a" with the process NDC (that contains two actions "b" and "c"). If another construction was to be combined (attached as a successor) then this structure would need a way to attach to both the p_b and p_c places.

B. Dummy transtion "dt3" and place "pdt3" are added to enable future compostions.

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.

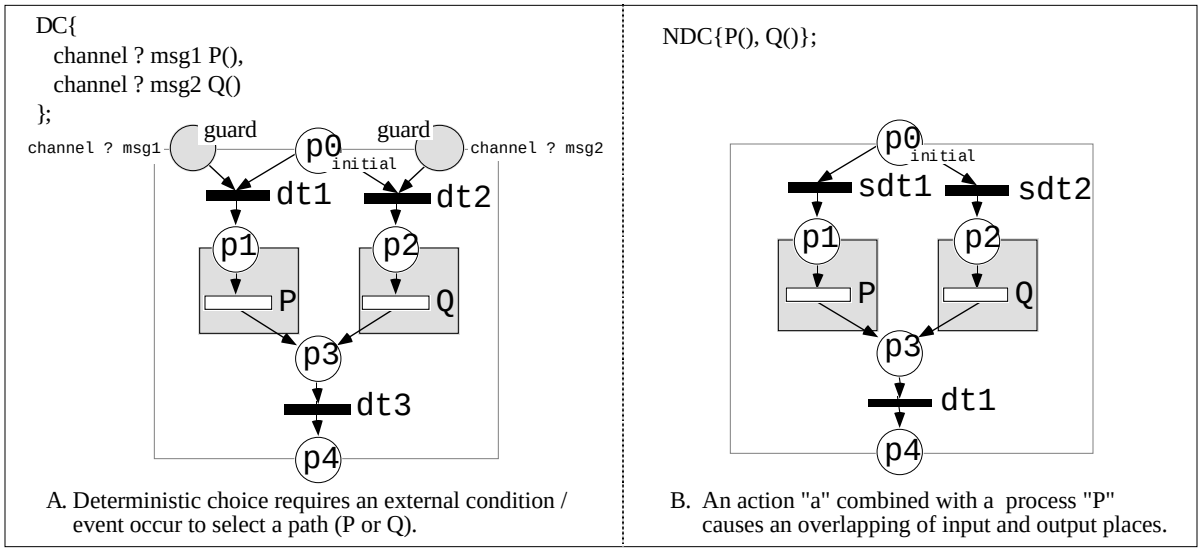


Figure 11. CSP to Petri net rules for Deterministic and nondeterministic choice.

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 form 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.

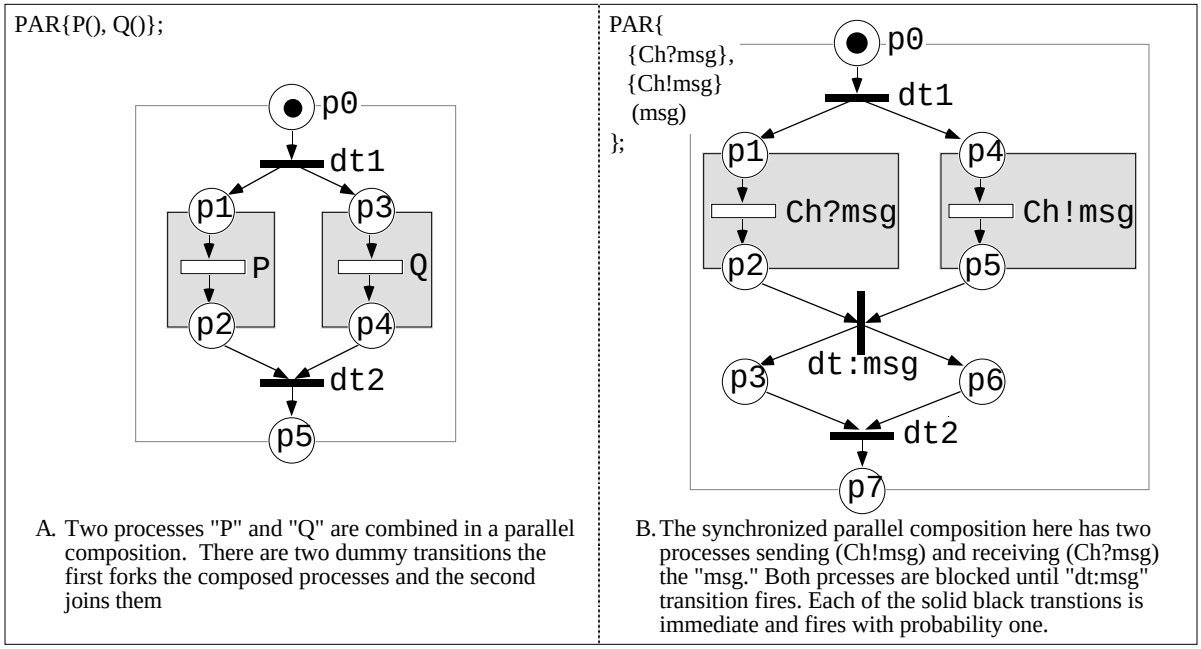


Figure 12. CSP to Petri net rules for Parallel and synchronized parallel composition.

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.

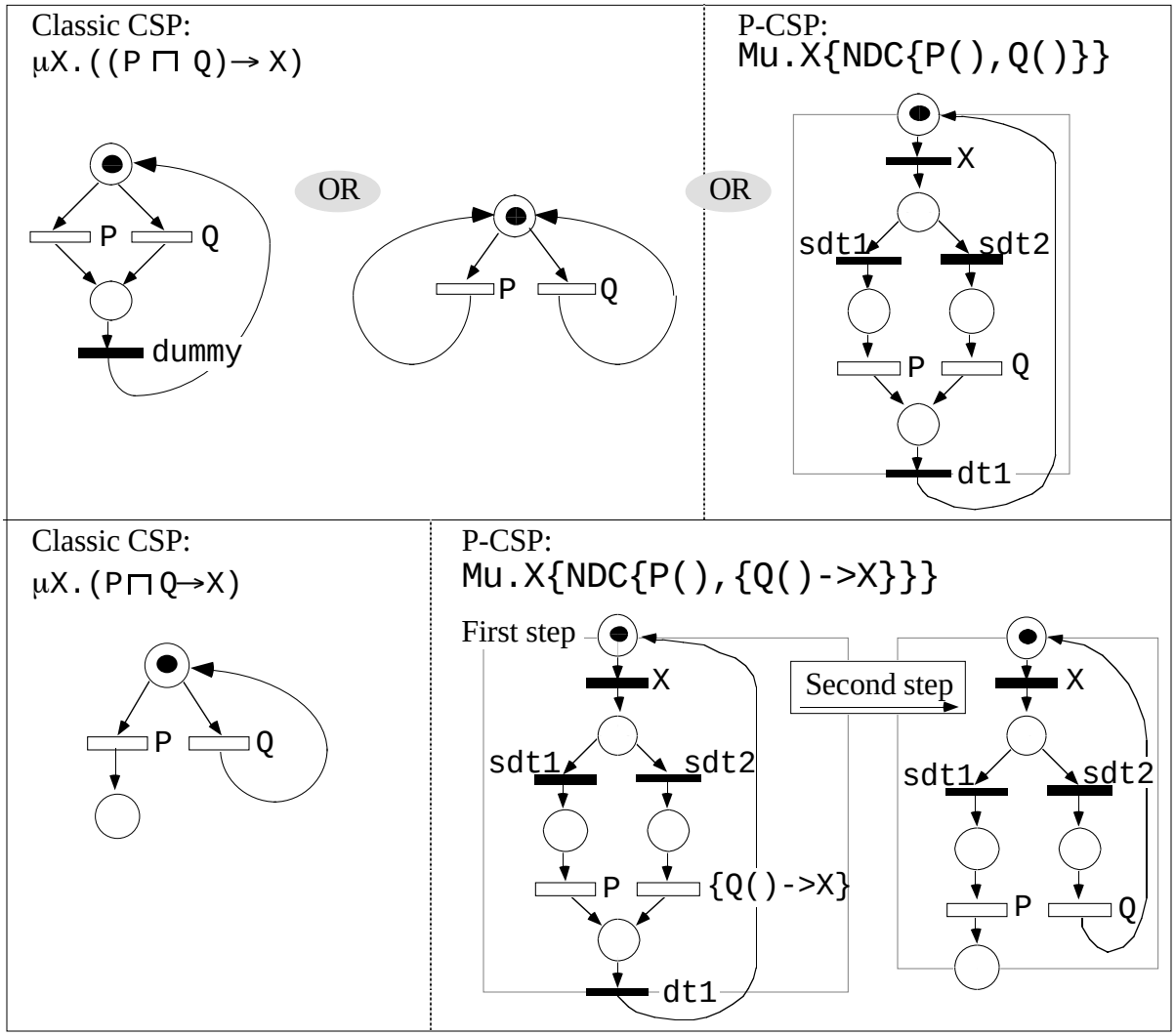


Figure 13. Comparing CSP and P-CSP to Petri net rules for recursion.

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 μX.((P ⊓ Q) → 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()→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.

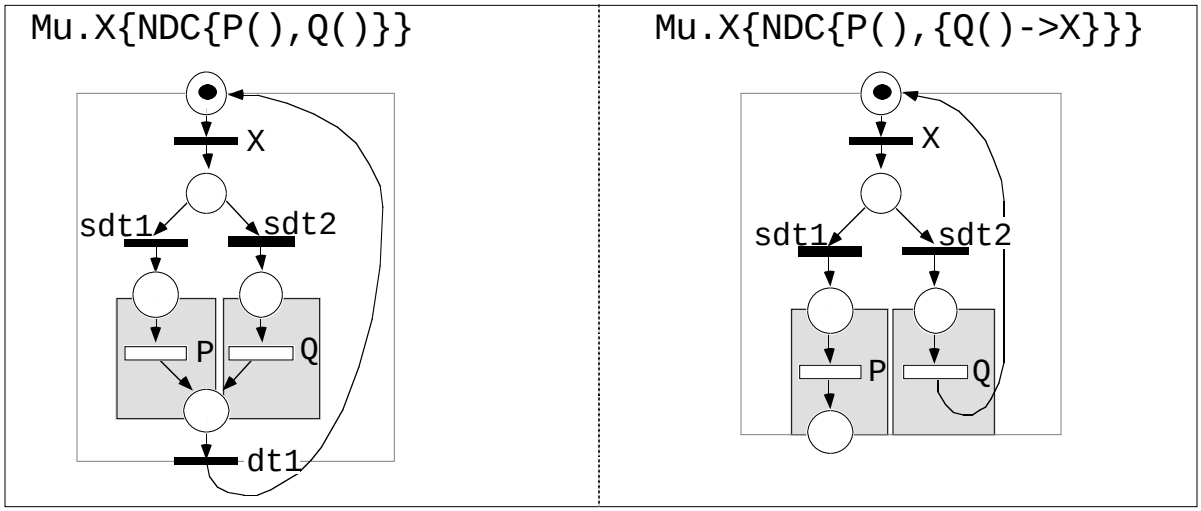


Figure 14. CSP to Petri net rules for recursion.

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.

```
VMC=

   PROCESS DispLg =
    {Tray ! large};

   PROCESS DispSm =
    {Tray ! small};

   PROCESS DispSm1p =
    SEQ{
       DispSm(),
       {Tray ! a1p}
    };

   PROCESS OnePenny =
    {Slot ? a1p}:FAIL(r= 0.0045);

   PROCESS TwoPenny =
    {Slot ? a2p}:FAIL(r= 0.0055);


Mu.X{
   NDC{
      SEQ{
         TwoPenny(),
         NDC{
            DispLg(),
            DispSm1p()
         }
      },
      SEQ{
         OnePenny(),
         NDC{
            DispSm(),
            SEQ{
               OnePenny(),
               DispLg()
            }
         }
      }
   }
}.
```

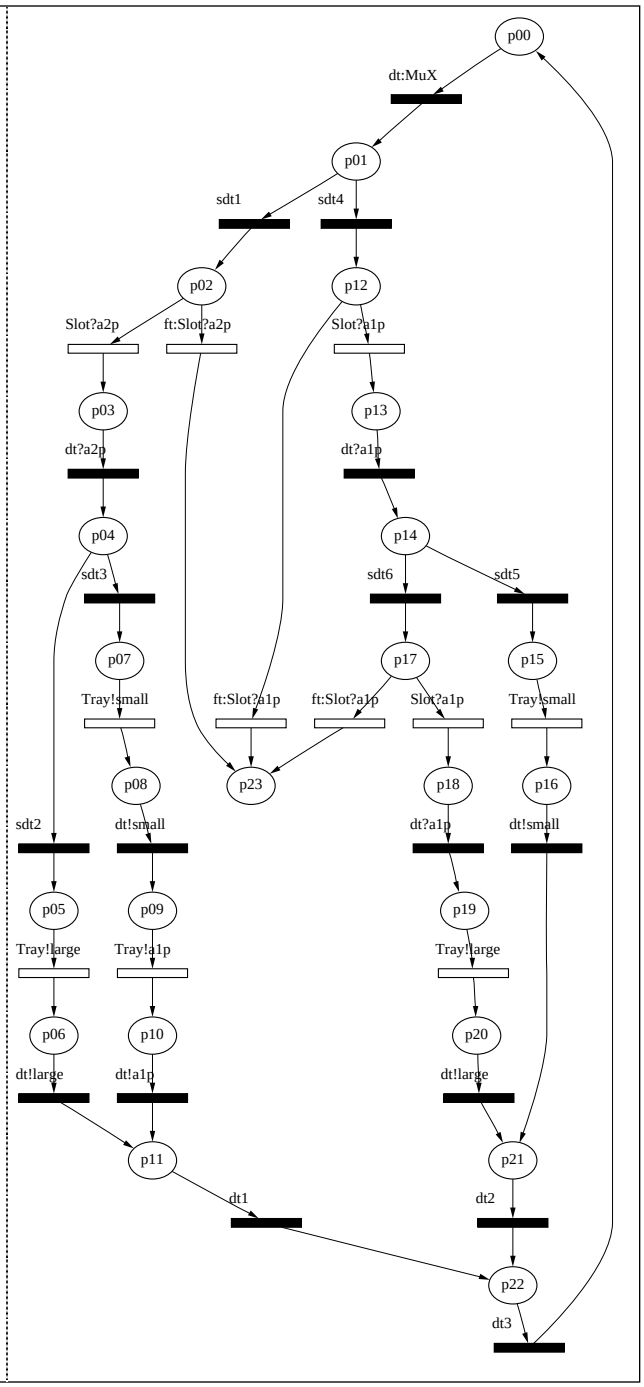


Figure 15. VMC specification for CSP to Petri net with failure annotations.

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

Lets 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)⊓
      (Tr!Sm→Tr!1p→VMC))
      ⊓
      (Slt?1p→((Tr!Sm→VMC)⊓
      (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};

Mu.X{
   NDC{
      SEQ{
         TwoPenny(),
         NDC{
            DispLg(),
            DispSm1p()
         }
      },
      SEQ{
         OnePenny(),
         NDC{
            DispSm(),
            SEQ{
               OnePenny(),
               DispLg()
            }
         }
      }
   }
}.
```

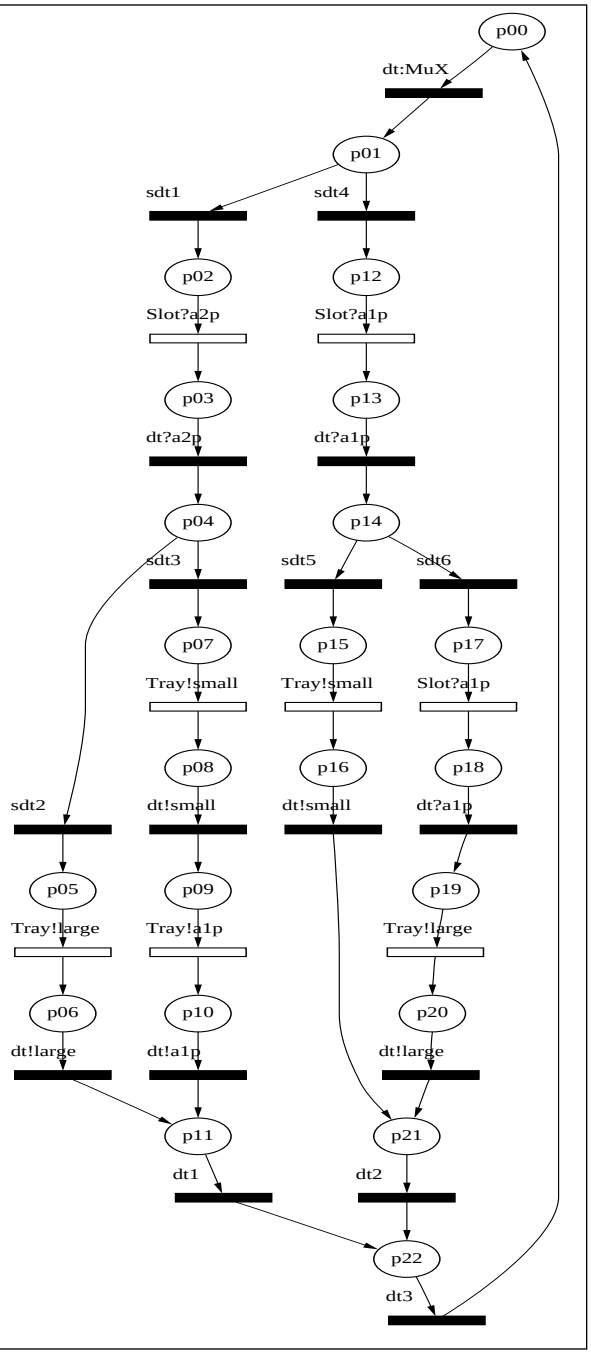


Figure 16. CSPN translation of the VMC example.

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.[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

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

[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 "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.

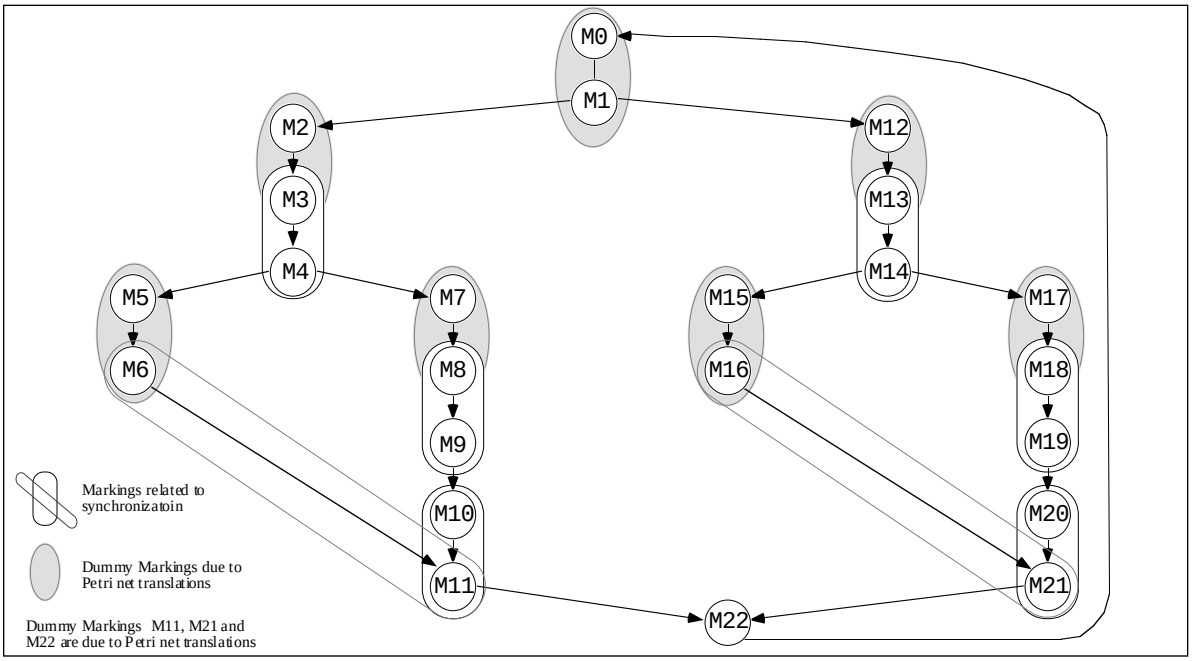


Figure 17. Structure of the feasible sequences of VMC Marking transitions.

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

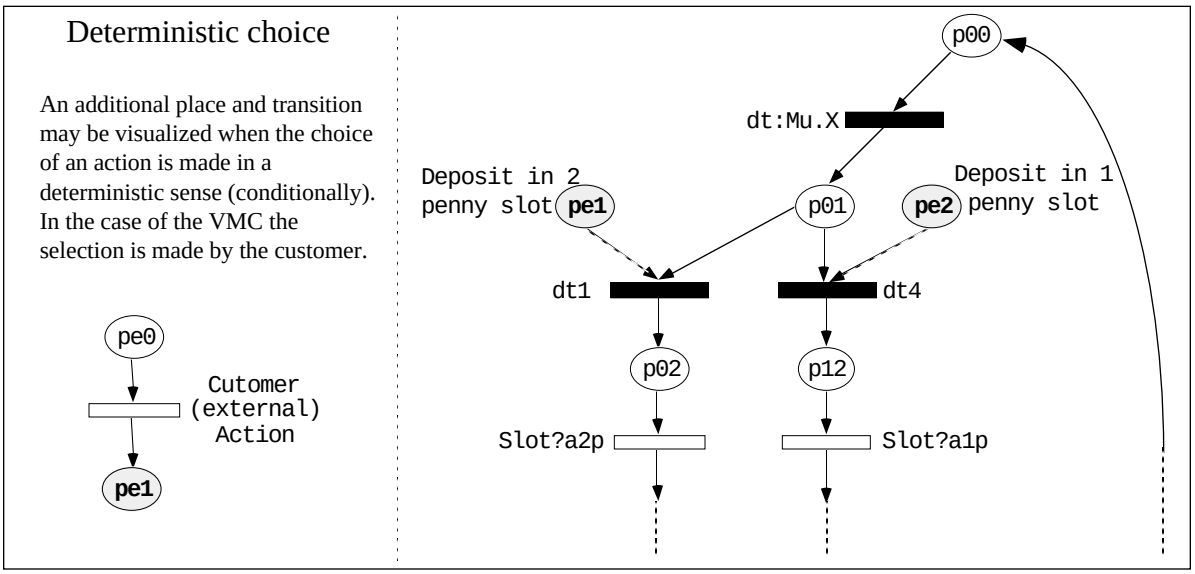


Figure 18. Partial VMC translation showing deterministic choice.

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,  ! large
2. ? a2p,  ! small,  ! a1p
3. ? a1p,  ! small
4. ? a1p,  ? a1p,    ! large
```

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

1. **M0** M1 M2 **M3** M4 M5 **M6** M11 M22
2. **M0** M1 M2 **M3** M4 M7 **M8** M9 M10 **M11** M22
3. **M0** M1 M12 **M13** M14 M15 **M16** M21 M22
4. **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).