1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Transformational Approach to Process Partitioning

by

GEORGE M. YEE

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Doctor of Philosophy

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering

Carleton University
Ottawa, Ontario
May 7, 1991

© Copyright
1991, George M. Yee

National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-70898-0

Canada

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis.

A Transformational Approach to Process Partitioning

submitted by
George M. Yee, M.Sc.
in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Chairman, Department of Systems and Computer Engineering

Thesis Supervisor

External Examiner

Carleton University

July, 1991

Abstract

The problem examined is the partitioning of a software requirements process specification to obtain a specification for a set of separate communicating subprocesses with appropriate communication interfaces. The partitioning is accomplished through the use of transformations which automatically preserve a form of behaviour correctness. Both the original specification and the partitioned specification are at a high-level of abstraction and describe software in terms of events and activities using timed Petri nets. The transformations deal with abstraction level, allowable event sequencing, construction of subprocesses, interprocess communication, control of resource access, and are applied using appropriate considerations such as modularity and performance.

Acknowledgements

I wish to thank my wife Corinne, my daughter Amy, my son Matthew, and my mother Mrs. N.F. Yee for their support and understanding throughout this endeavour.

I wish to express my gratitude to my supervisor, Professor Murray Woodside, for suggesting the thesis topic and for valuable guidance and critique throughout the research.

In addition, I wish to thank Professors Ray Buhr, Gerard Karam, and Luigi Logrippo, who were on my thesis proposal examining committee and gave me many useful comments regarding my proposal.

Finally, I am grateful for financial support from Bell-Northern Research, the Natural Sciences and Engineering Research Council of Canada, the Telecommunications Research Institute of Ontario, and Carleton University. I am also grateful for the support of NSERCC and the Telecommunications Research Institute of Ontario for other non-financial aspects of the research.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	The Problem	4
1.3	Contributions to Research	4
1.4	Document Outline	5
2	Background	6
2.1	Introduction	6
2.2	Processes, Subprocesses, and Activities	6
2.3	Descriptions of Software	7
2.3.1	Specification	7
2.3.2	Implementation and Design	8
2.3.3	Design and Implementation in the Life Cycle	9
2.4	Transformation of Specifications	9
2.5	Petri Nets	11
2.5.1	Basic Definitions	11
2.5.2	Extensions to Ordinary Petri Nets	15
2.6	Recent Use of Petri Nets in Software Modeling	17
2.6.1	Contribution	17
2.6.2	Petri Net Model	18
2.6.3	Application of the Model	22
2.6.4	Comparison of Approaches	32

2.7	Chapter Summary	34
3	Event Nets	36
3.1	Choice of Process Model	37
3.1.1	The Choice of Timed Nets	38
3.1.2	Other Petri Net Models	38
3.1.3	Provability	39
3.2	Timed Nets, Events, and Activities	39
3.2.1	Timed Nets	39
3.2.2	Events	39
3.2.3	Activities	40
3.3	Derivation of Event Nets	40
3.3.1	Definitions on Existing Components	40
3.3.2	Definitions of New Components	42
3.3.3	Formal Definition of Event Nets	43
3.3.4	Characterizing Event Nets	45
3.4	Comparison of Event Nets to Other Formal Models	48
3.4.1	Comparison to Petri Net Models in the Related Research	48
3.4.2	Comparison to LOTOS and Statecharts	50
3.5	Chapter Summary	54
4	Transformations	55
4.1	Introduction	55
4.2	Formal Descriptions of Event Nets	64
4.2.1	A Relational Notation for Event Nets	64
4.2.2	Specifying Event Net Structures with the path Relation	66
4.3	Required Properties of an Event Net Process Specification	68
4.4	A Formal Definition of Transformations	70
4.5	Formal Definitions of Non-communication Transformations	71
4.5.1	Abstraction (ABS)	71
4.5.2	Refinement (RFM)	72

4.5.3	Sequentialization (SEQ)	74
4.5.4	Subprocess Construction (SPC)	92
4.5.5	Resource Server (RSV)	94
4.6	Formal Definitions of Communication Transformations	97
4.6.1	Introduction	97
4.6.2	Send-No-Wait/Receive-Wait (SNW)	99
4.6.3	Rendezvous, Non-Recipient Acks (RNA)	99
4.6.4	Bounded FIFO Channel (BFC)	102
4.6.5	Rendezvous, Recipient Acks (RRA)	104
4.6.6	Transporter (TPT)	106
4.6.7	Bounded FIFO Buffer (BFB)	108
4.7	Chapter Summary	110
5	Behaviour Correctness and Deadlock-Freeness	112
5.1	Meaning of Behaviour Correctness	112
5.1.1	Forms of Behaviour Equivalence	112
5.1.2	Towards a Definition of Behaviour Correctness	114
5.2	Results for i-Correctness and Deadlock-Freeness	116
5.2.1	i-Correctness	116
5.2.2	Deadlock-Freeness	136
5.3	Proofs of i-Correctness and Deadlock-Freeness	139
5.3.1	Abstraction (ABS)	140
5.3.2	Refinement (RFM)	140
5.3.3	Sequentialization (SEQ)	140
5.3.4	Subprocess Construction (SPC)	142
5.3.5	Resource Server (RSV)	142
5.3.6	Bounded FIFO Channel (BFC)	143
5.3.7	Rendezvous, Recipient Activity Acks (RRA)	143
5.3.8	Transporter (TPT)	144
5.3.9	Bounded FIFO Buffer (BFB)	145

5.4	Chapter Summary	146
6	Using the Approach	148
6.1	Transforming the Process Specification	148
6.1.1	Procedure for Transformation	148
6.1.2	Order of Steps	151
6.1.3	Automation	156
6.2	Translating to Communicating Finite State Machines	158
6.3	Chapter Summary	162
7	Examples	164
7.1	Local Telephone Service	164
7.1.1	Process Specification	164
7.1.2	Transformation	165
7.1.3	Conversion to State Machines	174
7.2	Control Software for a Robotic Arm	175
7.2.1	Transformation	175
7.2.2	Experimental Verification of i-Correctness	184
7.3	Chapter Summary	190
8	Conclusions	191
8.1	Results	191
8.2	Strengths of this Approach	192
8.3	Weaknesses of this Approach	193
8.4	Operational Approach to Software Development	194
	Bibliography	195

List of Tables

2.1	Synthesis approaches using Petri nets.	35
3.1	Comparison of event nets to other Petri net models.	51
7.1	Abstracted transitions in robotic arm example.	176
7.2	More abstracted transitions in robotic arm example.	178
7.3	Partitioning in robotic arm example.	180
7.4	Application of communication transformations in robotic arm example. . .	180
7.5	Communications not failing the circular dependence conditions in robotic arm example.	183
7.6	Execution trace for verification of i-correctness.	187

List of Figures

2.1	An example Petri net structure.	12
2.2	Petri net graph for the example in Figure 2.1.	13
2.3	The example of Figure 2.2 in execution.	14
2.4	A software component in the extended model (from Yau and Caglayan [51]).	20
2.5	Tankoano's Petri net model (from Tankoano [46]).	22
2.6	Petri net interpretations of output control (from Yau and Caglayan [51]).	23
2.7	A TFG and its task tree (from Peng and Shin [37]).	25
2.8	GSPN model for SEND-RECEIVE-REPLY (from Peng and Shin [37]). . . .	26
2.9	SADT related to Petri nets (from Reisig [42]).	28
2.10	Semantics of $\langle e(\tau).c- \rangle!M \rangle$ and $\langle e(\tau).c- \rangle?M \rangle$ (from Tankoano [46]).	30
2.11	Semantics of $\langle e(\tau).c- \rangle!/M \rangle$ and $\langle e(\tau).c- \rangle?/M \rangle$ (from Tankoano [46]).	30
2.12	A partitioning rule (from Tankoano [46]).	31
2.13	A doubling rule (from Tankoano [46]).	31
2.14	Replacement by a well-formed block (from Tankoano [46]).	32
3.1	Events, activities, and predicate transitions.	44
3.2	Environmental components and interactions.	46
3.3	Interaction-connectedness.	47
4.1	A high-level view of the role required of transformations.	56
4.2	Abstraction and Refinement.	58
4.3	Non-abstractable nets, non-derivable refinements.	59

4.4	Sequentialization of alternate orders of event occurrence.	60
4.5	Subprocess Construction.	61
4.6	Inter-process communication.	62
4.7	Modeling resource access control.	63
4.8	Example path relations.	65
4.9	Event net structures by $\mathcal{G} = (\{a_1, a_4\}, \{\{a_1, a_4\}\}, \{\{a_2, a_3\} : [a_1, a_4]\})$	67
4.10	Strings, loops, and combinations.	69
4.11	Illustrations of step 3 constraints.	78
4.12	Subnet S for SEQ in First Example.	80
4.13	Applying the sequentialization algorithm to S_1 in First Example.	81
4.14	Applying the sequentialization algorithm to S_1 in First Example - Completed.	84
4.15	Subnet S' by SEQ in First Example.	87
4.16	More examples of SEQ.	88
4.17	Yet another example of SEQ.	89
4.18	Using a token to carry data through new intermediate transitions.	90
4.19	Example subprocess constructed from S in First Example.	93
4.20	Example subprocess from S containing a loop.	93
4.21	The resource server transformation.	96
4.22	“Send-No-Wait/Receive-Wait” communication style.	99
4.23	“Rendezvous - Non-Recipient Acks” communication style.	100
4.24	“Reverse Rendezvous” communication style.	101
4.25	“Bounded FIFO Channel” transformation.	102
4.26	Less waiting due to alternative choices T2 and T3.	103
4.27	“ k -way BFC” communication.	104
4.28	“Rendezvous - Recipient Acks” transformation.	105
4.29	“ k -way Rendezvous - Recipient Acks”.	106
4.30	“Transporter” transformation.	107
4.31	“Bounded FIFO Buffer” transformation.	109
5.1	A conceptual view of E , S , and $\bar{S}$; b is a boundary transition.	117

5.2	Example $R_E, R_S, R_{\bar{S}}$ where $R_E = R_S \cup R_{\bar{S}}$. The boundary transitions of S and $\bar{S}$ are a_1, a_3 , and a_4	118
5.3	Subnets $\bar{S}$ and $\bar{S}'$ are different for the "Rendezvous - Receiver Acks" transformation.	120
5.4	A conceptual view of $E' = T(E)$	120
5.5	Construction of E' by replacing S with S' - note $E' \subseteq E$ by Theorem 5.1.	125
5.6	α and $\vdash_{\mu'}$ relations.	127
5.7	$C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1 \vdash_{\mu'} C_3$ (circular waiting) resulting in deadlock.	127
5.8	Inadvertent introduction of a wait loop by the RRA transformation, where C_1 is the sender and C_3 the receiver.	129
5.9	Legitimate application of RRA to E , disallowed by the circular dependence conditions, but allowed by the circular waiting conditions.	133
6.1	Procedure for transformation described by an FSM.	151
6.2	Example PS.	152
6.3	Example sequentialization.	153
6.4	Example adjustment of detail.	153
6.5	Example subprocess construction.	154
6.6	Example installation of communication style.	154
6.7	Example re-doing of transformations.	156
6.8	Translation of a subprocess event net into a communicating FSM.	161
6.9	Translation of an example proto-design into communicating FSMs.	163
7.1	PS for local telephone service.	166
7.2	Switch for local telephone service.	167
7.3	Refinements for line connection mechanism.	168
7.4	Partitioning.	171
7.5	Resource access control.	172
7.6	Re-partitioning.	173
7.7	Communicating finite state machines corresponding to Figure 7.6.	174
7.8	PS for control of robotic arm (derived from Grigg [18]).	176

7.9	Net resulting from transformation steps 1 and 2.	177
7.10	Net resulting from transformation steps 1 to 4.	179
7.11	Net resulting from transformation steps 1 to 5.	181
7.12	Use of dependence graphs.	182
7.13	Proto-design for control of robotic arm.	185

Index of Definitions

Terminology	Definition	Page
Adjacency relation	4.1	64
boundary event set	5.2	116
boundary input event set	5.2	116
boundary output event set	5.2	116
boundary transition	5.2	116
circular dependence conditions		131
circular waiting conditions		128
continuous event net	3.8	48
deadlock-freeness	3.9	48
dependence relation α	5.4	126
ending transition	4.5	68
environmental component	3.5	45
event net generator	4.4	66
event net structure	3.1	43
event net	3.4	45
event subnet structure	3.2	44
event subnet	3.4	45
execution trace	3.7	47
finite state machine (FSM)	6.1	159

Terminology	Definition	Page
fired-first transition	4.5	68
input interaction	3.5	45
input interaction set	3.5	45
interaction	3.5	45
interaction set	3.5	45
interaction transition	3.5	45
interaction-connected	3.6	46
intermediate transitions	4.3	65
i-correctness	5.1	115
join transition		75
loop	4.5	68
marked event net structure	3.3	44
marked event subnet structure	3.3	44
marking	3.3	44
output interaction	3.5	45
output interaction set	3.5	45
parallel execution trace	3.7	47
parallel event net	3.8	48
parallel paths	4.6	68
Path relation	4.3	65
path-structured		68
potentially fireable	3.9	48

Terminology	Definition	Page
preservation of boundary event sets	5.3	117
proto-design		36
relational definition	4.2	64
sequential event net	3.8	48
sequential execution trace	3.7	47
single-choiced		68
software transition	3.5	45
starting transition	4.5	68
string	4.5	68
transformation	4.7	70
translation to communicating FSMs	6.2	160
wait loop	5.5	127
waits for in μ relation $\vdash_\mu$	5.4	126

Chapter 1

Introduction

1.1 Motivation

Advances in computing hardware have been developing rapidly, as evidenced by processors with ever increasing power, in tandem with memory chips of ever greater capacity. Computing software, on the other hand, has failed to keep pace with this fast progress (Cox [15]). In fact, the “software crisis” is becoming more and more exacerbated as the increasing complexity of hardware systems dictate increasing complexity in the software needed to control the hardware, e.g., avionics systems, and distributed systems in general. Software systems consisting of up to a million lines of code are now commonplace. Newly envisaged systems easily call for 10 million or more lines of code (Hatley and Pirbhai [20]). Software design methodology is only now beginning to make some headway in solving the “crisis”, but there is a shortage of good, effective software design techniques, especially for concurrent software, real-time software, and distributed systems (Brooks [9], Stankovic [44]).

Among the recent ideas in software development that have emerged to deal with this shortage is the *operational approach* to software development (Zave [52]). This approach subjects a system’s *operational*¹ specification to transformations in order to derive a design. The transformations preserve the system’s external behaviour, but rearrange the

¹The term “operational” means “executable” in some limited sense. The importance of operability is the ability to experiment and test.

internal components responsible for that behaviour. Such an approach is effective in the areas mentioned above, largely for two reasons: i) the approach is operational, allowing easier validation of requirements and other needed experimentation, and ii) the use of transformations that preserve external behaviour to obtain a design that is automatically correct with respect to the specification, is far less prone to error in the face of increased complexity than non-transform techniques that rely on verification for correctness.

Much of software research has been concerned with functionality rather than performance. Performance aspects have often been ignored. As a result, current design approaches fail to deal adequately with performance – for example, Hatley/Pirbhai [20] and Ward/Mellor [49] allow the specification of necessary response time, but do not deal with how to actually achieve a design giving the required response.

Traditionally, software performance has been handled by tuning after implementation, sometimes by extensive trial and error. Now, there are at least two reasons for addressing performance at the design stage *prior to implementation*. These reasons are i) increasing *complexity* in real time software – it is becoming extremely difficult to modify implemented real time software, and ii) economics – it is becoming too expensive to tune for performance after implementation.

This research considers the definition of transformations that may be used as first steps in a formal approach to concurrent software design that includes performance as a design concern. The transformations enable performance improvement in the design phase by impacting performance through their selection and use. However, performance analysis (e.g., for throughput, response time under actual system load) is outside the scope of this work, although the Petri net-based process model used in this work is often applied to performance analysis.

The transformations partition a model of a process, which represents software requirements, into models of sequential subprocesses which run concurrently. It is then only an additional step to obtain a design by assigning the sequential subprocesses to program structures such as procedures and modules. Since this additional step is needed to obtain a design, the description of software resulting from the transformations is called a *proto-design*. As in the operational approach, the transformations preserve the system's (the process's) external behaviour (in some sense that will be defined) but add structure by partitioning the specification into components (the subprocesses) responsible for that behaviour. The use of a particular transformation is guided by design concerns or

constraints, such as performance, modularity, and cohesion. Processes and transformations are expressed in terms of input/output events, using an executable FDT (Formal Description Technique) or formal model.

For example, consider the derivation of software for a computing system consisting of several parallel processors. Suppose software requirements are available in the form of a process specification expressed in terms of a formal model. Such a process specification could describe internal concurrency and require mutually exclusive access to shared resources. Applying the transformations to this specification would entail i) partitioning it into sequential subprocesses that run concurrently, using design concerns as constraints (e.g., required response time, modularity), ii) establishing interprocess communication between the subprocesses, choosing appropriate communication paradigms based again on design concerns, and iii) establishing suitable resource access control (e.g., mutual exclusion) for the subprocesses, as required by the original process specification. The resulting proto-design would be behaviourally correct with respect to the process specification, since the transformations preserve correctness in terms of external behaviour. However, although the proto-design may satisfy behavioural requirements, it may fail to satisfy response time requirements. In this case, steps i), ii), and iii) would be repeated until a proto-design is obtained that does have the required response, or no such proto-design can be found, in which case it may be necessary to relax the response requirements of the process specification, if possible. Once a proto-design satisfying all design concerns is obtained, the derivation of program structures such as procedures and modules may begin. For this example, a design can be obtained by simply assigning a subprocess to each processor, assuming there are at least as many processors as subprocesses and that program structures other than subprocesses are not required.

This work represents initial steps toward a new formal design approach for concurrent software that automatically satisfies behavioural requirements and provides an iterative procedure to satisfy design concerns such as modularity, cohesion, and performance. (The approach will turn out to be operational as well, due to the formal model selected for expressing requirements.) The work is only a step away from the identification of program structures, which would qualify this approach as a complete design technique. Even without this step the results can be used where program structures other than subprocesses (which are like Ada tasks) are not required, as in the above example. In the context of a design technique for concurrent software, this work is clearly applicable in such ar-

as telecommunications, avionics, and signal processing, where concurrent embedded and distributed real time systems abound. For example, this work is applicable to communication protocols, in which the above process specification corresponds to a protocol service specification and the subprocesses resulting from partitioning correspond to the protocol entities. In addition, with minor extensions, one can visualize this work finding application outside the software area: in identifying concurrent processors from a computing hardware requirements specification, or in identifying individual workers from an organizational human resources requirements specification, for example.

1.2 The Problem

This research proposed to solve the following problem:

Given a formal software requirements specification of a process, expressed using a suitable FDT, how can this specification be directly transformed into a proto-design (i.e., a description in terms of concurrently running sequential subprocesses) that is behaviourally correct with respect to the requirements specification and satisfies all design concerns, including performance?

A specification that is directly transformed is one that is restructured and augmented in terms of the elements of the FDT used. Restructuring may affect performance in that it may change previous structuring decisions which determine performance. Such decisions may concern, for example, execution order and inter-process communication. Transformations, therefore, may be used to satisfy performance requirements, in that they affect performance through restructuring.

1.3 Contributions to Research

The main contribution of this work is the *derivation of a set of correctness-preserving transformations that may be used to partition a formal process specification to obtain a proto-design.*

The following contributions enable the main contribution or allow it to be more easily applied:

- adaptation of a kind of *Petri nets* (Peterson [38], Murata [32]) called *timed nets* (Merlin and Farber [29]) for use in expressing software requirements (Section 3.3);
- definition of a formal concept of behaviour correctness between process specifications (a proto-design can be considered as simply a more structured process specification relative to the original process specification) appropriate for showing that the above transformations preserve correctness (Section 5.1.2);
- definition of a transformation procedure for applying the above transformations to a given process specification², to obtain a proto-design that takes account of design concerns, including performance and deadlock-freeness (Section 6.1);
- translation of the proto-design into a set of communicating finite state machines (Section 6.2);
- demonstration of the transformation procedure on real life examples (Chapter 7).

1.4 Document Outline

Chapter 2 introduces some terminology used in this work, gives background information on Petri nets, and describes related research. Chapter 3 defines the timed net model used in this research, and compares it to Petri net models used in related research as well as to the non-Petri net but well-known FDTs LOTOS (ISO [25]) and Statecharts (Harel [19]). Chapter 4 presents the transformations with illustrations, formal definitions, and discussion. Chapter 5 defines a form of behaviour correctness and shows that the transformations do indeed preserve this form of correctness. Chapter 6 defines a procedure for transforming a given process specification into a proto-design. Chapter 7 demonstrates the transformation procedure on two example problems, namely a local telephone service and control software for a robotic arm. Chapter 8 gives conclusions.

²The terms "process specification" and "requirements specification" are used interchangeably.

Chapter 2

Background

2.1 Introduction

This chapter provides background information needed to set this work in proper context and is organized into three parts. The first part explains the use of the terms *process*, *subprocess*, *activity*, *specification*, *design*, *implementation*, and *transformation*. This part also gives an overview of the use of transformations in design. The second part presents a summary of Petri net theory (Peterson [38], Murata [32]), upon which the formal description technique used in this work is based. The third part describes the use of Petri nets in other related research and compares the approach taken in this research to the approaches taken in the related research.

2.2 Processes, Subprocesses, and Activities

For this work, a *process* is an encapsulation of computation, which may be sequential, parallel, non-deterministic (i.e., at some point in the computation the ensuing computation can proceed along alternative paths and the choice of which path is non-deterministic), or any combination of sequential, parallel, and non-deterministic. A *subprocess* is contained within a larger process and is itself a process in which the computation can be sequential, non-deterministic, or both sequential and non-deterministic, but not parallel. Note that this is a particular definition of subprocess for this work. The general definition would identify a subprocess simply as a process contained within a larger process. An *activity* is a unit of computation contained within a subprocess and is identified with one particular function, requiring specific inputs and producing specific outputs (e.g., the computation

of x^2 is an activity requiring input x and producing output x^2). A process, unlike an activity, can be identified with more than one function (e.g., the computation of $x^2 + y^2$ is a process with three different functions: the computation of x^2 , the computation of y^2 , and the computation of $x^2 + y^2$). It is therefore natural to express a process in terms of activities, in which each activity corresponds to a particular function. There is thus a hierarchy of decomposition in which a process is decomposed into subprocesses, and a subprocess is decomposed into activities. Further, just as functions can be decomposed into component functions, an activity can be decomposed into component activities. This decomposition does not violate the condition that an activity can be identified with only one function. For example, by assigning $z^2 = x^2 + y^2$, an activity can be identified with the computation of z^2 and yet this activity can be decomposed into activities for x^2 , y^2 , and $x^2 + y^2$. The decomposition of an activity into component activities is expressed as *refinement*, which is a transformation introduced in Chapter 4.

2.3 Descriptions of Software

2.3.1 Specification

The term *specification* is used both in the absolute sense and in the relative sense. When used in the absolute sense, *specification* refers to the requirements capture phase of the software life cycle (Pressman [40]). When used in the relative sense, *specification* usually means a description of software relative to another description such that the two descriptions are related in that one is derived from the other by adding or removing detail (Brinksma et al. [8]).

The objective of specification, in the absolute sense, is to document user requirements for a software system either formally or informally, as the basis for software development. A specification describes system inputs, system outputs, required functionality, and the expected performance. A specification should be concerned only with *what* the system should do, not with *how* the system should do it (Cohen et al. [14]). A specification, therefore, says nothing about the internal arrangement or organization of system components (e.g., data structures, tasks) (Visser et al. [48]), which is concerned with the *how*. As the basis for software development, a specification should be precise, which usually means that it should be formal. An informal specification is subject to the vagueness and imprecisions of the informal description (e.g., natural language). However, a specification

may begin informally (e.g., in natural language) and then be re-expressed formally using a suitable formal model. Recently, the term *operational* specification has been used in connection with the *operational* approach to software development (Zave [52]). An operational specification is formal and can be executed. The advantages of an operational specification include the opportunity for the customer requesting the system to verify the system's operation through computer execution.

This work uses specification in both the absolute and relative sense; the intended meaning in each case will be clear from the context.

2.3.2 Implementation and Design

Brinksma et al. [8] offer four meanings for *implementation*:

1. "*implementation* as a synonym of the *real/physical system*";
2. "*implementation* as a (deterministic) *reduction* of a given specification; in this context specification and implementation are relative notions in a hierarchy of system descriptions, where one description is viewed as an implementation of another description, the specification, if the former results from the latter by resolving choices that were left open in that specification";
3. "*implementation* as an *extension* of a given specification; again specification and implementation are to be regarded as relative notions in a hierarchy; an implementation adds information that is consistent with the original specification; unlike refinement however, (proper) extension involves additional information about the observable behaviour of the system described; this notion is particularly relevant to partial specifications";
4. "*implementation* as a *refinement* of a given specification; also working with relative notions of implementation and specification, in this case an implementation provides more detail on the subdivision of the specification itself into smaller components; both descriptions are *extensionally* equivalent, i.e., their observable behaviour cannot be distinguished; the *intension* of both descriptions is not the same, as the implementation gives more details about the internal structure of the object of specification; for processes, this view of implementation was first introduced by Milner [30]".

In this research, the term *design* means a combination of 2 and 4 above, in which only the topmost description in the hierarchy is a specification (absolute sense). A design is thus both a reduction and a refinement of the specification. However, whereas concurrent software design usually extends the identification of subprocesses to include tasks, procedures, packages, modules, and other program structures, *this work considers only the identification of subprocesses*. For this reason, the term *proto-design* is used for the partitioned process specification, in place of *design*. The term *implementation* means item 1 above, i.e., an implementation consists of code on the target machine.

2.3.3 Design and Implementation in the Life Cycle

The meaning of design and implementation given above is consistent with their meaning in the software life cycle. Design is that part of the life cycle concerned with the organization of internal system components (Pressman [40], Vissers et al. [48]). There may be several possible organizations, all giving the same required functionalities. However, each separate organization will exhibit a different set of performance measurements. The objective of design is to find the proper arrangement of internal components that will satisfy both functionality (correctness) and performance requirements at the same time. Note that design is not implementation (Pressman [40]). Implementation is the mapping of software components to actual code in some programming language. Once the design of a software system has been decided, implementation is largely a mechanical process that, depending on the FDT used for the design (if an FDT is used), may be automated. Alternatively, again depending on the FDT used, the design itself may be executable. Automated implementation and executable design are forms of rapid-prototyping (Taylor and Standish [47], Vissers et al. [48]).

2.4 Transformation of Specifications

In the literature, transformational techniques have largely been applied at the programming level to derive automatically correct programs (Partsch and Steinbrüggen [36]). Such techniques have been applied to Petri nets to reduce complexity (Berthelot [2, 3]), i.e., complex subnets are transformed into behaviourally equivalent simpler ones. The use of transformational techniques at the specification level has been infrequent.

Tankoano [46, 45] applied transformational techniques to derive control software for

manufacturing systems. Tankoano's work is described below, along with other related research.

Zave [52] described the role of transformations when applied to an operational specification: "The specification is subjected to transformations that preserve its external behaviour, but alter or augment the mechanisms by which that behaviour is produced, so as to yield an implementation-oriented specification of the same system."

Zave described different types of transformations: "One type of transformation changes the modifiable and comprehensible mechanisms of the operational specification to equivalent ones lying at different points in the trade-off space balancing performance (timing, reliability) and the various implementation resources (processors, memory, communication channels, etc.) involved. Other transformations are needed so that specification structures can be mapped straightforwardly and efficiently onto a particular configuration of implementation resources. These transformations may introduce explicit representations of implementation resources, or resource allocation mechanisms, that were not present in the original specification." This description applies precisely to the use of transformations in this research, in which a design lies in Zave's trade-off space, and the introduction of mechanisms for resource access control and interprocess communication corresponds to Zave's representations of implementation resources. As examples of software development methods incorporating transformational techniques, Zave cited the Jackson System Development Method (JSD) and the PAISley Project.

Cohen et al. [14] defined transformations and described their features, uses, and support by automated tools. They (pp. 11-12, page references in the remainder of this section refer to Cohen et al. [14]) described a transformation as a manipulation of a model of a system in one form to produce another form. As examples of transformations, Cohen et al. cited programming language compilation and the generation of circuits from logic diagrams. Transformations must be "property-preserving" and can preserve only those properties that are not abstracted by the model. Transformations can be done by tools which "perform a variety of symbolic manipulations on system descriptions" (p. 99). Such tools must provide support for verification of manual transformation steps (p. 102).

Cohen et al. described the system development process in terms of transformations (p. 101): "The development process can be viewed as a sequence of transformations from some formal specification, each of which removes abstraction until an executable design with acceptable characteristics is produced. Some steps in this process can be automated,

or at least provided with considerable computational assistance.” This research holds the same view.

Cohen et al. also described transformations as the vehicle for going from one *style* of specification to another (p. 122-123). A particular style might be more appropriate for presentation to the customer while a different style would be more suitable for design. Visser et al. gave a good presentation of different specification styles in [48]. Cohen et al. went on to say that this traversal of specification styles raises the problem of determining behavioural equivalence between the different styles, which are meant to be representations of the same system. Ensuring correctness is a major goal of this research. A prerequisite for obtaining behaviour equivalence is mathematical formality, i.e., the use of a FDT for specification. As examples of style transformations, Cohen et al. cited the relationship between processes and traces in CSP (Hoare [23]), the transformation from recursive to iterative style (Burstall and Darlington [11]), and the Wide Spectrum Language of the CIP project (Brass et al. [6]), which was actually defined in terms of transformations.

Finally, Cohen et al. stated that “transformations tend to take a high-level (i.e., more abstract) specification into a lower level one, usually for the purpose of producing a representation which can be executed by a suitable processor” (p. 123).

2.5 Petri Nets

Petri nets, developed from the work of Carl Adam Petri in his doctoral thesis “Kommunikation mit Automaten” (communication with automata) in 1962 [39]. In this thesis, Petri presented the basis for a theory of communication between asynchronous components of a computer system.

Petri net theory is summarized here in terms of basic definitions and extensions (i.e., extended Petri net models). More detailed information on Petri nets may be found in Peterson [38] or Murata [32].

2.5.1 Basic Definitions

Petri nets have an algebraic representation known as a *Petri net structure* and a graphical representation known as a *Petri net graph*.

A *Petri net structure* C is a 4-tuple $C = (P, T, I, O)$ where $P = \{p_1, \dots, p_n\}$ is a finite set of *places*, $n \geq 0$, $T = \{t_1, \dots, t_m\}$ is a finite set of *transitions*, $m \geq 0$, $I : T \rightarrow P^\infty$ and

$O : T \rightarrow P^\infty$ are the *input* and *output* mappings respectively, from transitions to bags of places. (A bag of objects differs from a set of objects in that the bag can have more than one of each kind of object. Thus, this definition means that there can be more than one mapping from a transition to a place.)

A place p_i is an *input place* of transition t_j if $p_i \in I(t_j)$ and an *output place* if $p_i \in O(t_j)$. The *multiplicity* of input (output) place p_i for transition t_j is the number of occurrences of the place in the input (output) bag of the transition, $\#(p_i, I(t_j))$ (for multiplicity of output place, $\#(p_i, O(t_j))$).

Figure 2.1 gives an example Petri net structure.

$$\begin{aligned} C = (P, T, I, O), \quad P &= \{p_1, p_2, p_3, p_4\}, \quad T = \{t_1, t_2\} \\ I(t_1) &= \{p_1, p_4\}, \quad O(t_1) = \{p_2, p_3\} \\ I(t_2) &= \{p_2, p_3\}, \quad O(t_2) = \{p_4, p_1\} \end{aligned}$$

Figure 2.1: An example Petri net structure.

A *Petri net graph* G is a bipartite directed multigraph, $G = (V, A)$, where $V = \{v_1, \dots, v_s\}$ is a set of vertices and $A = \{a_1, \dots, a_r\}$ is a bag of directed arcs, $a_i = (v_j, v_k)$, with $v_j, v_k \in V$. In addition, $V = P \cup T$, $P \cap T = \emptyset$, and for each $a_i = (v_j, v_k) \in A$, either $v_j \in P$ and $v_k \in T$ (input arc a_i) or $v_j \in T$ and $v_k \in P$ (output arc a_i).

In a Petri net graph, a place p_i is represented by a circle $\bigcirc$, a transition t_i is shown as a bar $|$, and a directed arc a_i is represented by $\rightarrow$. An arc directed from a place to a transition defines the place to be an *input place* of the transition. Multiple inputs to a transition are depicted by multiple arcs from input places to the transition. An arc directed from a transition to a place defines the place to be an *output place* of the transition. As for input, multiple outputs are depicted by multiple arcs.

The Petri net graph of the above example is given in Figure 2.2:

A *marking* μ of a Petri net $C = (P, T, I, O)$ is a function $\mu : P \rightarrow N$, where N is the set of nonnegative integers. A *marked Petri net* $M = (C, \mu)$ (sometimes written as $M = (P, T, I, O, \mu)$) is a Petri net structure together with a marking μ . In other words, a marking μ is an allocation of tokens to the places. Tokens allow for execution of the Petri

Figure 2.2: Petri net graph for the example in Figure 2.1.

net and are portrayed in the Petri net graph as small dots • inside the places.

A Petri net executes by *firing* transitions. A transition can only fire if it is enabled. A transition $t_j \in T$ of a marked Petri net $M = (P, T, I, O, \mu)$ is *enabled* if for all $p_i \in P$,

$$\mu(p_i) \geq \#(p_i, I(t_j)).$$

In other words, a transition is *enabled* if each of its input places has at least as many tokens as it has arcs going to the transition.

The tokens enabling a transition are the transition's *enabling tokens*. The number of enabling tokens for a transition t_j is $\#(p_i, I(t_j))$. When a transition fires, it removes all its enabling tokens and deposits tokens in its output places – one token per output arc per output place. Only enabled transitions may fire. Firing an enabled transition t_j results in a new marking μ' defined as

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

If more than one transition is enabled, the choice of which transition to fire is made non-deterministically. Sometimes firing one will disable others that were previously enabled. The Petri net will continue to execute as long as there are enabled transitions. When such is not the case, the execution *halts*.

Figure 2.3(a) shows the above example with an initial marking μ . In this figure, only transition t_1 is enabled. Figure 2.3(b) shows marking μ' after t_1 fires. In this marking, only transition t_2 is enabled. Figure 2.3(c) shows the next marking μ'' after t_2 has fired. Here only t_1 is enabled but note that $\mu'' \neq \mu$ due to place p_2 , which will accumulate tokens without bound as the Petri net executes. Note also that the execution will never halt.

Figure 2.3: The example of Figure 2.2 in execution.

A Petri net has a *state* corresponding to its marking. A Petri net's *state space* is the set of all markings, i.e., N^n if n is the number of places. However, some of these markings may not be reachable (i.e., reached in its execution). The *next state function* defines the next marking (state) μ' when applied to the current marking μ and an enabled transition t_j . The *next state function* $\delta : N^n \times T \rightarrow N^n$ for $M = (P, T, I, O, \mu)$ and $t_j \in T$ is defined iff t_j is enabled. If $\delta(\mu, t_j)$ is defined, then $\delta(\mu, t_j) = \mu'$, where

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

for all $p_i \in P$.

The execution of a marked Petri net $M = (P, T, I, O, \mu^0)$ by successive firings yields the sequences $(t_{j_0}, t_{j_1}, t_{j_2}, \dots)$ and $(\mu^0, \mu^1, \mu^2, \dots)$ of transitions fired and corresponding markings, where $\delta(\mu^k, t_{j_k}) = \mu^{k+1}$ for $k = 0, 1, 2, \dots$. These sequences together provide a record of the execution. If μ' is obtained from μ by the firing of a transition, then μ' is *immediately reachable* from μ . If μ' is immediately reachable from μ and μ'' is immediately reachable from μ' , then μ'' is *reachable* from μ . The *reachability set* $R(M)$ for $M = (P, T, I, O, \mu)$ is the smallest set of markings satisfying i) $\mu \in R(M)$, and ii) if $\mu' \in R(M)$ and $\mu'' = \delta(\mu', t_j)$ for some $t_j \in T$, then $\mu'' \in R(M)$. That is, the reachability set $R(C, \mu)$ or $R(M)$ of $M = (C, \mu)$ is the set of all markings reachable from μ .

For the Petri net of Figure 2.2, let $\mu = (\mu_1, \mu_2, \mu_3, \mu_4)$ where $\mu(p_i) = \mu_i$. Then this Petri net has an infinite reachability set consisting of the markings

$$(1, 0, 0, 1) \rightarrow (0, 2, 1, 0) \rightarrow (1, 1, 0, 1) \rightarrow (0, 3, 1, 0) \rightarrow (1, 2, 0, 1) \rightarrow \dots$$

Deadlock-freeness in a Petri net is concerned with the potential firing of transitions. A transition is *potentially fireable* in a marking μ if the transition is enabled in some marking $\mu' \in R(M)$ (recall $M = (C, \mu)$). A transition is *deadlock-free* in a marking μ if it is potentially fireable in every marking in $R(M)$. A Petri net is *deadlock-free* in a marking μ if every transition is deadlock-free in μ .

2.5.2 Extensions to Ordinary Petri Nets

Several different types of Petri nets are obtained by adding extensions to ordinary Petri nets.

By associating time delays to the transitions and/or places of an ordinary Petri net, a (deterministic) *timed net* is obtained if the delays are deterministically specified or a

stochastic net (SPN) if the delays are probabilistically specified. Transitions in a timed or stochastic net are drawn as rectangles to distinguish them from ordinary transitions.

A particular kind of timed net (Merlin and Farber [29]) is obtained by assigning to each transition of an ordinary Petri net, two time values τ_1 and τ_2 . A transition can fire only if it has been enabled for at least time τ_1 and if enabled, it *must* fire before time τ_2 . These transitions may be labelled with their associated time values τ_1 and τ_2 .

An SPN is an ordinary Petri net in which each transition is associated with an exponentially distributed random variable that describes the delay between transition enabling and firing. Where several transitions are enabled, the one with the shortest delay will fire first, reflecting a stochastic choice based on the firing rates of the competing enabled transitions. It has been shown (Molloy [31]) that the reachability graph of a bounded SPN is isomorphic to a finite Markov Chain.

A *generalized stochastic Petri net* (GSPN) (Marsan et al. [28]) represents a class of Petri nets extended from SPNs in order to cope with the problem of state explosion. A GSPN has two types of transitions *timed* and *immediate*. A timed transition has an exponentially distributed firing rate; an immediate transition has no firing delay. State space reduction is achieved by discarding vanishing markings which are associated with intermediate states in which the system spends little time.

In a *predicate/transition net* (Genrich and Lautenbach [16]) predicates are associated with transitions, corresponding to data parameter tuples associated with arcs emanating from the transitions. Conditions are associated with places, similar to ordinary nets. A particular data parameter tuple on an arc is instantiated if its corresponding transition predicates hold true. Predicate/transition nets play an important role in complexity reduction (and thus abstraction), as the use of predicates affords a compact representation of many types of complex models.

Colored nets (Jensen [27]) represent a generalization of ordinary Petri nets in which there may be several types or colors of tokens. The availability of multiple token types could facilitate modeling of systems with complex interactions. For example, in process control systems there may be a need to model several streams of control (with differing priority) which could be represented by different colored tokens.

Noe [34, 35] has proposed *Pro-Nets* for modeling computing systems. In a Pro-Net, a single transition may have two different sets of input arcs and two different sets of output arcs, where the arcs of one set are "and-ed" and the arcs of the other set are "or-ed".

In addition, a time value may be associated with a transition as in timed nets. Further, a transition may be refined into a subnet or a subnet abstracted into a transition. Noe believed that the capability for abstraction and refinement was essential for any model used to represent complex systems.

2.6 Recent Use of Petri Nets in Software Modeling

It is useful to examine how other researchers have applied Petri nets in related work. Recently, Yau and Caglayan (1983) [51], Bruno and Marchetto (1986) [10], Peng and Shin (1987) [37], Reisig (1987) [42], and Tankoano (1988) [46, 45] all applied Petri nets to model software. The work of these researchers is summarized below in terms of i) Contribution, ii) Petri Net Model, and iii) Application of the Model. Following this, a comparison of approaches between this work and the work of the above researchers is given.

2.6.1 Contribution

Yau and Caglayan presented an extended Petri net model strictly for representing and analyzing the design of existing parallel and distributed systems. Their model considers software in terms of “components”, much like the “activities” we will consider in Chapter 3, and takes account of control and data in terms of control state variables, abstract data types, and data objects.

Bruno and Marchetto gave a methodology for the rapid prototyping of process control systems (e.g., flexible manufacturing systems), based on an extended Petri net model called *PROT nets* (short for *process translatable nets*). Their methodology supported i) building an operational specification model, ii) evaluation, simulation, and validation of the model, and iii) rapid prototyping or automatic translation into program structures – hence the name PROT (automatic translation into Ada is shown in the paper). Supported items i) and iii) were peculiar to PROT nets; item ii) relied on established techniques for Petri net analysis.

Peng and Shin presented an approach (employing a GSPN) for modeling the behaviour of real-time control software distributed over nodes. They claimed that their approach has high potential for resolving various distributed real-time system design issues such as task execution time estimation, message handling, time-out, and task allocation.

Reisig applied Petri nets to software requirements specification and design. His design

methodology is centered on the use of *refinement* and *embedding* (the addition of new components to an existing net). He also compared the use of Petri nets for design to other design methodologies.

Tankoano dealt with the design of distributed systems for the control of industrial processes. He gave a procedure [46] which allows one to proceed automatically from an external specification of a control application to be distributed, to the specification of the application's internal structure, through the use of correctness-preserving transformations. Tankoano's approach is most like the approach taken here, but his Petri net model and transformations are completely different. Tankoano's research did not come to the author's attention until this work was completed.

2.6.2 Petri Net Model

Yau and Caglayan

Yau and Caglayan claimed that a distributed system can be completely represented by a 7-tuple:

$$SYSTEM = (S, T, D, C, R, M_0, F)$$

where

S : set of control state variables,

T : set of abstract data types,

D : set of data objects,

C : set of software components,

R : interconnection relation, $R \subseteq C \times C$,

M_0 : initial marking function (or initial marking m_0),

F : final control state variables ($F \subseteq$ power set of S).

The corresponding extended Petri net model is obtained by mapping control state variables to Petri net places and software components to nonprimitive transitions (a non-primitive transition has a subgraph and does not fire instantaneously). Abstract data types and data objects are depicted by small squares. The execution of a software component is

represented by the firing of a nonprimitive transition. Yau and Caglayan generalized this firing by associating with the nonprimitive transition (software component) both a control transfer specification and a data transfer specification, which give the control flow and data flow, respectively, through the transition. Total distributed system state is defined as the combination of total system control state and total system data state (state of the data objects). The total system control state is the collection of the individual control states of each component of the distributed system.

Abstract data types have an algebraic specification consisting of a syntactic part, a semantic part, and a restriction part. The syntactic part identifies type names, domain and range value sets, operation names, and type checking information. The semantic part defines the meaning of operations. The restriction part identifies limitations on values and operations.

The initial control state, the initial data state, and the component transfer specifications determine the system's dynamic behaviour. A component starts execution once a selected set of input control state variables is enabled. The execution terminates once output control state variables and data objects are modified as per specifications. Figure 2.4 shows the graphical representation of a software component in the extended model (Yau and Caglayan [51]). The component itself is depicted as a rectangle. Data types are connected to the component by solid lines; data objects are connected by directed dotted lines.

Bruno and Marchetto

A PROT net represents an operational specification of a system based on processes and their synchronizations. A PROT net is an ordinary Petri net with the following extensions:

Types - a PROT net models interactions of different process types; these interactions achieve synchronization and data exchange at the transitions. Data structures are associated with process types through a type declaration which serves to list the process types in the net and specify a suitable data structure for each process.

Places - these have names in uppercase and types in lowercase (corresponding to process types); they represent states of the corresponding process type. Each process type is represented by the ordered collection of all the places of the corresponding type. This collection in general takes the form of a tree and is called a *process tree*. The

Figure 2.4: A software component in the extended model (from Yau and Caglayan [51]).

root of this tree represents the first state of the process. A path from the root to a leaf represents a single process execution cycle.

Tokens - represent process instances or the fact that a process is in the state corresponding to the place in which a token resides. Tokens carry attributes, which are pieces of information whose structure is specified in the type declaration. Tokens may also be given priorities.

Transitions - perform synchronizations among process instances; data exchange may be specified by means of actions written on outgoing arcs. The graphical representation is either a rectangle (for *simple* transitions with no subgraph) or a stretched ellipse (for transitions decomposable into subgraphs). The stretched ellipse representation allows top-down structuring and information hiding. Transitions may also have predicates (over the attributes of input tokens), timing requirements, and firing probabilities (for mutually exclusive transitions).

Arcs - actions on outgoing arcs specify attribute value assignments for transition enabling tokens. Whether or not this means that tokens are conserved at the transitions is not specified. Tokens would not need to be conserved if the attributes are temporary, i.e., not required after a particular transition.

Initial Marking - in the initial marking, all tokens of the same process type must be put into a single place, called an initial place; the initial marking also specifies the initial values of token attributes.

Transition Firing - a transition can fire if there is at least one token in each of its input places and its predicate (if any) is satisfied. Transition firing results in i) removal of tokens satisfying the transition predicate from the input places, and ii) adding tokens to the output places after performing the actions inscribed on the outgoing arcs.

Peng and Shin

Peng and Shin decomposed tasks into activities and precedence constraints, which they then modeled with a GSPN. As such, they did not employ any extended Petri net model of their own.

Reisig

Reisig employed ordinary Petri nets extended with the concepts of abstraction and refinement – a subgraph of a transition is a refinement of the transition, while a transition is an abstraction of its subgraph.

Tankoano

Tankoano described the external behaviour of a system by associating this behaviour with the execution of a Petri net as follows: conditions (characterizing the state of the environment) and events (characterizing the change of states in the environment) are associated with transitions, operations (describing the actions of the system on the environment) are associated with places. Figure 2.5 illustrates Tankoano's Petri net model. In this Figure, the condition c and the event e , as just explained, is associated with transition t . In addition, the symbol τ represents a type of communication which is tied to the occurrence of e . The event e occurs or does not occur depending on whether a particular communication type takes place at t . The designer is then able to use the communication type to control the occurrence of e . This, according to Tankoano, is a simple solution to the loss of events when such events occur at a time when the transition is not otherwise enabled. Thus

in Figure 2.5, t is enabled when simultaneously c is true and e occurs according to the communication type τ .

Figure 2.5: Tankoano's Petri net model (from Tankoano [46]).

2.6.3 Application of the Model

Yau and Caglayan

After representing a design in terms of their extended model, Yau and Caglayan proposed analysis of the design by transforming the control aspects into an equivalent ordinary Petri net and then applying standard analysis methods to this Petri net. Some of the interesting problems that can be treated in this way are safeness of the control states, mutual exclusion of executions for some of the software components, proper termination, and deadlock-freeness. Figure 2.6 shows transformations of some extended model output control specifications to ordinary Petri nets.

Yau and Caglayan stated that their approach is not a design method and that their model needs to be integrated into an overall design methodology, which *must address the critical issue of system partitioning*. Their extended model needs additional work to i) establish coverage (can all systems be described?), ii) incorporate performance and reliability constraints, and iii) represent dynamic software structures (e.g., task creation and destruction).

Bruno and Marchetto

To apply PROT nets to rapid prototyping, Bruno and Marchetto translated them to Ada program structures. This translation consists of i) identifying the process types, and ii) establishing synchronizations logically expressed in the net. A process type can be identified by extracting the corresponding process tree from the net, starting with the

Figure 2.6: Petri net interpretations of output control (from Yau and Caglayan [51]).

initial state or place. Unfortunately, process states are not contiguous in the net but are scattered, and can be isolated only by following net logic. Bruno and Marchetto gave an algorithm which performs a depth-first search of a given PROT net to obtain a process tree. Each process tree so obtained can then be translated into an Ada task type, whose body is made up of states (represented as comments to be replaced with statements) alternating with code implementing interactions with transitions. A transition is implemented as an Ada task which interacts with processes through rendezvous.

Bruno and Marchetto have a collection of software tools to facilitate PROT net management. These tools include a graphical syntax-directed net editor, a net translator that translates a PROT net into a program skeleton for the system prototype, and a syntax-directed program editor.

Peng and Shin

In their modeling approach, Peng and Shin first decomposed the tasks at a node into activities and precedence constraints, which they then modeled with a GSPN. In the final step, they derive a sequence of continuous time Markov chains from the GSPN in order to model concurrent task execution.

The input to the modeling process is a *task flow graph* or TFG and its associated task tree. A TFG is a graphical representation of a task to be executed in a node of the distributed system. A TFG is made up of four types of subgraphs: *chain*, *And-Fork to And-Join*, *Or-Fork to Or-Join*, and *loop*. A chain is a sequence of execution units or objects E_i . An And-Fork to And-Join subgraph is made up of several branches, all of which must be executed (possibly in parallel). An Or-Fork to Or-Join subgraph is again made up of several branches but only one of these branches is executed; the probability of choosing a particular branch is assumed given. A loop consists of a loop body with a loop-back probability p . A task tree is a hierarchical representation of the TFG showing how the TFG is organized in terms of the four types of subgraphs and execution objects. Figure 2.7 shows a TFG and its associated task tree (Peng and Shin [37]).

Peng and Shin defined the terms *module*, *activity*, *communication point*, *control point*, and *structure point*. A *module* is a software entity created by combining two or more code stretches or modules (recursive definition). An *activity* is the largest module that can be formed without violating any precedence constraints. A *communication point* is an execution object representing a communication primitive. A control point represents

Figure 2.7: A TFG and its task tree (from Peng and Shin [37]).

the boundary of an activity which is not a communication point. A *structure point* is a boundary point of a subgraph. Structure points (Figure 2.7) include the upper end and lower end points of a chain, the fork and join points, and the collecting and branching points of a loop.

Given the TFG, the next step is to determine the smallest activity set from the TFG so that a corresponding GSPN may be constructed. The smallest activity set is obtained from the TFG by an iterative process of *combination* and *expansion* phases. In the combination phase, modules within subgraphs of the same layer (in the task tree) are merged. In the expansion phase, the structure points of the subgraphs of the active layer are removed so that the next combination phase can be applied across the original boundaries.

Peng and Shin constructed their GSPN to model the following three aspects of the task system:

- precedence constraints among activities, imposed by the TFG;
- precedence constraints among activities of two communicating tasks, imposed by the semantics of communication primitives;

3. *Principle of embedding and sectioning* - an embedding is the addition of new components to an existing Petri net; a sectioning is a selection of a subnet and ignoring the rest.
4. *Principle of continuously proceeding toward dynamics* - the idea that a static Petri net representation transforms smoothly to a dynamic simulation model by the addition of tokens and firing rules.
5. *Principle of locality and independence* - the effect of a transition firing is restricted to its local environment, to its input and output places. Further, different transitions in general act independently from one another. According to Reisig, these attributes allow for freedom and precision in system design.
6. *Principle of non-determinism* - the representation of alternatives without making a decision as to which alternative is selected. For example, a place may have several outgoing arcs representing several alternative paths for a token to travel, and yet not specify in advance which path will be taken.

Reisig developed a methodology for system design based on the above principles, with emphasis on the use of refinement and embedding. To keep track of refinements and embeddings, he introduced a graphical aid called a *schedule*, which can also be used to determine when a particular refinement or embedding was illegal.

Petri nets, according to Reisig, are especially useful for *the design of embedded systems* because they can be used to model both the system and its environment. Correct representation of the environment is crucial to embedded systems, since such systems interact with the environment to a high degree.

Reisig compared Petri nets as a design methodology to other design methods. Some of his observations are:

- SADT (Structured Analysis and Design Technique) makes use of both activity diagrams and data diagrams. The relationship between these diagrams and Petri nets is given in Figure 2.9. Notice that Petri nets provide an integrated representation of both activity and data.
- Hierarchical refinement is used in both SADT and Petri nets but it is impossible to formulate dynamic behaviour, considered as a further refinement, in SADT.

- The use of Petri nets span the gamut from initial informal sketches to formal specifications of dynamic systems behaviour. Everything is developed on the basis of previous steps without having to change the representational model. No other software engineering method covers such a broad spectrum of the life cycle. SA (Structured Analysis) and SADT cover only the initial analytic stages. JSD and “pseudocode” only approximate the formulation of dynamic behaviour. Abstract data types come close but lack a concept for control and data flow. Petri nets are precisely what is needed for rapid prototyping.
- The use of Petri nets may be supplemented with various rules and conventions such as those found in SADT (e.g., SADT rules for arranging nodes “from top left to bottom right”) to arrive at a complete software design method.
- Petri nets have a simple graphical representation which is easily accessible to the non-expert, and are therefore suitable as a means of communication between various persons involved in a software project.

Figure 2.9: SADT related to Petri nets (from Reisig [42]).

Tankoano

Tankoano's approach is based on three abstraction levels: i) the conceptual level, ii) the organizational level, and iii) the operational level. A designer at the conceptual level would

describe a system using an external specification. At the organizational level, the designer would structure the system into a network of communicating modules (via ports). At the operational level, the designer would distribute the modules to different physical sites. In this way, the designer proceeds from an external conceptual specification of behaviour to an internal distributed specification of behaviour for a network of communicating modules.

The internal behaviour of a network of communicating modules is described using the same Petri net model. Each module is described by a separate Petri net. The internal behaviour is composed not only of the firing of transitions but also of the operations of synchronous and asynchronous communication. The occurrence of events at ports is considered to be independent of time.

Tankoano used a notation (taken from CSP) for identifying a communication as either synchronous (rendezvous) or asynchronous. Synchronous send at port PS and receive at port PE for message M are denoted as $PS!M$ and $PE?M$ respectively. The corresponding asynchronous send and receive are denoted as $PS!/M$ and $PE?/M$ respectively. Figure 2.10 shows the meaning of the notations $\langle e(\tau).c- \rangle!M \rangle$ and $\langle e(\tau).c- \rangle?M \rangle$. Note that Tankoano did not actually connect together the send and receive ports with arcs and places.

As shown in Figure 2.10, the firing of transition t is delayed until the rendezvous can take place. Figure 2.11 shows the meaning of the notations $\langle e(\tau).c- \rangle!/M \rangle$ and $\langle e(\tau).c- \rangle?/M \rangle$ for asynchronous communication. As expected, the asynchronous send does not introduce any delay; the asynchronous receive is treated like a condition.

Based on the above semantics, Tankoano allowed transitions to participate in a combination of synchronous and asynchronous communication.

Tankoano gave three sets of rules for decomposing an external specification expressed in terms of his Petri net model. These sets of rules fall under the headings of i) partitioning, ii) doubling, and iii) replacement of a well-formed block. Tankoano also gave advice for when a particular decomposition rule would be used.

In partitioning, a subnet containing a single transition is partitioned into two subnets by splitting the transition into two transitions and then synchronizing the firing of the two transitions by using rendezvous communication between them. Figure 2.12 shows such a partitioning rule.

In doubling, a subnet with one or more transitions is replaced by two subnets to which certain actions, events, or conditions are distributed from the initial subnet. The two sub-

Figure 2.10: Semantics of $\langle e(\tau).c- \rangle !M \rangle$ and $\langle e(\tau).c- \rangle ?M \rangle$ (from Tankoano [46]).

Figure 2.11: Semantics of $\langle e(\tau).c- \rangle !/M \rangle$ and $\langle e(\tau).c- \rangle ?/M \rangle$ (from Tankoano [46]).

Figure 2.12: A partitioning rule (from Tankoano [46]).

nets are constrained by means of rendezvous communication between transitions in each subnet to collectively behave in an equivalent manner to the initial subnet. Figure 2.13 shows a doubling rule. In Figure 2.13, places are labeled with actions and transitions are labeled with conditions or the sending/receiving of messages.

Figure 2.13: A doubling rule (from Tankoano [46]).

A well-formed block is a subnet which a token may enter by flowing through one and only one transition and which a token may exit by flowing through one and only one transition. The decomposition consists of i) replacing the well-formed block in the original net by a single place, ii) establishing a second net consisting of the well-formed block, the

block entry and exit transitions, and a wait place, and iii) constraining the collective behaviour of the two nets so formed to be equivalent to the original net by the use of rendezvous communication between transitions in the two nets. This decomposition rule is useful for establishing a remote procedure call, where the net resulting from i) above contains the call, and the net from ii) contains the remote procedure (the well-formed block). Figure 2.14 illustrates replacement by a well-formed block.

Figure 2.14: Replacement by a well-formed block (from Tankoano [46]).

Tankoano defined behaviour equivalence between two nets using a form of bi-simulation and showed ([45]) that this equivalence is indeed preserved by his decomposition rules.

2.6.4 Comparison of Approaches

Yau and Peng presented analysis models with which to analyze existing designs. They approached the analysis problem from different directions: Yau constructed the extended Petri net model directly, Peng used an activity network, the TFG. Bruno, Reisig, and Tankoano each presented synthesis models with which to synthesize new designs. These researchers also approached the problem differently: Bruno and Tankoano each used operational approaches, Reisig used a more traditional SADT-type of approach.

This research is of course a synthesis approach, but it differs fundamentally from the synthesis approaches described above in terms of i) model, ii) methodology, iii) validation of correctness, and iv) completeness. The comparison of models is deferred to Chapter 3, until after the model used in this work is defined. Here, the approach of this work is compared to each of the synthesis approaches in the related research, detailing fundamental differences in terms of ii), iii) and iv). Since the goals differ, the synthesis approach of this work cannot really be compared to the analysis approaches, except in terms of the

representational aspects of the model employed (Chapter 3).

Methodology

Bruno and Marchetto rely on process types that were declared for the PROT net (when it was first set up) in order to identify individual processes. Extraction of a process from the PROT net specification is synonymous with the extraction of a process tree, which involves the identification of all places of the same process type. The step from process identification to Ada structures is then achieved through the use of certain Ada constructs to realize synchronizations that are logically present in the PROT net (e.g., use of an Ada task with rendezvous to implement a transition).

In this research, subprocesses are dynamically derived through partitioning decisions made to satisfy design goals, via transformations. The proto-design is then obtained by using further transformations to establish communication styles and data access control.

Thus, a basic difference between the approach of this thesis and the approach of Bruno and Marchetto can be seen in the starting point – the process specification. The PROT net specification comes with processes already specified in the form of process type encodings in the places. In the approach of this thesis, processes usually do not already exist in the specification and it is necessary to derive them. Thus, the approach of Bruno and Marchetto uses a PROT net to model a system in which processes have already been identified and then translates that PROT net into Ada structures as a means of rapid prototyping. In this thesis, the primary concern is the design of the system's processes according to design goals.

Reisig emphasized the use of Petri nets as a powerful notation for software design in general, through the use of refinements and embeddings, in the same spirit as SADT. This thesis focuses on designs consisting of concurrently executing sequential processes. Perhaps due to his different focus, Reisig did not consider partitioning nor the modeling of interprocess communication as structuring steps toward a design. The approach of this thesis is broader in the sense that it considers more structuring techniques, but at the same time, it is narrower in the sense that it focuses on a particular kind of design. Also, Reisig did not deal with the modeling of time delays so that performance requirements cannot be incorporated into his design procedure.

Tankoano's approach is most like the one taken in this research but there are fundamental differences in the process model and the transformations. This research employs

transformations that are functional, e.g., establishing a particular communication mechanism, and multi-purposed, e.g., abstraction, refinement, and partitioning. Tankoano's transformations have one and only one purpose: that of decomposing the net. More significantly, Tankoano's transformations rely on the availability of rendezvous communication whereas the transformations of this work have no such limitation. Tankoano's approach also does not deal with time which again rules out the incorporation of performance requirements into his design procedure.

Validation of Correctness

The use of correctness-preserving transformations in Tankoano's work as well as in this work precludes the necessity of validating the design for correctness. However, this is not the case for the approaches of Bruno and Reisig, who did not employ correctness-preserving transformations. Bruno applied Petri net analysis to his PROT nets to determine properties such as deadlock-freeness, place boundedness, and so on. He also executed his Ada proto-types to ensure correct functionality. Reisig determined the correctness of his Petri net designs by executing them.

Completeness

This section compares the designs for how close they are to code. Bruno and Marchetto's approach yields a design that is closest to code since they actually produce Ada skeleton code. The remaining synthesis approaches, including that of this research, do not yield code, although in this research, it is only an additional step or two to generate code (a translation to convert a proto-design to finite state machines is already provided).

Table 2.1 summarizes the above discussion of the different synthesis approaches. In this table, "annotated Petri nets" refer to Tankoano's use of CSP-like notation to annotate his transitions with communication semantics.

2.7 Chapter Summary

This chapter has presented background information for this research. In particular, the terms *process*, *subprocess*, *activity*, *specification*, *design*, *implementation*, and *transformation*

	Bruno and Marchetto (1986)	Feisig (1987)	Tankoano (1989)	Yee (1991)
Methodology	identification of processes thru process states and net logic	construction using refinements and embeddings	transformations based on communication semantics	transformations based on functional requirements
Validation of Correctness	Petri net analysis and execution of the design	execution of the design	correctness automatically preserved	correctness automatically preserved
Completeness	Ada skeleton code	structured Petri net	annotated Petri nets	structured Petri net

Table 2.1: Synthesis approaches using Petri nets.

tion have been explained in the context of this work. This chapter also reviewed Petri net theory, and looked at recent research which employed Petri nets as software models.

The next chapter will define the Petri net model for this research and compare it to other software models.

Chapter 3

Event Nets

The form of Petri nets used in this research is called *event nets*. This chapter gives the rationale for the choice of event nets and derives them from timed nets. Formal definitions of event nets and their characterizations are then given. The chapter concludes with an evaluation of event nets by comparing them to i) the Petri net based models used in the related research described in Chapter 2, and ii) the well known formal description techniques LOTOS (ISO [25]) and Statecharts (Harel [19]).

Since event nets will be used for expressing both the requirements specification and the proto-design, it is useful to define the terms “requirements specification” and “proto-design” before proceeding any further. A *requirements specification* or *process specification* is a formal description of required software behaviour in terms of control and event flow. A requirements specification may contain internal parallelism, non-determinism, or execution time requirements.

All non-determinism in the requirements specification are assumed to be “don’t care” non-determinism, i.e., all non-deterministic choices are equally valid, and the correct behaviour of the software system does not depend on the selection of some particular choice.

Execution time requirements are combined to compare the total execution times of alternate designs. The purpose of this comparison is to choose a design that conforms to execution time requirements, and is not performance analysis, which is outside the scope of this work.

A *proto-design* is a formal description of software behaviour in terms of a set of separate, concurrently executing, communicating sequential processes in which each process is expressed in terms of control and event flow.

3.1 Choice of Process Model

Cohen et al. ([14], p. 9) described requirements for system representation. Based on this description, criteria for selecting an executable formal model for this research include:

- expressive power - capability of representing requirement details of the software system (including performance);
- direct transformability - a requirements specification expressed in the language of the model must potentially be directly transformable into a design, i.e., without recourse to mappings into an auxiliary language;
- provability - formal model support for proofs of hypotheses concerning correctness (e.g., "behaviour correctness" of a proto-design with respect to a requirements specification);
- clarity, familiarity, and ease of use for a wide cross-section of both technical and non-technical people;
- supporting theory - availability of a wide base of results that have been obtained by other researchers using the formal model; the existence of such a base means that the model has been proven.

In addition, Brinksma [7] cites the following requirement:

- abstraction facilities and modularity - availability of facilities to abstract from irrelevant details and to structure specifications following meaningful composition principles - such facilities increase the readability and analyzability of specifications for complex systems.

The following criterion must be added to allow specification of performance requirements:

- specification of timing constraints - capability of the model to express execution timing constraints for subsequent performance analysis.

Finally, executability of the model would be highly desirable for requirements validation, design verification, and testing:

- executability - capability of the model to be executed, using a language interpreter or simulation tool.

3.1.1 The Choice of Timed Nets

Using the above criteria, timed nets were chosen as the basis for the formal model. At first, ordinary Petri nets, timed nets, activity sequence networks (Woodside [50]), and CCS (Milner [30]) were all considered as potentially suitable formal models. Activity sequence networks are close to timed nets and offer a good way of representing data. However, activity sequence networks are lacking in familiarity and supporting theory. CCS is strong in direct transformability, provability, and supporting theory but lacks timing information for performance analysis. Ordinary Petri nets also fail to capture timing information needed for performance analysis. Timed nets appear strong on all criteria.

3.1.2 Other Petri Net Models

Other adaptations of Petri net notation for software modeling, such as those discussed in the related research of Chapter 2 are largely unsuitable for this work. Yau and Caglayan's extended model is too detailed: it has representations for state variables and abstract data types, which implies that the model is at too low an abstraction level for use here, although it is appropriate for representing the details of existing distributed implementations. On the other extreme, ordinary Petri nets, as used in Reisig's approach, are missing certain features required in this research, e.g., predicates. Tankoano's Petri net model is also unsuitable here since it is specialized to model communication event semantics, e.g., communication links are not modeled by arcs! A "Petri net" model for this research must be recognizable as a Petri net by Petri net analysis tools, so that such tools may be applied. GSPNs, used by Peng and Shin for analysis, are too restrictive for software specification in general since their transition firing rates must be exponential and they lack predicates. Further, all models discussed in the related research, except for GSPNs and PROT nets, lack the representation of time. PROT nets would not be recognizable as a Petri net by many existing Petri net analysis tools since decomposable transitions are represented as ellipses. Further, PROT net transitions do not model functional activities but instead model synchronization among processes.

Other Petri net models in the literature (see Section 2.5.2) include predicate/transition nets (Genrich and Lautenbach [16]) and coloured Petri nets (Jensen [27]). These are also collectively known as *high-level nets* and have powerful features for modeling individuals and their changing properties and relations. Also, Noe [34, 35] (Section 2.5.2) has proposed

a net model called *Pro-Nets* in which transitions may be refined into subnets and subnets abstracted to transitions. Both Noe and Reisig, therefore, proposed the use of abstraction and refinement for specifying software systems.

The Petri net based models mentioned in this section are either too low-level, lack simplicity, tied to a particular process model that is semantically incompatible with the objectives of this work, or lack certain required features. Event nets were formulated to obtain the proper mix of components needed for this research without sacrificing simplicity. Event nets enjoy the analytical advantages of timed nets (e.g., performance analysis) and improve upon timed nets for this work through features added to facilitate software modeling.

3.1.3 Provability

To deal with provability, one possibility was to translate results between event nets and CCS so that proofs could be done in CCS. However, CCS proofs can be cumbersome, and supporting translation theory would be required to justify translating from one model to the other. Rather than pursue this indirect route to provability, a more direct approach was taken – namely to develop a proof theory on top of event nets (Chapter 5).

3.2 Timed Nets, Events, and Activities

3.2.1 Timed Nets

A *timed net* (Merlin and Farber [29] and Chapter 2) is an ordinary Petri net in which each transition has two time values τ_1 and τ_2 . Such a transition is called a *timed transition*. A timed transition can fire only if it has been enabled for at least time τ_1 , and if enabled, it *must* fire before time τ_2 . A timed transition is drawn as a rectangle to distinguish it from an ordinary transition.

3.2.2 Events

An *event* is an instantaneous change in a program associated with one or more of i) the occurrence of an action, ii) the enabling or disabling of a function, iii) the transfer of data, or iv) a time value. For example, a program requiring access to a data store may originate a “request access” event, which is the occurrence of the requesting action at an instant

in time. Events may be *external* or *internal*, corresponding respectively to a system's external events with its environment and its internal events between system components (e.g., subprocesses). A system's external events with its environment are also called its *interactions*.

3.2.3 Activities

An *activity* is a period of time in a program associated with i) input/output events, ii) the input and output of data (modeled by events), and iii) execution of some function. For example, the computation of the square of a number is an activity, which may have the input event "input number", output event "output number squared", and function "compute the square of the input number".

In this work, activities are used to express required functionality. An activity is identified by relating output events to input events using the following simple rule: *given a set of input events, an activity is required in order to generate the corresponding set of output events*.

3.3 Derivation of Event Nets

Event nets are derived from the timed net model of Merlin and Farber [29] by i) defining events, data, and activities in terms of the existing components of this timed net model and ii) adding new components to this timed net model, namely predicate transitions, refinement/abstraction, and queuing. Sections 3.3.1 and 3.3.2 below give (i) and (ii) respectively, and together define the behaviour semantics of event nets. A formal definition of event nets is given in Section 3.3.3.

3.3.1 Definitions on Existing Components

Events

Events have been defined in Section 3.2.2. The movement of a token along an arc from a transition to an output place models the occurrence of an output event. The movement of a token along an arc from an input place to a transition models the occurrence of an input event. An arc associated with an event in this way may be labelled with the name of the event.

Repeated events are considered as distinct events and arise through repeated execution of activities due to looping or process cycling. The names of repeated events are assumed to be automatically indexed so that such events are distinguishable from one another (e.g., the indexing of events in order to tell them apart is also done in Real-Time Logic by Jahanian and Mok [26]). An index of a repeated event is assumed to be incremented from one cycle to the next by the cycle activity that is executed upon start-up or first entry into the cycle.

Activities

A timed transition models an activity, with execution time $\tau \in [\tau_1, \tau_2]$, where the latter interval consists of positive real numbers and possibly zero. The execution time τ is the time from the instant when the transition was enabled, i.e., when execution started, to the instant when execution completes. A transition is enabled when there is at least one token in each of its input places, as in ordinary Petri nets. Tokens leave input places and are absorbed by the timed transition at the instant the transition is enabled. The transition does not fire until execution completes, at which point it fires instantaneously sending one token to each output place. In principle, the value of τ may be modeled deterministically or stochastically (e.g., exponential distribution). However, a model for the value of τ is not considered in this work.

Data

A token may carry data values or entire data structures. Such data values or structures are collectively known as *data attributes* (much like the token attributes of PROT nets). Data variables and storage structures exist implicitly and are not explicitly represented apart from their presence in event names. The assignment of data to variables may be modeled as data transfer events (recall from the above discussion that a data transfer is an event). The data carrying capacity of a token is taken to be unlimited.

A token flowing from one transition to a second transition may model the conveyance of data values from a data producing activity (the first transition) to a data consuming activity (the second transition). On its way to the consuming activity, the token may pass through other activities, which may themselves place data on the token for conveyance to recipient activities or consume data on the token that were meant for them.

3.3.2 Definitions of New Components

Predicate Transitions

Predicate transitions are used to model alternative paths of execution. A predicate transition is a timed transition to which is associated a *predicate*, i.e., a Boolean expression relating data values that are carried by a token. The execution of a predicate transition is divided up into phase 1 and phase 2. Phase 1 is mandatory and consists of predicate evaluation. Phase 2 is optional and consists of any assigned computation other than predicate evaluation. Phase 2 (if any) may be entered only if the predicate evaluated to true. The execution time τ of a predicate transition is the total time taken to execute both phase 1 and phase 2. As for activities, no model for the value of τ is considered in this work.

A predicate transition starts phase 1 execution at the instant when it becomes enabled, i.e., when there is at least one token in each of its input places. At this point, data required for the predicate evaluation is carried by some token or tokens residing in the predicate transitions input places. To avoid taking in such tokens and then somehow returning them should the predicate evaluate to false, it is understood that *for predicate evaluation, the predicate transition can access the data that it needs for the evaluation while this data is carried by tokens residing in its input places*. At the instant when a predicate evaluates to true, a token from each input place is absorbed into the transition. Upon completion of execution, the predicate transition fires and deposits a token into each of its output places.

A phase 2 computation is in effect an activity. A predicate transition therefore may have input and output events that are associated with the phase 2 computation as an activity.

A predicate transition may have the constant predicate 'true', which is always true. Such a predicate transition is equivalent to an activity, and will fire if it becomes enabled. This type of predicate transition is used to model choice paths where the choice is not determined by a predicate, but by whether or not the predicate transition becomes enabled.

In this model, one is not interested in the actual evaluation of predicates but rather in the subsequent flow of control that is dependent on the outcome of such evaluation. Therefore, predicate evaluation is considered non-deterministic, which allows all outcomes of predicate evaluations to be modeled.

A predicate transition is distinguished from other timed transitions by drawing it as

a timed transition with a line through its center and labeling it with its predicate. A predicate transition that is not labeled with any predicate has the constant predicate 'true' by default.

Refinement and Abstraction

A timed transition (either an activity or a predicate transition) may contain a subgraph which is its *refinement*. At the same time, a timed transition is the *abstraction* of its subgraph. Subgraphs must conform to the following rules: i) arcs terminating on a transition must all terminate on a single transition within its subgraph, and ii) arcs originating from a transition must all originate from a single transition within its subgraph. These rules are required to ensure a) that the subgraph has the same conditions for execution as its abstraction and b) that output events generated at the same time by the abstraction are generated at the same time by its subgraph.

Queuing

Every token arriving at a place by default joins an implicit FIFO queue (other queuing disciplines can be used). An arriving token joins the tail of the queue; a departing token leaves the head of the queue.

Figure 3.1 shows activity *a* with input event *E1*, output event *E2*, and predicate transitions *pred1* and *pred2*.

The preceding behaviour semantics describe a timed net process model called *event nets*, since the model expresses the temporal order of events. Event nets are high-level in the sense that functional activities may be abstracted in terms of timed transitions. Event nets allow for the expression of software structure and enable performance analysis (due to event modeling and the availability of Petri net analysis tools).

3.3.3 Formal Definition of Event Nets

This section formally defines event nets and event subnets.

DEFINITION 3.1 *An event net structure C is a 5-tuple $C = (P, A, T, I, O)$ where $P = \{p_1, \dots, p_n\}$ is a finite set of places, $n \geq 0$, $A = \{a_1, \dots, a_k\}$ is a finite set of activities, $k \geq 0$, $T = \{t_1, \dots, t_m\}$ is a finite set of predicate transitions, $m \geq 0$, $I : A, T \rightarrow P^\infty$ and $O : A, T \rightarrow P^\infty$ are the input and output mappings, respectively, from activities and*

Figure 3.1: Events, activities, and predicate transitions.

predicate transitions to sets of places. A place $p \in P$ is both an input place and an output place, i.e., $p \in I(d)$ for some $d \in (A \cup T)$ and $p \in O(d')$ for some $d' \in (A \cup T)$.

The restriction of the ranges of I and O to sets of places rather than to bags of places, as in ordinary Petri nets, means that only one mapping is allowed from an activity or predicate transition to a place. The requirement for a place to be both an input place and an output place means that a place is always on the token path between two timed transitions. These properties facilitate the development of a relational notation for event nets in Chapter 4.

DEFINITION 3.2 *Given an event net structure $C = (P, A, T, I, O)$, an event subnet structure D of C is a 5-tuple $D = (P_D, A_D, T_D, I_D, O_D)$ where*

$$P_D \subseteq P, A_D \subseteq A, T_D \subseteq T, \text{ and} \\ I_D : A_D, T_D \rightarrow P_D^\infty, O_D : A_D, T_D \rightarrow P_D^\infty$$

such that $p \in I_D(d)$ and $p \in O_D(d')$ only if $p \in I(d)$ and $p \in O(d')$ for $p \in P_D$ and $d, d' \in (A_D \cup T_D)$. An event subnet structure is also an event net structure, by Definition 3.1.

DEFINITION 3.3 *A marking μ of an event net structure $C = (P, A, T, I, O)$ is a function $\mu : P \rightarrow N$ where N is the set of non-negative integers. A marked event net structure $M = (C, \mu)$ is an event net structure together with a marking μ . Similarly, a marked event*

subnet structure $K = (D, \mu)$ is an event subnet structure D together with a marking μ of D .

DEFINITION 3.4 *An event net $E = (M, B)$ is a marked event net structure M together with the set B of behaviour semantics defined above by Section 3.3. An event subnet $S = (K, B)$ is a marked event subnet structure K together with B .*

3.3.4 Characterizing Event Nets

Event nets may be characterized according to structure and execution as follows:

Structure

DEFINITION 3.5 *The following components characterize event net structure:*

- *An environmental component of a software system may be a user, a process that is not part of the software system, or a hardware device from which (whom) the software system receives input data, described by input events, and to which (whom) the software system sends output data, described by output events. An event net process specification models both the software system and its environmental components. An environmental component is modeled by an activity and its input or output places. Only tokens for input to an environmental component may appear in that component's input places. Only tokens output by an environmental component may appear in that component's output places.*
- *A software transition is either a software activity of the system under design (as opposed to an activity modeling an environmental component) or a predicate transition.*
- *An input interaction of a software transition is an input event describing the input of data from an environmental component to a software transition. An input interaction set is the set of all input interactions of an software transition. An output interaction of a software transition is an output event describing the output of data from a software transition to an environmental component. An output interaction set is the set of all output interactions of a software transition. An interaction is either an input interaction or an output interaction (interactions are also known as external*

events). An interaction set is either an input interaction set or an output interaction set.

- An interaction transition is a software transition that has associated input interactions, output interactions, or both.

Figure 3.2 illustrates Definition 3.5.

Figure 3.2: Environmental components and interactions.

The following definition gives a condition that appears reasonable for modeling systems in which every activity either participates in interactions or generates some output which is required by an activity that does participate in interactions:

DEFINITION 3.6 Let E be an event net with marking μ and reachability set $R(E, \mu)$. E is interaction-connected iff for any non-interaction transition c in E , $\exists$ interaction transition d in E and markings μ' , μ'' in $R(E, \mu)$ such that in order for d to be enabled in μ'' , c must first be enabled in μ' .

A subnet S of E is interaction-connected iff for any non-interaction transition c in S , $\exists$ interaction transition d in E and markings μ' , μ'' in $R(E, \mu)$ such that in order for d to be enabled in μ'' , c must first be enabled in μ' .

Figure 3.3 illustrates Definition 3.6.

Interaction-connectedness is important for showing safety properties such as deadlock-freeness and will be further examined in Chapter 5.

E:

Non-Interaction transitions: c1, c2, c3
 Interaction transitions: d1, d2

E is interaction-connected.

Figure 3.3: Interaction-connectedness.

Execution

Event nets may be characterized in terms of execution traces, parallel execution, sequential execution, continuous execution, and deadlock-freeness as follows:

DEFINITION 3.7 Let E be an event net with structure (P, A, T, I, O) in which time is measured from the start of execution. Let $A_E(t) = \{d \mid d \in A \cup T, d \text{ is enabled at time } t\}$. Then the sequence $A_E(t_0), A_E(t_1), \dots, A_E(t_n)$ is a description of the execution of E up to time t_n . Let $u_0 = 0$ and u_k be the earliest time from u_{k-1} at which $A_E(u_k) \neq A_E(u_{k-1})$. Then the sequence $A_E(u_0), A_E(u_1), \dots, A_E(u_m), \dots$ is called an *execution trace* of E . Let $\#A_E(u_k)$ denote the number of transitions in $A_E(u_k)$. Then $A_E(u_0), A_E(u_1), \dots, A_E(u_m), \dots$ is called a *sequential execution trace* iff $\#A_E(u_k) = 1$ for all integers k . The sequence $A_E(u_0), A_E(u_1), \dots, A_E(u_m), \dots$ is called a *parallel execution trace* iff $\exists$ some $A_E(u_k)$ such that $\#A_E(u_k) > 1$.

An event net may have more than one execution trace due to non-determinism in its execution, caused by parallel fork branches and non-deterministic choices of which next transition to fire. Parallel fork branches lead to alternate orders of transition firing for those transitions lying on different parallel branches and involve the same transitions. Non-deterministic choices of which next transition to fire result in alternate execution paths that involve different transitions.

DEFINITION 3.8 *Let E be an event net. E is sequential iff all its execution traces are sequential. E is parallel iff at least one of its execution traces is parallel. E is continuous iff every execution trace of E is non-terminating.*

The event nets (processes) considered in this work are all continuous and may be parallel or sequential. The meaning of deadlock-freeness for an event net is the same as for an ordinary Petri net (Chapter 2).

DEFINITION 3.9 *Let E be an event net with marking μ and reachability set $R(E, \mu)$. A transition d of E is potentially fireable in μ iff $\exists \mu' \in R(E, \mu)$ such that d is enabled in μ' . A transition of E is deadlock-free in μ iff it is potentially fireable in every marking in $R(E, \mu)$. E is deadlock-free in μ iff every transition of E is deadlock-free in μ .*

The deadlock-freeness of an event net E is a *safety* property of E (a safety property of E says that “nothing bad will happen” to E).

3.4 Comparison of Event Nets to Other Formal Models

Event nets will now be compared to Petri net models in the related research of Chapter 2. Event nets will also be compared to the well known FDTs LOTOS (ISO [25]) and Statecharts (Harel [19]), using the criteria at the beginning of this chapter.

3.4.1 Comparison to Petri Net Models in the Related Research

The related research of Chapter 2 applied the following Petri net models: an extended model (Yau and Caglayan), PROT nets (Bruno and Marchetto), GSPN (Peng and Shin), ordinary Petri nets with refinement (Reisig), and ordinary Petri nets with annotated communication semantics (Tankoano).

In the following comparison, the corresponding event net feature is in brackets. None of these models, it would appear, have the event net feature of a queuing discipline for tokens arriving at places. On the other hand, all of these models are forms of Petri nets and therefore all share the basic features of places and transitions and all obey the Petri net transition firing rule, even though some models conjoin it with other rules (e.g., predicate evaluation).

Yau and Caglayan

Event nets share with the extended model the use of decomposable software components (decomposable activities). Event nets are at a higher abstraction level than Yau and Caglayan's extended model and do not incorporate the following low level features of the extended model: control state variables, abstract data types, data objects, control transfer specifications, and data transfer specifications. Such features are probably required in the extended model in order for it to capture a design represented by existing code (recall the extended model is used for analysis of existing designs). An important deficiency of the extended model is that it does not have any representation for time.

Bruno and Marchetto

Event nets are closest to PROT nets and share the use of token attributes (data carrying tokens), decomposable transitions (although a PROT net transition models synchronization rather than functional activity), predicates on transitions, and timing requirements on transitions. However, event nets do not share the following low level details of PROT nets: process types and type declarations, data structures, token priorities, action annotation on arcs for data exchange, and transition firing probabilities. Process types are a consequence of Bruno and Marchetto's approach to process identification and are not needed in this work. The remaining detailed features of PROT nets are excluded in order to achieve a high level of abstraction, although transition firing probabilities may prove useful.

Peng and Shin

Peng and Shin translated their task flow graph to a GSPN for subsequent analysis. A GSPN is comparable to a timed net, which means that for software modeling, event nets are richer than GSPNs. Event nets share in Peng and Shin's use of GSPNs to model precedence constraints among activities and precedence constraints among activities of two communicating tasks, imposed by communication semantics. Peng and Shin's work is also of interest to us in their use of a task flow graph. Although a task flow graph is not a Petri net model, event nets share the task flow graph's use of activities and activity chains, split-joins, and loops.

Reisig

Reisig's contribution was oriented more towards a rationalization of the use of Petri nets in software engineering rather than towards a new and better Petri net model. The use of event nets for software modeling conforms to the six principles of Petri net usage formulated by Reisig (see Chapter 2). In particular, event nets incorporate Reisig's principles of abstraction/refinement and embedding/sectioning, which are realized as follows: abstraction/refinement are built-in components of the event net model; embedding/sectioning are done using transformations, which replace selected subnets (sectionings) with new subnets (embeddings).

Tankoano

Event nets and Tankoano's Petri nets both model conditions, events, and interprocess communication. However, Tankoano expresses these features by annotating them on his transitions rather than through Petri net execution. Tankoano ties the firing of a transition to the occurrence of a certain communication type at the transition and uses communication semantics to partition his net. Thus, even though conditions, events, and communication are present in both models, they are used in totally different ways. Tankoano's model does not have a representation for time, nor does it feature decomposable transitions and data carrying tokens. Tankoano's model cannot be run on standard Petri net simulation and analysis tools since it expresses interprocess communication with CSP-like notation, rather than through Petri net constructs as done in this work.

Table 3.1 summarizes the above comparison of event nets to other Petri net models in the related research. In this table, "Commun. by Annotation" refers to Tankoano's use of CSP-like notation to annotate transitions with communication styles; "Commun. by PN Struct." refers to modeling communication styles using Petri net structures, namely transitions, places, and arcs.

3.4.2 Comparison to LOTOS and Statecharts**LOTOS**

LOTOS (ISO [25]) is a textual language for describing distributed systems in terms of processes. A system is described as a single process that may consist of several interacting subprocesses. These subprocesses in turn may be refined into sub-subprocesses,

	Yau & Cag. (1983)	Bruno & Mar. (1986)	Peng & Shin (1987)	Reisig (1987)	Tankoano (1989)	Yee (1991)
Activity Trans.	yes	no	yes	yes	no	yes
Decompos- able Trans.	yes	yes	no	yes	no	yes
Predicate Trans.	no	yes	no	no	yes	yes
Data Types	yes	yes	no	no	no	no
Data in Tokens	no	yes	no	yes	yes	yes
Represent Time	no	yes	yes	no	no	yes
Commun. by Annotation	no	no	no	no	yes	no
Commun. by PN Struct.	yes	yes	yes	yes	no	yes
Queues in Places	no	no	no	no	no	yes

Table 3.1: Comparison of event nets to other Petri net models.

and so on, so that a LOTOS specification of a system is essentially a hierarchy of process definitions. Since LOTOS can describe processes, one would expect it to possess capabilities for describing sequential, parallel, and non-deterministic execution and indeed it does. Inter-process communication is achieved by synchronization at process ports or gates. Asynchronous communication may be modeled with the use of buffer processes. LOTOS also possesses capabilities for the specification of abstract data types. A LOTOS specification may be expressed in a number of different extensional and intensional styles with the extensional styles more suitable to requirements specification and the intensional styles more suitable to design (see Vissers et al. [48]).

Event nets are comparable to LOTOS in terms of all criteria listed in Section 3.1 except for the specification of timing constraints (which enable performance analysis) and ease of learning.

Standard LOTOS does not possess any facility for the specification of timing constraints but this is changing, as various researchers are developing ways of incorporating time into LOTOS (see Hulzen et al. [24] and Quemada et al. [41]). The results of this effort are quite elegant (especially the effort of Quemada et al. summarized below) and would appear to be effective. However, since these developments are relatively new, they have not been tried and proven for their expression of time, unlike timed nets. Hulzen et al. proposed to extend LOTOS with clocks in which each process has an implicit clock table and an implicit TICK gate. A process is always ready to synchronize with a clock tick from a master clock, increment its clock table, and continue with its normal behaviour. Operators are defined for starting and reading clocks, allowing time values to be part of conditions which determine the occurrence of events. Quemada et al. proposed a timed calculus interpretation of basic LOTOS, in which time is introduced by adding time restrictions to the occurrence of events. Each event in this calculus happens at one instant of time, which is appended to it as a numerical value, e.g., a_3 represents that event a happens at instant 3. In addition, this time value is always relative to the previous event, or to the start of the behaviour if there is no previous event. By identifying these events with the activities of event nets it is seen that such expression of time is similar to the expression of time in event net activities (timed transitions).

In terms of ease of learning, event nets appear to be conceptually easier to grasp than LOTOS. This is primarily due to the fact that event nets describe the flow of data and control, which is a familiar concept that has been exhibited ubiquitously via the flowchart.

Statecharts

Statecharts (Harel [19]) is a kind of extended communication finite state machine model which allows for hierarchical states (i.e., states within states), broadcast communication of events, and composition of independent concurrent statecharts. The main advantages of statecharts over conventional communication state machines are the capabilities to express state hierarchy and the composition of concurrent statecharts. These features reduce significantly the effort required in state space exploration. This effort reduction is due to the fact that the state space of the system can be explored by exploring the much smaller state spaces of individual machines, although the actual size of the system state space is not reduced in any way. Statecharts therefore help to solve the state explosion problem which has hampered reachability analysis of ordinary state machine based descriptions.

Comparing statecharts with event nets, one advantage of statecharts is that the composition of concurrent statecharts allows an elegant, precise, short form way to express requirements with state space the cartesian product of separate statechart state spaces. Such composition may be compared to the parallel operator in LOTOS and as in LOTOS, leads naturally to a structured description of software requirements in terms of resources (the concurrent statecharts) which is an intensional description supportive of design. However, the broadcast communication paradigm in statecharts is weak from a software modeling viewpoint, when compared to the essentially unlimited number of communication paradigms possible in event nets. Firstly, broadcast is the only communication paradigm in statecharts, whereas real life concurrent software uses many different communication paradigms. Secondly, if the receiving statechart is not in the proper state to receive the broadcasted event, that event is lost: real life software seldom uses such a communication model. Another disadvantage of statecharts is that as finite state machines, they express control only, whereas event nets express not only control but also functionality in terms of activities and execution sequence. Still another drawback of statecharts is that they do not have a natural way of specifying timing requirements, unlike the activities of event nets. In statecharts, timing requirements would have to be expressed in a round-about way in terms of events. However, statecharts, as a form of finite state machine, are probably easier to understand and use than event nets.

3.5 Chapter Summary

The main contribution of this chapter has been the derivation of event nets, the software model for use in this research. After an informal description of event nets, formal definitions of event nets and their characterizations were given, followed by a comparison of event nets to the Petri net models used by other researchers described in Chapter 2. The chapter concluded by comparing event nets to two well known formal description techniques, namely LOTOS and statecharts.

Given software requirements expressed in terms of an event net, a next question is how are transformations expressed using event nets, and what kind of transformations are required to obtain a proto-design? This question will be answered in the next chapter.

Chapter 4

Transformations

The purpose of this chapter is to define transformations on event nets, so that given an event net process specification of a software system, the transformations may be used to convert the process specification into a proto-design. Section 4.1 motivates the development of the transformations and informally describes them. Section 4.2 develops a formal notation for specifying event nets. Section 4.3 lists certain required properties of an event net process specification. Section 4.4 formally defines the transformations.

4.1 Introduction

To discover the nature of the transformations that will be needed, it is helpful to consider “where one is coming from”, i.e., the requirements specification, and “where one is going to”, i.e., the proto-design. An event net requirements specification of a software system is an event net which, through execution, expresses requirements in terms of interactions between the event net and its environment. Such an event net would probably be unstructured since structure is not the goal at the requirements specification stage. In addition, this event net may exhibit internal concurrency (possibly through internal concurrent processes) or non-determinism, reflecting required concurrency or non-determinism in interactions. The proto-design, on the other hand, is an event net “design” specification in terms of separate *sequential* subprocesses, which execute continuously and which are tied to one another through different styles of inter-process communication. The transformations to be defined must allow progression from the requirements specification to the proto-design. Figure 4.1 gives a high-level view of the role required of transformations.

Given the structure of requirements specification and proto-design, as just described,

Figure 4.1: A high-level view of the role required of transformations.

it is immediately clear that in order to partition the requirements specification into sequential subprocesses, a *subprocess construction* transformation that constructs sequential subprocesses is needed. This construction results in subprocesses that are inter-connected by token paths, arising from inter-activity connections in the requirements specification. These token paths are then candidates for *communication* transformations, needed to establish different styles of interprocess communication (this is further explained below). Another effect of the partitioning has to do with access to shared resources. A resource which is accessed sequentially in the requirements specification may, in the proto-design, be accessed concurrently by concurrently executing subprocesses. Depending on the type of access, this may give rise to a need to enforce mutual exclusion of resource access in the proto-design. This type of transformation is called *resource access control*. The next two transformations are needed to support the definition of event nets. Since transitions can have a subgraph, a *refinement* transformation for replacing a transition with its subgraph is needed. Conversely, since subgraphs can be abstracted, an *abstraction* transformation for replacing a subgraph with its abstracting transition is needed. It is not suggested, of course, that these two transformations are needed just to support event nets. Refinement allows the designer to delay the details of certain activities to a later stage, when certain needed information, for example, becomes available. As noted in Chapter 3, refinement is also featured in the models of Yau and Caglayan, Bruno and Marchetto, and Reisig.

Abstraction allows the designer to reduce the complexity of an event net for greater ease of understanding. The abstracted subnet may be displayed by itself and appropriately identified as belonging to the abstracting transition, so that no behavioural information is lost. Finally, the part of the subprocess construction transformation that sequentializes activities for a new sequential subprocess, is useful in its own right as a *sequentialization* transformation. For example, the requirements specification may specify the execution order of two activities a_1 and a_2 to be either a_1, a_2 , or a_2, a_1 . Suppose, however, that better throughput is achieved if a_2 is executed before a_1 (a required output from a_2 is then available sooner). In this case, the designer may use the sequentialization transformation to sequentialize or chain the activities so that a_2 is executed before a_1 .

Six types of transformations have been derived for transforming a requirements specification into a proto-design. Resource access control and communication are really classes, each containing many individual transformations, since there are many types of resource access control and many communication styles. The subprocess construction and communication transformations are collectively referred to as *partitioning* since both transformations are required to partition the requirements specification. Each of these transformations will now be more closely examined.

Abstraction and Refinement

The abstraction transformation hides or abstracts a subnet into a single timed transition. The refinement transformation refines or exposes a single timed transition to reveal its underlying subnet (the refinement subnet). Thus, abstraction and refinement are opposites of each other. For consistency with the event net model, input arcs to the transition to be refined (the outer transition) are required to terminate on a *single* transition of the refinement subnet and output arcs from the outer transition are required to originate from a *single* transition of the refinement subnet. Further, abstraction must not hide any events which need to be visible for comparing the behaviour of two nets. Refinement may introduce new intermediate events.

For example, in Figure 4.2(a) the subnet encircled with dashed lines is abstracted into the single transition (encircled with dashed lines) of Figure 4.2(b). Conversely, the transition of Figure 4.2(b) is refined into the subnet of Figure 4.2(a).

Figure 4.3 illustrates the requirements for consistency with the outer transition. Neither the encircled subnet of Figure 4.3(a) nor the encircled subnet of Figure 4.3(b) can be

Figure 4.2: Abstraction and Refinement.

abstracted into the encircled transition of Figure 4.2(b), due to the single transition arc termination and origination requirement. Conversely, the encircled transition of Figure 4.2(b) cannot be refined into the encircled subnet of either Figure 4.3(a) or Figure 4.3(b).

Sequentialization

The requirements specification may allow certain events to occur in more than one order. For example, where two activities are forked by a single transition, either of the two activities may finish first or both may finish at the same time. Correspondingly, the order of events associated with the two activities is undetermined, and in fact, several such orders are possible. The sequentialization transformation transfers to the proto-design only a subset of the set of orders of event occurrence allowed by the requirements specification. For example, Figure 4.4(a) shows a subnet in which the events A, B, C, D, E, and F may occur in any sequence provided i) event A is first with event F last (relative to the subnet), ii) events B and C occur after event A, iii) event D occurs after event B, iv) event E occurs after event C, and v) event F occurs after events D and E. In particular, the sequences A,B,C,E,D,F and A,B,C,D,E,F are both members of the set of possible event sequences allowed by Figure 4.4(a). Figure 4.4(b) models the sequence A,B,C,E,D,F and

Figure 4.3: Non-abstractable nets, non-derivable refinements.

Figure 4.4(c) models both sequences A,B,C,E,D,F and A,B,C,D,E,F. Figures 4.4(b) and 4.4(c) therefore model subsets of the set of event sequences modeled by Figure 4.4(a) and are the result of applying sequentialization transformations to Figure 4.4(a).

Subprocess Construction

In terms of event nets, a subprocess is made up of one or more alternative transition firing sequences. In such a firing sequence, the transition that was enabled upon startup can only be enabled again if it is next in line to fire, according to its position in the sequence.

Subprocess construction involves i) selecting a subnet to turn into a subprocess, ii) applying the sequentialization transformation to this subnet, and iii) if necessary, enabling the subnet resulting from part ii) to cycle, i.e., execute repeatedly.

For example, suppose the subnet with interactions A and B in Figure 4.5(a) is identified from the requirements specification as a potential subprocess following part i) above. The sequentialization transformation is applied to this subnet, giving the subnet in Figure 4.5(b), completing part ii) above. Figure 4.5(c) shows the resulting subprocess after part iii) above.

Figure 4.4: Sequentialization of alternate orders of event occurrence.

Communication

The subprocess construction transformation results in stand-alone subprocesses which are interconnected to one another by *communication links* represented by communication token paths. The communication transformation consists of replacing these token paths with specific event net constructs that model particular communication mechanisms such as rendezvous. By default, without any such replacement, the existing communication token paths model “Send-No-Wait/Receive-Wait” asynchronous unbounded FIFO channel communication, due to the underlying event net semantics.

Figure 4.6 gives an example of establishing rendezvous communication. Figure 4.6(a) shows the communication token path from subprocess 1 to subprocess 2. This token path by default models “Send-No-Wait/Receive-Wait” communication as stated above. Figure 4.6(b) is obtained by applying the “Rendezvous - Recipient Acks” communication transformation to the communication token path in Figure 4.6(a).

Resource Access Control

Resource access control refers to the control of a program’s access to a logical resource such as a disk file, or a physical resource such as a printer, for writing, reading, or both writing and reading. For example, mutual exclusion of write and read access is one type of resource access control that is modeled by the place labeled “resource” in Figure 4.7(a),

Figure 4.5: Subprocess Construction.

Figure 4.6: Inter-process communication.

where subprocesses 1 and 2 require write and read access to “resource” respectively. The single token inside place “resource” effectively models mutually exclusive access to the resource.

The resource access control transformation replaces arcs and places that model resource access control at a *high abstraction level* with particular resource access control mechanisms that give the same access control at a *lower abstraction level*. For example, the “Resource Server” transformation is one type of resource access control transformation that provides mutual exclusion for read and write access. This transformation introduces an additional subprocess to act as a server to control access requests. Figure 4.7(b) shows the result of applying the “Resource Server” transformation to the net of Figure 4.7(a).

(a) Modeling mutually exclusive access to a resource.

(b) Application of the "Resource Server" transformation to part (a).

Figure 4.7: Modeling resource access control.

4.2 Formal Descriptions of Event Nets

4.2.1 A Relational Notation for Event Nets

DEFINITION 4.1 Let $E = (P, A, T, I, O)$ be an event net structure and $D_1, D_2 \subset (A \cup T)$. Then an adjacency relation is an ordered pair $\langle D_1; D_2 \rangle$, where

$$\begin{aligned} \langle D_1; D_2 \rangle \text{ iff } \exists p \in P, \text{ such that for every } d \in (D_1 \cup D_2), \\ p \in O(d) \text{ if } d \in D_1 \text{ and} \\ p \in I(d) \text{ if } d \in D_2. \end{aligned}$$

Further, if D_1 contains all d such that $p \in O(d)$, and D_2 contains all d such that $p \in I(d)$, then $\langle D_1; D_2 \rangle$ is a full adjacency relation. An adjacency relation is not reflexive, not symmetric, and not transitive. Notation: $p : \langle D_1; D_2 \rangle$ denotes place p belonging to adjacency relation $\langle D_1; D_2 \rangle$ (there is only one such p ; if there is more than one such p , then all but one are redundant and are removed).

Notation: In Definition 4.1 let

$$D_1 = \{d_1, d_2, \dots, d_k\}, \quad D_2 = \{c_1, c_2, \dots, c_m\}.$$

Then $\langle D_1; D_2 \rangle$ may be written as

$$\begin{aligned} \langle D_1; c_1, c_2, \dots, c_m \rangle, \quad \langle d_1, d_2, \dots, d_k; D_2 \rangle, \quad \text{or} \\ \langle d_1, d_2, \dots, d_k; c_1, c_2, \dots, c_m \rangle. \end{aligned}$$

The notation $p : \langle D_1; D_2 \rangle$ is extended to these forms of $\langle D_1; D_2 \rangle$ in the obvious way. As an example of Definition 4.1, the net in Figure 4.8 gives

$$\langle a_1; a_2, a_4 \rangle, \langle a_1; a_2 \rangle, \langle a_1; a_4 \rangle, \langle a_2, a_4; a_3 \rangle, \langle a_2; a_3 \rangle, \langle a_4; a_3 \rangle, \langle a_3; a_1 \rangle.$$

A set of adjacency relations may be used to define an event net structure:

DEFINITION 4.2 Let $E = (P, A, T, I, O)$ be an event net structure and

$$R_E = \{ \langle D_1; D_2 \rangle \mid D_1, D_2 \subset (A \cup T), \langle D_1; D_2 \rangle \text{ is full} \}.$$

Then R_E is a relational definition of E .

The next theorem ensures that R_E defines E unambiguously.

THEOREM 4.1 R_E defines E uniquely and vice versa.

Figure 4.8: Example path relations.

Proof: Suppose R_E does not uniquely define E . Then $\exists R_{E'}$ which also defines E with $R_E \neq R_{E'}$. That is, $\exists D_1, D_2$ such that

$$\langle D_1; D_2 \rangle \in R_{E'} \quad \text{and} \quad \langle D_1; D_2 \rangle \notin R_E.$$

But since R_E and $R_{E'}$ both define E , this means $\exists p \in E$ and $\nexists p \in E$ such that $p \in O(d)$ for $d \in D_1$ and $p \in I(d)$ for $d \in D_2$. Contradiction.

Suppose E does not uniquely define R_E . Then there is another event net structure E' which also defines R_E with E' distinct from E . That is, $\exists D_1, D_2$ and place p such that $\langle D_1; D_2 \rangle$ is true for E but not for E' . But since E and E' both define R_E , this means $\langle D_1; D_2 \rangle \in R_E$ and $\langle D_1; D_2 \rangle \notin R_E$. Contradiction. $\square$

The following path relation holds for two transitions if the transitions are connected together by a token path. It is defined in terms of a composition of adjacency relations and will be used to define certain transformations.

DEFINITION 4.3 Let $E = (P, A, T, I, O)$ be an event net structure and $d_1, d_2 \in (AUT)$. Then a path relation is an ordered pair $[d_1, d_2]$ where

$$[d_1, d_2] \text{ iff } \langle d_1; d_2 \rangle \text{ or } \exists c_1, c_2, \dots, c_m \text{ such that} \\ \langle d_1; c_1 \rangle, \langle c_1; c_2 \rangle, \dots, \langle c_m; d_2 \rangle.$$

The transitions $c_1, c_2, \dots, c_m$ are called intermediate transitions of $[d_1, d_2]$ and are all distinct from one another and from d_1 and d_2 . The path relation is transitive but in general, it is not reflexive and not symmetric. However, the path relation is reflexive and symmetric

for transitions on a cycle. The term "path" is synonymous with "path relation". Notation: $d : [d_1, d_2]$ denotes d is an intermediate transition of $[d_1, d_2]$; $D : [d_1, d_2]$ denotes D is a set of intermediate transitions of $[d_1, d_2]$; $\neg d : [d_1, d_2]$ denotes d is not an intermediate transition of $[d_1, d_2]$; similarly, $\neg D : [d_1, d_2]$ denotes D is a set of transitions none of which are intermediate transitions of $[d_1, d_2]$.

For example, the net in Figure 4.8 has paths $[a_1, a_2]$, $[a_2, a_3]$, $[a_1, a_4]$, $[a_4, a_3]$, $[a_3, a_1]$, $[a_1, a_3]$, $[a_1, a_1]$, $[a_2, a_1]$, $[a_2, a_2]$, $[a_4, a_1]$, $[a_4, a_4]$, $[a_3, a_2]$, $[a_3, a_4]$, and $[a_3, a_3]$. Note that no path is traversed more than once, since intermediate transitions are distinct as required by Definition 4.3.

4.2.2 Specifying Event Net Structures with the path Relation

An Event Net Structure Generator

Given $[d_1, d_2]$, a set of paths may be defined by specifying the intermediate activities of $[d_1, d_2]$ and their order of execution. This idea is used to derive a notation for an event net structure "generator", which is a compact way of specifying event net structures that are made up of paths.

DEFINITION 4.4 *Let*

$$Y = \{d \mid d \text{ is a transition in each generated net}\} \quad (\text{compulsory transitions}),$$

$$H = \{[d_1, d_2] \mid [d_1, d_2] \text{ holds in each generated net}\} \quad (\text{compulsory paths}),$$

$$J = \{D : [d_1, d_2] \mid D : [d_1, d_2] \text{ holds in some generated net, } [d_1, d_2] \in H\}$$

(optional transitions).

Then, viewing E not as an existing event net structure but as an event net structure to be specified, $\mathcal{G} = (Y, H, J)$ is a generator of event net structures E , which is sometimes written as $\mathcal{G}(E) = (Y, H, J)(E)$. The generated event net structures are those which satisfy Y , H , and J . All transitions making up the generated structures are found in Y , H or J . A generated structure vacuously satisfies H , or J if transitions required for their satisfaction are absent from the net.

For example, consider $\mathcal{G} = (\{a_1, a_4\}, \{[a_1, a_4]\}, \{\{a_2, a_3\} : [a_1, a_4]\})$. Then $\mathcal{G}$ generates the following event net structures (Figure 4.9):

$$R_{E_1} = \{< a_1; a_4 >\},$$

$$R_{E_2} = \{ \langle a_1; a_2 \rangle, \langle a_2; a_4 \rangle \},$$

$$R_{E_3} = \{ \langle a_1; a_3 \rangle, \langle a_3; a_4 \rangle \},$$

$$R_{E_4} = \{ \langle a_1; a_2 \rangle, \langle a_2; a_3 \rangle, \langle a_3; a_4 \rangle \},$$

$$R_{E_5} = \{ \langle a_1; a_3 \rangle, \langle a_3; a_2 \rangle, \langle a_2; a_4 \rangle \},$$

$$R_{E_6} = \{ \langle a_1; a_2 \rangle, \langle a_1; a_3 \rangle, \langle a_2; a_4 \rangle, \langle a_3; a_4 \rangle \},$$

$$R_{E_7} = \{ \langle a_1; a_2, a_3 \rangle, \langle a_2; a_4 \rangle, \langle a_3; a_4 \rangle \},$$

$$R_{E_8} = \{ \langle a_1; a_2 \rangle, \langle a_1; a_3 \rangle, \langle a_2, a_3; a_4 \rangle \}.$$

Note that R_{E_1} vacuously satisfies J since no intermediate transitions are used.

Figure 4.9: Event net structures by $\mathcal{G} = (\{a_1, a_4\}, \{[a_1, a_4]\}, \{\{a_2, a_3\} : [a_1, a_4]\})$.

As another example, $\mathcal{G} = (\{a_1, a_4\}, \{[a_1, a_4], [a_2, a_3]\}, \{\{a_2, a_3\} : [a_1, a_4]\})$ generates only the event net structures (Figure 4.9):

$$R_{E_1} = \{ \langle a_1; a_4 \rangle \},$$

$$R_{E_2} = \{ \langle a_1; a_2 \rangle, \langle a_2; a_4 \rangle \},$$

$$R_{E_3} = \{ \langle a_1; a_3 \rangle, \langle a_3; a_4 \rangle \},$$

$$R_{E_4} = \{ \langle a_1; a_2 \rangle, \langle a_2; a_3 \rangle, \langle a_3; a_4 \rangle \}$$

due to the additional ordering constraint $[a_2, a_3]$. Note that R_{E_1} vacuously satisfies J since a_2 and a_3 do not appear. R_{E_1} vacuously satisfies H for $[a_2, a_3]$. Also, R_{E_2} vacuously satisfies H for $[a_2, a_3]$ and vacuously satisfies J for a_3 . R_{E_3} vacuously satisfies H for $[a_2, a_3]$ and vacuously satisfies J for a_2 . R_{E_4} satisfies Y , H , and J .

The author planned to use the event net structure generator as a filter to qualify subnets for transformation. However, this proved to be unnecessary for the transformations given in this work. The generator may be useful in future formulations of transformations or in other areas of event net specification.

Strings and Loops

Strings and loops are path structures that are useful for specifying event subnets. They are defined in terms of "starting", "ending", and "fired-first" transitions.

DEFINITION 4.5 *A transition a is a starting transition of a path if there is no other transition d in the path such that $\langle d; a \rangle$ is true. A transition z is an ending transition of a path if there is no other transition d in the path such that $\langle z; d \rangle$ is true. A transition a is a fired-first transition of a path if it is the transition in the path that is fired first in any execution cycle. A string is a path which has both a starting transition and an ending transition. A loop is a path in which $[d, d]$ is true for every transition d in the path. A loop has a fired-first transition h , such that there is a transition g in the loop with $\langle g; h \rangle$ true and place $p : \langle g; h \rangle$ contains one or more tokens. Figure 4.10 illustrates strings, loops, and their combinations.*

The concept of "parallel" paths is also useful.

DEFINITION 4.6 *Two or more paths are parallel if their starting transitions, or fired-first transitions in the case of loops, are all forked by a single common transition.*

4.3 Required Properties of an Event Net Process Specification

The transformations of this work require an event net process specification to be:

- *deadlock-free* - the specification must be deadlock-free as in Definition 3.9,

Strings:

Loop:

Combination of a String and a Loop:

Transitions a, b, and c are fired-first transitions of the corresponding loops.

Figure 4.10: Strings, loops, and combinations.

- *interaction-connected* - the specification must be interaction-connected (Definition 3.6),
- *path-structured* - the specification is made up of strings or loops joined together by arcs and places,
- *single-choiced* - where two or more transitions are forked from a place, one and only one of the forked transitions will be executed,
- *complete* - the specification must not leave out any required behaviour.

Further, as mentioned at the start of Chapter 3, all non-determinism modeled in the process specification are assumed to be "don't care" non-determinism.

Deadlock-freeness and interaction-connectedness are needed to show that transformed nets are deadlock-free. The path-structured requirement serves to define the topology of the event net process specification for compatibility with the transformations. path-structured is not overly restrictive since most Petri nets have this structure. Single-choiced rules out cases where two or more transitions forked by a place are enabled at the same time and helps to keep event nets simple. Single-choiced is also not considered to be overly restrictive since a decision between alternatives in software generally results in the selection of a single alternative.

4.4 A Formal Definition of Transformations

A transformation is formally defined as follows:

DEFINITION 4.7 *A transformation T of an event net E is a replacement of an event subnet S of E by an event subnet S' distinct from S . This replacement is carried out by first disconnecting S from E at the border transitions and then connecting up S' at either the same border transitions or at border transitions as defined by the transformation. This replacement transforms E to a new event net E' . Symbolically, $E' = T(E)$. A specific transformation R is written as T_R .*

In the following sections, transformations are defined by specifying S (through R_S) and S' (through $R_{S'}$). Subnet S is specified as a template which implicitly expresses conditions on the structure of those S for which the transformation can be applied.

The following transformation definitions use the letter "a" to represent transitions which can only be activities, and the letter "d" to represent transitions which can be

either activities, predicate transitions, or both. Each transformation definition is followed by an application example and a discussion.

4.5 Formal Definitions of Non-communication Transformations

This section gives formal definitions of the following transformations: abstraction, refinement, sequentialization, subprocess construction, and resource server (a type of resource access control transformation). A short form name for each transformation is also given in brackets next to each full name.

4.5.1 Abstraction (ABS)

Subnet S

S is any subnet with transitions d_k and d_l such that all token paths into S go into d_k and all token paths out of S go out of d_l .

Subnet S'

S' consists of a single transition d whose input arcs are the input arcs of d_k and whose output arcs are the output arcs of d_l . In addition, if d_k is a predicate transition, then d is a predicate transition whose predicate is the predicate of d_k . Transition d is said to abstract or hide subnet S .

Examples

For example, consider the event subnets of Figure 4.2. Corresponding to Figure 4.2(a), $R_S = \{ \langle a_3; a_4 \rangle, \langle a_3; a_5 \rangle, \langle a_4; a_6 \rangle, \langle a_5; a_6 \rangle \}$ with transitions a_3 and a_6 such that all token paths into S go into a_3 and all token paths out of S go out of a_6 .

Corresponding to Figure 4.2(b), S' consists of the transition a whose input arcs are the input arcs of a_3 and whose output arcs are the output arcs of a_6 .

As another example, consider Figure 4.3(a). This event net does not match the stated form for R_S and therefore cannot be abstracted. Similarly, the event net of Figure 4.3(b) cannot be abstracted.

Discussion

That tokens must flow in and out of S through d_k and d_l is required to avoid any distortion in event timing upon abstracting S . For example, suppose this condition is violated by two output arcs from different transitions within S . Then the output events corresponding to these arcs would be shown as output events of the abstracting transition d , distorting the actual output timing. In addition, this condition implies that the only transitions of S that may participate in interactions are d_k and d_l (unless some internal transition of S models an environmental component and another transition of S other than d_k or d_l interacts with this environmental component – this possibility will be ruled out by excluding transitions that model environmental components from the event net to be transformed (Chapter 6)).

The definition of S is sufficiently loose to allow it to take on many forms. For example, S may consist of parallel branches, be a subprocess, and so on.

No information is lost through abstraction. The abstracted subnet is not discarded but rather hidden from view. The abstracted subnet resides inside d and participates fully in event net execution by virtue of its connections to the rest of the net through the transitions d_k and d_l .

4.5.2 Refinement (RFM)**Subnet S**

S consists of a single transition d .

Subnet S'

S' is any subnet with transitions d_k, d_l such that

1. all token paths into d go into d_k and all token paths out of d go out of d_l (as satisfied by S in the abstraction transformation);
2. if d is a predicate transition, then d_k is a predicate transition whose predicate is the predicate of d (as in the abstraction transformation);
3. S' is interaction-connected;
4. if d_k fires, d_l will be enabled;

Example

As an example, consider again Figure 4.2. In Figure 4.2(a), S' defined by

$$R_{S'} = \{ \langle a_3; a_4 \rangle, \langle a_3; a_5 \rangle, \langle a_4; a_6 \rangle, \langle a_5; a_6 \rangle \}$$

is a refinement of S consisting of transition a in Figure 4.2(b). S' has transitions a_3 and a_6 such that all token paths into a go into a_3 and all token paths out of a go out of a_6 .

Discussion

As in the abstraction transformation, d_k and d_l are the only transitions which may participate in interactions, in which case the interactions are those belonging to d . If other transitions of S' were allowed to participate in interactions, such interactions would be either i) new or ii) belong to d . Case i) would upset any behaviour correctness based on having the same interactions for both the pre-transformed and the transformed net. Case ii) would upset the relative timing of input/output events as explained above for the abstraction transformation.

Qualitatively, the refinement transformation is the reverse of the abstraction transformation and vice versa but some refinements cannot be abstracted. If S qualifies for the abstraction transformation and at the same time qualifies as S' for the refinement transformation, then $S = T_{RFM}(T_{ABS}(S))$. Similarly, if S qualifies for the refinement transformation and at the same time qualifies as S' for the abstraction transformation, then $S = T_{ABS}(T_{RFM}(S))$.

The conditions that if d_k fires, d_l will be enabled, and that $R_{S'}$ be interaction-connected, are needed to prove preservation of behaviour correctness and deadlock-freeness, respectively.

If E is interaction-connected, a way to ensure that S' is interaction-connected is to verify that every transition d' in S' is required to fire in order for d_l to be enabled. In other words, $[d', d_l]$ must be true for every transition d' in S' . For then, if d_l is an interaction transition, S' would be interaction-connected by definition. If d_l is a non-interaction transition, the definition of T_{RFM} , together with the fact that E' is cyclic, imply $\exists$ transition c in E' such that d_l is required to fire in order for c to be enabled. That is, d' has to fire for c to be enabled. Then if c is an interaction transition S' is interaction-connected by definition. If c is a non-interaction transition, E interaction-connected (as

a requirements specification or through a previous transformation) implies $\exists$ interaction transition c' in E' such that c must fire in order for c' to be enabled. That is, d' must fire in order for c' to be enabled and again S' is interaction-connected by definition.

4.5.3 Sequentialization (SEQ)

The sequentialization transformation replaces subnet S , consisting of paths, some of which may be parallel, with a subnet S' , consisting of paths, none of which are parallel. Some of the paths of S may be loops, in which case there are corresponding loops in S' . Subnet S' is constructed using transitions and ordering information from subnet S , such that the firing order of transitions, as specified by S , is automatically maintained.

Subnet S

The subnet S is made up of strings or loops. paths in S with distinct starting transitions are assumed parallel. If S consists of a number of unconnected paths, such paths are assumed to be parallel. Since SEQ results in a S' in which activities execute once per execution of S' , activities in S must execute once per execution of S (otherwise, certain activities in S' may not execute the required number of times - see "Discussion" subsection below). Examples of S are given below.

Subnet S'

Subnet S' is obtained as follows. First, subnet S_1 is obtained from subnet S by cutting or abstracting all loops in S (including those that are part of strings); S_1 then consists of only strings, no part of which are loops. Next, subnet S_2 is constructed from S_1 using the sequentialization algorithm below, which results in S_2 consisting of paths, none of which are parallel. Finally, subnet S' is obtained from S_2 by re-connecting any loops that were cut in obtaining S_1 .

Obtaining S_1 from S Construct the subnet S_1 as follows: i) set $R_{S_1} = R_S$, $R_p = \phi$ (R_p is a repository for adjacency relations temporarily removed from loops due to cutting); ii) for every loop in S_1 , either abstract the loop into a single transition using the abstraction transformation, or cut the loop by setting $R_{S_1} = R_{S_1} \setminus \{ \langle g; h \rangle \}$, $R_p = R_p \cup \{ \langle g; h \rangle \}$, where g and h are transitions in the loop such that $\langle g; h \rangle$ holds and h is fired-first.

This choice of removing $\langle g; h \rangle$ for a cut allows h to be fired before any other transition of the loop in S_2 , thereby retaining the intended loop firing order found in S . In other words, the removal of some other $\langle e; f \rangle$ distinct from $\langle g; h \rangle$ may distort this intended loop firing order. That h is fired-first is indicated by the presence of one or more tokens in place $p : \langle g; h \rangle$, by definition of a loop.

The decision to abstract or cut a loop depends on whether or not it is desired to retain the existing loop activity execution succession. If this retention is desired, one would abstract. Such retention may be important for performance reasons, where loop activities are required to execute in quick succession, without some non-loop activity sequenced into the firing order.

It is useful to describe S_1 formally. S_1 is characterized by a set $A = \{a_1, a_2, \dots, a_m\}$ of starting transitions and a set $Z = \{z_1, z_2, \dots, z_n\}$ of ending transitions, $A \cap Z = \emptyset$, such that S_1 satisfies the path relations $[a, z]$ for all $a \in A$ and all $z \in Z$ and

- for every transition d of S_1 , either i) $d \in A$, ii) $d \in Z$, or iii) d is an intermediate transition of some path $[a, z]$, $a \in A$, $z \in Z$;
- for any transition d in S_1 , there is no path $[d, d]$ in S_1 .

Obtaining S_2 from S_1 The sequentialization algorithm constructs S_2 by copying the transition firing orders given in S_1 , so that S_2 automatically maintains the same transition firing order as specified by S_1 . The construction begins by the user selecting a starting transition a_s (based on design goals). The subnet S_2 is then built-up by adding adjacency relations $\langle D; F \rangle$ to R_{S_2} , where D is a subset of K_D (with initial element a_s) and F is a subset of K_F (with initial elements in $A \setminus \{a_s\}$). The user chooses D and F based on design concerns and certain constraints given in Step 3 of the algorithm, which ensure that correct transition firing order is maintained when transitions are moved in and out of event net branches. The sets K_D and K_F are updated after every addition to R_{S_2} : transitions move from K_F to K_D and new transitions enter K_F via a set L which contains the next transitions f that may be fired, after each transition d . The next transitions f are those for which $\langle d; f \rangle$ holds in S_1 . A transition d of a subnet U is a *join transition* iff $\exists$ transition sets $D_1, D_2, \dots, D_k$ in U such that $\langle D_1; d \rangle, \langle D_2; d \rangle, \dots, \langle D_k; d \rangle$ holds in U , $k \geq 2$. A useful feature of the sequentialization algorithm is that it allows the user to build legal alternative transition firing sequences by re-sequencing transitions of S_1 which connect

to join transitions (Step 5). This feature is important to allow alternative sequences of activities in which a particular sequence is triggered by the occurrence of certain input events. Example 1 below illustrates this aspect of the algorithm. Finally, the algorithm stops when all transition, in S_1 have been transferred to S_2 , i.e., when $K_F = L = \phi$.

The resulting S_2 consists of a set of strings $[a, z]$, none of which are parallel, and no part of which are loops, where $a \in A_1$, $A_1 \subseteq A$, $z \in Z_1$, and $Z_1 \subseteq Z$.

Sequentialization Algorithm

Step 1 : Choose a starting transition $a_s \in A$ and set $K_D = \{a_s\}$, $K_F = A \setminus \{a_s\}$, $R_{S_2} = \phi$.

Step 2 : Choose $D \subset K_D$ (to extend R_{S_2}). Set

$$L = \{f \mid \langle d, f \rangle \text{ holds in } S_1, d \in D, f \text{ in } S_1\}$$

$$K_F = K_F \cup L$$

Step 3 : Choose $F \subseteq K_F$ subject to *all* of the following *constraints*:

- (a) : if F consists of more than one transition, then the transitions of F are all predicate transitions, and $\exists$ set B of transitions in S_1 such that $\langle B; F \rangle$ holds in S_1 (see Figure 4.11(a));
- (b) : if F consists of a single transition f , then
 - (i) : f is not a predicate transition;
 - (ii) : if e is a predicate transition such that $[e, f]$ holds in S_1 , then $[e, f]$ holds in S_2 ;
 - (iii) : if f is a join transition, then for any transition d such that $[d, f]$ holds in S_1 , $[d, f]$ also holds in S_2 ; further, if f joins choice branches starting with predicate transitions in some set C in S_1 , f joins choice branches starting with the same set of predicate transitions C in S_2 (see Figure 4.11(b));
- (c) : Introduction of new transitions into a choice branch (see Figure 4.11(c)): Suppose new transitions $f \in F$ have been moved into a choice path starting with predicate transition e_1 . Then the transitions f must be in every choice path that is alternate to the choice path starting with e_1 . Formally, for any F , suppose e_1 is a predicate transition such that $[e_1, f]$ holds in S_2 for every

$f \in F$ but $[e_1, f]$ does not hold in S_1 for any $f \in F$. Let $e_1 \in C$ where C is a set of predicate transitions specifying alternative paths from a place. Then the following constraint must be satisfied prior to the completion of S_2 (meaning it may take several passes of Step 3 to satisfy it): for each $e_k \in C$, $[e_k, f]$ holds in S_2 for all $f \in F$. Further, $[f, f]$ is always false in S_2 for any $f \in F$ (this guards against putting 2 or more copies of f on the same choice path).

If no F is possible that satisfies constraints (a) to (c) above, set $K_F = K_F \setminus L$ and go to step 2 to choose a different D .

Step 4 : Set $R_{S_2} = R_{S_2} \cup \{ \langle D; F \rangle \}$, $K_D = (K_D \setminus D) \cup F$. If constraint 3(c) is applicable and is not yet satisfied, then:

(a) : if f occurs in K_F as f rather than $f.i$, set $i = 1$ and append i to f in K_F so that f in K_F becomes $f.i$; this notation will distinguish multiple occurrences of f in S_2 , in order to avoid confusion in the construction of R_{S_2} (N.B.: for verifying that $[f, f]$ is false in Step 3(c), $f.i$ is understood to be f); go to step 2.

(b) : if f occurs in K_F as $f.i$, increment i by 1 and go to step 2.

Otherwise (i.e., constraint 3(c) is not applicable or is already satisfied), set $K_F = K_F \setminus F$.

Step 5 : If constraint 3(b)(iii) is applicable, i.e., f is a join transition, repeat the following substeps as many times as desired (including not at all), in order to generate alternative paths terminating at f :

(a) : if this is the first repetition, set $D_0 = D$ as obtained in step 2; set $j = 0$;

(b) : set $j = j + 1$ and re-do steps 1 to 4 using the set of transitions already in S_2 to obtain a new set of paths terminating at the same join transition f ; in this procedure, append/replace (j) to/for each transition in the new set of paths before adding it to R_{S_2} so that a transition d or $d(j - 1)$ becomes $d(j)$;

(c) : this substep updates the connection of alternative paths to the join transition: let $\langle D; F \rangle$ be the relation just added to R_{S_2} in step 4 of the current j -th

Figure 4.11: Illustrations of step 3 constraints.

repetition of these substeps, in which F consists of the join transition f and $j \geq 1$; set

$$D_j = D,$$

$$R_{S_2} = R_{S_2} \setminus \{ \langle D_{j-1}; F \rangle, \langle D_j; F \rangle \},$$

$$D_j = D_j \cup D_{j-1},$$

$$R_{S_2} = R_{S_2} \cup \{ \langle D_j; F \rangle \}.$$

Step 6 : (Stopping Condition) If $K_F \neq \phi$, go to step 2. Otherwise, set

$$L = \{ f \mid \langle d; f \rangle \text{ holds in } S_1, d \in K_D, f \text{ in } S_1 \}.$$

If $L \neq \emptyset$, go to step 2. Otherwise, S_2 is complete. STOP.

If step 5 was done, the K_F and K_D obtained in the last iteration of steps 1 to 4 are the current K_F and K_D .

Obtaining S' from S_2 Construct the subnet S_2 as follows: i) set $R_{S'} = R_{S_2} \cup^* R_p$, where $\cup^*$ means that for every $\langle g; h \rangle \in R_p$, $\langle g^*; h^* \rangle$ is also added to the union for every $g^* = g.i$ or $g(l)$, $h^* = h.i$ or $h(l)$ found in S_2 (g^* and h^* arise from Steps 4 and 5 of the sequentialization algorithm); ii) Add the same number of tokens to every place $p : \langle g; h \rangle$ or $p : \langle g^*; h^* \rangle$ as found in place $p : \langle g; h \rangle$ of S .

Examples

Some examples of applying the SEQ transformation follow.

First Example Figure references in this example are highlighted by enclosing them in boxes.

Obtaining S_1 from S Figure 4.12 shows subnet S for the SEQ transformation.

The loop in S has transition 6 as fired-first transition; therefore, the loop is cut by removing $\langle 9; 6 \rangle$ from R_S , giving the subnet S_1 shown in Figure 4.13(a). $R_p = \{ \langle 9; 6 \rangle \}$.

Obtaining S_2 from S_1 Applying the sequentialization algorithm to S_1 :

Step 1 : $A = \{1, 6\}$, user chooses $a_s = 1$, $K_D = \{1\}$, $K_F = \{6\}$, $R_{S_2} = \phi$.

2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Figure 4.12: Subnet S for SEQ in First Example.

Step 2 : user “chooses” (in this case, really no choice) $D = \{1\}$, $L = \{2\}$, $K_F = \{6, 2\}$.

Step 3 : user chooses $F = \{6\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 1; 6 >\}$, $K_D = \{6\}$, $K_F = \{2\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$

Step 2 : user “chooses” $D = \{6\}$, $L = \{7\}$, $K_F = \{2, 7\}$.

Step 3 : user chooses $F = \{2\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 6; 2 >\}$, $K_D = \{2\}$, $K_F = \{7\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user “chooses” $D = \{2\}$, $L = \{3, 5\}$, $K_F = \{7, 3, 5\}$.

Step 3 : user chooses $F = \{7\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 2; 7 >\}$, $K_D = \{7\}$, $K_F = \{3, 5\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user “chooses” $D = \{7\}$, $L = \{8\}$, $K_F = \{3, 5, 8\}$.

Step 3 : user chooses $F = \{3, 5\}$, constraints satisfied.

(a) S_1 for sequentialization algorithm.

(b) Partial S_2 after encountering transition 9.

(c) Subnet obtained by re-doing steps 1 to 4.

Figure 4.13: Applying the sequentialization algorithm to S_1 in First Example.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 7; 3, 5 >\}$, $K_D = \{3, 5\}$, $K_F = \{8\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{3\}$, $L = \{4\}$, $K_F = \{8, 4\}$.

Step 3 : user chooses $F = \{8\}$, for $d = e = 3$, $[3, 8]$ does not hold in S_1 ; constraint 3(c) is applicable and prior to the completion of S_2 , $[5, 8]$ must hold in S_2 .

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 3; 8 >\}$, $K_D = \{5, 8\}$, constraint 3(c) is applicable and is not yet satisfied; therefore $K_F = \{8.1, 4\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{5\}$, $L = \{9\}$, $K_F = \{8.1, 4, 9\}$.

Step 3 : user chooses $F = \{9\}$, transition 9 is a join transition, constraint 3(b)(iii) (second half) is violated.

Step 3 : user chooses $F = \{4\}$, constraint 3(b)(ii) is violated since this choice of F will make it impossible for $[3, 4]$ to hold in S_2 .

Step 3 : user chooses $F = \{8.1\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 5; 8.1 >\}$, $K_D = \{8, 8.1\}$, constraint 3(c) is satisfied for $f = 8$, $K_F = \{4, 9\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{8\}$, $L = \{9\}$, $K_F = \{4, 9\}$.

Step 3 : user chooses $F = \{9\}$, transition 9 is a join transition, constraint 3(b)(iii) (second half) is violated.

Step 3 : user chooses $F = \{4\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 8; 4 >\}$, $K_D = \{8.1, 4\}$, $K_F = \{9\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{8.1\}$, $L = \{9\}$, $K_F = \{9\}$.

Step 3 : user "chooses" $F = \{9\}$, transition 9 is a join transition, constraint 3(b)(iii) (second half) is violated. It is not possible to choose a different F to satisfy all constraints of step 3. Therefore $K_F = \phi$ and step 2 is now re-done:

Step 2 : user chooses $D = \{4\}$, $L = \{9\}$, $K_F = \{9\}$.

Step 3 : user "chooses" $F = \{9\}$, transition 9 is a join transition, constraint 3(b)(iii) (second half) is violated. Again, it is not possible to choose a different F to satisfy all constraints of step 3. Therefore $K_F = \phi$ and step 2 is now re-done:

Step 2 : user chooses $D = \{8.1, 4\}$, $L = \{9\}$, $K_F = \{9\}$.

Step 3 : user "chooses" $F = \{9\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 8.1, 4; 9 >\}$, $K_D = \{9\}$, $K_F = \phi$. At this point the subnet S_2 appears as shown in Figure 4.13(b).

Step 5(a) : $D_0 = \{8.1, 4\}$, $j = 0$ (user chooses Step 5 to add alternative sequential paths terminating at join transition 9).

Step 5(b) : $j = 1$. Suppose steps 1 to 4 above are re-done, resulting in the subnet shown in Figure 4.13(c).

Step 5(c) : $D = \{8(1), 8.1(1)\}$, $F = \{9\}$, and $< 8(1), 8.1(1); 9 >$ was just added to R_{S_2} in step 4 of the current 1st repetition of the substeps. Then

$$D_1 = D = \{8(1), 8.1(1)\}$$

$$R_{S_2} = R_{S_2} \setminus \{< 8.1, 4; 9 >, < 8(1), 8.1(1); 9 >\}$$

$$D_1 = \{8(1), 8.1(1), 8.1, 4\}$$

$$R_{S_2} = R_{S_2} \cup \{< 8(1), 8.1(1), 8.1, 4; 9 >\}$$

At this point, steps 5(b) and 5(c) may be repeated but the user chooses to stop here. The subnet S_2 now appears as shown in Figure 4.14(a).

Step 6 : $K_F = \phi$, $K_D = \{9\}$, $L = \{10, 11\} \neq \phi$.

Step 2 : user "chooses" $D = \{9\}$, $L = \{10, 11\}$, $K_F = \{10, 11\}$.

Step 3 : user chooses $F = \{10\}$, constraints satisfied.

(a) Partial S_2 after adding a new set of alternative paths.

(b) Completed S_2 .

Figure 4.14: Applying the sequentialization algorithm to S_1 in First Example - Completed.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 9; 10 >\}$, $K_D = \{10\}$, $K_F = \{11\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user "chooses" $D = \{10\}$, $L = \{12, 13\}$, $K_F = \{11, 12, 13\}$.

Step 3 : user chooses $F = \{11\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 10; 11 >\}$, $K_D = \{11\}$, $K_F = \{12, 13\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user "chooses" $D = \{11\}$, $L = \{14, 15\}$, $K_F = \{12, 13, 14, 15\}$.

Step 3 : user chooses $F = \{12, 13\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 11; 12, 13 >\}$, $K_D = \{12, 13\}$, $K_F = \{14, 15\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{12\}$, $L = \{16\}$, $K_F = \{14, 15, 16\}$.

Step 3 : user chooses $F = \{14, 15\}$, for $d = e = 12$, $[12, 14]$ and $[12, 15]$ do not hold in S_1 ; constraint 3(c) is applicable and prior to the completion of S_2 , $[13, 14]$ and $[13, 15]$ must hold in S_2 .

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 12; 14, 15 >\}$, $K_D = \{13, 14, 15\}$, constraint 3(c) is applicable and is not yet satisfied. Therefore $K_F = \{14.1, 15.1, 16\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{14\}$, $L = \phi$, $K_F = \{14.1, 15.1, 16\}$.

Step 3 : user chooses $F = \{14.1\}$, violates constraint 3(b)(i) since 14 is a predicate transition.

Step 3 : user chooses $F = \{15.1\}$, violates constraint 3(b)(i) since 15 is a predicate transition.

Step 3 : user chooses $F = \{14.1, 15.1\}$, violates constraint 3(c) since $[14, 14]$ cannot hold in S_2 .

Step 3 : user chooses $F = \{16\}$, transition 16 is a join transition, constraint 3(b)(iii) (second half) is violated. It is not possible to choose a different F to satisfy all constraints of step 3. Therefore $K_F = \{14.1, 15.1, 16\}$ and step 2 is now re-done:

Step 2 : user chooses $D = \{13\}$, $L = \{16\}$, $K_F = \{14.1, 15.1, 16\}$.

Step 3 : user chooses $F = \{16\}$, transition 16 is a join transition, constraint 3(b)(iii) (second half) is violated.

Step 3 : user chooses $F = \{14.1, 15.1\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 13; 14.1, 15.1 >\}$, $K_D = \{14, 15, 14.1, 15.1\}$, constraint 3(c) is satisfied, therefore $K_F = \{16\}$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F \neq \phi$.

Step 2 : user chooses $D = \{14, 15, 14.1, 15.1\}$, $L = \phi$, $K_F = \{16\}$.

Step 3 : user "chooses" $F = \{16\}$, constraints satisfied.

Step 4 : $R_{S_2} = R_{S_2} \cup \{< 14, 15, 14.1, 15.1; 16 >\}$, $K_D = \{16\}$, $K_F = \phi$.

Step 5 : constraint 3(b)(iii) is not applicable.

Step 6 : $K_F = \phi$, $L = \phi$. S_2 is complete, STOP. The completed S_2 is shown in Figure 4.14(b).

Obtaining S' from S_2 Since $R_p = \{< 9; 6 >\}$, and transition 6(1) is also in S_2 , $R_{S'} = R_{S_2} \cup \{< 9; 6 >, < 9; 6(1) >\}$. Since place $p : < 9; 6 >$ in S contains a single token, a single token is added to place $p : < 9; 6 >$ and place $p : < 9; 6(1) >$ in S' . Figure 4.15 shows S' .

More Examples Figure 4.16 shows subnets S and S' for two other applications of SEQ in which S consists of strings without any loops. Figure 4.17 shows subnets S , S_1 , S_2 , and S' for an application of SEQ in which S consists of two unconnected paths (assumed parallel) where one path is a loop and the other path a string.

Discussion

The subnet S is not required to contain any parallel paths, so that S need not be checked for parallel paths before SEQ can be applied. In any case, the sequentialization algorithm

Figure 4.15: Subnet S' by SEQ in First Example.

allows the construction of extra alternative paths, which may be useful even if S does not have any parallel paths.

To illustrate the requirement that activities in S must execute once per execution of S , consider a loop and a string forked by the same transition. If in an execution of S , the loop executes twice for every single execution of the string, using SEQ to combine them into a single sequence will force loop activities to execute once per execution of the sequentialized subnet, contrary to requirements. Such a situation may be handled by abstracting (using the ABS transformation) the activity or activities that execute more than once per execution of S into a single transition. The abstracted activities would then still execute their required number of times in the sequentialized subnet. In the illustration just given, the loop would be abstracted into a single transition.

Suppose the sequentialization algorithm changes the relations $\langle d_1; d_2 \rangle$ and $\langle d_3; d_4 \rangle$ into the relations $\langle d_1; d_3 \rangle$, $\langle d_3; d_2 \rangle$, and $\langle d_2; d_4 \rangle$ (e.g., by combining two parallel paths into a single path, see Figure 4.18). Then any output from d_1 to d_2 in the old structure is assumed to be carried by the token which flows from d_1 to d_2 through d_3 in

a)

b)

Figure 4.16: More examples of SEQ.

Figure 4.17: Yet another example of SEQ.

the new structure. Similarly, any output from d_3 to d_4 in the old structure is assumed to be carried by the token which flows from d_3 to d_4 through d_2 in the new structure, and this token is the one referred to earlier as flowing from d_1 to d_2 through d_3 . In general, this illustrates a situation that can occur whenever transitions are re-sequenced. In the new structure, it is assumed that any data required by a consuming transition will be carried to it by the token that flows from the producing transition to the consuming transition, even if this token is required to flow through new intermediate transitions.

Figure 4.18: Using a token to carry data through new intermediate transitions.

In steps 1 to 3 of the sequentialization algorithm, the user exercises design freedom by choosing a_s , D , and F , respectively, to reflect design goals. For example, if the user is able to choose F to satisfy the constraints of step 3, and at the same time, F gives better throughput by way of some output being available sooner, a design goal of better performance will have been partly achieved. The choices arise out of the fact that S_1 does not specify the relative firing order of certain transitions participating in parallel event net constructs, e.g., two transitions from different concurrent paths.

Step 3 of the sequentialization algorithm ensures that if any $[d_1, d_2]$ is true in S_1 , $[d_1, d_2]$ is also true in S_2 . Thus the sequentialization algorithm preserves the path relations of S_1 in S_2 . In fact, the SEQ transformation preserves the path relations of E in E' , which may be seen as follows: Consider any $[d_1, d_2]$ that holds in E and define $\bar{S}$ by $R_{\bar{S}} = R_E \setminus R_S$.

There are three cases to examine depending on the location of d_1 and d_2 in E : (i) if $[d_1, d_2]$ holds in S , then $[d_1, d_2]$ holds in E' , since

$$\begin{aligned} \{\text{path relations of } S_1\} &= \{\text{path relations of } S\} \\ &\quad \setminus \{\text{path relations of } S \text{ using adjacency relations in } R_p\}, \\ \{\text{path relations of } S_2\} &= \{\text{path relations of } S_1\}, \\ \{\text{path relations of } S'\} &= \{\text{path relations of } S_2\} \\ &\quad \cup \{\text{path relations of } S \text{ using adjacency relations in } R_p\}, \end{aligned}$$

and the result follows by algebraic substitution; (ii) if $[d_1, d_2]$ holds in $\bar{S}$, then again $[d_1, d_2]$ holds in E' , since SEQ does not affect the structure of $\bar{S}$; (iii) if d_1 lies in $\bar{S}$ and d_2 lies in S , there is a border transition b belonging to both $\bar{S}$ and S such that $[d_1, b]$ holds in $\bar{S}$ and $[b, d_2]$ holds in S . Then $[d_1, b]$ holds in E' since SEQ does not affect the structure of $\bar{S}$, and $[b, d_2]$ holds in E' from part (i) of this argument. Then by transitivity of the path relation, $[d_1, d_2]$ holds in E' .

There are at least as many alternative paths in S' as there are in S . This follows from the sequentialization algorithm, specifically i) the retention of choice places and their associated predicate transitions, and ii) the step 5 re-sequencing of transitions in concurrent paths which meet at a join transition, giving new alternative paths.

Step 4 of the sequentialization algorithm is a transition identification method which allows transitions to be duplicated in S_2 by giving them unique identifiers. Otherwise, S_2 's relational definition would define it to be quite different from what was intended.

The sequentialization transformation necessarily reduces concurrency. This reduction in itself may be detrimental to performance but on the other hand, the resulting new sequence of events may improve performance. If reduction of concurrency is unacceptable, then perhaps the subprocess construction transformation, which can retain or increase parallelism, should be considered rather than the sequentialization transformation.

It is important to note that S' does not automatically have a sequential execution trace – this depends on its initial marking. Subnet S' may be marked in such a way that two or more transitions are enabled at any one time, resulting in parallel execution traces. However, S' 's execution trace is sequential if i) it is initially unmarked (except for input places to fired-first transitions in loops), and ii) after a starting or fired-first transition is enabled, no other starting or fired-first transition can be enabled until after an ending

transition has fired or a return to the initial state in case S' has no ending transition (e.g., S' is a loop).

4.5.4 Subprocess Construction (SPC)

Subnet S

S is the same as for the SEQ transformation.

Subnet S'

The subnet S' is obtained by applying the following procedure:

Step 1 : Apply the following parts of the SEQ transformation to S : i) obtain S_1 from S ,
ii) obtain S_2 from S_1 .

Subnet S_2 is described as follows: Let $A_1 \subseteq A$ and $Z_1 \subseteq Z$. Then S_2 satisfies the path relations $[a, z]$ for all $a \in A_1$ and all $z \in Z_1$ such that

- (a) : for every transition d of S_2 , either i) $d \in A_1$, ii) $d \in Z_1$, or iii) d is an intermediate transition of some path $[a, z]$, $a \in A_1$, $z \in Z_1$;
- (b) : the path relations $[a, z]$ for all $a \in A_1$ and all $z \in Z_1$ define alternative execution paths in that an execution cycle of S_2 traces out one and only one such path;
- (c) : at most one transition is enabled in S'' .

Step 2 : Set $R_{S_2} = R_{S_2} \cup \{ \langle Z_1; A_1 \rangle \}$.

Step 3 : Check if any transitions are enabled in S_2 to determine if place $p : \langle Z_1; A_1 \rangle$ should be marked. If two or more transitions are enabled in S_2 , an error has occurred; re-do this transformation and correct the error. If one transition is enabled in S_2 , leave place $p : \langle Z_1; A_1 \rangle$ unmarked. If no transitions are enabled in S_2 , add a token to place $p : \langle Z_1; A_1 \rangle$.

Step 4 : Apply the final part of the SEQ transformation: obtain S' from S_2 .

Examples

Figure 4.12 is an example subnet S for subprocess construction. Figures 4.13(a) and 4.20(b) show S_1 and S_2 , respectively, following Step 1 of SPC, using the S of Figure 4.12. Figure 4.19 shows the corresponding subprocess S' following Steps 2, 3 and 4 of SPC.

Figure 4.19: Example subprocess constructed from S in First Example.

As another example, consider S in Figure 4.17 as a candidate for subprocess construction. Figure 4.17 also shows S_1 and S_2 due to Step 1 of SPC. Figure 4.20 shows the corresponding subprocess S' following Steps 2, 3 and 4 of SPC.

Figure 4.20: Example subprocess from S containing a loop.

In both of these examples, notice that a single token was added to place $p : \langle Z_1; A_1 \rangle$, according to Step 3 of SPC.

Discussion

The subnet S may already be sequential, in which case SEQ may not be necessary. However, specifying that SEQ be applied in all cases avoids the need to check if S is indeed sequential. Also, the sequentialization algorithm allows the construction of extra alternative paths, even if S is already sequential.

The addition of $\langle Z_1; A_1 \rangle$ to R_{S_2} threatens to change the sequence of transition firings of S_2 and thereby affect behavioural correctness. However, the addition does not affect the sequence of firings. The only possible effect of the addition is to delay the re-execution of some transition in A_1 . Adding $\langle Z_1; A_1 \rangle$ to R_{S_2} allows S' to cycle only after the last transition in a sequence of transition firings has fired, without affecting the required firing orders.

If S_2 has one enabled transition, it suffices to leave place $p : \langle Z_1; A_1 \rangle$ unmarked, since the firing of the enabled transition will eventually put a token in p . If S_2 has no enabled transitions, it is necessary to have a token in p so that some starting or fired-first transition may eventually be enabled.

4.5.5 Resource Server (RSV)

The resource server transformation replaces a subnet S , which models the sharing of a single resource between several transitions, with a subnet S' , which models the same sharing of the single resource but at a lower level of detail in terms of a server process controlling the sharing.

Subnet S

The subnet S is labeled as R_S in Figure 4.21, where the single token in place p models the resource, and the transitions sharing the resource are identified as transitions d_i . In terms of adjacency relations, S is defined as (refer to Figure 4.21):

$$R_S = \{ \langle d_1, d_2, \dots, d_n; d_1, d_2, \dots, d_n \rangle \}, n \geq 2,$$

with one token in place p , where $p : \langle d_1, d_2, \dots, d_n; d_1, d_2, \dots, d_n \rangle$.

Subnet S'

The subnet S' is labeled as $R_{S'}$ in Figure 4.21, where place p_1 replaces p as the resource place containing the resource token. The server process consists of activity transitions a_1 , a_2 , and predicate transitions $t_1, t_2, \dots, t_n$. Transition a_1 governs the return of a resource. Transition a_2 allocates a resource. Predicate transitions t_i route a resource grant to a requestor based on the requestor ID carried by a request token. Also, each transition d_i of S has been refined into two transitions, namely $d_{i,1}$ and $d_{i,2}$, where $d_{i,1}$ requests access to the resource, and $d_{i,2}$ receives access to the resource. Once transition $d_{i,2}$ finishes with the resource, it sends the server process a "finished access" message. In terms of adjacency relations, S' is defined as (refer to Figure 4.21):

$$\begin{aligned}
 R_{S'} &= \bigcup_{i=1}^4 A_i \text{ with} \\
 A_1 &= \bigcup_{i=1}^n \{ \langle d_{i,1}; d_{i,2} \rangle \}, \\
 A_2 &= \{ \langle a_2; a_1 \rangle, \langle a_1; a_2 \rangle \}, \\
 A_3 &= \{ \langle d_{1,1}, d_{2,1}, \dots, d_{n,1}; a_2 \rangle, \langle a_2; t_1, t_2, \dots, t_n \rangle, \\
 &\quad \langle d_{1,2}, d_{2,2}, \dots, d_{n,2}; a_1 \rangle \}, \\
 A_4 &= \bigcup_{i=1}^n \{ \langle t_i; d_{i,2} \rangle \}
 \end{aligned}$$

where place p_1 contains a single token, $p_1 : \langle a_1; a_2 \rangle$, and $\{ \langle d_{i,1}; d_{i,2} \rangle \}$ is a refinement of d_i , $1 \leq i \leq n$.

Example

This transformation is applied in the telephone switch example of Chapter 7.

Discussion

The resource server transformation may be used to model mutually exclusive access to both logical (e.g., file) and hardware resources (e.g., printer). Each d_i in S may belong to a separate subprocess, be part of the same subprocess along with a proper subset of other d_i 's, or be part of the same subprocess along with all other d_i 's. However, the resource server transformation is best applied where each d_i belongs to a separate subprocess, since it is this case which is most likely to require control of concurrent access.

Figure 4.21: The resource server transformation.

For each i , $d_{i,1}$ outputs a request token to the server process. If the resource is available, predicate transition t_i returns a “permission-to-access” token to $d_{i,2}$. Predicate transition t_i evaluates its predicate based on a requester ID which is part of the request token. The predicate for t_i contains the requestor ID of the requestor to which it connects and evaluates to “true” if there is a match-up between this requestor ID and the requester ID in the request token. If the resource is not available, request tokens arriving at place p_2 join a FIFO queue, i.e., resource requestors are served on a first-come-first-served policy.

The transitions d_i in S and $d_{i,1}, d_{i,2}$ in S' are the only transitions which may participate in interactions, i.e., the transitions of the server process do not participate in interactions, in order that behaviour correctness may be based on identical interactions between requirements specification and proto-design (Chapter 5).

The fact that for each i , $\langle d_{i,1}; d_{i,2} \rangle$ in S' is a refinement of d_i in S means that all input events of d_i in S become input events of $d_{i,1}$ in S' and all output events of d_i in S become output events of $d_{i,2}$ in S' .

The places p (in S) and p_1 (in S') may model $n \geq 1$ shared resources by each holding

n resource tokens, provided that each requestor does not care which of the n resources it accesses. For example, the n resources may be n printers, where printing may be done on any printer. In this case, each printer resource token would contain the corresponding printer ID so that the requestor could tell the user where to pick up the printed output. If the requestor does care which resource it accesses, S and S' would be modified to allow each requestor to have a predicate transition, which would select the required resource token from the n resource tokens, based on a resource type ID in each resource token.

4.6 Formal Definitions of Communication Transformations

4.6.1 Introduction

Communication Styles

Software processes which execute separately and which together make up a system need to inter-communicate for the purpose of information exchange and activity coordination. A software system may be compared to a human organization. In order to achieve the organization's goals, the people (processes) in the organization have to exchange information and coordinate work by talking to one another.

In recognition of the need for inter-process communication, certain computer operating systems and languages which support multi-processing, also support various inter-process communication (IPC) forms or styles. For example, UNIX/C supports sockets, processes in a known proprietary switch operating system/language communicate using letters and mailboxes, and Ada is well known for its rendezvous. Rendezvous communication is a synchronous style in which either sender or receiver may be blocked until both processes are ready to participate in the communication (i.e., sender sends if the receiver is waiting to receive and receiver receives if the sender is waiting to send) at which point both processes are synchronized in their execution and are said to be in rendezvous. Mailboxes are asynchronous in the sense that the sender places a message (letter) in the recipient's mailbox without having to synchronize with the receiver picking up the message. The receiver blocks on its mailbox if the mailbox is empty. Asynchronous styles such as the use of mailboxes may be described simply as "Send-No-Wait/Receive-Wait" (i.e., non-blocking send with blocking receive) styles. Other asynchronous styles include the use of transporter or courier processes, and the use of buffer processes. A description of IPC

styles may be found in Andrews and Schneider [1], and Gentleman [17].

Communication styles may incorporate acknowledgements, here abbreviated to "acks". There are basically two types of acks: "pure acknowledgement acks" and "data acks". A pure acknowledgement ack is a simple acknowledgement of receipt. A data ack not only acknowledges receipt but also represents requested data such as the results of some computation. For example, the event "output $f(x)$ " can be a data ack to the event "input x ". Computed results passed back in an Ada rendezvous may be considered data acks; however, an Ada rendezvous may also be programmed to use pure acknowledgement acks.

Communication Transformations

Communication transformations are used to implement event net models of the above communication styles between subprocesses. There is a one-to-one correspondence between communication transformation and communication style. The choice of which communication style or transformation to use depends on the intended target language or operating system. Thus, if the user is designing software in Ada and intends to use the Ada rendezvous as the IPC style, communication transformations that install event net models of the rendezvous would be used. If on the other hand, the user is working in an environment that employs letters and mailboxes for IPC, transformations that install event net models of mailboxes would be used. If appropriate, the user may also use a mixture of styles in one system. For example, the user may choose communication transformations that install rendezvous and mailboxes for an Ada system.

The communication styles modeled in this work assume perfect communication channels. This is not a restrictive assumption since a designer designing software in UNIX/C or in Ada would not ordinarily be required to also design the communication protocol that underlies the message sends and receives. In principle, however, such protocol may be modeled if desired, through the use of appropriate event net structures.

Some communication styles do not require transformations for their realization. The communication style "Send-No-Wait/Receive-Wait" is modeled by existing event net structures by default and is referred to below as the default communication style. The style "Rendezvous, Non-Recipient Acks" arises from the flow in the original specification and may be expressed using two "Send-No-Wait/Receive-Wait" styles. "Send-No-Wait/Receive-Wait" and "Rendezvous, Non-Recipient Acks" may be used to communicate data corresponding to interaction events.

Formal definitions of communication styles and transformations follow.

4.6.2 Send-No-Wait/Receive-Wait (SNW)

In this IPC style, a sender subprocess sends a message to a receiver subprocess via a communication channel and immediately continues with its execution. The receiver subprocess is blocked if there is no message for it to receive. A sending activity in the sender subprocess is modeled by a transition d_s (see Figure 4.22). Similarly, a receiving activity in the receiver subprocess is modeled by a transition d_r . The communication channel is modeled by the place p between transitions d_s and d_r , and the message itself is modeled by a token which flows from d_s to d_r . If the sender sends but the receiver is not ready to receive, any queuing of messages in the channel is modeled by the queuing of tokens in place p . Note that here the channel is assumed unbounded so that place p is unbounded. The case of a bounded channel is treated below as the bounded FIFO channel transformation.

Formally, the “Send-No-Wait/Receive-Wait” style is modeled by a subnet S (Figure 4.22), where

$$R_S = \{ \langle d_s; d_r \rangle \}.$$

Figure 4.22: “Send-No-Wait/Receive-Wait” communication style.

4.6.3 Rendezvous, Non-Recipient Acks (RNA)

RNA is a rendezvous style in which a non-recipient activity, i.e., an activity other than the activity that received the message token, generates the ack. Thus, the term “non-recipient” refers to non-recipient activity. The ack may be either a pure acknowledgement

ack or a data ack. Since this is the rendezvous style, only one message can ever be in the channel at any one time, so that channel capacity is not a problem.

The RNA style is expressed as two SNW styles – one used for the sender to send its message, the other used for the receiver to send its ack. Thus, sending activities, receiving activities, and communication channels are modeled by transitions and places for two SNW styles as shown in Figure 4.23, where subprocess i initially sends a message (token) to subprocess j and then waits at place p to receive an ack (token) from subprocess j .

Formally, the “Rendezvous, Non-Recipient Acks” style is modeled by a subnet S (Figure 4.23), where

$$R_S = \{ \langle d_s; d \rangle, \langle d_s; d_r \rangle, \langle d_c; d \rangle \} \cup R_{S_1} \text{ where}$$

$$R_{S_1} = \{ \langle d_1; d_2 \rangle \mid \langle d_1; d_2 \rangle \text{ holds in a particular path } [d_r, d_c] \}.$$

The set of adjacency relations R_{S_1} defines the path $[d_r, d_c]$. Since $\langle d_r; d_c \rangle$ may or may not hold depending on subprocess j , the connection between d_r and d_c must be expressed as a path, which covers all cases. It is, of course, possible to use an event net generator to express this path, but the set R_{S_1} expresses it more simply (a generator would be better to express multiple paths).

The subnets $\{ \langle d_s; d \rangle \}$ and R_{S_1} may have arisen naturally in the original specification or were constructed by the SPC transformation which created the subprocesses i and j .

Figure 4.23: “Rendezvous - Non-Recipient Acks” communication style.

The so-called “reverse” rendezvous, in which the receiver subprocess sends a request to the sender subprocess in advance and the sender subprocess returns the results of the request as a data ack, is just a special case of the “Rendezvous, Non-Recipient Acks”

style with the roles of sender and receiver interchanged. Figure 4.24 illustrates the reverse rendezvous style.

Figure 4.24: "Reverse Rendezvous" communication style.

The following communication styles *do* require transformations for their realization, since modeling them calls for modifying existing event net structures. The transformations are called by the same names as the styles and are defined by specifying the subnets S and S' as was done for non-communication transformations. In the following transformations, $R_{S'}$ is expressed in terms of R_S , where possible, to emphasize "what must be done" to R_S to obtain $R_{S'}$.

Each of the following communication transformations modifies the default communication style "Send-No-Wait/Receive-Wait" into a new style. Thus, R_S for each of these transformations is R_S for SNW, shown in Figure 4.22. However, although the SNW style may be used without transformation and also used as R_S for transformation, there is an important difference between these two applications: without transformation, the SNW style may model the transfer of data corresponding to interactions, but *as R_S for transformation, the SNW style cannot model the transfer of data corresponding to interactions*. In other words, *communication styles established by transformation do not model the transfer of data corresponding to interactions*. This observation is due to the fact that as R_S for transformation, the SNW style models an internal communication between non-environmental components, arising from partitioning.

The application of a communication transformation is a good example of proceeding from a higher level of abstraction (SNW style) to a lower level of abstraction (a more detailed style), which is closer to code.

4.6.4 Bounded FIFO Channel (BFC)

The BFC transformation modifies the SNW communication style by placing a bound of n messages (one token corresponding to one message) on the communication channel (in contrast, the SNW style uses an unbounded channel). Thus, in Figure 4.25, R_S models the SNW style as before, but now, the BFC transformation adds place p_1 to model the capacity of the SNW channel represented by place p .

Formally, the "Bounded FIFO Channel" transformation is defined by S and S' as follows (Figure 4.25):

Subnet S

$$R_S = \{ \langle d_s; d_r \rangle \}$$

Subnet S'

$$R_{S'} = R_S \cup \{ \langle d_r; d_s \rangle \} \text{ with } \mu(p_1) = n, n \geq 1$$

where $p_1 : \langle d_r; d_s \rangle$ and n is the channel bound.

Figure 4.25: "Bounded FIFO Channel" transformation.

Discussion

If subprocess j is not ready to receive, the messages sent by subprocess i accumulate in FIFO order in place p , modeling message accumulation in the communication channel. The accumulation order is FIFO by definition of event nets. Once n messages have accumulated, subprocess i must wait until subprocess j has begun to receive messages, before it can send the $n + 1$ -st message. Thus, the communication between subprocesses i and j is asynchronous provided $\mu(p_1) > 0$ ($p_1 :< d_r; d_s >$). Otherwise, the communication is constrained asynchronous. Since subprocess i must wait to send a message if $\mu(p_1) = 0$, the BFC transformation may result in less waiting in event nets where the sending transition is only one of a number of choices for next transition to fire, as in Figure 4.26.

Figure 4.26: Less waiting due to alternative choices T2 and T3.

The BFC transformation may be applied k -times to model message sending from the same transition to k distinct receiving subprocesses (Figure 4.27), resulting in a k -way BFC communication style.

Because waiting may occur if the channel is full, deadlock may be introduced by this transformation and it will not be used in such cases. A check against this problem is described in Section 5.2.1.

Figure 4.27: “ k -way BFC” communication.

4.6.5 Rendezvous, Recipient Acks (RRA)

RRA is a rendezvous style in which the recipient activity, i.e., the activity that received the message token, generates the ack (Figure 4.28). Thus, the term “recipient” refers to recipient activity. Like the RNA style, the ack may be either a pure acknowledgement ack or a data ack and channel capacity is not a problem.

Sending and receiving activities, the send channel, and messages are modeled by event net components as in the SNW style. The RRA transformation adds places p_1 and p_2 , where p_1 models the return channel for the ack, and p_2 models the waiting place for subprocess i . In reality, the send and return channels may be a single two-way channel. The waiting place for subprocess i may be at the receiver’s receive port (as in the Ada rendezvous).

Formally, the “Rendezvous - Recipient Acks” transformation is defined by S and S' as follows (Figure 4.28):

Subnet S

$$R_S = \{ \langle d_s; d_r \rangle \}$$

Subnet S'

$$R_{S'} = (R_S \setminus \{ \langle d_s; d_r \rangle \}) \cup \{ \langle d_{s_1}; d_{s_2} \rangle, \langle d_{s_1}; d_r \rangle, \langle d_r; d_{s_2} \rangle \}$$

where $\{ \langle d_{s_1}; d_{s_2} \rangle \}$ is a refinement of d_s .

Figure 4.28: "Rendezvous - Recipient Acks" transformation.

Discussion

Unlike "Bounded FIFO Channel", the existence of choice in an application of "Rendezvous - Recipient Acks", where the sending transition is only one of a number of candidates for next transition to fire, cannot result in less waiting. This result is due to the fact that for "Rendezvous - Recipient Acks", the sender waits *after* the message is sent whereas for "Bounded FIFO Channel", the sender waits *before* the message is sent.

The "Rendezvous - Recipient Acks" transformation may be applied k -times to model message sending from the same transition to k distinct receiving subprocesses, which would all return acks to the same transition in the sending subprocess (Figure 4.29). This results

in a "*k*-way Rendezvous - Recipient Acks" communication style, also known as a "*k*-way rendezvous".

Again, because waiting may occur if the channel is full, deadlock may be introduced by this transformation and it will not be used in such cases. A check against this problem is described in Section 5.2.1.

Figure 4.29: "*k*-way Rendezvous - Recipient Acks".

4.6.6 Transporter (TPT)

The "Transporter" transformation modifies S by introducing a transporter process to carry a message from the sender subprocess to the receiver subprocess, much as a human courier carries a letter from a human sender to a human receiver. Thus, in subnet S' (Figure 4.30), the transporter process consists of transitions a_1 and a_2 , with its execution token in place p_2 . Subprocess i uses a communication style similar to BFC with bound 1 to call the transporter process with a message for delivery. The transporter process then delivers this message to subprocess j , using RRA. Place p_1 models whether or not the transporter process is free to carry a message for subprocess i since it may be busy waiting on subprocess j to receive.

As usual, software sending and receiving activities are modeled by transitions d_s and d_r .

The communication channel between d_s and d_r is modeled by the transporter subprocess. A message from d_s to d_r is again modeled by a token.

Formally, the "Transporter" transformation is defined by S and S' as follows (Figure 4.30):

Subnet S

$$R_S = \{ \langle d_s; d_r \rangle \}$$

Subnet S'

$$R_{S'} = (R_S \setminus \{ \langle d_s; d_r \rangle \}) \cup \{ \langle d_s; a_1 \rangle, \langle a_1; d_r \rangle, \langle a_1; a_2 \rangle, \langle d_r; a_2 \rangle, \langle a_2; a_1 \rangle, \langle a_2; d_s \rangle \}$$

with $\mu(p_1) = 1$, $\mu(p_2) = 1$ for places $p_1 : \langle a_2; d_s \rangle$ and $p_2 : \langle a_2; a_1 \rangle$.

Figure 4.30: "Transporter" transformation.

Discussion

The use of the "Transporter" transformation can reduce sender waiting time by letting the transporter process wait for the receiver to receive in place of the sender having to do this waiting. This arrangement works best when the transporter process is always

waiting to receive the message from the sender. Since this is not always the case – the transporter process may be carrying a message and waiting for the receiver to receive the “Transporter” transformation is best applied where the sending transition is only one of a number of choices for next transition to fire. As for “Bounded FIFO Channel”, this tends to reduce sender waiting time.

The “Transporter” communication style in fact behaves like the “Bounded FIFO Channel” style with channel capacity 1 (i.e., $n = 1$). Place p_1 (Figure 4.30) may be considered as modeling this channel capacity. Thus, as for “Bounded FIFO Channel”, the “Transporter” style is asynchronous if $\mu(p_1) > 0$. Otherwise, it is constrained asynchronous.

Because the sender may have to wait if the transporter process is waiting on the receiver, deadlock may be introduced by this transformation and it will not be used in such cases. A check against this problem is described in Section 5.2.1.

4.6.7 Bounded FIFO Buffer (BFB)

The Bounded FIFO Buffer transformation modifies subnet S by introducing a separate buffer process, to hold messages from sender subprocess i (Figure 4.31), until they are taken up by receiver subprocess j . Messages from the sender accumulate in FIFO order in place p_1 , which models a bounded buffer with a capacity of n messages (one token corresponding to one message).

The sender calls the buffer process with a message using the RRA style and in return receives a pure acknowledgement ack. The receiver calls the buffer process to obtain a message using the RRA style and in return receives the message as a data ack. If there is no message in the buffer, the receiver is blocked until the sender sends the buffer process a message.

Software sending activities correspond to transitions d_{s_1} and d_{r_1} ; software receiving activities are represented by transitions d_{s_2} and d_{r_2} . The buffer is modeled by place p_1 and its capacity is modeled by place p_2 . Communication channels among sender, buffer process, and receiver are modeled by the places associated with the use of the RRA style (see definition of RRA style above). Messages and acks are represented by tokens.

Formally, the “Bounded FIFO Buffer” transformation is defined by S and S' as follows (Figure 4.31):

Subnet S

$$R_S = \{ \langle d_s; d_r \rangle \}$$

Subnet S'

$$R_{S'} = (R_S \setminus \{ \langle d_s; d_r \rangle \}) \cup \{ \langle d_{s_1}; d_{s_2} \rangle, \langle d_{r_1}; d_{r_2} \rangle \}$$

$$\cup \{ \langle d_{s_1}; a_1 \rangle, \langle a_1; a_2 \rangle, \langle a_2; a_1 \rangle, \langle d_{r_1}; a_2 \rangle, \langle a_1; d_{s_2} \rangle, \langle a_2; d_{r_2} \rangle \}$$

with $\mu(p_2) = n$ where $p_2 : \langle a_2; a_1 \rangle$ and n is the bound on the size of the buffer (place p_1). Subnets $\{ \langle d_{s_1}; d_{s_2} \rangle \}$ and $\{ \langle d_{r_1}; d_{r_2} \rangle \}$ are refinements of activities d_s and d_r respectively. Activities a_1, a_2 belong to the buffer subprocess.

Figure 4.31: "Bounded FIFO Buffer" transformation.

Discussion

If the receiver does not call for its messages, the sender can store up to n messages in the buffer before having to wait for the receiver to start taking its messages. The FIFO order of messages is, as for Bounded FIFO Channel, a consequence of the event net feature that tokens arriving at a place enter a FIFO queue.

The advantage in using the Bounded FIFO Buffer style lies in the fact that provided the buffer capacity has not been reached, i.e., $\mu(p_2) > 0$, the sender can send a message to the

buffer process in the RRA style and receive back an ack almost immediately. The sender can thus send messages asynchronously to the receiver until the buffer capacity is reached, i.e., $\mu(p_2) = 0$, at which point the sender must wait for the receiver to call for its messages, i.e., the sender must wait to synchronize with the receiver. Once the receiver calls for its first message from the full buffer, synchronization is achieved and communication will proceed synchronously if buffer capacity is one and messages are transferred one at a time. However, if there is a pause in the sending of messages sufficient to allow the receiver to take up $1 < m \leq n$ messages from the full buffer before the sending resumes, then the sending resumed may be asynchronous until the buffer is again full.

Channel capacity is not a problem here since there is no accumulation of messages in the channels. Also, unlike BFC, the existence of choice, where the sending transition is only one of a number of candidates for next transition to fire, cannot result in less waiting for the sender. This conclusion follows from the fact that in BFB, the sender waits *after* the message is sent whereas in BFC, the sender waits *before* the message is sent.

Because the sender may have to wait if the buffer is full, deadlock may be introduced by this transformation and it will not be used in such cases. A check against this problem is described in Section 5.2.1.

4.7 Chapter Summary

This chapter began by motivating the need for transformations, and deriving six types of transformations based on the objective of transforming an event net requirements specification into a proto-design. The transformations derived were abstraction, refinement, sequentialization, subprocess construction, communication, and resource access control. In preparation for composing formal definitions of transformations, a relational notation was derived with which event nets may be specified clearly and unambiguously. Before defining the transformations using this notation, a number of required properties of an event net process specification were listed. The relational notation was then used to formulate formal definitions of each of the six types of transformations mentioned above. Using the notation, six communication styles were defined, namely “Send-Nc Wait/Receive-Wait”, “Rendezvous, Non-Recipient Acks”, “Bounded FIFO Channel”, “Rendezvous, Recipient Acks”, “Transporter”, and “Bounded FIFO Buffer”. Of these six styles, the last four are realized through transformations of the same name.

Transformations that will turn an event net requirements specification into a proto-design have been presented. However, a further objective is that the proto-design obtained be behaviourally correct with respect to the requirements specification. Therefore, in the next chapter the meaning of behaviour correctness will be considered and defined. It will be shown that correctness, according to this meaning, is preserved by the transformations. Then, any proto-design obtained solely through the use of the transformations will be automatically correct with respect to the requirements specification.

Chapter 5

Behaviour Correctness and Deadlock-Freeness

This chapter defines a form of behaviour correctness for use in this work, presents a number of concepts useful for showing correctness and deadlock-freeness, and derives various results in terms of these concepts. The concepts and results are then employed to show that the transformations defined in Chapter 4 preserve both behaviour correctness and deadlock-freeness.

5.1 Meaning of Behaviour Correctness

A reasonable definition of behaviour correctness is sought, which can be used to ascertain whether or not a proto-design is behaviorally correct with respect to its specification. The search begins in the literature, where many forms of behaviour equivalence relations exist for the comparison of software behaviour between processes.

5.1.1 Forms of Behaviour Equivalence

Some forms of behaviour equivalence found in the literature are *strong or weak observational equivalence* (Milner [30], Bolognesi and Smolka [4]), *testing equivalence* (Hennessy [21]), and *failures equivalence* (Hoare et al. [22], also discussed by Nicola and Hennessy [33]). A brief summary of each equivalence together with a statement on possible application to this work follows:

- *Failures Equivalence*: Define a refusal set of a process as a set of actions that the process can refuse to perform at a particular state. Two processes are then failures equivalent if after performing the same string of actions they both have the same refusal sets.

Showing failures equivalence requires determining actions which a process can refuse. Since this work intends to compare actions which are accepted rather than actions which are refused, failures equivalence is not a good choice for use here.

- *Testing Equivalence*: Two processes, suspected of being behaviourally equivalent, are subjected to experiments to compare their behaviours. An experiment consists of offering each process an identical input and then comparing the processes' outputs. The two processes are testing equivalent unless an experiment is conducted in which the outputs differ.

Testing equivalence could possibly have been adapted for use here but the selection of suitable experiments appears difficult.

- *Strong Observational Equivalence*: This equivalence is also known simply as *strong equivalence*. Two processes are strongly equivalent when it is impossible for an external observer to tell them apart merely by observing their actions which are all visible.

If all actions of a software system consisted only of inputs and outputs with the user, then using this equivalence, two software systems are behaviourally equivalent if their inputs and outputs with the user are indistinguishable, which at first appears to be reasonable. However, inputs and outputs with the user do not cover all possible actions of a software system – some actions, such as input and output between system components, are internal to the system. Strong equivalence requires that these internal actions must also be indistinguishable. Since a proto-design, or any design obtained through refinement, would likely have internal actions which are in addition to those in the requirements specification, strong equivalence is not a good candidate for use here (it is too strong).

- *Weak Observational Equivalence*: This equivalence is also known simply as *observational equivalence*. Two processes are observationally equivalent when it is impossible for an external observer to tell them apart merely by observing their visible actions. The processes may have invisible internal actions.

Weak observational equivalence appears to be the best candidate for use here. Under this equivalence, a proto-design may have internal actions that are different from those of the requirements specification, and yet be behaviourally equivalent

to it. This equivalence also appears reasonable, in that a user would not care how the internals of a software system work (the invisible internal actions), so long as the system does what is expected (the visible external actions). The definition of correctness below is based on this equivalence.

5.1.2 Towards a Definition of Behaviour Correctness

Taking a cue from the literature, it appears that the ideas of weak observational equivalence can be used to formulate a definition of behaviour correctness. This definition of behaviour correctness might be: *A proto-design is behaviourally correct wrt (with respect to) a requirements specification iff the interactions of the proto-design are indistinguishable from the interactions of the requirements specification. Both proto-design and requirements specification may have non-interaction events, which are ignored.* This definition is obtained as a direct application of weak observational equivalence to event nets and is an equivalence relation. Under this definition, for every execution of the specification, there is an execution of the proto-design, and vice versa, such that i) the same interactions occur in both nets, and ii) the interactions occur in the same order in both nets.

The definition of behaviour correctness just given, although based on weak observational equivalence, is nevertheless too strong for this work. The definition requires that if the specification expresses some non-determinism in the order of interactions, then this non-determinism must be fully present in the proto-design. Accordingly, if a specification specifies three possible orders of occurrence for a particular set of interactions, but in the proto-design only two of these orders of occurrence are possible, for the same set of interactions, then the proto-design is not behaviorally correct using this definition. As an example, consider a software requirements specification for an automatic teller machine, in which the user may enter account and amount in either order. Suppose the proto-design requests the user to enter account before amount. Since opening an account takes longer than merely registering an amount, the system could be opening the account while the user is entering the amount. Then, this seemingly reasonable proto-design fails the definition of correctness in the previous paragraph, even though the specification allows both orders to be possible. In this case, allowing both orders in the proto-design would be confusing and absurd!

The path from requirements specification to design is characterized by reductions in non-determinism, as the designer makes design decisions along the way and adds detail

(recall the discussion of specification and design in Chapter 2). Since the above definition of correctness is unable to accomodate this necessary characteristic of the design process, it must be modified as follows: *A proto-design is behaviourally correct wrt a requirements specification iff i) corresponding to every execution of the proto-design there is an execution of the requirements specification such that their interactions are indistinguishable, and ii) corresponding to every execution of the requirements specification, there is an execution of the proto-design such that their interactions are indistinguishable except possibly for the order of occurrence of interactions. Again, both proto-design and requirements specification may have non-interaction events, which are ignored.* This definition of correctness gives the proto-design the freedom to implement only a subset of the specification's possible interaction sequences that involve the same interactions, as required by the automatic teller example. Behavioural correctness for event nets, which is called *i-correctness* (after interaction correctness) will now be defined:

DEFINITION 5.1 *Given an event net E , an event net E' is i-correct wrt E (written $E' \sqsubseteq E$) iff i) not counting the order of occurrence, E' has the same interaction sets as E (i.e., corresponding to every execution of one, there is an execution of the other, such that not counting the order of occurrence, both executions exhibit the same interaction sets), and ii) E' 's interaction sets occur in the same order as in E or in a subset of the orders in E (i.e., corresponding to every execution of E' , there is an execution of E such that their interaction sets occur in the same order, but there may be executions of E for which there are no executions of E' such that their interaction sets occur in the same order).*

This definition is in terms of interaction *sets* rather than individual interactions, since event net interactions occur as *sets* when a transition fires.

Definition 5.1 states that E' 's behaviour can always be duplicated by E . On the other hand, E' may or may not be able to duplicate E 's behaviour. In the case of 'may not', the difference in behaviour is due solely to the order of occurrence of certain interaction sets. If every execution of E gives the same order of occurrence of its interaction sets, then E' can always duplicate E 's behaviour. All this suggests that E' 's set of behaviours is a proper subset of or equal to E 's set of behaviours, where equality corresponds to the case where E 's interaction sets occur in only one order. This subset relationship is reflected in the choice of the symbol " $\sqsubseteq$ " to represent the i-correctness relation. The fact that in general, E may behave differently from E' takes away any symmetry in the i-correctness

relation and therefore i-correctness is not an equivalence relation. However, i-correctness is reflexive and transitive.

The proofs of i-correctness below are based on Definition 5.1, which conveniently separates the conditions on interaction sets into a condition on sameness (same values) (part (i)) and a condition on order of occurrence (part (ii)). To determine if $E' \sqsubseteq E$, Definition 5.1 is checked for satisfaction. This check is done by comparing E' to E to determine i) if they have the same interaction sets (corresponding to part (i) of Definition 5.1) and ii) if the interaction sets of E' occur in an order or orders that can be duplicated by E (corresponding to part (ii) of Definition 5.1).

It is interesting to note that the i-correctness of a proto-design event net E' with respect to a requirements event net E is a *liveness* property of E' . A liveness property of an event net says that "something good will happen" to the event net. In this case i-correctness qualifies as "something that is good" for E' .

5.2 Results for i-Correctness and Deadlock-Freeness

5.2.1 i-Correctness

The objective of this section is to derive some theoretical results which can be used to prove that the transformations defined in Chapter 4 preserve i-correctness. A proto-design obtained from the requirements specification through applying the transformations of Chapter 4 will then be automatically i-correct wrt requirements. Since a transformation consists of replacing a subnet by a new subnet, it is reasonable to look at the points of inter-connection between a subnet and the rest of the net. Such points of inter-connection are termed boundary transitions, defined as follows:

DEFINITION 5.2 Let E be an event net with subnet S such that $R_S \neq \emptyset$. Let $\bar{S}$ be defined by $R_{\bar{S}} = R_E \setminus R_S$ (see Figure 5.1). A transition b is a boundary transition of S and $\bar{S}$ iff $\exists$ sets of transitions A, B, C, D such that $\langle A; b \rangle \in R_S$ and $\langle C; b \rangle \in R_{\bar{S}}$ or $\langle b; D \rangle \in R_{\bar{S}}$. A boundary input event set with respect to S ($\bar{S}$) consists of the set of all input events describing input from S ($\bar{S}$) to a boundary transition. A boundary output event set with respect to S ($\bar{S}$) consists of the set of all output events describing output from a boundary transition to S ($\bar{S}$). A boundary event set is either a boundary input event set or a boundary output event set.

Figure 5.1: A conceptual view of E , S , and $\bar{S}$; b is a boundary transition.

Definition 5.2 means that tokens moving in and out of a subnet are constrained to pass through the boundary transitions, as illustrated in Figure 5.2.

Preserving i-correctness under transformation requires interaction sets to remain the same and occur in pre-determined order(s). Since a transformation can affect boundary event sets, which can in turn affect interaction sets, it seemed reasonable to look at the preservation of boundary event sets under transformation. The preservation of boundary event sets by transformations is defined as follows:

DEFINITION 5.3 *Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' . Let $\bar{S}'$ be defined by $R_{\bar{S}'} = R_{E'} \setminus R_{S'}$. Then T preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ iff*

- *the boundary event sets with respect to $\bar{S}'$ are the same as the boundary event sets with respect to $\bar{S}$, and*
- *the sequence or sequences of boundary event sets with respect to $\bar{S}'$ are unchanged from the sequence or sequences of boundary event sets with respect to $\bar{S}$.*

T preserves boundary event sets with respect to S and S' iff

- *the boundary event sets with respect to S' are the same as the boundary event sets with respect to S ,*
- *the sequence or sequences of boundary event sets with respect to S' are unchanged from the sequence or sequences of boundary event sets with respect to S , and*
- *the following ordering relationship is maintained: let B be a boundary event set with respect to S and S' , let D be an interaction set of a non-boundary transition of S*

Figure 5.2: Example R_E , R_S , $R_{\bar{S}}$ where $R_E = R_S \cup R_{\bar{S}}$. The boundary transitions of S and $\bar{S}$ are a_1 , a_3 , and a_4 .

and S' ; then if B occurs before D in S' , B occurs before D in S , and if B occurs after D in S' , B occurs after D in S . This is a requirement on T for preservation of i -correctness (not necessary for the $\bar{S}$, $\bar{S}'$ case above since $\bar{S}$ is not transformed).

T preserves boundary event sets iff T preserves boundary event sets with respect to S and S' and T preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$.

Notice that T preserving boundary event sets with respect to S and S' in Definition 5.3 does not mean that $S' \subseteq S$. Subnets S' and S may still have different interaction sets, or some of their respective interaction sets may occur in different orders.

Notice also that if T preserves boundary event sets, the boundary transitions of S and S' are the same. However, if T only preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$, the boundary transitions of S and S' are not necessarily the same, in which case $\bar{S}$ and $\bar{S}'$ are not necessarily the same. This is illustrated in Figure 5.3 for the "Rendezvous - Receiver Acks" transformation, which only preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Figure 5.3(a) shows the pre-transformed net E with the subprocess at the left communicating with the subprocess at the right. Figure 5.3(b) shows the transformed net E' , in which $\bar{S}$ and $\bar{S}'$ are different.

To determine if T preserves boundary event sets, it suffices to check the definition of T , specifically the definitions of S and S' since they determine E' . For example, the definition of the "Refinement" transformation indicates that it preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. The definition of the "Rendezvous - Recipient Acks" transformation, indicates that it only preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$.

All transformations in Chapter 4 preserve boundary event sets with respect to $\bar{S}$ and $\bar{S}'$; some also preserve boundary event sets with respect to S and S' .

The meaning of the i -correctness of subnets requires some clarification, where a transformation has been applied to E to obtain E' . Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' (see Figure 5.4). Then $\bar{S}' \subseteq \bar{S}$ is understood in the context of observing $\bar{S}$ and $\bar{S}'$ during the execution of E and E' . Similarly, $S' \subseteq S$ is understood in the context of observing S and S' during the execution of E and E' . Some of the results below are expressed in terms of the i -correctness of subnets.

Notation: In the proofs below, lower case roman numerals, lower case roman numerals with letter suffixes, and arabic numbers are used as labels (e.g., (v), (v-a), (2)). A colon

Figure 5.3: Subnets $\bar{S}$ and $\bar{S}'$ are different for the “Rendezvous - Receiver Acks” transformation.

Figure 5.4: A conceptual view of $E' = T(E)$.

after such a label signifies that the statement following the colon is the item being labeled (e.g., $(v):, (v-a):, (2):$). No colon means that the label is being referenced. Arabic numbers are used to label intermediate results.

i-Correctness Results for Transformations

If a transformation replaces a subnet S with a subnet S' and the transformation preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ (see Figure 5.4), then one would expect the event sets of $\bar{S}$ and $\bar{S}'$ to be unaffected as well, i.e., one would expect $\bar{S}' \subseteq \bar{S}$. This expected result is exactly the statement of the following lemma. The proof of this lemma uses the definition of i-correctness by first showing that $\bar{S}$ and $\bar{S}'$ have the same interaction sets, and then showing that the interaction sets of $\bar{S}'$ occur in the required order.

LEMMA 5.1 *Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Then $\bar{S}' \subseteq \bar{S}$.*

Proof: It is given that (i): T replaces S with S' and (ii): T preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Item (i) implies (iii): except possibly for boundary transitions, $\bar{S}$ and $\bar{S}'$ have the same structure.

To show that $\bar{S}$ and $\bar{S}'$ have the same interaction sets, let D be an interaction set of some transition d in $\bar{S}$. Transition d is either (iv): a boundary transition or (v): a non-boundary transition. If (iv), (ii) implies D is an interaction set of $\bar{S}'$. If (v), either (v-a): d is the only non-boundary transition in $\bar{S}$ or (v-b): d is not the only non-boundary transition in $\bar{S}$. If (v-a), (iii) implies d is the only non-boundary transition in $\bar{S}'$. Then d fires in $\bar{S}$ together with (ii) and (iii) imply d fires in $\bar{S}'$. That is, D is an interaction set of $\bar{S}'$. If (v-b), $\exists$ a sequence s of input/output events in $\bar{S}$ which results in the enabling of d in $\bar{S}$. But then (ii) and (iii) imply that s occurs in $\bar{S}'$ and results in the enabling of d in $\bar{S}'$. That is, D is an interaction set of $\bar{S}'$. Therefore, if D is an interaction set of $\bar{S}$, D is also an interaction set of $\bar{S}'$. Similarly, if D is an interaction set of $\bar{S}'$, D is also an interaction set of $\bar{S}$. Therefore, (1): $\bar{S}$ and $\bar{S}'$ have the same interaction sets.

To show that the interaction sets of $\bar{S}'$ occur in the required sequence or sequences, let A be any set of event sets in $\bar{S}'$ occurring in some sequence s_1 . Then (ii) and (iii) imply (vi): the event sets of A also occur in $\bar{S}$ in the sequence s_1 . This result is seen as follows: (iii) implies that any possible difference in the occurrence of event sets in $\bar{S}'$ compared to

$\bar{S}$ can only be due to a difference in the boundary transitions. But any such difference has no effect on the occurrence of event sets in $\bar{S}$ and $\bar{S}'$, as long as the boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ are preserved according to Definition 5.3, and this is indeed the case. Now let B be any set of interaction sets of $\bar{S}'$ and suppose these interaction sets occur in some sequence s_2 . Then (1) means B also belongs to $\bar{S}$ and (vi) implies the interaction sets of B occur in sequence s_2 in $\bar{S}$. Therefore (2): the interaction sets of $\bar{S}'$ occur in the same sequence or subset of sequences as specified in $\bar{S}$.

Results (1) and (2) imply $\bar{S}' \subseteq \bar{S}$. $\square$

Suppose that a transformation replaces a subnet S with a subnet S' and preserves the boundary event sets (see Figure 5.4). Then if the transformed net is i-correct wrt the untransformed net, one might expect $S' \subseteq S$. On the other hand, if $S' \subseteq S$, then one might expect the transformed net to be i-correct wrt the untransformed net. The following Theorem 5.1 shows these expected results to be true.

In the proof of Theorem 5.1, given that the transformed net is i-correct wrt the untransformed net, showing that $S' \subseteq S$ is relatively easy, using the definition of i-correctness. Given that $S' \subseteq S$, showing that the transformed net is i-correct wrt the untransformed net is a little more difficult. The difficulty rests in showing that the transaction sets of the transformed net occur in the required sequence or sequences. The proof is accomplished by considering any two interaction sets of the transformed net and showing that no matter which subnet their corresponding transitions are from, the interaction sets occur in the required order for i-correctness.

THEOREM 5.1 *Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' and preserves boundary event sets. Then $E' \subseteq E$ iff $S' \subseteq S$.*

Proof: First, suppose $E' \subseteq E$. It is required to show that $S' \subseteq S$. Now $E' \subseteq E$ implies E' and E have the same interaction sets. By lemma 5.1, $\bar{S}'$ and $\bar{S}$ have the same interaction sets. Then since E' and E have the same interaction sets, it follows that (1): S and S' have the same interaction sets.

To show that the interaction sets of S' occur in the required sequence or sequences, let A be the set of interaction sets of S' (same as the set of interaction sets of S). Then these interaction sets are in E' and $E' \subseteq E$ implies that these interaction sets occur in the same sequence or subset of sequences as specified by E . That is, (2): the interaction sets

of S' occur in the same sequence or subset of sequences as specified in S . It follows from (1) and (2) that $S' \subseteq S$.

Second, suppose $S' \subseteq \bar{S}$. It is required to show that $E' \subseteq E$. To show that E' and E have the same interaction sets, let D be any interaction set of E' . Then (i): D is an interaction set of $\bar{S}'$, or (ii): D is an interaction set of S' . If (i), Lemma 5.1 implies D is an interaction set of E . If (ii), $S' \subseteq S$ implies D is an interaction set of E . Therefore, if D is any interaction set of E' , D is an interaction set of E . Similarly, if D is any interaction set of E , D is an interaction set of E' . It follows (3): that E' and E have the same interaction sets.

To show that the interaction sets of E' occur in the required sequence or sequences, let D_1, D_2 be any two interaction sets of E' . Without loss of generality, suppose D_1 occurs before D_2 (note that "occurs before" is unambiguous, even if D_1 and D_2 are in a loop, since each event is considered distinct - see discussion following this proof). Then one of the following is true: (iii): D_1 and D_2 are both interaction sets of S' , (iv): D_1 and D_2 are both interaction sets of $\bar{S}'$, (v): D_1 is an interaction set of S' and D_2 is an interaction set of $\bar{S}'$, (vi): D_1 is an interaction set of $\bar{S}'$ and D_2 is an interaction set of S' .

If (iii), $S' \subseteq S$ implies D_1 occurs before D_2 in E .

If (iv), Lemma 5.1 implies D_1 occurs before D_2 in E .

If case (v) is true, D_1 corresponds to some interaction transition d_1 in S' , D_2 corresponds to some interaction transition d_2 in $\bar{S}'$. Then (v-a): d_1 and d_2 are boundary transitions which may or may not be distinct, (v-b): d_1 is not a boundary transition, d_2 is a boundary transition, (v-c): d_1 is a boundary transition, d_2 is not a boundary transition, and (v-d): d_1 is not a boundary transition, d_2 is not a boundary transition.

If (v-a), $\mathcal{T}$ preserves boundary event sets means D_1 occurs before D_2 in E , since D_1 and D_2 correspond to boundary event sets whose order of occurrence is preserved. Alternatively, D_1 and D_2 are also interaction sets of S and S' . Then $S' \subseteq S$ implies D_1 occurs before D_2 in E .

If (v-b), D_2 is also an interaction set of S and S' . Then $S' \subseteq S$ implies D_2 occurs before D_1 in E .

If (v-c), $[d_1, d_2]$ is true in $\bar{S}'$ such that d_1 fires before d_2 . Since $\mathcal{T}$ does not affect the structure of $\bar{S}$, $[d_1, d_2]$ is true in $\bar{S}$. That is, D_1 occurs before D_2 in E .

If (v-d), $[d_1, d_2]$ is true in E' such that d_1 fires before d_2 . Then $\exists$ some boundary transition b such that $[d_1, b]$ is true in S' and $[b, d_2]$ is true in $\bar{S}'$. Let b have event set

D with respect to S' . Then D_1 occurs before D in S' . Then T preserves boundary event sets with respect to S and S' means D_1 occurs before D in E (using the preserved ordering relationship between boundary event sets and interaction sets of non-boundary transitions). Since T does not affect the structure of $\bar{S}$, $[b, d_2]$ is true in $\bar{S}$. That is, D occurs before D_2 in E . Then D_1 occurs before D in E and D occurs before D_2 in E means D_1 occurs before D_2 in E .

If case (vi) is true, the proof that D_1 occurs before D_2 in E is similar to the proof for (v).

It follows (4): that the interaction sets of E' occur in a sequence or subset of sequences specified by E . Then (3) and (4) imply that $E' \subseteq E$. $\square$

The expression " D_1 occurs before D_2 " was used in the above proof assuming that the relation "occurs before" is unambiguous. That this is indeed the case will now be more fully explained. The question of ambiguity arises where D_1 and D_2 are visualized as occurring in a loop, in which D_1 is followed by D_2 , which is then followed by D_1 , and so on. Then, is it D_1 that occurs before D_2 or D_2 that occurs before D_1 ? As briefly mentioned in the proof above, this is not a problem since all events occurring in a loop are distinct through automatic indexing (see Chapter 3). That is, each event and hence each interaction set can be visualized as having an index number which sets it apart. Thus if D_1 and D_2 occur in a loop, it is really the following sequence that occurs: D_{11} is followed by D_{21} , which is then followed by D_{12} , which in turn is followed by D_{22} and so on. In other words, the statement " D_1 occurs before D_2 " is to be understood as the statement " D_{1i} occurs before D_{2i} " which is always unambiguous.

Figure 5.5 shows an example of applying Theorem 5.1.

Suppose that a transformation replaces a subnet S with a subnet S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ (see Figure 5.4). Then, if S and S' do not participate in interactions, one would expect $E' \subseteq E$. This gives the following theorem, which is useful in cases where the subnets defining a transformation do not participate in interactions.

Figure 5.5: Construction of E' by replacing S with S' - note $E' \sqsubseteq E$ by Theorem 5.1.

THEOREM 5.2 *Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Then $E' \sqsubseteq E$ if S and S' do not participate in interactions.*

Proof: The result follows from Lemma 5.1. $\square$

The above results on i-correctness may only be applied to transformations which preserve boundary event sets or preserve boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Results applicable to communication transformations, which do not necessarily preserve boundary event sets with respect to $\bar{S}$ and $\bar{S}'$, are addressed next.

i-Correctness Results for Communication Transformations

The following definitions express the condition in which the receiving process of a communication must wait for a message from the sender process.

DEFINITION 5.4 *Let E be an event net with initial marking μ and let C_i, C_j be two distinct subnets of E modeling sequential subprocesses. Then*

1. $C_j \propto C_i$ (" C_j depends on C_i ") iff $\exists \langle d_s; d_r \rangle \in R_E$ with d_s in C_i and d_r in C_j ,
2. $C_j \vdash_{\mu'} C_i$ (" C_j waits for C_i in marking μ' ") iff $C_j \propto C_i$ and $\exists \mu' \in R(E, \mu)$ such that $\mu'(p) = 0$ for $p : \langle d_s; d_r \rangle$ and $\mu'(p') \geq 1$ for $p' \in I(d_r)$, $p' \neq p$.

The symbol " $\propto$ " denotes the "dependence" relation, which is clearly not reflexive ($C_i \propto C_i$ is undefined), not symmetric, and not transitive. The symbol " $\vdash_{\mu'}$ " denotes the "waits for in marking μ' " relation, which is not reflexive ($C_i \vdash_{\mu'} C_i$ is undefined), not symmetric ($\vdash_{\mu'}$ is defined in terms of $\propto$, which is not symmetric), and not transitive ($\vdash_{\mu'}$ is defined in terms of $\propto$, which is not transitive).

Example: The event net E in Figure 5.6 has $\langle T2; T3 \rangle, \langle T3; T4 \rangle \in R_E$, so by Definition 5.4, $C_3 \propto C_2 \propto C_1$. Also, E has marking μ' with $\mu'(p_1) = 0$, $\mu'(p_2) = 0$, $\mu'(p_3) = 0$, $\mu'(p_4) = 0$, $\mu'(p_5) = 1$, $\mu'(p_6) = 1$, and $\mu'(p_7) = 1$. By Definition 5.4, $C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1$.

Example: Figure 5.7 shows $E' = T(E)$ for E in Figure 5.6 and T defined by $R_{E'} = R_E \cup \{ \langle T5; T1 \rangle \}$. The components $\langle T2; T3 \rangle, \langle T3; T4 \rangle, \langle T5; T1 \rangle \in R_{E'}$, so by

Figure 5.6: α and $\vdash_{\mu'}$ relations.

Definition 5.4, $C_3 \alpha C_2 \alpha C_1 \alpha C_3$. Also, E' has marking $\mu'(p_1) = 0$, $\mu'(p_2) = 0$, $\mu'(p_3) = 0$, $\mu'(p_4) = 0$, $\mu'(p_5) = 1$, $\mu'(p_6) = 1$, $\mu'(p_7) = 1$, and $\mu'(p_8) = 0$. By Definition 5.4, $C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1 \vdash_{\mu'} C_3$. Clearly, $E' \not\sqsubseteq E$ for E in Figure 5.6 since E' is deadlocked.

Figure 5.7: $C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1 \vdash_{\mu'} C_3$ (circular waiting) resulting in deadlock.

The last example illustrates how a transformation may not preserve i-correctness, through the inadvertent introduction of a “wait loop” defined as follows:

DEFINITION 5.5 Let E be an event net with initial marking μ , consisting of distinct subnets $C_1, C_2, \dots, C_n$, $n \geq 2$, where each C_i models a sequential subprocess. A **wait loop**

in E is a condition

$$C_i \vdash_{\mu'} C_{k_1} \vdash_{\mu'} C_{k_2} \vdash_{\mu'} \dots \vdash_{\mu'} C_{k_m} \vdash_{\mu'} C_i$$

for some $\mu' \in R(E, \mu)$ and some $C_i, C_{k_1}, C_{k_2}, \dots, C_{k_m}$, where $1 \leq i \leq n$, $1 \leq m \leq n-1$, and $k_1, k_2, \dots, k_m \in \{1, 2, \dots, n\} \setminus \{i\}$.

Circular Waiting Conditions

The next theorem establishes a condition under which the application of the communication transformations “Bounded FIFO Channel” (BFC) or “Rendezvous - Recipient Acks” (RRA) results in an event net that is i -correct wrt the pre-transformed net. The condition prevents the BFC or RRA transformation from inadvertently introducing a wait loop by checking for its presence. Figure 5.8 shows an application of the RRA transformation (nets (a) and (b)) which inadvertently introduces a wait loop (nets (b) and (c)) resulting in a deadlock (net (c)).

THEOREM 5.3 *Let E be a deadlock-free event net with initial marking μ , consisting of subnets $C_1, C_2, \dots, C_n$, $n \geq 2$, where each C_i models a sequential subprocess. Let $E' = T(E)$ where T is defined by $R_S = \{ \langle d_s; d_r \rangle \}$, d_s in C_i , d_r in C_j , for $i \neq j$ and*

$$R_{S'} = \{ \langle d_s; d_r \rangle, \langle d_r; d_s \rangle \} \quad (\text{BFC})$$

with $\mu(p) = n \geq 1$ for $p : \langle d_r; d_s \rangle$ or

$$R_{S'} = \{ \langle d_s; d_e \rangle \} \cup \{ \langle d_s; d_r \rangle, \langle d_r; d_e \rangle \} \quad (\text{RRA}).$$

Define conditions

$$\mathcal{L}_1 = (\exists \mu' \in R(E', \mu) \text{ such that } C_i \vdash_{\mu'} C_j \vdash_{\mu'} C_i),$$

$$\mathcal{L}_2 = (\exists \mu' \in R(E', \mu) \text{ and } C_{k_1}, C_{k_2}, \dots, C_{k_m}, \text{ such that}$$

$$C_i \vdash_{\mu'} C_j \vdash_{\mu'} C_{k_1} \vdash_{\mu'} C_{k_2} \vdash_{\mu'} \dots \vdash_{\mu'} C_{k_m} \vdash_{\mu'} C_i,$$

$$\text{for } 1 \leq m \leq n-2, k_1, k_2, \dots, k_m \in \{1, 2, \dots, n\} \setminus \{i, j\}).$$

Then $E' \sqsubseteq E$ if $\mathcal{L}_1$ and $\mathcal{L}_2$ are false. The conditions $\mathcal{L}_1$ and $\mathcal{L}_2$ are called the circular waiting conditions.

Figure 5.8: Inadvertent introduction of a wait loop by the RRA transformation, where C_1 is the sender and C_3 the receiver.

Proof: Suppose $\mathcal{L}_1$ and $\mathcal{L}_2$ are false and that $E' \not\sqsubseteq E$. There are 2 cases.

Case 1: $R_{S'} = \{ \langle d_s; d_r \rangle, \langle d_r; d_s \rangle \}$ with $\mu(p) \geq 1$ for $p : \langle d_r; d_s \rangle$ (BFC). Since $E \sqsubseteq E'$, $E' \not\sqsubseteq E$ is due to the addition of the component $\langle d_r; d_s \rangle$ to R_E giving $R_{E'}$, and since E' has the same initial marking μ as E , with $\mu(p) \geq 1$ for $p : \langle d_r; d_s \rangle$, E' has the same behaviour as E provided d_s is enabled in E' when d_s is enabled in E . That is, $E' \not\sqsubseteq E$ is due to $\exists \mu' \in R(E', \mu)$ such that $C_i \vdash_{\mu'} C_j$ for μ' and all markings in $R(E', \mu)$ after μ' (if $C_i \vdash_{\mu'} C_j$ for μ' only, it may still be true that $E' \sqsubseteq E$ since C_i 's events are not ordered relative to C_j 's events, i.e., $E' \not\sqsubseteq E$ implies C_i stopped executing). That is, $C_j \vdash_{\mu'} C_{k_1}$ for some process C_{k_1} , $k_1 \neq j$, and this is true for μ' and all markings in $R(E', \mu)$ after μ' . That is, $C_{k_1} = C_i$, in which case condition $\mathcal{L}_1$ is true, or $C_{k_1} \neq C_i$. If $C_{k_1} \neq C_i$, C_{k_1} must be waiting on some other process C_{k_2} ; otherwise, C_{k_1} would be deadlock-free in E' since it is deadlock-free in E and $\mathcal{T}$ did not change the structure of C_{k_1} . That is, $C_{k_1} \vdash_{\mu'} C_{k_2}$, $k_1, k_2 \neq j$, $k_1 \neq i$, for μ' and all markings in $R(E', \mu)$ after μ' . If $C_{k_2} = C_i$, condition $\mathcal{L}_2$ is true for $m = 1$. If $C_{k_2} \neq C_i$, C_{k_2} must be waiting on some process C_{k_3} ; otherwise, C_{k_2} would be deadlock-free in E' since it is deadlock-free in E and $\mathcal{T}$ did not change the structure of C_{k_2} . That is, $C_{k_2} \vdash_{\mu'} C_{k_3}$, $k_1, k_2, k_3 \neq j$, $k_1, k_2 \neq i$, for μ' and all markings in $R(E', \mu)$ after μ' . If $C_{k_3} = C_i$, condition $\mathcal{L}_2$ is true for $m = 2$. If $C_{k_3} \neq C_i$, C_{k_3} must be waiting on some process C_{k_4} ; otherwise, C_{k_3} would be deadlock-free in E' since it is deadlock-free in E and $\mathcal{T}$ did not change the structure of C_{k_3} . The proof may continue in this fashion until reaching $C_{k_m} \vdash_{\mu'} C_i$ as the final alternative. Therefore, $E' \not\sqsubseteq E$ implies either condition $\mathcal{L}_1$ is true or condition $\mathcal{L}_2$ is true. Contradiction. Therefore, $E' \sqsubseteq E$.

Case 2: $R_{S'} = \{ \langle d_s; d_e \rangle \} \cup \{ \langle d_s; d_r \rangle, \langle d_r; d_e \rangle \}$ (RRA). The proof of this case is similar to case 1 with $\langle d_r; d_e \rangle$ replacing $\langle d_r; d_s \rangle$ in case 1. $\square$

Theorem 5.3 suggests that after applying the BFC or RRA transformation, the event net E' must be examined to ensure that the circular waiting conditions are false, in order to ensure that the transformations preserve i-correctness. This examination could be done manually for small examples, but for any net of meaningful size it requires a computer tool. For example, checking the circular waiting conditions manually on the net of Figure 5.8(c) correctly identifies the wait loop $C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1 \vdash_{\mu'} C_3$.

If the circular waiting conditions are not false, the BFC or RRA transformation has to be undone and cannot be re-applied until E' has been “adjusted” (through the application of other transformations or re-consideration of transformations that have already been

applied) so that the circular waiting conditions are false.

The circular waiting conditions identify a particular marking and verification of their falsity may be done via a reachability analysis to see if this marking is reachable from the initial marking. The conditions are false if this marking is not reachable from the initial marking. In this case, the computational complexity of checking the circular waiting conditions is that of doing a reachability analysis. The latter can take a long time and is subject to state explosion. Fortunately, efficient algorithms for doing reachability analysis may be applied here too.

Given its computational complexity, reachability analysis to determine the falsity of the waiting conditions is to be avoided if at all possible. Two ways to avoid this analysis will now be given. The first method derives "circular dependence conditions" whose falsity automatically implies the falsity of the circular waiting conditions. The second method is to show that the number of messages over a communication channel is bounded, which for certain communication transformations means that the circular waiting conditions are false. The circular dependence conditions are presented first.

Circular Dependence Conditions

The following corollary to Theorem 5.3 gives a faster method for determining whether or not the circular waiting conditions are false. The corollary makes use of the fact that $\vdash_{\mu'}$ is defined in terms of α , and therefore the circular waiting conditions are false if the underlying α relations are false. The latter relations define "circular dependence conditions" which correspond to the circular waiting conditions.

COROLLARY 5.1 *Let E and E' be as in Theorem 5.3. Define conditions*

$$\mathcal{M}_1 = (\exists \langle d_p; d_q \rangle \in R_E \text{ giving } C_j \alpha C_i, \langle d_p; d_q \rangle \neq \langle d_s; d_r \rangle),$$

$$\mathcal{M}_2 = (\exists C_{k_1}, C_{k_2}, \dots, C_{k_m}, \text{ in } E \text{ such that}$$

$$C_j \alpha C_{k_1} \alpha C_{k_2} \alpha \dots \alpha C_{k_m} \alpha C_i$$

$$\text{for } 1 \leq m \leq n-2, k_1, k_2, \dots, k_m \in \{1, 2, \dots, n\} \setminus \{i, j\}).$$

Then $E' \sim E$ if $\mathcal{M}_1$ and $\mathcal{M}_2$ are false. The conditions $\mathcal{M}_1$ and $\mathcal{M}_2$ are called the circular dependence conditions.

Note that the circular dependence conditions are defined as $C_j \alpha C_i$ in expectation of the dependence $C_i \alpha C_j$ being provided by the return link of the BFC or RRA transfor-

mation. For then, $C_i \propto C_j \propto C_i$ corresponding to the circular waiting conditions. This allows the circular dependence conditions to be checked prior to applying the BFC or RRA transformation.

The circular dependence conditions are much easier to check than the circular waiting conditions, since they are describing structure only and do not involve any marking. In fact, the circular dependence conditions may be checked by visual inspection, even if E is large, and a computer tool may of course be used. Moreover, as has been mentioned, these conditions are checked on E *prior* to the application of the transformation, unlike the circular waiting conditions, which are examined on E' *after* the transformation. This approach could save some work, especially if it turns out that the transformation introduces a wait loop and has to be undone. Thus, the use of the circular dependence conditions to avoid the inadvertent introduction of wait loops is a more practical alternative to the waiting conditions.

The drawback in the use of the circular dependence conditions to identify potential wait loops is that they can rule out perfectly legitimate loop structures such as the one shown in Figure 5.9(b). Checking the dependence conditions for the communication from C_1 to C_3 in Figure 5.9(a), one finds $C_3 \propto C_2 \propto C_1$, i.e., the dependence conditions are not both false. One then concludes that neither the BFC nor the RRA transformation can be applied to the communication without introducing a wait loop, which is a false conclusion as shown by Figure 5.9(b). On the other hand, the circular waiting conditions correctly rule out any wait loop in applying the RRA transformation to the communication, since there is no $\mu' \in R(E', \mu)$ such that $C_1 \vdash_{\mu'} C_3 \vdash_{\mu'} C_1$ or $C_1 \vdash_{\mu'} C_3 \vdash_{\mu'} C_2 \vdash_{\mu'} C_1$ (Figure 5.9(b)).

The use of the circular waiting conditions to avoid wait loops may be called a *fine-grained* method, in contrast to the use of the circular dependence conditions, which may be called a *coarse-grained* method. This distinction is due to the fact that the former method discerns only true wait loops, whereas the latter method can rule out legitimate cases that are not wait loops.

Nevertheless, the use of the circular dependence conditions may be justified in design, where it would be consistent with *rapid* iteration among alternative structures, as in this work. Therefore, to avoid the introduction of wait loops (and preserve i-correctness), it is recommended that the circular dependence conditions be checked first; the circular waiting conditions may be checked where the former has ruled out a transformation and

Figure 5.9: Legitimate application of RRA to E, disallowed by the circular dependence conditions, but allowed by the circular waiting conditions.

the designer feels strongly that the loop structure resulting from the transformation should be retained, if at all possible (such as in Figure 5.9).

The following procedure may be used to check the circular dependence conditions prior to applying a BFC or RRA transformation:

Procedure for Checking the Circular Dependence Conditions Let E and E' be as in Corollary 5.1. Subnet S for transformation is defined by $R_S = \{ \langle d_s, d_r \rangle \}$ with d_s in subprocess C_i and d_r in subprocess C_j .

- Step 1 :** Check condition $\mathcal{M}_1$ by determining if there is any transition d_p in C_i , $d_p \neq d_s$ and any d_q in C_j , $d_q \neq d_r$, such that there is a path from d_p to d_q . Condition $\mathcal{M}_1$ is false if no such transitions d_p, d_q exist.
- Step 2 :** Check condition $\mathcal{M}_2$ by determining if there are subprocesses $C_{k_1}, C_{k_2}, \dots, C_{k_m}$, such that there is a path from C_i to C_{k_m} , from C_{k_m} to $C_{k_{m-1}}$, ..., from C_{k_2} to C_{k_1} , and from C_{k_1} to C_j . Condition $\mathcal{M}_2$ is false if no such subprocesses $C_{k_1}, C_{k_2}, \dots, C_{k_m}$ exist.
- Step 3 :** If $\mathcal{M}_1$ and $\mathcal{M}_2$ are both false, apply a BFC or RRA transformation to subnet S prior to re-applying this procedure to another S , so that the effect of the transformation (e.g., new paths) may be taken into account when this procedure is re-applied. If either $\mathcal{M}_1$ or $\mathcal{M}_2$ is true, neither the BFC nor the RRA transformation can be applied to S based on the circular dependence conditions.

The paths in steps 1 and 2 of this procedure are communication paths, i.e., dependence relations (Definition 5.4) between subprocesses. A graph using arrows between subprocess identifiers to represent dependence relations between the subprocesses may be drawn up as an aid for steps 1 and 2. An arrow from subprocess A to subprocess B in this graph would signify that there is an event net communication arc from A to B, i.e., that B depends on A. Step 1 may be performed on this graph by checking if there is an arrow from C_i to C_j (corresponding to the path from d_p to d_q) other than the arrow representing the communication from d_s to d_r . Step 2 may be done on this graph by substituting "arrow" for "path" in the above description of step 2. This graph is called a *dependence graph*.

Example applications of this procedure using dependence graphs are given in Chapter 7. If in step 3 of this procedure, the user finds that the BFC or the RRA transformation

cannot be applied, the transformation may be applied anyway, followed by a check of the circular waiting conditions, which as explained above, are more accurate. The circular waiting conditions would be checked either by applying the procedure for bounded number of messages below or by performing a reachability analysis. If the waiting conditions also do not allow the transformation, then it may be necessary to "adjust" E' until either the dependence conditions or the waiting conditions are false. This adjustment may consist of un-doing certain transformations and then applying different transformations in order to re-arrange the paths between subprocesses.

The circular waiting conditions and the circular dependence conditions are also applicable to the TPT and BFB transformations, as shown by their i-correctness proofs below. That is, the TPT and the BFB transformations preserve i-correctness if either the circular waiting conditions or the circular dependence conditions are found to be false.

The second way of determining the falsity of the circular waiting conditions without doing a reachability analysis will now be considered:

Bounded Number of Messages

By reasoning over the topology of the event net, it is sometimes possible to show that the application of a communication transformation will result in the falsity of the circular waiting conditions, without a reachability analysis. This approach is most beneficial where the circular dependence conditions cannot be shown to be false and the only other recourse is a reachability analysis. The objective here is to show that a maximum number n of messages traverse the communication channel in question at any one time. Then, if $n > 1$, application of the BFC or BFB transformations with bound n will result in the falsity of the waiting conditions, since the sender will never be forced to wait for the receiver to receive. If $n = 1$, the TPT transformation may be included with BFC and BFB as transformations that will result in the falsity of the waiting conditions. Whether or not the number of messages traversing a channel is bounded may be determined by using the following procedure:

Procedure for Bounded Number of Messages For each communication channel for which it is desired to show that application of either the BFC, BFB, or TPT transformations will fail the circular waiting conditions:

- Step 1 :** Show that there is never more than n messages in the communication channel, for some $n \geq 1$, by examining the structure and markings of the two subprocesses and reasoning that no more than n messages can be in the channel at any one time.
- Step 2 :** If step 1 succeeds and $n > 1$, either the BFC transformation or the BFB transformation with bound n may be applied to the communication; if step 1 succeeds and $n = 1$, the BFC, BFB, or the TPT transformation may be used; the resulting communication style will then fail the circular waiting conditions because the sender will never be forced to wait for the receiver to receive;
- Step 3 :** If step 1 fails, i.e., it is indeterminate that there is never more than n messages in the channel for some $n \geq 1$, then it is necessary to do a reachability analysis as discussed above.

Example applications of this procedure are given in Chapter 7.

5.2.2 Deadlock-Freeness

The meaning of deadlock-freeness for an event net was given in Definition 3.9. In this work, *it is assumed that the requirements specification event net is deadlock-free*. The transformations used to transform a requirements specification into a proto-design must preserve deadlock-freeness as well as i-correctness. These two concepts are separate in that by themselves, one does not imply the other. However, the following theorems identify some conditions under which deadlock-freeness may be obtained.

Interaction-connectedness (Definition 3.6) pulls together the concepts of i-correctness and deadlock-freeness. Informally, an event net is interaction-connected if corresponding to each non-interaction transition of the net, there is some interaction transition of the net such that the non-interaction transition must fire in order for the interaction transition to be enabled. In this work, *it is assumed that the requirements specification event net is interaction-connected*.

Suppose that a transformation replaces a subnet S with a subnet S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ (see Figure 5.4). Then, if the pre-transformed net is interaction-connected, and the replacement subnet due to the transformation is also interaction-connected, one would expect the transformed net to be interaction-connected also. This expected result is stated by the following theorem, which can be used to conclude that a transformed net is interaction-connected.

THEOREM 5.4 *Let E be an event net and $E' = T(E)$, where T replaces subnet S in E with subnet S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Then E' is interaction-connected if E and S' are both interaction-connected.*

Proof: Suppose E and S' are both interaction-connected. It is required to show that E' is interaction-connected. This requirement is fulfilled by showing that E' satisfies the definition of interaction-connectedness. Let c be any non-interaction transition of E' . Then (i): c is in $\bar{S}'$, c is not a boundary transition, or (ii): c is in S' .

If (i), T replaces S with S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ implies (iii): c is in $\bar{S}$. Then E interaction-connected means $\exists$ interaction transition d in E such that c must fire in order for d to be enabled. Then (iii-a): d is in S or (iii-b): d is in $\bar{S}$, d is not a boundary transition.

If (iii-a), (i.e., c is in $\bar{S}$, d is in S , and c must fire in order for d to be enabled) $\exists$ boundary transition b in E such that c must fire in order for b to be enabled ($b = d$ is possible). Then T replaces S with S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ implies $\exists$ boundary transition b' in E' such that c must fire in order for b' to be enabled. If b' is an interaction transition, (1): c must fire in order for interaction transition b' to be enabled. If b' is not an interaction transition, S' interaction-connected means (by definition) $\exists$ interaction transition d' in E' such that b' must fire in order for d' to be enabled. That is, (2): c must fire in order for interaction transition d' to be enabled.

If (iii-b), (i.e., c is in $\bar{S}$, d is in $\bar{S}$, d is not a boundary transition, and c must fire in order for d to be enabled) T replaces S with S' and preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$ implies d is in $\bar{S}'$ and (3): c must fire in order for interaction transition d to be enabled.

If (ii), S' interaction-connected means (by definition) (4): $\exists$ interaction transition d in E' such that c must fire in order for d to be enabled. Then (1), (2), (3), and (4) show that for any non-interaction transition c in E' , there is some interaction transition d in E' , such that c must fire in order for d to be enabled. That is, E' is interaction-connected. $\square$

Suppose that a transformation of an interaction-connected event net results in an event net that is both i-correct wrt the pre-transformed net and interaction-connected. Then since i-correctness, loosely speaking, means that both nets execute the same interaction transitions, and interaction-connectedness, loosely speaking, means that in both

nets, execution of all interaction transitions requires the execution of all non-interaction transitions, then one would expect that if either net is deadlock-free, the other net must also be deadlock-free. This expected result is stated formally by the following theorem.

THEOREM 5.5 *Let E be an event net and $E' = T(E)$. Suppose E and E' are interaction-connected and $E' \subseteq E$. Then E' is deadlock-free iff E is deadlock-free.*

Proof: Suppose E is deadlock-free. Then $\exists \mu$ such that every transition of E is deadlock-free in μ .

Let $d'_1, d'_2, \dots, d'_m$ be the interaction transitions of E' for some m . Then $E' \subseteq E$ means E has interaction transitions $d_1, d_2, \dots, d_n$ for some n such that (i): $d'_1, d'_2, \dots, d'_m$ have the same interaction sets as $d_1, d_2, \dots, d_n$ and (ii): these interaction sets occur in E' in the same sequence or subset of sequences as they occur in E . In order for (ii) to be true, (iii): $d'_1, d'_2, \dots, d'_m$ must fire in the same sequence or subset of sequences as $d_1, d_2, \dots, d_n$. Then $d_1, d_2, \dots, d_n$ deadlock-free in μ together with (iii) imply $\exists$ some marking μ' of E' such that $d'_1, d'_2, \dots, d'_m$ are deadlock-free in μ' .

Let $c'_1, c'_2, \dots, c'_p$ be the non-interaction transitions of E' for some p . Consider non-interaction transition c'_k for some $1 \leq k \leq p$. Then $\exists d'_j, 1 \leq j \leq m$, such that c'_k is required to fire in order for d'_j to be enabled (E' is interaction-connected). Since d'_j is deadlock-free in μ' and c'_k is required to fire in order for d'_j to be enabled, it follows that c'_k is deadlock-free in μ' . That is, $c'_1, c'_2, \dots, c'_p$ are deadlock-free in μ' .

Therefore, E' has a marking μ' such that every transition of E' is deadlock-free in μ' . That is E' is deadlock-free.

Similarly, E is deadlock-free if E' is deadlock-free. $\square$

Theorems 5.4 and 5.5 provide a way of determining the deadlock-freeness of a transformed net: If both the replacement subnet in the transformation and the pre-transformed net are interaction-connected, then the transformed net is also interaction-connected, by Theorem 5.4. If the pre-transformed net is deadlock-free and interaction-connected, and the transformed net is i-correct wrt the pre-transformed net and interaction-connected, then the transformed net is deadlock-free, by Theorem 5.5.

For S' to be interaction-connected, Definition 3.6 requires that if c is a non-interaction transition in S' , then c is required to fire in order for some interaction transition d in E' to

be enabled. For a particular c , checking for d may be tedious, since d could be in $\bar{S}'$. The following theorem gives conditions for S' to be interaction-connected, which, for some of the transformations in this work, are easier to verify than Definition 3.6.

THEOREM 5.6 *Let E be an interaction-connected event net, and $E' = T(E)$, where T replaces subnet S in E with subnet S' . Let c be any non-interaction transition in S' . Then S' is interaction-connected if $\exists$ boundary transition b' in S' , such that c is required to fire in order for b' to be enabled and i) b' is an interaction transition or ii) b' is a non-interaction transition and $\exists$ non-interaction transition b in E ($b = b'$ is possible) such that if b is required to fire in order for some transition e to be enabled, then e is in E' and b' is required to fire in order for e to be enabled in E' .*

Proof: Suppose condition (i) of the theorem is true. Then $\bar{S}'$ is interaction-connected by Definition 3.6.

Suppose condition (ii) of the theorem is true. Then E interaction-connected means $\exists$ interaction transition d in E such that b must fire in order for d to be enabled. But then d is in E' and b' must fire in order that d be enabled in E' . That is, c must fire in order that d be enabled in E' . That is, $\bar{S}'$ is interaction-connected by Definition 3.6. $\square$

The reason that the conditions given in the previous theorem are easier to check is that for certain transformations, condition ii) is automatically satisfied by boundary transitions, and an examination of $\bar{S}'$ is not required. Note that if S' has only interaction transitions, S' is automatically interaction-connected (there is nothing to check). The following corollary to Theorem 5.6 is useful in cases where S' is made up only of boundary transitions.

COROLLARY 5.2 *Let E, E' be as in Theorem 5.6 and suppose the transitions in S' are all boundary transitions. Let b' be any non-interaction boundary transition of S' . Then S' is interaction-connected if $\exists$ non-interaction transition b in E ($b = b'$ is possible) such that if b is required to fire in order for some transition e to be enabled, then e is in E' and b' is required to fire in order for e to be enabled in E' .*

5.3 Proofs of i-Correctness and Deadlock-Freeness

A transformation T preserves i-correctness iff $T(E) \sqsubseteq E$. Given that E is deadlock-free, a transformation T preserves deadlock-freeness iff $T(E)$ is deadlock-free. This section uses

the results of the previous section to show that the transformations defined in Chapter 4 preserve both i-correctness and deadlock-freeness.

The following proofs are given in a two-part format, in which part 1 shows preservation of i-correctness and part 2 shows preservation of deadlock-freeness. To describe the transformations, the proofs use the same notation as used for their definition in Chapter 4.

5.3.1 Abstraction (ABS)

This transformation is defined in Section 4.5.1.

Since the abstraction transformation only hides subnets that do not have interactions, and any execution of $E' = T_{ABS}(E)$ is really an execution of E , it is clear that this transformation preserves i-correctness and deadlock-freeness. It is also clear that if E is interaction-connected, so is $E' = T(E)$.

5.3.2 Refinement (RFM)

This transformation is defined in Section 4.5.2.

i-Correctness: Since T_{RFM} assigns the input events of d to d_k and the output events of d to d_l , and since d_l will be enabled if d_k fires, it follows that T_{RFM} preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Also, S and S' do not participate in interactions. By Theorem 5.2, $E' \subseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4). Also, S' is interaction-connected by construction. Then Theorem 5.4 implies E' is interaction-connected. In addition, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \subseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.3.3 Sequentialization (SEQ)

This transformation is defined in Section 4.5.3.

i-Correctness: Let D be any interaction set in E . Then $\exists$ interaction transition d in E with interaction set D . Now d is either in $\bar{S}$ or S . If d is in $\bar{S}$, then d is in $\bar{S}'$ and hence in E' . If d is in S , d is in S' , since the sequentialization algorithm (SA) does not stop (step 6)

until all transitions in S have been placed in S' . Therefore d is in E' . Further, since SEQ preserves path relations (see discussion for SEQ in Chapter 4), any input available to d in E is also available to d in E' . That is, if d can be enabled in E , d can be enabled in E' . That is, D is an interaction set of E' .

Let D' be any interaction set of E' . Then $\exists$ interaction transition d' in E' with interaction set D' . Now d' is either in $\bar{S}'$ or S' . If d' is in $\bar{S}'$, then d' is in $\bar{S}$ and hence in E . If d' is in S' , d' is in S , since all transitions in S' come from S (the SA uses transitions in S to build S'). Therefore d' is in E . Since E is deadlock-free (either as a requirements specification or by a previous transformation that preserved deadlock-freeness), $\exists$ marking μ' such that d' is live in μ' . That is, D' is an interaction set of E . Therefore every interaction set of E is an interaction set of E' and vice versa. That is, (1): E and E' have the same interaction sets.

Let a, z be any interaction transitions in both E and E' , with interaction sets A and Z respectively. Then SEQ preserves path relations means that if $[a, z]$ holds in E , $[a, z]$ also holds in E' . That is, if A occurs before Z in E , A occurs before Z in E' . Suppose a and z are in S . It may be the case that S does not specify any path between a and z (a and z on different parallel branches). Then A may occur either before or after Z in S , and the user chooses either $[a, z]$ or $[z, a]$ for S' through the SA. That is, the user chooses to have A occurring either before or after Z in S' . That is, where S allows A occurring before Z or Z occurring before A , S' allows only one of these choices. It follows that (2): the interaction sets of E' occur in the same order as they occur in E if E specifies only one occurrence order, or, the interaction sets of E' occur in a subset of their occurrence orders in E if E specifies more than one occurrence order.

Results (1) and (2) imply that $E' \subseteq E$. $\square$

Deadlock-Freeness: It is necessary first to show that E' is interaction-connected. Let d be any non-interaction transition of E' (if every transition of E' is an interaction transition, E' is interaction-connected vacuously). Then the SA implies d is a non-interaction transition in E . Since E is interaction-connected (as a requirements specification or by a previous transformation using Theorem 5.4), $\exists$ interaction transition d' in E such that d must fire in order for d' to be enabled. That is, $[d, d']$ is true for E . But the SEQ transformation implies d' is in E' and SEQ preserves path relations implies $[d, d']$ is true in E' . That is, $\exists$ interaction transition d' in E' such that d must fire in order for d' to be

enabled. That is, E' is interaction-connected.

Now E deadlock-free (as a requirements specification or by a previous deadlock-freeness-preserving transformation) with E and E' interaction-connected and $E' \subseteq E$, means E' is deadlock-free, by Theorem 5.5. $\square$

5.3.4 Subprocess Construction (SPC)

This transformation is defined in Section 4.5.4.

i-Correctness: By definition of SPC, S' for SPC may be obtained by adding $\{ \langle Z_1; A_1 \rangle \}$ to $R_{S'}$ resulting from SEQ, with $\mu(p) = 1$ or $\mu(p) = 0$, where $p : \langle Z_1; A_1 \rangle$. But this addition serves only to delay the re-execution of a transition in A_1 until the execution of a transition in Z_1 is completed. This addition cannot change the interaction sets of S' or their order of occurrence resulting from SEQ. Therefore, since $E' \subseteq E$ for SEQ, $E' \subseteq E$ for SPC. $\square$

Deadlock-Freeness: Again, by definition of SPC, S' for SPC may be obtained by adding $\{ \langle Z_1; A_1 \rangle \}$ to $R_{S'}$ resulting from SEQ, with $\mu(p) = 1$ or $\mu(p) = 0$, where $p : \langle Z_1; A_1 \rangle$. Now the only possible effect of this addition is to delay the re-enabling of a transition in A_1 . Therefore, since E' resulting from SEQ is deadlock-free, E' resulting from SPC is deadlock-free. $\square$

5.3.5 Resource Server (RSV)

This transformation is defined in Section 4.5.5.

i-Correctness: Transitions $d_{i,1}, d_{i,2}$ are enabled in S' iff transition d_i is enabled in S , $1 \leq i \leq n$. Then E' and E have the same boundary event sets with respect to $\bar{S}$ and S' and these event sets occur in E' in the same sequence or sequences as they occur in E . Therefore $\mathcal{T}_{RSV}$ preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Also, S and S' do not participate in interactions (they do not actually interact with the environment they only model access control). By Theorem 5.2, $E' \subseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4).

S' is interaction-connected as follows: For any transition (and hence any non-interaction transition) c in S' , $\exists$ boundary transition $d_{i,2}$ for some i such that c must fire in order for $d_{i,2}$ to be enabled. If transition $d_{i,2}$ is an interaction transition, condition i) of Theorem 5.6 is satisfied. If $d_{i,2}$ is not an interaction transition, transition d_i plays the role of transition b in condition ii) of Theorem 5.6. Since d_i is refined into $d_{i,1}$ and $d_{i,2}$, if e is a transition in E such that d_i must fire in order for e to be enabled, e is a transition in E' such that $d_{i,2}$ must fire in order for e to be enabled in E' (by definition of refinement plus the fact that T_{RSV} has no effect on transitions such as e). Thus, if $d_{i,2}$ is not an interaction transition, condition ii) of Theorem 5.6 is satisfied. Then S' is interaction-connected by Theorem 5.6.

The preceding, along with Theorem 5.4 imply E' is interaction-connected. Also, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \sqsubseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.3.6 Bounded FIFO Channel (BFC)

This transformation is defined in Section 4.6.4.

i-Correctness: This transformation is applied only if the circular dependence conditions or the circular waiting conditions are false. By Theorem 5.3, $E' \sqsubseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4). S' is interaction-connected if both d_s and d_r are interaction transitions (definition of interaction-connectedness satisfied with nothing to check). Otherwise, S' is interaction-connected by Corollary 5.2 with $b = b' = d_s$ or $b = b' = d_r$. Then Theorem 5.4 implies E' is interaction-connected. In addition, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \sqsubseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.3.7 Rendezvous, Recipient Activity Acks (RRA)

This transformation is defined in Section 4.6.5.

i-Correctness: This transformation is applied only if the circular dependence conditions or the circular waiting conditions are false. By Theorem 5.3, $E' \sqsubseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4). S' is interaction-connected if d_{s_1} , d_{s_2} , and d_r are all interaction transitions (definition of interaction-connectedness satisfied with nothing to check). Otherwise, S' is interaction-connected by Corollary 5.2 with $b' = d_{s_1}$ and $b = d_s$, or $b' = d_{s_2}$ and $b = d_s$, or $b = b' = d_r$. Then Theorem 5.4 implies E' is interaction-connected. In addition, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \subseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.3.8 Transporter (TPT)

This transformation is defined in Section 4.6.6.

i-Correctness: Although Theorem 5.3 is not a result concerning the TPT transformation, it turns out that this theorem is still required in the following proof of i-correctness. Hence, the circular dependence conditions or the circular waiting conditions must be false for any application of the TPT transformation.

Define transformations $\mathcal{T}_{TPT1}$, $\mathcal{T}_{TPT2}$ such that

$$E' = \mathcal{T}_{TPT}(E) = \mathcal{T}_{TPT2}(\mathcal{T}_{TPT1}(E)).$$

Transformation $\mathcal{T}_{TPT1}$ is defined as

$$\begin{aligned} R_S &= \{ \langle d_s; d_r \rangle \}, \\ R_{S'} &= R_S \cup \{ \langle d_r; d_s \rangle \} \end{aligned}$$

with $\mu(p) = 1$ where $p : \langle d_r; d_s \rangle$.

Transformation $\mathcal{T}_{TPT2}$ is defined as

$$\begin{aligned} R_S &= \{ \langle d_s; d_r \rangle, \langle d_r; d_s \rangle \}, \\ R_{S'} &= (R_S \setminus \{ \langle d_s; d_r \rangle, \langle d_r; d_s \rangle \}) \cup \{ \langle d_s; a_1 \rangle, \langle a_1; d_r \rangle, \langle a_1; a_2 \rangle, \\ &\quad \langle d_r; a_2 \rangle, \langle a_2; a_1 \rangle, \langle a_2; d_s \rangle \} \end{aligned}$$

with $\mu(p) = \mu(p_1) = \mu(p_2) = 1$ for places $p : \langle d_r; d_s \rangle$, $p_1 : \langle a_2; d_s \rangle$, and $p_2 : \langle a_2; a_1 \rangle$.

Let $E_1 = \mathcal{T}_{TPT1}(E)$ and of course $E' = \mathcal{T}_{TPT2}(E_1)$. Since either the circular dependence conditions or the circular waiting conditions are false, Theorem 5.3 implies $E_1 \subseteq E$. It is necessary to show that $E' \subseteq E_1$.

In the definition of $\mathcal{T}_{TPT_2}$, d_s is enabled in S' iff d_s is enabled in S , and d_r is enabled in S' iff d_r is enabled in S . It follows that $\mathcal{T}_{TPT_2}$ preserves boundary event sets. Further, since S and S' for $\mathcal{T}_{TPT_2}$ do not participate in interactions, it follows that $E' \subseteq E_1$ by Theorem 5.2. Therefore $E' \subseteq E_1 \subseteq E$ which implies $E' \subseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4). Also, S' (for $\mathcal{T}_{TPT}$) is interaction-connected by Theorem 5.6 as follows: transition a_1 is required to fire in order for d_r to be enabled and transition a_2 is required to fire in order for d_s to be enabled; thus condition i) of Theorem 5.6 is satisfied if d_s and d_r are interaction transitions; otherwise, condition ii) of Theorem 5.6 is satisfied for $b = b' = d_s$ with $c = a_2$, $c = d_r$ or $b = b' = d_r$ with $c = a_1$, $c = d_s$. Then Theorem 5.4 implies E' is interaction-connected. In addition, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \subseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.3.9 Bounded FIFO Buffer (BFB)

This transformation is defined in Section 4.6.7.

i-Correctness: Although Theorem 5.3 is not a result concerning the BFB transformation, it turns out that this theorem is still required in the following proof of i-correctness. Hence, the circular dependence conditions or the circular waiting conditions must be false for any application of the BFB transformation.

Define transformations $\mathcal{T}_{BFB1}$, $\mathcal{T}_{BFB2}$ such that

$$E' = \mathcal{T}_{BFB}(E) = \mathcal{T}_\lambda \circ \mathcal{T}_{B2}(\mathcal{T}_{BFB1}(E)).$$

Transformation $\mathcal{T}_{BFB1}$ is defined as

$$R_S = R_S \text{ for } \mathcal{T}_{BFB},$$

$$\begin{aligned} R_{S'} = (R_S \setminus \{ \langle d_s; d_r \rangle \}) \cup \{ & \langle d_s; a_1 \rangle, \langle a_1; a_2 \rangle, \\ & \langle a_2; a_1 \rangle, \langle d_{r_1}; a_2 \rangle, \langle a_2; d_{r_2} \rangle, \\ & \langle d_{r_1}; d_{r_2} \rangle \} \end{aligned}$$

where $\langle d_{r_1}; d_{r_2} \rangle$ is a refinement of d_r and $\mu(p_2) = n$ for place $p_2 : \langle a_2; a_1 \rangle$.

Transformation $\mathcal{T}_{BFB2}$ is defined as

$$R_S = \{ \langle d_s; a_1 \rangle \},$$

$$R_{S'} = (R_S \setminus \{ \langle d_s; a_1 \rangle \}) \cup \{ \langle d_{s_1}; a_1 \rangle, \langle d_{s_1}; d_{s_2} \rangle, \langle a_1; d_{s_2} \rangle \}$$

Let $E_1 = \mathcal{T}_{BFB1}(E)$ and of course $E' = \mathcal{T}_{BFB2}(E_1)$. Since either the circular dependence conditions or the circular waiting conditions are false, Theorem 5.3 implies $E' \subseteq E_1$. It is necessary to show that $E_1 \subseteq E$.

In the definition of $\mathcal{T}_{BFB1}$, d_s is enabled in S' iff d_s is enabled in S , and d_{r_1}, d_{r_2} are enabled in S' (d_{r_1} enabled before d_{r_2}) iff d_r is enabled in S . It follows that $\mathcal{T}_{BFB1}$ preserves boundary event sets with respect to $\bar{S}$ and $\bar{S}'$. Also, S and S' for $\mathcal{T}_{BFB1}$ do not participate in interactions. Then $E_1 \subseteq E$ by Theorem 5.2 and therefore $E' \subseteq E_1 \subseteq E$ which implies $E' \subseteq E$. $\square$

Deadlock-Freeness: E is interaction-connected (as a requirements specification or by a previous transformation in conjunction with Theorem 5.4). Also, S' (for $\mathcal{T}_{BFB}$) is interaction-connected by Theorem 5.6 as follows: transition a_1 is required to fire in order for d_{s_2} to be enabled and transition a_2 is required to fire in order for d_{r_2} to be enabled; thus condition i) of Theorem 5.6 is satisfied if all boundary transitions are also interaction transitions or if just d_{s_2} and d_{r_2} are interaction transitions, since d_{s_1} must fire in order for d_{s_2} to be enabled and d_{r_1} must fire in order for d_{r_2} to be enabled; otherwise, condition ii) of Theorem 5.6 is satisfied for $b = d_s, b' = d_{s_2}$ with $c = d_{s_1}, c = a_1$ or $b = d_r, b' = d_{r_2}$ with $c = d_{r_1}$ or $c = a_2$. Then Theorem 5.4 implies E' is interaction-connected. In addition, E is deadlock-free (as a requirements specification or by a previous transformation) and $E' \subseteq E$. Then Theorem 5.5 implies E' is deadlock-free. $\square$

5.4 Chapter Summary

In this chapter, i-correctness has been derived for measuring the behaviour of a transformed event net with respect to the pre-transformed net. This measure has its roots in weak observational equivalence but is not itself an equivalence relation since it is not symmetric. As a measure of behaviour, i-correctness is applicable to comparing the successive behaviours of processes along the design trajectory from requirements specification to proto-design. This usage means that i-correctness cannot be symmetric, since these

successive behaviours are not symmetric – they become more and more deterministic with each additional design decision made. Next, concepts such as boundary transitions, preservation of boundary event sets, circular waiting conditions, and circular dependence conditions were introduced and used to formulate and prove various results. Finally, these concepts and results were used to show that the transformations defined in Chapter 4 preserve i-correctness and deadlock-freeness.

In the next chapter, a procedure for applying the transformations of this work to design software will be given.

Chapter 6

Using the Approach

This chapter shows how transformations may be used for software design. A procedure is presented for transforming a given process specification (with all required properties as listed in Section 4.3) into a proto-design. In addition, a translation is defined for converting a proto-design into a set of communicating state machines.

6.1 Transforming the Process Specification

This section gives a procedure for applying the transformations presented in Chapter 4, to transform an event net process specification (PS for short) into a proto-design. The procedure suggests an order of application for the transformations (e.g., communication transformations may only be applied after subprocess construction transformations), and explains how the transformations may be used to achieve the design goals.

Environmental components in the PS do not take part in the transformations, i.e., they are not part of any subnet S to be replaced by a transformation. Nevertheless, environmental components must be retained in the PS because they are part of requirements, i.e., removing them from the PS could lead to a proto-design that fulfills a different set of requirements from that expressed by the PS.

6.1.1 Procedure for Transformation

The procedure for transformation is a framework in which individual tools, the transformations, may be used to accomplish the work of turning a PS into a proto-design. The procedure produces a proto-design which is not necessarily the *best* proto-design for meeting all of the design goals. The resulting proto-design is i-correct wrt the requirements

specification, but its quality and performance depend on the skill and knowledge of the designer, as reflected in the designer's application of the transformations.

A PS may be transformed into a proto-design as follows:

Adjustment of Detail

Apply the abstraction and refinement transformations to the event net to adjust the level of activity detail. Abstraction reduces activity detail, which may improve clarity. Refinement increases activity detail, which could implement mechanisms for achieving certain functions or allow a finer grained allocation of timing requirements to activities.

Sequentialization

Apply the sequentialization transformation to the PS as one way of dealing with performance and modularity goals. A particular sequence of event occurrence may be better for performance in that a required event is provided sooner (such a sequence may be surmised through a combination of net simulation and knowing which activity input is required when). Increasing the use of this transformation can result in less parallelism, resulting in fewer subprocesses and hence lower modularity. In some cases, it is better to curb the use of this transformation in order to retain parallelism which is needed for good performance (illustrated by the example below).

Partitioning

As preparation for applying the subprocess construction and communication transformations, identify a partitioning of the PS based on design goals. For example, a high degree of parallelism may be needed for good performance. Certain activities may need to be grouped together and other activities kept apart in order to address cohesion and modularity requirements. Activities may also be grouped together to satisfy data input requirements or the need to execute as separate concurrent processes at different physical nodes of a network.

A partitioning of the PS includes the selection of interprocess communication styles. Such selection is based on logical requirements and design goals. For example, if there is no logical requirement for a message to be acknowledged, and there is a need for fast response, an asynchronous communication style may be chosen.

Emphasizing certain design goals over others leads to the identification of alternate partitionings and ultimately to alternate proto-designs. For example, emphasizing a goal of fast response over a goal of high modularity may result in fewer subprocesses in order to reduce delays caused by interprocess communication.

Once a partitioning is identified which the designer thinks will achieve the design goals, realize this partitioning in the net by applying the subprocess construction and communication transformations to create subprocesses and establish interprocess communication styles. To ensure preservation of i-correctness, each application of a communication transformation is accompanied by a verification that the circular dependence or circular waiting conditions are false, as explained in Section 5.2.1.

A proto-design is obtained after this step if resource access control and performance verification are not required.

Resource Access Control

Apply the resource access control transformation to install specific resource access control mechanisms, such as the use of a resource server where mutual exclusion access to a single resource is required.

A proto-design is obtained after this step if performance verification is not required.

Performance Verification

If performance is a design goal, verify the proto-design for performance by simulating it with a timed Petri net simulation tool. There are many such tools available (e.g., Chiola [12]).

If this verification fails, re-do the above steps until a verification passes. Re-doing these steps can identify new partitionings missed earlier or improve performance by exploiting certain degrees of freedom that exist in the application of the transformations. For example, as already mentioned, sequentialization may schedule certain events ahead of others for better response. Subprocess construction and communication may increase the number of subprocesses or reduce communication delay by replacing synchronous communication with asynchronous communication. If this verification fails even after re-doing the above steps a number of times, the performance requirements may need to be relaxed, and the above steps repeated.

6.1.2 Order of Steps

Figure 6.1 shows the procedure for transformation as a finite state machine, in which abstraction, refinement, and sequentialization are alternated with identifying a partitioning. Communication and resource access control are done after subprocess construction. Performance verification is done after a proto-design is obtained, provided timing data are available. This state machine assumes that it is possible to pass performance verification (if performance verification is done) after re-doing the transformations a finite number of times.

Figure 6.1: Procedure for transformation described by an FSM.

Example

Consider the following textual requirements for a software system:

"The software system allows a single user to enter two numbers x , y and receive back $f(x)$, $g(y)$, and $h(x, y) = f(x) + g(y)$. The user does not care in which order the values x and y are entered, nor does the user care in which order the results $f(x)$ and $g(y)$ are returned. The computation times for $f(x)$, $g(y)$, and $h(x, y)$ are 3, 2, and 2 respectively. The total elapsed time from the input of x and y to the output of $h(x, y)$ must not exceed 7."

Corresponding to this textual description of requirements, the PS of Figure 6.2 is given.

Figure 6.2: Example PS.

The activities of this PS have execution times as follows: activity producing OG: 2, activity producing OF: 3, and activity producing OH: 2. The procedure for transformation was applied to this PS as follows:

1. *Sequentialization*: To reduce parallelism (thought to be in excess), the sequentialization transformation was applied to the PS, sequencing events IY before IX and OG before OF, resulting in the net of Figure 6.3.
2. *Adjustment of Detail*: The two activities from the previous net with events ST and RY were abstracted into a single activity to simplify the net; the activity involving

Figure 6.3: Example sequentialization.

events IX and OF were refined into 2 activities to assign times for the input of IX and the computation generating OF. This step gave the net in Figure 6.4.

Figure 6.4: Example adjustment of detail.

3. **Partitioning:** Modularity, response time, and functional requirements suggested that a possible partitioning was to have a separate subprocess for each computation. Rendezvous was selected as the IPC mechanism in order that each message be acked, which was considered good design practice (another reason might have been that the implementation language supports only rendezvous IPC).

Accordingly, the subprocess construction transformation was applied to the net in Figure 6.4 to create three separate subprocesses, as in Figure 6.5.

After the circular dependence conditions were checked for each communication and found to be false each time (using the procedure in Section 5.2.1) the "Rendezvous - Recipient Acks" transformation was applied to the net in Figure 6.5 giving the net in Figure 6.6.

Figure 6.5: Example subprocess construction.

Figure 6.6: Example installation of communication style.

4. *Resource Access Control*: This step was not needed since there was no resource access control modeled in the PS. The proto-design is then the previously obtained net.
5. *Performance Verification*: Suppose that the proto-design was subjected to a performance verification using a Petri net tool and it was found that timing requirements as specified were not met.
6. The procedure for transformation was repeated on the PS and a new design obtained as follows:

- (a) *Sequentialization*: sequentialization was not applied, based on a perceived need to retain the parallelism in the PS, since the previous proto-design did not meet timing requirements.
- (b) *Adjustment of Detail*: This step was not applied - this time there was no perceived need to adjust activity complexity levels.
- (c) *Partitioning*: A partitioning to retain the parallelism present in the PS was identified in order to improve the response time. To further improve the response time it was decided that interprocess communication employ the asynchronous "Send-No-Wait/Receive-Wait" style.

Accordingly, the subprocess construction transformation gave the net in Figure 6.7.

Since "Send-No-Wait/Receive-Wait" is modeled by default, no communication transformation was needed. It was therefore not necessary to check the circular dependence or circular waiting conditions.

- (d) *Resource Access Control*: This step was not needed - no resource access control was modeled in the PS. The new proto-design is then the previously obtained net.
 7. *Performance Verification*: Suppose that the new proto-design is subjected to a performance verification and this time it is found that timing requirements are met.
- The new proto-design has more concurrency than the previous design, due to the decision not to apply sequentialization. It satisfies the PS in terms of i-correctness (by i-correctness-preserving transformations) and achieves the design goals of fast response and high modularity.

Figure 6.7: Example re-doing of transformations.

6.1.3 Automation

Despite the dependence of the procedure for transformation on user knowledge and experience, a computer tool can significantly facilitate its use. Such a tool would essentially provide three sets of capabilities: i) event net construction, display, editing, storage/retrieval, and printing, ii) application of the transformations under user direction, and iii) simulation and performance evaluation. Capability sets i) and iii) are already provided by Petri net tools such as GSPN1.4 (see Chiola [12]), although GSPN has a slightly different operational semantics from event nets. GSPN tokens do not move until the transition fires, whereas event net input tokens are absorbed as soon as the transition is enabled (event net output tokens, like GSPN output tokens, are not produced until the transition fires). This difference manifests itself only visually in the movement of input tokens and does not invalidate the use of GSPN1.4 for event net simulation and performance evaluation. However, the visual representation of predicate transitions might be added to GSPN1.4, even though the actual evaluation of predicates is not needed since they are fired non-deterministically. For capability set ii), the following features are desirable in a computer

tool:

- automatic application of transformations under user direction, with the following implications for transformations:
 - Sequentialization - allow the user to select which activities to sequentialize and in what order;
 - Abstraction - allow the user to select which activities to abstract;
 - Refinement - allow the user to select i) which activity to refine, and ii) what refined structure to use (a number of standard refinements would be stored in a library); if the desired refinement is not in the library, allow the user to construct the refinement and then optionally store it in the library for future use;
 - Subprocess Construction - invoke sequentialization and automatically add the arcs required for cycling;
 - Communication - allow the user to select i) the communication path which will be transformed, and ii) the allowable communication transformation to use (a number of communication transformations would be stored in the library) after automatically checking the circular dependence or circular waiting conditions;
 - Resource Access Control - allow the user to select i) the resource access control to be transformed, and ii) the resource access control transformation to use (a number of resource access control transformations would be stored in the library);
- an “undo” feature for use after applying a transformation, which would restore the net back to the way it was prior to applying the transformation, would be useful;
- online help on transformation selection by providing estimated execution times, especially for communication and resource access control transformations; for example, if fast response is a design goal and the designer is selecting which communication transformation to use, estimated execution time for point-to-point messaging for each communication style would assist the designer in choosing the appropriate style and transformation;

- proceeding beyond the previous user help feature, the computer tool could possess an expert system capability for partitioning the event net to achieve the design goals; to implement such an expert system, a list of partitioning rules corresponding to different design goals could be assembled and a Prolog program written to suggest different partitionings based on the structure of the PS and the design goals; this feature requires further research.

6.2 Translating to Communicating Finite State Machines

Finite state machines (FSMs) have been used successfully for the specification of concurrent software. In addition, FSMs are appreciated for their simplicity and effectiveness for specifying the “what” of software behaviour without undue emphasis on the “how” of achieving that behaviour. It is therefore useful to be able to translate from event nets to FSMs.

The translation given here maps an event net proto-design into a set of communicating finite state machines (see Brand and Zafiropulo [5]) in which an event produced by one FSM is used as input by another FSM via a FIFO duplex channel. The designation “communicating” is applied to an FSM which communicates with at least one other FSM in this way.

The translation obtains communicating state machines with transitions identified by “condition/action” labels. A condition may consist of the input or receiving of events, a predicate, or both input (receiving) of events and predicate, with “?” meaning “input” or “receive”. An action consists of the output or sending of events, with “!” meaning “output” or “send”.

DEFINITION 6.1 *A finite state machine (FSM) is a 5-tuple $(Q, \Sigma, \Delta, \theta, \gamma)$ where*

Q is a finite set of states $\{q_1, q_2, \dots, q_n\}$,

Σ is a finite alphabet of input events,

Δ is a finite alphabet of output events,

θ is the next state function,

$\theta : Q \times \Sigma^ \rightarrow Q$, mapping the current state*

and set of input events into the next state,

where Σ^ is the set of all subsets of Σ ,*

γ is the output function,

$\gamma : Q \times \Sigma^ \rightarrow \Delta^*$, mapping the current state*

and set of input events into a set of output events,

where Δ^ is the set of all subsets of Δ .*

This definition is seen to be equivalent to the definition of an FSM in Peterson [38] by taking each of Peterson's symbols to represent a set of events.

The translation from a proto-design to a set of communicating FSMs follows. The communication channel between two communicating FSMs is taken as defined in Brand and Zafiropulo [5].

DEFINITION 6.2 *Let C be an event subnet with structure $(P_C, A_C, T_C, I_C, O_C)$ modeling a sequential subprocess in a proto-design (note: A_C does not contain environmental components). Then a corresponding communicating FSM $M_C = (Q_C, \Sigma_C, \Delta_C, \theta_C, \gamma_C)$ is obtained from C by setting*

$$\Sigma_C = \{\text{input interactions of } C\} \cup \{\text{message events from other subprocesses}\} \\ \cup \{\text{predicates of predicate transitions in } C\},$$

$$\Delta_C = \{\text{output interactions of } C\} \cup \{\text{message events to other subprocesses}\},$$

$$Q_C = \{p \mid p : < d_i; d_j >, d_i, d_j \in A_C \cup T_C, d_j \text{ participates in events of } \Sigma_C \cup \Delta_C \\ \text{or has a predicate in } \Sigma_C\},$$

$$\theta_C \text{ defined by } \theta_C(p, \sigma) = p' \text{ where } p, p' \in Q_C, \sigma \in \Sigma_C^* \cup \Delta_C^*, \text{ and } p \in I_C(d), \\ p' \in O_C(d') \text{ for some transitions } d, d' \in A_C \cup T_C \\ \text{with either i) } d = d' \text{ or ii) } [d; d'] \text{ if } d \neq d',$$

$$\gamma_C \text{ defined by } \gamma_C(p, \sigma) = \delta \text{ where } p \in Q_C, \sigma \in \Sigma_C^*, \delta \in \Delta_C^*, \text{ and} \\ p \in I_C(d), \delta \text{ is output by } d' \text{ for some transitions } d, d' \in A_C \cup T_C \\ \text{with either i) } d = d' \text{ or ii) } [d; d'] \text{ if } d \neq d'.$$

The definition of Q_C essentially turns into states those places in C which are i) input places of transitions having interactions or message events or ii) input places of predicate transitions. In the latter case, the predicate must be true for the associated FSM transition to take place. The translation ignores non-predicate transitions which deal only with internal events (i.e., such transitions are not represented as FSM transitions). Figure 6.8 gives an example of applying Definition 6.2. It is proper to designate the FSM resulting from the above translation as “communicating”, since a proto-design always contains at least two communicating subprocesses.

If there are no interactions involved in a communication, it may happen that the communication is not annotated with any events at all, since showing non-interaction events is optional. However, the above translation requires at least one message event to be shown in a communication between two subprocesses, in order that the communication be modeled between the two corresponding FSMs. Therefore, prior to converting a proto-design into a set of communicating FSMs, any interprocess communication that does not

(a) Subprocess event net with interactions I1, I2, and messages M1, M2.

(b) Communicating FSM corresponding to the event net in part (a).

Figure 6.8: Translation of a subprocess event net into a communicating FSM.

show any message event must be annotated with at least one message event before the translation can proceed.

Figure 6.9 shows the communicating FSMs translated from the new proto-design of Figure 6.7. Notice that interprocess communication paths have been annotated with message events that were not shown before. An additional example of translating a proto-design into a set of communicating FSMs may be found in Chapter 7.

6.3 Chapter Summary

This chapter has shown how event nets may be used for software design. A procedure has been presented for transforming a PS into a proto-design. The procedure for transformation does not guarantee an optimal proto-design, but it does guarantee one that is i-correct wrt the PS. In addition, a translation has been given for converting a proto-design into a set of communicating finite state machines.

The next chapter will present application examples of the procedure for transformation.

(a) Proto-design of Figure 6.7.

(b) Communicating FSMs corresponding to the proto-design in part (a).

Figure 6.9: Translation of an example proto-design into communicating FSMs.

Chapter 7

Examples

This chapter presents two application examples of the procedure for transformation described in Chapter 6. The first example involves the design of software for a local telephone service. The second example begins from a given PS which was developed by Grigg [18] to model real time control software for a robotic arm. Grigg developed the PS in the course of his master's thesis, as a model of existing control software that he had written. This example serves to demonstrate the transformations on a PS representing actual concurrent real time software.

Both examples were developed with a graphical Petri net construction and analysis tool called GSPN1.4 (Chiola [12]) which runs on SUN workstations. This tool facilitated the construction of the PS and the application of the transformations. Although the transformations have been shown to preserve i-correctness, it was felt that as a matter of interest, empirical evidence that the generated proto-design was indeed i-correct wrt the PS should be obtained using the tool. This evidence was obtained for the control software example, using the simulation capabilities of GSPN1.4.

7.1 Local Telephone Service

7.1.1 Process Specification

Suppose one wishes to design software for a local telephone service in a small town with a population of under five hundred. Because of the small population and geographical area, only one switch is needed, with no requirement for the software to be distributed over different physical sites. Since customers would not tolerate long delays in receiving a dial tone or in establishing a connection, a design goal is fast response. Also, there is an

implicit requirement for some mechanism for establishing a connection between the caller and the callee.

Consider the sequence of events associated with making a local telephone call. This sequence of events may be described by words as follows:

From the caller's point of view, the caller picks up the receiver, i.e. goes off-hook, and receives a dial tone. The caller then inputs the callee's telephone number after which the caller hears either a busy tone or ring-back. In the case of a busy tone, the caller replaces the receiver, i.e. goes on-hook. In the case of ring-back, the caller waits for either the callee to go off-hook or a certain number of rings to pass after which the caller goes on-hook. If the callee goes off-hook, the caller may start talking. Once the conversation is finished, the caller goes on-hook.

From the callee's point of view, the callee may be unavailable to take the call, already talking on the phone, or available to take the call. If available, the callee hears ringing, goes off-hook, and starts talking. Once the conversation is finished, the callee goes on-hook.

Based on this description, a PS for a local telephone service was developed, as shown in Figure 7.1, by defining activities in terms of input/output events and sequencing the activities to model the telephone service. In this PS, "start" and "end" events have been postulated for continuous dial tone, busy tone, ringing, and ring-back. For simplicity, the input of a telephone number was treated as a single event. This PS models a local switch for POTS (Plain Old Telephone Service) with caller (left side of net) and callee (right side of net) environmental components. For the sake of clarity, only one caller and one callee are shown in Figure 7.1, but it is understood that there are actually *many* callers and *many* callees receiving service from the switch.

Figure 7.2 shows the switch extracted from the PS of Figure 7.1. Immediate transitions are used as token sources and sinks connecting the switch to the many subscribers, where a token arriving at the switch is assumed to contain addressing information to enable an associated departing token to be routed correctly to the proper caller or callee. This separation of the software (i.e. the switch) from its environmental components is not recommended in general since it could alter requirements. It is done here only to increase readability (here, the requirements have not changed).

7.1.2 Transformation

The procedure for transformation was applied to the switch subnet (Figure 7.2) as follows:

Figure 7.1: PS for local telephone service.

Figure 7.2: Switch for local telephone service.

1. Adjustment of Detail: In order to establish a mechanism for line connection (one that the designer knows), the refinement transformation was applied to Figure 7.2 resulting in the event net of Figure 7.3, in which the refined subnets are enclosed in dashed rectangles.

Figure 7.3: Refinements for line connection mechanism.

The refinements implemented a mechanism for line connection involving i) checking "line status" (a look-up table), i.e. whether the line corresponding to the callee's number is busy or free, ii) reserving the callee's line if it is free and then updating the callee's line status to "busy", iii) releasing the callee's line if there is no answer, and then updating the callee's line status to "free", and iv) releasing the callee's line once the call is over (finished talking), and then updating the callee's line status to "free". This mechanism requires mutually exclusive access to "line status" and introduces the following events which are all internal to the switch:

RLS - request line status,
GLS - grant line status,
ULS - update line status,
GA - grant access,
FA - finish access.

Numbers are appended to these mnemonics to label distinct events.

It is important to note that the above mechanism for line connection is not unique. The designer can choose to use the refinement transformation to implement *different* mechanisms with possibly different implications for system response. The refinement transformation therefore offers yet another degree of freedom with which to improve performance.

2. Partitioning: considering the need for fast response, and rules of good design practice such as modularity and cohesion, the following subprocesses were arrived at:

subprocess 1 - activity with events OFFH0, SDT;

subprocess 2 - activities with events ITN, EDT, RLS, GLS, GA0, FA0, SBT, ONH0, EBT, ULS0, GA1, FA1;

subprocess 3 - activities with events SRB, SR, ONH1, ULS1, GA2, FA2, ER0, ERB0, ERB1, OFFH1, ER1, ONH2, ONH3, ULS2, GA3, FA3.

Subprocess 1 was formed for the sole purpose of providing dial tone, since subscribers dislike having to wait for a dial tone. Subprocess 2 was identified for cohesiveness and may be called "line identification". Subprocess 3 similarly emphasizes cohesion and may be called "line connect and disconnect".

To illustrate the allocation of IPC style, suppose subprocesses 1 and 2 are shared among all callers (there are many callers and callees) but that there is a separate copy of subprocess 3 for each caller. Then IPC may be handled as follows: Since subprocesses 1 and 2 are shared, the communication between subprocesses 1 and 2 must be fast. Therefore the asynchronous "Bounded FIFO Channel" style with bound three was chosen for use between subprocesses 1 and 2. The bound of three was selected for illustration but could be based on an estimated number of call attempts. Since subprocess 3 is dedicated to each caller it was felt that the "Rendezvous - Recipient

Acks" style for the communication from subprocess 2 to subprocess 3 would not incur excessive delay (subprocess 3 would be quick to "ack"). Also, because there could be many subprocess 3's active in the switch at any one time, it was felt that subprocess 3 should "ack" its message from subprocess 2 for logical integrity. This requirement was another reason for selecting rendezvous here.

Accordingly, the subprocess construction transformation was applied to the previous net to create the subprocesses identified above. After the subprocesses were created, the "Bounded FIFO Channel" with bound three and the "Rendezvous - Recipient Acks" transformations were applied to the resulting net to establish the above identified communication styles. Prior to applying each communication transformation, the circular dependence conditions of Corollary 5.1 were checked using the procedure in Section 5.2.1 to ensure that the transformation would preserve i-correctness (which would preclude the possibility of introducing a wait loop). It was found that each communication path to be transformed failed the circular dependence conditions, ensuring that the subsequent application of the communication transformation preserved i-correctness.

Figure 7.4 shows the result of applying the subprocess construction and communication transformations.

3. Resource Access Control: the Resource Server transformation (RSV) was applied to the event net of Figure 7.4 to implement a server mechanism (see definition of RSV transformation in Chapter 4) for mutual exclusion of access to the line status table. This server is shared by subprocesses 2 and 3, and serves possibly many callers and callees at any one time. The application of the RSV transformation resulted in the net of Figure 7.5.

As can be seen in Figure 7.5, the RSV transformation introduced a new server subprocess 4 to control access to the line status table. Predicate transitions are employed in the server to properly route a GA token back to the requesting subprocess. It is assumed that appropriate source identification information is also carried by each RA token. Also, each arriving RA token implicitly enters a FIFO queue according to the event net model.

Figure 7.4: Partitioning.

Figure 7.5: Resource access control.

Performance Verification

This step was omitted due to the decision to leave out timing requirements. Therefore, the proto-design is the event net of Figure 7.5.

Re-doing the Design

Suppose that timing requirements were included, and performance verification failed due to excessive delays for starting busy tone and ringing. Partitioning is then re-done, in the course of which it is decided that faster busy tone and ringing response is possible if subprocess 2 in Figure 7.5 is divided up into subprocesses 2a and 2b, with "Bounded FIFO Channel" communication, bound three, between them (for the same reasons as given above for this transformation), as shown in Figure 7.6.

Figure 7.6: Re-partitioning.

Using the procedure of Section 5.2.1, it was found that the communication path between subprocesses 2a and 2b fails the circular dependence conditions of Corollary 5.1, allowing the “Bounded FIFO Channel” transformation to preserve i-correctness. Suppose that this new proto-design does indeed pass performance verification. The new proto-design has been obtained by placing more emphasis on fast response time as a design goal.

7.1.3 Conversion to State Machines

The communicating subprocesses in the proto-design of Figure 7.6 were converted into communicating state machines (Figure 7.7) using Definition 6.2.

Figure 7.7: Communicating finite state machines corresponding to Figure 7.6.

7.2 Control Software for a Robotic Arm

Figure 7.8 shows the PS developed by Grigg [18] to model his control software for a robotic arm. This PS is an event net for which Grigg has analyzed activity function and timing and one which he has shown to meet requirements [18]. This net has a high degree of parallelism, especially in the parallel fork branches of transitions T14, T5, and T8.

Based on his analysis of activity function and performance, Grigg arrived at a partitioning of this net into subprocesses. Grigg discusses the functional and performance implications of different partitionings, so these issues are not addressed here, and the transformations for a given partitioning can be addressed directly. On the other hand, to illustrate certain transformations, a partitioning somewhat different from Grigg's was adopted.

7.2.1 Transformation

A proto-design (Figure 7.13) was obtained from the PS as follows: first of all, given the desired partitioning, the sequentialization transformation had to be applied within each subprocess boundary (since each subprocess must be sequential). At the same time, the abstraction transformation was used to reduce the level of detail. Specifically, the following sequentialization and adjustment of detail steps of the procedure for transformation were done:

1. Sequentialization: the sequentialization transformation was applied to the PS three times, giving the following transition firing orders (from left to right):
 - (a) T36, T43, T37, T38, T39;
 - (b) T32, T30, T29, T28, T27, T26, T25, T24;
 - (c) T14, T17, T18, T16, T15, T13.
2. Adjustment of Detail: the abstraction transformation was applied three times to the net resulting from the previous step, giving the abstracted transitions in Table 7.1. Steps 1 and 2 result in Figure 7.9.
3. Sequentialization: the sequentialization transformation was applied to Figure 7.9 once giving the firing order T41, T42, T44 (the marked input-output place of T42 was removed as it no longer served any purpose).

<i>Abstraction</i>	<i>Activities (left-to-right firing order)</i>
A1	T33, T34, T35
A2	T36, T43, T37, T38, T39, T40
A3	T30, T29

Table 7.1: Abstracted transitions in robotic arm example.

3 of/de 3

PM-1 3 1/2"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Figure 7.9: Net resulting from transformation steps 1 and 2.

<i>Abstraction</i>	<i>Activities (left-to-right firing order)</i>
A4	T27, T45
A5	T16, T18, T15, T14, T17, T13
A6	T48, T47

Table 7.2: More abstracted transitions in robotic arm example.

4. **Adjustment of Detail:** the abstraction transformation was applied to Figure 7.9 three times yielding the abstracted transitions in Table 7.2.

Steps 1 to 4 result in Figure 7.10.

5. **Partitioning:** the subprocess construction transformation was applied to Figure 7.10 giving the subprocesses in Table 7.3 (Figure 7.11).
6. **Partitioning:** the communication transformations in Table 7.4 were applied to inter-process communication in Figure 7.11, giving the proto-design of Figure 7.13. (All communication transformations in this table were selected for illustrative purposes only and do not reflect any functional or performance tradeoffs.)

Having selected these transformations, it was necessary to check that their application preserved *i*-correctness (includes the avoidance of wait loops). This check was done by applying the procedure for checking the circular dependence conditions (Section 5.2.1) and drawing the dependence graph of Figure 7.12(a), in which the arrows represent sender-receiver communication paths or interprocess dependence relations found in Figure 7.11. An arc from subprocess C_i to subprocess C_j indicates that $C_j \propto C_i$ in the sense of Definition 5.4.

Recall that in terms of a dependence graph (Section 5.2.1), a communication fails the circular dependence conditions if it is not possible to find an alternate arrow path, possibly chained, from the sender to the receiver, other than the arrow representing the communication itself. For example, the communication from SP2 to SP7 does not fail the circular dependence conditions, since there is an alternate arrow path from SP2 to SP6 to SP7. Accordingly, Table 7.5 gives all communications in Figure 7.11 that do not fail the circular dependence conditions:

Figure 7.10: Net resulting from transformation steps 1 to 4.

<i>Subprocess</i>	<i>Activities (left-to-right firing order)</i>
SP1	A1, A2, T41, T44, T46
SP2	T31, T32, A3, T28, A4, T26, T25, T24, T23
SP3	T21, T22
SP4	T52, T49, A6
SP5	T50, T55
SP6	T7, T51, A5
SP7	T19, T5
SP8	T20, T8
SP9	T6, T53
SP10	T1
SP11	T2
SP12	T3
SP13	T4
SP14	T12
SP15	T11
SP16	T10
SP17	T9

Table 7.3: Partitioning in robotic arm example.

<i>Sender</i>	<i>Receiver</i>	<i>Communication Transformation</i>
SP1	SP2	Bounded FIFO Channel with bound 1
SP2	SP6	Rendezvous - Recipient Acks
SP6	SP7	Rendezvous - Recipient Acks
SP6	SP8	Bounded FIFO Channel with bound 3
SP9	SP2	Transporter
SP9	SP4	Bounded FIFO Buffer with bound 1

Table 7.4: Application of communication transformations in robotic arm example.

Figure 7.11: Net resulting from transformation steps 1 to 5.

(a) Dependence graph corresponding to communication paths in Figure 7.11.

(b) Updated dependence graph after initial rendezvous transformation.

Figure 7.12: Use of dependence graphs.

<i>Sender</i>	<i>Receiver</i>	<i>Alternate Path</i>
SP1	SP2	SP1 to SP3 to SP2
SP2	SP1	SP2 to SP5 to SP4 to SP1
SP2	SP5	SP2 to SP1 to SP3 to SP5
SP2	SP7	SP2 to SP6 to SP7
SP3	SP1	SP3 to SP2 to SP1
SP3	SP2	SP3 to SP1 to SP2
SP3	SP4	SP3 to SP5 to SP4
SP3	SP5	SP3 to SP2 to SP5
SP9	SP2	SP9 to SP4 to SP1 to SP2
SP9	SP4	SP9 to SP2 to SP5 to SP4

Table 7.5: Communications not failing the circular dependence conditions in robotic arm example.

The selected communication transformations are now examined in light of these results. The communication from SP2 to SP6 fails the circular dependence conditions and hence, according to Corollary 5.1, the selected “Rendezvous - Recipient Acks” transformation may be applied without loss of *i*-correctness. After applying this transformation the dependence graph of Figure 7.12(a) needs to be updated with the arrow from SP6 to SP2, which represents the path used by the receiver to “ack” the sender’s message, introduced by the “Rendezvous - Recipient Acks” transformation just applied. It is necessary to update the dependence graph after every communication transformation that introduces a new dependence relation. The updated dependence graph is shown in Figure 7.12(b). Now looking at this updated dependence graph, it is seen that the communication from SP6 to SP8 fails the circular dependence conditions, and again, according to Corollary 5.1, the selected “Bounded FIFO Channel” transformation may be applied without loss of *i*-correctness. Note that the communications in the preceding table which do not fail the circular dependence conditions are also reflected in the updated dependence graph. However, now, according to the updated dependence graph, an additional communication path does not fail the circular dependence conditions, namely the

communication from SP6 to SP7 with alternate path SP6 to SP2 to SP7. Communications which do not fail the circular dependence conditions in a particular dependence graph will not fail them in an updated dependence graph if the update consisted only of adding arrows to the graph and not removing any arrows (removal could result from re-doing communication transformations). However, the updated dependence graph may show additional communication paths which do not fail the circular dependence conditions, as is the case here.

For the remaining selected transformations, the corresponding communications do not fail the circular dependence conditions, as shown by the preceding table and the remark regarding the communication from SP6 to SP7 in the last paragraph. For these transformations, it is necessary to show that they preserve *i*-correctness by failing the circular waiting conditions of Theorem 5.3. This objective is achieved by applying the procedure for bounded number of messages (Section 5.2.1).

For example, in Figure 7.11 the number of messages in the communication channel from SP1 to SP2 can never exceed one, since once T46 fires, T46 can not fire again until A4 has fired, i.e. once there is a message in the channel, that message is received and the channel cleared before the sender sends again. This result is evident from the fact that starting from the initial marking, the order of transition firings for key transitions is T46 followed by A4, followed by T31, followed by A2, followed by T46. Therefore the sender never has to wait and application of the "Bounded FIFO Channel" transformation with bound one fails the circular waiting conditions. Accordingly, by Theorem 5.3, the application of this transformation preserves *i*-correctness. Similarly, the application of the remaining selected communication transformations preserves *i*-correctness.

7.2.2 Experimental Verification of *i*-Correctness

Although the proto-design of Figure 7.13 was obtained by means of *i*-correctness-preserving transformations and hence is guaranteed to be *i*-correct wrt the PS of Figure 7.8, it was felt useful and interesting to verify this *i*-correctness for a number of execution cycles of the proto-design. Accordingly, the proto-design was executed using the simulation capabilities of GSPN1.4 (see Chiola [13]) through 120 simulation steps (sufficient to capture execution cycling), in which each step enabled or fired some transitions. At each step, if there were

Figure 7.13: Proto-design for control of robotic arm.

more than one transition enabled, the simulator selected the transition to fire next as the one which would contribute the least execution time to the overall execution time. The execution trace of this simulation is given in Table 7.6.

It follows from the definition of i-correctness (Definition 5.1), that a proto-design is i-correct wrt its PS if all of the following conditions are satisfied:

1. all transitions fired in the PS are fired in the proto-design;
2. any transition fired in the proto-design that is not fired in the PS does not involve interactions; and
3. the transitions referred to in condition 1 are fired in the proto-design in the same order as they are fired in the PS, or they are fired in one of a number of possible orders defined by the PS.

Using these criteria to test for i-correctness, a verification of i-correctness may proceed by comparing the transitions fired in the proto-design (Table 7.6) to the transitions fired in the PS (Figure 7.8), confirming first that condition 1 is satisfied and then confirming that conditions 2 and 3 are satisfied for each of the 120 simulation steps. This verification was done manually. The results of the comparison for conditions 2 and 3 are included in Table 7.6 under the "iCC" (for i-Correctness Compatibility) column, in the form of a letter code giving the reason for i-correctness compatibility. This letter code is as follows:

- A - same firing order as in PS;
- B - one of a number of firing orders defined by PS;
- C - does not involve interactions.

For example, transition T28 is fired in step 2. Since the PS does not impose any precedence order among transitions T46, T28, T21, which are all enabled in step 1, T28 is fired in step 2 in one of a number of possible orders, and accordingly, iCC receives the letter B. On the other hand, the PS requires T22 to fire after T21 upon initialization. Since T22 is fired (step 4) after T21 is fired (step 3), as required by the PS, iCC for step 4 receives the letter A. There are cases in which a transition is fired after some other transition, as required by the PS, but at the same time, it is fired as one of a number of possible orders of firing with respect to a third transition with which it was forked. In

<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>	<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>
1		T46 T28 T21		61	TP1	TP2	C
2	T28		B	62	T8	T12 T11 T10 T9	A
3	T21	T22	B	63	T12		A
4	T22	T21 T49 T50	A	64	A6	D3	A
5	T49		A	65	D3		C
6	T21		B	66	TP2		C
7	T50	A6	A,B	67	T11		A,B
8	A6	D3	A	68	T10		A,B
9	D3		C	69	T26	T25	A
10	T46	A1 A4	B	70	T9	T7	A,B
11	A1		A	71	T7	T51	A
12	A4	A5	A	72	T51		A
13	A5	T19 T20 D1	A	73	T25	T24	A
14	T20	T8	A	74	T24	T23 T55	A
15	T8	T12 T11 T10 T9	A	75	T55		A
16	T12		A	76	T23	T31 T5	A
17	T10		A,B	77	T5	T1 T2 T3 T4	A
18	D1	T26	C	78	T4		A
19	T19	D2	A,B	79	T3		A,B
20	T9		A,B	80	T2		A,B

Table 7.6: Execution trace for verification of i-correctness.

<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>	<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>
21	D2		C	81	T31	A2 T32	A
22	T26	T25	A	82	A2	T41	A
23	T25	T24	A	83	T1	T6	A,B
24	T24	T23 T55	A	84	T6	BF2	A
25	T55		A	85	T41	T42	A
26	T23	T31 T5	A,B	86	T42	T44 T22	A
27	T5	T1 T2 T3 T4	A	87	T22	T21 T50	A
28	T3		A	88	T21		A
29	T11	T7	A,B	89	BF2	D4 BF1	C
30	T4		A,B	90	T44	T46	A,B
31	T2		A,B	91	T46	A1	A
32	T1	T6	B	92	BF1	T52	C
33	T7	T51	A	93	D4	T53	C
34	T51		A	94	T52	T49	A
35	T31	A2 T32	A,B	95	T32	A3	A
36	T32	A3	A	96	T50		A,B
37	A3	T28	A	97	A1		A
38	T28		A	98	T49	A6	A
39	A2	T41	A	99	T53	TP1	A
40	T6	BF2	A	100	A6	D3	A

Table 7.7: Execution trace (cont'd).

<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>	<i>Step</i>	<i>Fired</i>	<i>Enabled</i>	<i>iCC</i>
41	T41	T42	A	101	A3	T28	A
42	T42	T44 T22	A	102	TP1	TP2	C
43	T22	T21 T50	A	103	D3		C
44	T44	T46	A	104	T28	A4	A
45	T21		A	105	TP2		C
46	T46	A1 A4	A	106	A4	A5	A
47	A1		A	107	A5	T19 T20 D1	A
48	BF2	BF1 D4	C	108	T20	T8	A
49	BF1	T52	C	109	D1	T26	C
50	D4	T53	C	110	T26	T25	A
51	A4	A5	A,B	111	T8	T12 T11 T10 T9	A
52	T52	T49	A	112	T25	T24	A
53	T50		A	113	T11		A
54	T49	A6	A	114	T9		A,B
55	A5	T19 T20 D1	A	115	T10		A,B
56	D1	T26	C	116	T12		A,B
57	T19	D2	A	117	T24	T23 T55	A
58	T53	TP1	A	118	T23	T31	A
59	D2		C	119	T19	T5 D2	A,B
60	T20	T8	A	120	T55		A,B

Table 7.7: Execution trace (completed).

such cases, iCC is labeled by both A and B. For example, in step 7, T50 is fired after T22 as required by the PS. However, at the same time, T50 is fired after T49 in step 5, which is one of a number of possible orders of firing, since T49 could just as well be fired after T50 according to the PS. Notice that transitions T49 and T50 are both forked by transition T22.

The above comparisons of transitions fired and transition firing orders satisfied conditions 1, 2, and 3. Accordingly, the simulation verified that the proto-design is indeed i-correct wrt the PS.

Although the above manual comparison may be automated as a means of verifying i-correctness, there is no need for such an automated procedure on an ongoing basis. The proto-design is guaranteed to be i-correct wrt the PS provided that it was obtained using i-correctness-preserving transformations, as discussed in Chapters 5 and 6. The verification was carried out here only as a matter of interest.

Conversion to State Machines

The communicating subprocesses in the proto-design of Figure 7.13 may be converted into communicating state machines to give a state machine description of process behaviour, as was done in the first example. This conversion has been omitted here as it would add little new insight into the conversion procedure.

7.3 Chapter Summary

This chapter has presented two examples of applying the procedure for transformation defined in Chapter 6. The first example obtained a proto-design for a local telephone service. The second example of transforming a PS modeling actual control software for a robotic arm also verified the i-correctness of the resulting proto-design wrt the PS using simulation. In addition, the latter example illustrated the application of the circular waiting conditions to ensure that communication transformations preserved i-correctness. The circular waiting conditions were shown to be false by i) showing that the circular dependence conditions were false and ii) reasoning that the number of messages traversing a communication channel was bounded.

The next and final chapter presents conclusions on the results of this research.

Chapter 8

Conclusions

This chapter summarizes the approach taken in this research in terms of results, strengths, and weaknesses. In addition, it is shown that the use of event nets for specification and design is part of an operational approach to software development.

8.1 Results

The following problem was posed in Chapter 1:

“Given a formal software requirements specification of a process, expressed using a suitable FDT (Formal Description Technique), how can this specification be directly transformed into a proto-design (i.e., a description in terms of concurrently running sequential subprocesses) that is behaviourally correct with respect to the requirements specification and satisfies all design concerns, including performance?”

This research has solved the above problem by providing a set of transformations (Chapter 4) for use in partitioning an event net process specification into a set of sequential subprocesses which run concurrently, using design goals to constrain the partitioning. The resulting partitioned specification or proto-design is behaviourally correct (i-correctness) with respect to the original process specification, is deadlock-free, and satisfies all design goals, including performance.

In terms of research contributions (mentioned in Chapter 1), this work has

- obtained event nets from timed nets for use in expressing software requirements (Section 3.3);

- defined i-correctness, a formal concept of behaviour correctness between process specifications appropriate for showing that transformations preserve correctness (Section 5.1.2);
- defined a procedure for applying the transformations to a process specification, to obtain a proto-design that takes into account design concerns, including performance and deadlock-freeness (Section 6.1);
- defined a translation of the proto-design into a set of communicating finite state machines (Section 6.2);
- demonstrated the transformation procedure on real life examples (Chapter 7).

8.2 Strengths of this Approach

The approach taken in this research has the following strengths (weaknesses are identified below):

- Event Nets:
 - based on Petri nets, which are well-supported in the literature;
 - integrated model representing both control and functionality;
 - can be executed, and existing Petri net simulators facilitate validation and performance verification;
 - may be analyzed for temporal error (e.g., deadlock) using existing Petri net tools for reachability analysis;
 - may be analyzed for performance using existing timed Petri net tools;
 - an event net proto-design may be translated into communicating finite state machines as shown in this work.
- Transformations:
 - preserve behaviour correctness and deadlock-freeness (provided communication transformations are applied only after checking for wait loops);
 - are open-ended, i.e., new transformations may be defined and shown to preserve i-correctness as the need arises;

- some may be automated (e.g., Communication and Resource Access Control).
- Procedure for Transformation:
 - directly transforms an event net process specification into an event net proto-design which is automatically behaviourally correct with respect to the specification and is deadlock-free;
 - incorporates the knowledge and experience of the designer to establish desired functionality (e.g., line status detection in the switch example) and improve performance (through the exercise of degrees of freedom in applying the transformations);
 - is in principle a candidate procedure for rapid-prototyping (Taylor and Standish [47], Scharer [43]) since event nets are executable;
 - using the term “object-based design” to identify the use of objects to hide data and functionality behind interfaces, the procedure addresses the identification of objects, such as subprocesses, from a flat specification.

8.3 Weaknesses of this Approach

The most serious weakness of this partitioning approach is that it has not had the benefit of wide usage so that it can be ascertained empirically whether or not the approach works in all cases. Specifically, one can ask the following questions: i) Is the event net model expressive enough for the specification needs of all concurrent and distributed software?, ii) Is it straightforward to come up with an event net specification (PS) as was done in the switch example?, iii) Is i-correctness a reasonable definition of behavioural correctness?, iv) Are the degrees of freedom present in the application of transformations sufficient to allow the designer to iterate and arrive at a proto-design that satisfies performance requirements?, v) How well does a proto-design map onto implementation in code, i.e., is it easy to turn a proto-design into a design, so that it can be implemented in code?, vi) How well does the approach size-up for use on large applications (e.g., event net diagrams for these applications may grow to unmanageable proportions)? Further research is required to answer these questions.

Another weakness concerns the detection of possible wait loops, prior to introducing a particular communication style, through checking the circular dependence or circular

waiting conditions. Verification of failing the circular waiting conditions by reachability analysis (where the circular dependence conditions and the bounded number of messages method (Section 5.2.1) cannot be used) is not appealing, especially where the reachability space is large.

8.4 Operational Approach to Software Development

The operational and conventional approaches for software development have been explained and compared by Zave [52]. The proposed partitioning methodology given in Chapter 6 is part of an operational approach to software development, based on the following description of specification and design in the operational approach given by Zave.

- The specification is operational (i.e., executable) and specifies a system in terms of implementation-independent structures (satisfied by an event net specification).
- The structures provided by the FDT are independent of specific resource configurations or resource allocation strategies (during design implementation, i.e., coding) (true for event nets as the FDT).
- The operational specification is problem-oriented and not implementation-oriented (true for an event net specification).
- The operational specification undergoes transformations that preserve its external behaviour but which change or augment the mechanisms by which that behaviour is produced, yielding an implementation-oriented specification of the same system (true for the transformational approach of this work).
- The transformations introduce explicit representations of implementation resources or resource allocation mechanisms that were absent from the original specification (true for this work, e.g., Resource Server transformation establishes a server subprocess to control resource access).
- The same FDT is used for both the operational specification and the transformed specification (event nets are used for both).

Bibliography

- [1] G. Andrews and F. Schneider, "Concepts and notations for concurrent programming," *ACM Computing Surveys*, Vol. 15, pp. 3-43, March 1983.
- [2] G. Berthelot, "Checking properties of nets using transformations," in *Lecture Notes in Computer Science*, Vol. 222, Springer-Verlag, pp. 19-40, 1986.
- [3] G. Berthelot, "Transformations and Decompositions of Nets," in *Lecture Notes in Computer Science*, Vol. 254, Springer-Verlag, pp. 359-376, 1987.
- [4] T. Bolognesi and S. Smolka, "Fundamental results for the verification of observational equivalence: a survey," *Proceedings of the 7th IFIP International Workshop on Protocol Specification, Testing, and Verification*, pp. 165-179, 1987.
- [5] D. Brand and P. Zafiropulo, "On communicating finite state machines," *Journal of the ACM*, Vol. 30, No. 2, pp. 323-342, April 1983.
- [6] B. Brass, F. Erhart, A. Horsch, H. Riethmayer, and R. Steinbruggen, "CIP-S: an instrument for program transformation and rule generation," Report I8211, Technical University of Munich, 1982.
- [7] E. Brinksma, "A tutorial on LOTOS," in *Protocol Specification, Testing, and Verification, V*, pp. 171-194, North Holland, 1986.
- [8] E. Brinksma, G. Scollo, and C. Steenberg, "LOTOS specifications, their implementations and their tests," in *Protocol Specification, Testing, and Verification, VI*, edited by B. Sarikaya and G. Bochmann, North-Holland, pp. 349-360, 1987.
- [9] F. Brooks, "No silver bullet: essence and accidents of software engineering," *Computer*, Vol. 20, No. 4, pp. 10-19, April 1987.

- [10] G. Bruno and G. Marchetto, "Process-translatable Petri nets for the rapid prototyping of process control systems," *IEEE Trans. on Software Engineering*, Vol. SE-12, pp. 346-357, February 1986.
- [11] R. Burstall and J. Darlington, "A transformation system for developing recursive programs," *Journal of the ACM*, Vol. 24, pp. 44-67, January 1977.
- [12] G. Chiola, "A graphical Petri net tool for performance analysis," in *Proc. Third Int. Workshop on Modeling Techniques and Performance Evaluation*, (France), AFCET, pp. 323-333, May 1987.
- [13] G. Chiola, "GreatSPN Users' Manual, Version 1.3", corso Svizzera 185, 10149 Torino, Italy: Dipartimento di Informatica, Università degli Studi di Torino, September 1987.
- [14] B. Cohen, W. Harwood, and M. Jackson, *The Specification of Complex Systems*. Addison-Wesley, 1986.
- [15] B. Cox, *Object-Oriented Programming - An Evolutionary Approach*. Reading, Mass.: Addison-Wesley, 1986.
- [16] H. Genrich and K. Lautenbach, "System modeling with high-level Petri nets," *Theoretical Computer Science*, Vol. 13, pp. 109-136, 1981.
- [17] W. Gentleman, "Message passing between sequential processes: the reply primitive and the administrator concept," *Software - Practice and Experience*, Vol. 11, pp. 435-466, 1981.
- [18] K. Grigg, "Design of parallel adaptive robotic control software," Master's thesis, Faculty of Engineering, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6, 1990.
- [19] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, Vol. 8, pp. 231-274, 1987.
- [20] D. J. Hatley and I. A. Pirbhai, *Strategies for Real-Time System Specification*. 353 West 12 Street, New York, NY 10014: Dorset House Publishing Co., Inc., 1987.
- [21] M. Hennessy, *Algebraic Theory of Processes*. The MIT Press, 1988.

- [22] C. Hoare, S. Brookes, and A. Roscoe, "A theory of communicating sequential processes," Technical Monograph Prg-16, Computing Laboratory, University of Oxford, 1981.
- [23] C. Hoare, *Communicating Sequential Processes*. London, England: Prentice-Hall International, UK, Ltd., 1985.
- [24] W. Hulzen, P. Tilanus, and H. Zuidweg, "Lotos extended with clocks," *The 2nd International Conference, FORTE'89, Participant's Proceedings*, pp. 230-244, 1989.
- [25] ISO-IEC/JTC1/SC21/WG1/FDT/C, *LOTOS, A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*. ISO International Standard IS8807, February 1989.
- [26] F. Jahanian and A. Mok, "Safety analysis of timing properties in real-time systems," *IEEE Trans. on Software Engineering*, Vol. SE-12, pp. 890-904, September 1986.
- [27] K. Jensen, "Coloured Petri nets and the invariant-method," *Theoretical Computer Science*, Vol. 14, pp. 317-336, 1981.
- [28] M. A. Marsan, G. Balbo, and G. Conte, "A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems," *ACM Trans. on Computer Systems*, Vol. 2, pp. 93-122, May 1984.
- [29] P. Merlin and D.J. Farber, "Recoverability of communication protocols - implications of a theoretical study", *IEEE Trans. on Communications*, Vol. COM-24, pp. 1036-1043, Sept. 1976.
- [30] R. Milner, *A Calculus of Communicating Systems, Lecture Notes in Computer Science*, Vol. 92. Springer-Verlag, 1980.
- [31] M. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Trans. on Computers*, Vol. C-31, pp. 913-917, September 1982.
- [32] T. Murata, "Petri Nets: Properties, analysis and applications," *Proceedings of the IEEE*, Vol. 77, No. 4, pp. 541-580, April 1989.
- [33] R. D. Nicola and M. Hennessy, "Testing equivalences for processes," *Theoretical Computer Science*, Vol. 34, pp. 83-133, 1984.

- [34] J. Noe, "Nets in modeling and simulation," in *Lecture Notes in Computer Science*, Vol. 84, Springer-Verlag, pp. 347-368, 1980.
- [35] J. Noe, "Abstractions of net models," in *Lecture Notes in Computer Science*, Vol. 84, Springer-Verlag, pp. 369-388, 1980.
- [36] H. Partsch and R. Steinbrüggen, "Program transformation systems," *ACM Computing Surveys*, Vol. 15, pp. 199-236, September 1983.
- [37] D. Peng and K. Shin, "Modeling of concurrent task execution in a distributed system for real-time control," *IEEE Trans. on Computers*, vol. C-36, pp. 500-516, April 1987.
- [38] J. Peterson, *Petri Net Theory in the Modeling of Systems*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.
- [39] C. Petri, "Kommunikation mit automaten", PhD thesis, University of Bonn, Bonn, West Germany, 1962. (in German).
- [40] R. Pressman, *Software Engineering, A Practitioner's Approach*. McGraw-Hill, 1987. second edition.
- [41] J. Quemada, A. Azcorra, and D. Frutos, "A timed calculus for lotos," *The 2nd International Conference, FORTE'89, Participant's Proceedings*, pp. 245-264, 1989.
- [42] W. Reisig, "Petri nets in software engineering," in *Lecture Notes in Computer Science*, Vol. 255, Springer-Verlag, pp. 63-96, 1987.
- [43] L. Scharer, "The prototyping alternative," *ITT Programming*, Vol. 1, No. 1, pp. 34-43, 1983. Reprinted in *New Paradigms for Software Development* by W. Agresti, IEEE Computer Society Press, pp. 59-68, 1986.
- [44] J. Stankovic, "Misconceptions about real-time computing: a serious problem for next generation systems," *Computer*, Vol. 21, No. 10, pp. 10-19, October 1988.
- [45] J. Tankoano, "M2C: une approche méthodique pour la conception certifiée des systèmes de commande des automatismes industriels répartis", PhD thesis, Nancy, 1988. Thèse de Docteur d'Etat ès-sciences.
- [46] J. Tankoano et J. Derniame, "Réseaux de petri et applications distribuables," *Techniques et Science Informatiques*, vol. 8, no. 4, pp. 339-359, 1989.

- [47] T. Taylor and R. Standish, "Initial thoughts on rapid prototyping techniques," *ACM SIGSOFT Software Engineering Notes*, Vol. 7, pp. 160-166, December 1982.
- [48] C. Vissers, G. Scollo, and M. v. Sinderen, "Architecture and specification style in formal descriptions of distributed systems," *Proc. IFIP WG6.1 Symposium on Protocol Specification, Testing, and Verification VIII*, Atlantic City, USA, pp. 189-204, June 1988 (North Holland, 1989).
- [49] P. R. Ward and S. J. Mellor, *Structured Development of Real-Time Systems*, Vol. I, II and III. Englewood Cliffs, New Jersey 07632: Prentice-Hall, Inc., 1985.
- [50] C. M. Woodside, "Activity sequence network models for software performance engineering," Dept. of Systems and Computer Engineering, Carleton University, Ottawa, Canada K1S 5B6, October 1988. Internal note.
- [51] S. Yau and M. Caglayan, "Distributed software system design representation using modified Petri nets," *IEEE Trans. on Software Engineering*, vol. SE-9, pp. 733-745, November 1983.
- [52] P. Zave, "The operational versus the conventional approach to software development," *Communications of the ACM*, Vol. 27, No. 2, pp. 104-118, February 1984.

END

2/7/05/9/2

FIN