National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

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.

Designing Reactive Systems for Strong Traceability

by

Joanne Lee Boyd, B.Eng., M.Eng.

A thesis submitted to the
Faculty of Graduate Studies and Research
in partial fulfillment 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, Canada

January 7, 1993

© 1993, Joanne Lee Boyd

**National Library
of Canada**

**Acquisitions and
Bibliographic Services Branch**

**395 Wellington Street
Ottawa, Ontario
K1A 0N4**

**Bibliothèque nationale
du Canada**

**Direction des acquisitions et
des services bibliographiques**

**395, rue Wellington
Ottawa (Ontario)
K1A 0N4**

Your file Votre référence

Our file Notre référence

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.

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.

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 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 0-315-84017-X

Canada

Name

Joanne L. Boyd

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science

SUBJECT TERM

0484

U.M.I.

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0120
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0298
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Plant Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0370
Ecology 0329
Entomology 0353
Genetics 0369
Limnology 0793
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0414
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0390
Chemotherapy 0992
Dentistry 0567
Education 0351
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463
Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal System Science 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

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

Designing Reactive Systems for Strong Traceability

submitted by

Joanne Lee Boyd, B.Eng., M.Eng.

in partial fulfilment of the requirements
for the degree of Doctor of Philosophy

Chair, Department of Systems
and Computer Engineering

Thesis Supervisor

External Examiner

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
March 4, 1993

Abstract

This thesis proposes a design approach for *strong traceability* of behavioural requirements in reactive systems. The method includes software specification, a systematic design approach for translating the specification into a design, and a validation process to illustrate strong traceability between the specification and its design. The thesis suggests that this type of approach is not currently practiced. Current popular trends provide either *complete traceability* (for automatic transformations using formal relations), or *weak traceability* (for ad hoc, and creative design methodologies based on informal, trial-and-error relations). The thesis defines: (1) a software specification model for reactive systems: *Entity Behaviour Descriptions* which is unique in its separation of application-specific behaviour from the semantics of the communications, (2) a verification technique for EBD specifications with unbounded process pools accessing bounded resources, (3) a technique for developing and documenting *design fragments* which represent reusable and formalized design alternatives for communications, (4) a design process which uses design fragments to modularize the design of communications and lead to strong traceability of the behavioural requirements, (5) a validation process to illustrate conformance through the demonstration of completeness and correctness of the design, and (6) a documentation process to support strong traceability. Together, these contributions define a systematic design technique for reactive system software with strong traceability. The process and its components are evaluated by examples which illustrate the following benefits: (i) strong traceability of behaviour with minimized proof requirements, (2) systematic, piece-meal design, (3) creative and critical design decisions left in the body of the design process, and (4) structured documentation. The evaluation also indicates that the process can be improved with practical experience and tool support.

Acknowledgments

I would like to thank my thesis supervisor and friend, Dr. Gerald M. Karam, for his continuous and generous support, his enthusiastic insight, his recommendations, and those periodic pats on the back. I would like to thank all of the professors, staff, students, and researchers of the Department of Systems and Computer Engineering at Carleton for providing a welcome, encouraging, and professional atmosphere in which to study.

I would also like to thank my family and friends for their support, encouragement, and babysitting; with special thanks going to Gareth and David, my perfect husband and son, respectively.

Financial support was provided by the Natural Sciences and Engineering Research Council (NSERC), the Ontario Ministry of Colleges and Universities, the Telecommunications Research Institute of Ontario (TRIO), and Carleton University.

Contents

1	Introduction	1
1.1	Reactive Systems and Strong Traceability	1
1.2	Influential Trends	4
1.3	Objectives: A Method for Strong Traceability	7
1.4	Overview: Designing for Strong Traceability	8
1.5	Contributions	11
1.6	Thesis Outline	13
2	Background	14
2.1	MachineCharts	15
2.2	Specification Models and Methods	21
2.2.1	Entity Modelling	21
2.2.2	STATEMATE and Statecharts	25
2.2.3	Jackson System Development	30
2.2.4	Protocol Specifications	31
2.2.5	Summary	32
2.3	Knowledge Based Software Engineering	35
2.3.1	General Transformation Systems	37
2.3.2	General Programming Environments	38
2.3.3	Summary	40
2.4	Formal Validation Techniques	43
3	Specification of Software Requirements	45
3.1	Specification in the Design Framework	46
3.2	EBDs – A Model for Communicating Entities	50
3.2.1	Architectural Description	51
3.2.2	Entity Behaviour Descriptions	53
3.2.3	EBD Semantics	68
3.2.4	Examples of Communicating Entity Specifications	76
3.2.5	Related Research	82
3.3	Methods for Developing EBD Models	84

3.3.1	Analysing and Decomposing Large Systems	85
3.3.2	Developing EBD Models	89
3.4	Analysis of EBD Specifications	89
3.4.1	Representation Model	93
3.4.2	Reachability Analysis with Process Pools	97
3.4.3	An Example	116
3.4.4	Evaluation	123
4	Designing within the Design Fragments Framework	131
4.1	Allocation	132
4.1.1	Concepts	132
4.1.2	Models	134
4.1.3	Methods	136
4.2	Designing with Design Fragments	138
4.2.1	Relation to Current Work in Knowledge-Based Software Engineering	138
4.2.2	Design Fragments – A Conceptual View	140
4.2.3	Design Fragments – A Separation of Concerns	143
4.2.4	Design Fragments – The Setup	154
4.2.5	Design Fragments – A Detailed View	161
4.2.6	Design Fragments Framework	189
4.2.7	Method for Designing with Design Fragments	190
5	Validation	196
5.1	Validating Correctness	197
5.1.1	Validating Specification Contingencies	197
5.1.2	Validating Design Contingencies	199
5.1.3	Properties and Performance Issues	200
5.1.4	Typical Contingencies	201
5.1.5	Demonstrating Contingencies: Formal Techniques	203
5.1.6	Demonstrating Contingencies: Intuitive Reasoning	205
5.1.7	Example Design – 1: Readers/Writers	209
5.1.8	Example Design – 2: Producer/Consumer	220
5.2	Validating Completeness	225
5.2.1	Partition Tables	227
5.2.2	An Algorithm for Validating Completeness	235
5.2.3	Example Completeness Proof	236
6	Example System Design	239
6.1	Analysing and Modelling the System	239
6.2	Allocation and Decomposition of Work	255

6.3	Tasking Model for Example Subsystem	264
6.4	Summary	274
7	Conclusions	276
7.1	Contributions	278
7.2	Evaluation	282
7.3	Future Research	286
A	EI Details	302
B	Readers/Writers Generalized System States	308
C	Design Fragments Framework	311
C.1	Populating the Framework	311
C.1.1	Inter-Actor Communications — IPC Design Fragments	312
C.1.2	Intra-actor Communications — SEQ Design Fragments	320
C.1.3	Collapsed Behaviours	333
C.2	Framework	334
C.2.1	Asynchronous Communications	334
C.2.2	Synchronous Communications	345
C.2.3	Transient Broadcast Communications	350
C.2.4	Patterns of Interaction	353
C.2.5	Intra-actor Communications	355
C.2.6	Collapsed Behaviours	360
C.3	Framework Extensions	361
D	Final EBDs and Tables for Example Design of Chapter 6	366
D.1	Initial Partition Tables	367
D.2	Final EBDs (if changed)	369
D.3	Final Actor Tables (if changed)	375
D.4	Final Partition Tables	378
D.5	Build Tables	385

List of Figures

2.1	MachineChart Design for an Ada Rendezvous	17
2.2	MachineChart Design Notations	17
2.3	MachineChart Design with ACMs	19
2.4	Illustration of default exits in statecharts	29
2.5	Illustration of Default Entries in statecharts	29
2.6	Examples of statechart history notation	29
3.1	Communications Meta-Model	52
3.2	EBD and EI Views of Event Queuing	53
3.3	EBD segment illustrating transitions from a null state.	62
3.4	EBD segment illustrating a direct flowchart transition.	63
3.5	EBD segment illustrating alterative flowchart transitions.	63
3.6	EBD segment illustrating event driven transitions.	64
3.7	EBD segment illustrating use of message data.	64
3.8	EBD segment illustrating a message data directive.	65
3.9	EBD segment illustrating the ordering and priority concepts.	66
3.10	EBD of an EBD Meta-entity	70
3.11	EBD for the EI	72
3.12	Example 1 - Communicating Entities	77
3.13	Example 2 - An Ada-like Rendezvous	78
3.14	Example 3 - Blocking Send and Non-blocking Receive	78
3.15	Example 4 - Binary Semaphore Interaction	79
3.16	Example 5 - Interrupt Servicing	79
3.17	Example 6 - Illustration of the <i>order</i> and <i>priority</i> directives	80
3.18	EBD Specification for the Readers and Writers Problem	83
3.19	Interfacing to Distributed Entities in a Large Design	88
3.20	Classical Reachability Analysis Process Model Example	94
3.21	Readers/Writers Specification for Verification	96
3.22	Classical Reachability Analysis Matching Process	100
3.23	Expanded versus Compressed Reachability Trees	103
3.24	Counter Value Example	108

3.25	Multiple Conditioned Self-loops	109
3.26	Generalized Matching Process	113
3.27	Algorithm For Ripple Generalizations	115
3.28	Readers/Writers generalized reachability tree	120
3.29	S-state 2 initially generates this new s-state 5	120
3.30	S-state 5 self generalizes to a new version of s-state 5	121
3.31	S-state 14 self-generalizes with s-state C	123
3.32	S-state 14 initially generates s-states A and B as shown	124
3.33	A matches s-state 2, and B matches s-state 14"	125
3.34	Minimized Reachability Tree for Readers/Writers	128
3.35	Deadlock Specification for the 2 of 3 Problem	130
4.1	Actor Table Example	135
4.2	Designs Illustrating Fragments For Asynchronous Event Transfer	141
4.3	Three designed components of an actor	145
4.4	First Example Basic Model and Design Fragment Set	148
4.5	Second Example Basic Model and Design Fragment Set	149
4.6	Third Example Basic Model and Design Fragment Set	150
4.7	Fourth Example Basic Model and Design Fragment Set	151
4.8	MachineChart Notations for Alternate Queueing Disciplines	152
4.9	Refining a Specification to Match a Basic Model Design	153
4.10	Initial Event/Action Interface for Agenda Design	155
4.11	Solution to Satisfy Assumed Basic Model Design	159
4.12	Defining Contingencies by Equating Visible Behaviours	165
4.13	Design Fragment for an Asynchronous Interaction	168
4.14	Second Design Fragment for an Asynchronous Interaction	171
4.15	Third Design Fragment for an Asynchronous Interaction	173
4.16	Behaviour of a Solution uses a Courier	175
4.17	Inter-actor Client/Server Design Fragment	178
4.18	Readers/Writers Specification	181
4.19	Readers/Writers Design using Synchronous Communications	182
4.20	Actor-level SEQ Design Fragment	186
4.21	Readers/Writers Simulation using DF-Inter-SEQ-1	188
4.22	Readers/Writers Simulation Without Placeholders	189
5.1	Readers/Writer Specification	210
5.2	Readers/Writers Design Solution	211
5.3	Client/Server Design Fragment Bound for Readers/Writers Design	212
5.4	Description Referenced by ?WriterFree? label	214
5.5	Proof of ??WriterFree?? as Collapsible Operation	214
5.6	Proof of BT3ref Contingency of Readers/Writers Design	217

5.7	Producer/Consumer Specification	221
5.8	Producer/Consumer Design after initial EBD to actor allocation . . .	222
5.9	Producer/Consumer Design	222
5.10	Proof of 'producer_buffer' as a Statement	229
5.11	Proving that Sends are Initiated by the Consumer	230
5.12	Initial Decomposition of Specification	230
5.13	Unbound Top-Level Partition Table	231
5.14	Top-Level Partition Table for 'READER?'	231
5.15	Unbound Layered Partition Table	232
5.16	Unbound Layered Partition Table	233
5.17	Example Decomposition Schema for Readers/Writers	234
5.18	Algorithm for proving completeness	236
6.1	Analysis and Modelling Steps	240
6.2	Display for Example Sonar System	241
6.3	Context Diagram for Example Sonar System	242
6.4	Sonar System Design	243
6.5	Entity Relationships for the Display Subsystem	248
6.6	SPA EBD	249
6.7	STA EBD	250
6.8	TA EBD	250
6.9	ERR EBD	251
6.10	CA EBD	251
6.11	CHC EBD	252
6.12	TRK EBD	252
6.13	SB EBD	252
6.14	JC EBD	253
6.15	JSTK EBD	253
6.16	SOFT EBD	254
6.17	ANC EBD	254
6.18	KBD EBD	255
6.19	KEY EBD	256
6.20	EVT EBD	256
6.21	DSPLY EBD	257
6.22	SPA Actor Table	257
6.23	SEH Actor Table	258
6.24	TA Actor Table	259
6.25	HC Actor Table	260
6.26	HC into Actor Table	261
6.27	OIH Actor Table	262
6.28	DISPLAY Actor Table	263

6.29	Primitive Partition Table for ?EVT?	263
6.30	Primitive Partition Table for ?STA?	263
6.31	Tasking Relationships for the Display Subsystem	264
6.32	Design for the Display Subsystem	265
6.33	Refined Specification for TA Entity using the Reset Directive	268
C.1	Emulation Design for the Readers/Writers Problem	315
C.2	Producer/Consumer EBD Specification	325
C.3	Emulation of the Readers/Writers Problem Allocated to One Actor - Part 1	327
C.4	Emulation of the Readers/Writers Allocated to One Actor - Part 2	328
C.5	Refined Readers/Writers Specification	329
C.6	Simulation of the Readers/Writers Allocated to One Actor	332

List of Tables

2.1	Special events, conditions, and actions of statechart transitions. . . .	28
3.1	Syntax Used to Declare the Various Communications Paradigms . . .	56
3.2	Transformations on pool states and events	99
3.3	Counter Range Updates for Given Transition Conditions and Updates	110
3.4	Ripple Matching Requirements	115
3.5	Outcome of Generalizing Two Components	116
4.1	Semantics of Asynchronous Interactions in Design Terms	163
4.2	Semantics of Synchronous Interactions in Design Terms	164
4.3	Semantics of Transient Broadcast Interactions in Design Terms	165
4.4	Generic Build Table for DF-ClientServer-IPC-1	193
4.5	Build Table for Design Decision in Readers/Writers Design	194
5.1	Build Table for Design Decision in Readers/Writers Design	211
5.2	Build Table for Second Design Decision in Readers/Writers Design . .	213
5.3	Build Table for Third Design Decision in Readers/Writers Design . .	216
5.4	Build Table for Fourth Design Decision in Readers/Writers Design . .	218
5.5	Build Table for Fourth Design Decision in Readers/Writers Design . .	219
5.6	Build Table for First Design Decision in Producer/Consumer Design .	223
5.7	Build Table for Second Design Decision in Producer/Consumer Design	224
5.8	Build Table for Third Design Decision in Producer/Consumer Design	225
5.9	Build Table for Fourth Design Decision in Producer/Consumer Design	226
5.10	Build Table for Fourth Design Decision in Producer/Consumer Design	227
5.11	Build Table for Fourth Design Decision in Producer/Consumer Design	228
5.12	Actor Table Example	229
A.1	EI state descriptions	303
A.2	Legend for EI EBD (table 1 of 2)	304
A.3	Legend for EI EBD (table 2 of 2)	305
A.4	Operations and Condition Functions of the EI EBD (table 1 of 2) . .	306
A.5	Operations and Condition Functions of the EI EBD (table 2 of 2) . .	307

C.1	Processing for <i>start(op)</i> action in DF-Prim-SEQ-N	321
C.2	Processing for <i>receive</i> action in DF-Prim-SEQ-N	322
C.3	Processing for <i>send(E)</i> action in DF-Prim-SEQ-N	323

Chapter 1

Introduction

1.1 Reactive Systems and Strong Traceability

The development of reactive system software [21, 58] is a complex exercise, over and above typical data processing problems, for at least the following reasons: (1) reactive systems usually employ concurrent elements, (2) responses to environmental events must often occur in a timely fashion, and (3) they are often the heart of safety or mission-critical applications such as process control, avionics, flight control, robotics, defense, and telecommunications, where high reliability is a significant requirement.

The term, *reactive system*, was coined by Harel [54] to distinguish real-time systems which are *transformational* from those which must interact continuously with their environment, and are therefore called *reactive*. As an example, a payroll system is transformational because once started, it converts personnel information into pay cheques, and then stops. It may have a real-time constraint to complete within a given time period, but it does not interact with the environment during its main processing. An example of a reactive system is an automatic teller machine (ATM) which includes a human-machine interface; this software has a relatively long execution lifetime. It may operate for hours or days before the machine is closed for maintenance, and the execution of the software components depends on the customer initiating, interacting with, and completing a banking session. The

customer is part of the environment in which the ATM must operate, and with which it must interact.

These systems are defined by a combination of functional and behavioural requirements. The behavioural requirements define the timing and precedence relations (i.e., when and in what order) of the functional requirements (i.e., what is to be done). For example, the behavioural requirements define the state changes in a system as a function of the external events. These requirements often complicate the software design process because they frequently need solutions with multitasking and synchronization (e.g., via message passing).

The automatic teller machine example illustrates a reactive system with dynamic behaviour. The system is dependent on the number of concurrent users, the types of transactions which are being requested, and the response times of humans and of a large shared database. Harel introduced the term *reactive system* and the Statecharts specification modelling technique to address these issues and, in particular, to deal with the added problems introduced by *hard* real-time systems; i.e., those with strict timing constraints.¹ Reactive systems with hard constraints are characterized by the need to define predictable behaviour through a static, *a priori* scheduling of the various system functions [109]. The need for such stringent reliability is dependent on the potential for catastrophic results if the temporal behaviour is errant.

For this thesis, the particular concern is with systems whose requirements can be satisfied by the implementation hardware, and a correct and reasonably efficient, message passing design. Several authors have proposed alternative approaches to software development for such reactive systems. Zave, for example, uses the term *embedded systems* to represent this type of system [122], and has developed the executable specification language, PAISLey, to support the development process. Hatley and Pirbhai [59] have defined real-time extensions to data flow analysis in order to support the specification and design of such systems, and Jackson [112] has developed another unique process of specification and design for systems of this type. In all cases, the methods of transformation from specification to design

¹Associating Statecharts with hard real-time systems is an interpretation made by the thesis author based on the applicability and semantics of Statecharts.

are strong in terms of the functional requirements: where explicit transformations are generally supported by potential implementation languages. However, transformations to design of behavioural requirements are typically either: (1) explicit transformations which are often not supported by the implementation domain (i.e., language, hardware, or performance requirements); or, as a result, (2) ad hoc.

Generally, an experienced designer uses a wealth of domain knowledge, background in target environments (languages, operating systems, and hardware), and good analytical skills to produce designs that supposedly satisfy the behavioural requirements; and, at the same time, fit into the implementation domain. The validation of these designs often takes the form of prototyping to show expected behaviour, design walkthroughs, performance estimations, and requirements traceability matrices (which show where a requirement has been implemented in the design). Prototypes, while useful, are limited in what they can reveal because: (1) their breadth of validation can cover no more than that of conventional testing, and (2) their performance may be misleading and misinterpreted. The other methods are similarly weak because they do not require, nor do they introduce, a formal relation between the requirements and design representation. Thus, the traceability known to industry is *weak traceability*, because it is based on informal arguments over informal relations.

Strong traceability exists if there is a systematic or rigorous mapping between requirements and design; i.e., given a formal relation between representations, a systematic mapping has no proof that the relation is valid, a rigorous mapping requires an informal proof, and a formal mapping requires a formal proof [11]. *Complete traceability* exists if there is a correctness preserving transformation from a requirement to a design (formal proof over formal relations; e.g., as in automated transformations). Having strong traceability *strengthens* the validation procedure, while still exploiting the domain knowledge and experience of the designers.²

In summary, this thesis views the notions of specification, design or implementa-

²We assume that, in many cases, allowing such input from the designers generates a more favourable design over one generated through automatic transformation given the current state of the art in that area.

tion, and validation of reactive systems as follows. (1) Specification defines what and when services are to be provided by the system (i.e., the functional and behavioural requirements). (2) Implementation defines how these services will be provided by the software within a given target environment. (3) Validation studies conformance between the specification and the implementation providing a measure of traceability.

1.2 Influential Trends

Specification

Current trends in specification for reactive systems often employ *Entity Modelling* for decomposition [122, 59, 112, 23, 14, 58], and state machines for specifying concurrent system behaviour [61, 55, 59, 23, 14, 83, 66, 42, 105]. Entities are defined to model a reactive system as a set of communicating components. To identify entities in an informal specification, Booch [14] supports the proposals of several authors who (collectively) define the following list of potential candidates: (1) tangible things, (2) people roles, (3) places, (4) events, (5) organizations, (6) concepts, and (7) devices. However, in reactive system problems, abstraction, encapsulation and concurrency must all interact, therefore “real-world” objects need the support of “objects of invention” which serve as mechanisms to provide some higher level behaviour. Booch refers to this process as *entity abstraction*; i.e., entities of the problem domain are placed within a “useful model” forming a basis for *software requirements* (often referred to as *system design*) [88].

The behaviour of each entity can be defined by a state machine which communicates (by channel interconnections) with other entities in the *system under development* (SUD). Communicating state machines are composed of two fundamental elements: (1) the application behaviour (i.e. the conditions and actions of the state transitions), and (2) the communication semantics (which describe how events are transferred between the entities). Together these elements describe the abstract behaviour of the entities and the system as a whole. Generally these features are intertwined by the modelling technique. This leaves the designer with the

difficult task of separating the application-specific details from the generic details of communications as a part of the design process when the communications model of the specification language is not appropriate for, or supported by, the target implementation language.

A considerable amount of research in the specification of behaviour for tightly constrained reactive systems exists or is under further consideration [55, 63, 64, 65, 83, 42]. Most are inappropriate for reactive systems with soft constraints because the models are restricted in terms of how entities are allowed to interact. These restrictions are imposed to reduce the set of possible outcomes of the system execution, and thus make the execution predictable. This simplifies the verification of the specification, and therefore limits the choices of predictable designs to satisfy the requirements (leaning towards automatic transformation). As an example, Statcharts [55] restrict the specification to broadcast communication semantics where all entities are expected to be waiting when an interacting event is signalled. This becomes a drawback when an implementation is to be created from the specification. The system must either be reworked with a different modelling paradigm or emulated by, for example, (1) a broadcast system (i.e., either shared data is required, or a central controller with a lot of message traffic will be necessary), or (2) a sequential program (i.e., with a strict, deterministic, compile-time schedule).

Design and Implementation

A current trend for the preliminary design of reactive systems, employs iconic design languages [20, 23, 14, 58]. Buhr's MachineCharts is one example which is specifically geared to the development of reactive system designs [23, 21, 22]. The icon set is large and flexible, and thus it is often the case that one design can be described by several MachineChart layouts, and the corresponding specification by at least as many. A system is designed with MachineCharts by first studying the structure of the design and the required system behaviour (called place and time by Buhr). The functional requirements (the work) are put into the design once the structure and behaviour have been determined. The use and combinations of these icons is described as a creative process [23].

MachineCharts are unique in their inclusion of the *Abstract Controller Machine*

(ACM) component which encapsulates application-specific behaviour defined by, for example, a state machine. As such, there is the potential for strong traceability between a specification defined using entities and state machine models, and a MachineChart design focusing on the use of ACMs. The specification is maintained by the design through the ACM component. Buhr et al. propose that an ACM, encapsulating the application-specific behaviour of an entity, can be *plugged* into a variety of alternative *concrete* designs which implement the required inter-entity communications [22]. This conceptual approach to design is used in the research to achieve strong traceability of behavioural requirements in reactive systems.

The desire for *traceability* often leads to the ideas of automated translation and transformation. Research in program transformation ranges from the introduction of compilers, through code generators which translate correct programs to efficient programs [37], through specification to implementation transformations [45, 103], and the translation of informal requirements to formal specifications [44]. However, most of the work dealing with the design phase (or implementations), is limited to sequential programs: there is little work in the development of reactive system software design. (Closed reactive systems, i.e., simulations, are considered by Fickas [45]. The translation process results in a uniprocessor solution which corresponds to a general event-driven simulation.)

Design assistants, as conceptualized in [5] and implemented in systems such as the Programmer's Apprentice [117, 95] (which includes the Designer's Apprentice), PDS (Program Development System) [31, 70], and AFFIRM [111], are knowledge-based software engineering projects which assist in the software production by co-operating with the designers. They establish alternative and selective abstract solutions (so called *programming plans*) for problems in sequential programming. The programming plans are constructed using: (1) approaches familiar to programmers, and (2) well-defined programming elements [108]. They assist the designer in choosing a *cliche* [117], *template* [53], or *scheme* [32] and instantiating the parameters for the given problem.

The work done in knowledge-based software engineering has not yet focussed on the general design problems of interprocess communications in reactive systems.

Here, time may permit many, and significantly different, designs which all produce the same externally visible behaviour [12]. Reactive system design plans are immature relative to those of sequential programming and even those of hard real-time systems where predictability constrains the options and creativity of the design process [8]. Therefore, the development of correct designs for reactive systems is still a significant problem (as opposed to the focus being on more efficient versions of a design).

Validation

As discussed earlier, the measure of a validation technique typically falls into one of two categories: (1) *complete traceability* which is achieved when automatic transformation is used (however the result often requires tuning to the particular target), or (2) *weak traceability* when conventional testing by cases is applied.

1.3 Objectives:

A Method for Strong Traceability

The goal of this thesis research is to define a design process for reactive systems which supports strong traceability of the behavioural requirements using relatively familiar and popular techniques. Strong traceability is defined by the illustration of completeness and correctness of the behavioural requirements in a design. With few exceptions the difficulty in meeting strong traceability is the lack of any systematic, let alone rigorous method of migrating a system from its requirements specification to its software design. The objective here is to address this problem by a design framework that captures five significant notions:

Concurrent Programming Plans — Predetermined, reasonable, reusable design alternatives called *design fragments* which represent *programming plans* for reactive systems. Design fragments capture the semantics of the communications defined by the specification; they define a formal relation between independent, non-overlapping specification segments and corresponding design alternatives.

Domain Tailored Design — Deliberate but managed use of creative decision-

making in the design process (such as deciding which components are tasks, and which viable design fragments should be used in relation to the properties of the implementation domain). The design is not removed from the designer, it is merely focused and structured.

Traceability in the Small – Correctness — The steps needed to verify (with respect to the behavioural requirements) the inclusion of design fragments; i.e., to support strong traceability by defining how conformance between the specification and the design can be illustrated. The designer decides on the degree to which conformance must be shown at each stage (e.g., no illustration, informal argument, or formal proof). The validation associated with the application of a design fragment provides information on the design process; i.e., why a particular design decision is used.

Traceability in the Large – Completeness — MachineChart ACMs are employed to directly capture the application behaviour defined by the software specification. The application-behaviour component of the specification is made resident in the design. This defines the coverage required to complete the design; i.e., the event interactions which must be satisfied by the application of design fragments.

Systematic Design and Documentation — Design fragments are used in a systematic design process which is amenable to systematic documentation.

1.4 Overview: Designing for Strong Traceability

The key elements of the design framework are: (1) a software requirements specification model known as *Communicating Entities*, similar to communicating finite state machines used in communication protocol specification [120], but unique in its distinct separation of application-specific behaviour and the generic behaviour of the communication semantics; (2) an extensible set of design fragments expressed in MachineCharts; and (3) a method for translating the requirements specification to a design while establishing strong traceability between the two levels of abstraction. The main steps in the method are: specify, design, and validate as outlined below.

Specification:

Requirements — The requirements specification is developed using communicating entities which support a variety of communications semantics (synchronous, asynchronous, transient broadcast) and event orderings (e.g., FIFO and priority-based). The abstract behaviour of each entity is expressed as a state machine called an *entity behaviour description* (EBD), where the application-specific behaviour is viewed separately from the semantics of the communications. This separation is critical because it constitutes a factoring of the behavioural requirements into application-specific details and generic details of the communications; the application-specific details are transferred “as-is” into the design, and the generic details are solved using appropriate design fragments. Thus, the entities must be chosen so that each can be mapped to a single design object (such as a task in Ada).

Design:

Allocation of Concurrency — The engineer allocates entities to *actors* (the MachineCharts representation of tasks), and each EBD is encapsulated by an *Abstract Controller Machine* (ACM) design component. Any allocation is possible; from one entity per actor, to all entities in one actor, and any combination in between. The only constraint is that an entity is not decomposed across actor boundaries (this is enforced by encapsulation of an EBD by an ACM). The allocation step is creative because the engineer may have to resolve many complex requirements, such as: load balancing in a multiprocessor environment, proper representation of apparent concurrency, reducing concurrency for high performance operation, and characteristics of the target environment. The allocation step is documented in *Actor Tables*.

Selection of Design Fragments — The engineer chooses design fragments that provide an implementation of the communications expressed in the requirements. Fragments are broadly categorized by how they support communication in a given concurrency allocation: i.e., they support either communications between two entities allocated to the same actor (*intra-actor*), or they provide communications between entities allocated to different actors (*inter-actor*). They are used incrementally to add to a basic design format of an actor, and are documented in *Build Tables*. In some cases, these solutions are contingent on design properties (e.g.,

buffer limits), and/or features of the specification which are application-specific. The *contingencies* are defined as components of the design fragment, and provide the means to show correctness of the corresponding design decision with respect to the specified behaviour. Therefore, each fragment would have associated with it, certain design properties (e.g., requires shared memory), and a series of verification steps corresponding to the contingencies. The verification steps are used to show that the integration of the design fragment into the design *does not violate* the communication semantics specified in the requirements.

Validation:

Proving Correctness of the Design — Design fragments define formal relations between independent specification segments and partial design solutions (*programming plans*). They are developed such that most of the conformance proof is a property of matching the design fragment with a segment of the specification. The contingencies represent only those properties which are not automatically enforced by this matching process. Therefore, the proposed design approach leads to a number of much smaller proof requirements (contingencies) because much of the proof effort is already done through the definition of the design fragments.

Once bound to a specification segment, the contingencies are expressed in terms of the application. There are two types of contingencies: (1) specification contingencies which depend on the specification and can be proven formally, and (2) design contingencies which are dependent on a basic design structure for actors. The engineer uses whatever appropriate techniques are available (no argument, informal argument, or formal and perhaps automatic proof) to *demonstrate* that the selected design fragments do not violate the communication semantics of the requirements.³ These correctness *proofs* are generally independent of one another, and are also documented in the *Build Tables*. The satisfaction of functional requirements are not considered in this work.

Proving Completeness of the Design — Completeness is demonstrated by showing that all requirements of the specification are covered by the design. Each design

³The term *demonstrated* is used to represent the *proof* of a contingency be it formal, informal, or as a statement of assumed correctness.

fragment satisfies a segment of the specification, therefore the Build Tables (which document the design at a design fragment level) are used to check off the requirements defined by the specification. This completeness proof can be automated, and is documented in *Partition Tables* which are generated mechanically from Actor Tables using the same hierarchical decomposition found in the original EBD specification.

The final product is a design which *conforms* to the specification. Conformance is defined by completeness, and, if proven, correctness of a design over a specification. Completeness is a function of the specification, and placed in terms of the design through the allocation of EBDs to actors. Correctness is a function of the verification of design fragment *contingencies*. Different levels of conformance are possible given the degree to which proofs of contingencies are taken (e.g., informal argument, or formal proof). At any level of proof, strong traceability is provided by completeness, and the definition of correctness.

1.5 Contributions

There are four main areas of contribution: (1) a technique for specifying reactive system software, (2) a method for designing for strong traceability, (3) a validation process for the design, and (4) a documentation process which supports strong traceability.

Contributions to Reactive System Specification:

Specification Model — The EBD specification technique is defined to model communicating entities. This method is unique in how it views the application-specific behaviour separately from the semantics of the communications. It is useful in its own right, and fundamental to the proposed design approach.

Verification of the Specification — EBDs are amenable to typical verification techniques applicable to state machines (e.g., reachability analysis [118], and temporal logic [34]). For a special case, reachability analysis is enhanced by regular expression syntax to verify systems with unbounded clients using bounded resources.

Contributions to Reactive System Design:

Design Fragments — A method for developing and documenting design fragments is defined. It is based on a design model for actors which includes the encapsulation of the specified application-behaviour using MachineChart AC'Ms. Design fragments represent formalized and independent *programming plans* for the communication requirements of reactive systems. The method provides an approach to formally document design fragments, which is amenable to establishing a framework in a handbook format. An initial population of a design fragments library is given.

Design Method for Strong Traceability — A design method including design fragments is defined. The approach is 2-phase: (1) entities of the specification are allocated to actors of the design, and (2) design fragments are used to design the specified communications. The method supports strong traceability through: (1) rules for correct allocation, (2) a basic design structure for actors, (3) a corresponding set of design fragments, (4) methods for demonstrating contingencies defined by the design fragments, (5) documentation for illustrating the design and its validation (correctness and completeness), and (6) a systematic method for demonstrating completeness of the design.

Contributions to Design Validation – Strong Traceability:

Traceability in the Small – Correctness — Much of the correctness proof is implicit in the underlying design structure for actors, the design fragments, and the proposed design method. The contingencies represent only those issues which are not enforced through the binding of a specification segment to a design fragment. Algorithms are defined for handling the set of contingencies along with methods for their individual, formal and informal proof.

Traceability in the Large – Completeness — A systematic algorithm for showing completeness of a design is defined in terms of the specified requirements (documented in Actor and Partition Tables) and the design decisions (documented in Build Tables).

Contributions to Design Documentation:

Actor, Partition, and Build Tables — Tables are used to document each design decision, and the proofs of correctness. They provide support for strong traceability by recording: (1) allocation decisions (which are documented in Actor Tables), (2) design fragment decisions (which are documented in Build Tables), (3) demonstrations of correctness (which are also documented in Build Tables), and (4) a framework for validating completeness (via Partition Tables).

1.6 Thesis Outline

Chapter 2 provides detailed background information on the “influential trends” introduced in Section 1.2.

Chapters 3 through 5 cover each step in the design process as follows. Chapter 3 discusses the specification of communicating entities. Section 3.1 relates the specification technique to designing with design fragments, and Section 3.2 defines the EBD modelling technique. Section 3.3 provides a brief discussion on developing EBD specifications, and Section 3.4 covers the verification technique to manage unbounded client pools with bounded resources.

Chapter 4 covers the design steps. Allocation of EBDs to actors is discussed in Section 4.1, and designing with design fragments is found in Section 4.2.

Chapter 5 cover the validation process. Proving correctness through proof of contingencies is discussed in Section 5.1, and proving completeness is covered in Section 5.2.

Chapter 6 illustrates the design process using a real-life, albeit simplified example, and the conclusions are drawn in Chapter 7.

Chapter 2

Background

This chapter provides background information for the thesis work. As defined in the introduction, there are three main areas of contribution: (1) a specification model and method for defining reactive system software requirements, (2) a systematic design process to transform these specifications to designs using design fragments, and (3) a validation process which yields strong traceability between the specification and the design. The development of the specification model is heavily influenced by the design goal for strong traceability which is to be characterized by the *abstract controller machine* (ACM) component of Buhr's MachineCharts. The ACM is a novel notational element which directly encapsulates a state machine specification of temporal behaviour requirements. Because of this, MachineCharts have been chosen as the language with which to express the *design fragments* in this thesis. As such, a discussion on MachineCharts is provided in the first section of this chapter.

Existing Specification Models and Methods — The next section provides a discussion on some of the existing specification models and methods. Jaffe's paper [62] is recommended reading for those interested in gaining an appreciation of the complexity of specifying requirements for reactive systems. *Entity modelling* is a popular technique for structural decomposition of reactive system specifications and designs. Buhr suggests that *statecharts* might be appropriate as a specification tool for direct translation of the behaviour requirements to the ACM design concept. This section looks at *STATEMATE*, the specification technique including

statecharts. This is followed by a brief discussion on the JSD (Jackson's System Development) software development methodology [112]. In JSD there is no clear line between specification and design, but rather the design phase flows naturally from the specification which defines functional and data flow requirements (temporal requirements are not specified in a formal, verifiable model but rather through English descriptions called *structured text*). Also in this section are descriptions of other notions used in specification of communications protocol software which, as Buhr indicates, is one type of reactive system for which MachineCharts are intended, and which offer some interesting properties for specification techniques. This section concludes with a summary of the desired properties of a specification model for reactive system software development.

Design by Program Transformations (Automatic Programming) — This chapter continues with a section on the current state of research into knowledge based software engineering which is often referred to as automatic programming. After a brief introduction, the main contributions to this field are discussed. These are tools for software development: i.e., (1) SAFE/TI (Specification Acquisition From Experts/Transformational Implementation) which is being developed at the Information Science Institute, (2) Harvard's PSD (Program Development System), (3) MIT's Programmer's Apprentice, and (4) the PSI system of Stanford [89].

Design Validation — In Chapter 1 validation was defined as the study of conformance between the specification and design, and was attributed with a measure of traceability: *complete traceability* for formal (automated) proof or transformation techniques, *weak traceability* for testing techniques applied to informal specifications, and *strong traceability* for any level of demonstration given potentially formal relations and the means to prove conformance (i.e., contingencies). This chapter lists the current techniques in formal validation and discusses their limitations.

2.1 MachineCharts

MachineCharts is an iconic language used to describe software designs with pictures [23]. The basic components of the notation are boxes, actors, buttons, gates, fin-

gers, stores and engines. Boxes and actors support software modularization and encapsulation. The perimeter of the box defines its interface and buttons touching this surface represent access points to the box contents analogously to a software module and its interface functions. An actor is drawn as a parallelogram and defines a concurrent component of the SUD. Its perimeter protects the contents by affecting mutual exclusion on the buttons which are now called gates; i.e., only one gate can be executing at any time. An actor with gates is analogous to an Ada task and its entry procedures [20]. Box buttons represent re-entrant procedures and actor gates represent synchronization points of inter-process communications (IPC). Fingers are the iconic notation to illustrate invocation paths of the software design. A finger *pushing* a button describes an operation being invoked. A finger pushing a gate button must be complemented by another finger (of the actor which owns the button¹) pulling on the same button. This push-pull action is intended to illustrate that synchronization is required to activate the code of a gate button. A store icon simply defines a data store, while an engine icon defines the mainline processing of a box, actor, or button icon and is therefore always drawn hierarchically within the controlled component. (Button engines are implicit; they do not appear on the diagram.)

For example, Figure 2.1 shows two actors communicating with send-wait-reply, and receive-wait semantics (an Ada rendezvous). These icons can be enhanced by other notational elements to indicate such principles as (1) Wait-OR which is illustrated by drawing a continuous line through all of the gate buttons involved in the *choice* selection, (2) masking any combination of an Wait-OR structure is shown by placing a *waiting place* (a dot icon) at a gate which is set according to some boolean expression, and (3) conditional or timeout communications which are illustrated by adding a clock symbol to the finger icon (or curling the finger at the destination end to illustrate balking). These concepts are illustrated in Figure 2.2.

The component of most interest to this chapter is the *abstract controller machine* since it supports the novel concept of tracing behaviour requirements from specification to design in a potentially systematic and rigorous fashion. An engine defines

¹Ownership is not necessary in the full set of MachineChart notations.

Figure 2.1: MachineChart Design for an Ada Rendezvous

Choice Bars
provide WAIT-OR
of gates

Waiting place:
e.g. gate "get"
is closed when
the internal store
"buffer" is empty.

Conditional and timed fingers

Figure 2.2: MachineChart Design Notations

the mainline agenda of a software component, therefore it accomplishes the same thing as would a flow chart description of the component. Some of the engine description is found in the buttons of boxes and actors. That is, buttons have implicit engines which execute according to the semantics of the drawing. For example, if an actor engine pulls a choice bar (see Figure 2.2), the engine of the now active gate defines the current flow of control. This approach minimizes redundancy by removing the syntax for OR synchronization from within the definition of the actor engine, (in which the gate engine information would otherwise reside), since it is already defined by the diagram.

However, reactive systems typically consist of interacting, communicating components. The interactions will sometimes have specific required precedence relations and timing requirements which will change according to the current state of the environment; i.e., the flow of control can only be determined by a sequence of interactions instead of just an initial start event needed to invoke a flowchart description. The environment of a reactive system is assumed to be unpredictable, and therefore a path of the execution cannot always be determined until some *event*, or sequence (history) of events, have affected the system. State machines can be used to describe the alternative behaviour patterns of interacting components working within such an environment. The ACM icon (illustrated by a small box encapsulating a circle) is intended to encapsulate an *abstract state machine* (ASM) which is used by an engine to determine the correct execution path from the current system state. The mainline *agenda*, which could not decide the next step on its own, now passes the incoming *event* to the ACM expecting to be returned an *action* from which the appropriate execution path can be determined.

For example, Figure 2.3 illustrates three interacting components of the SUD and one *passive* component (which does not need mutual exclusion protection even within a design that assumes multiple concurrent threads of execution as realized by the three actor icons). Each button of the passive object defines a re-entrant procedure which can be described by a sequential program agenda (its implicit engine). The actor engines are shown to be pulling on gate buttons to accept the system events, which make further calls based on state information supplied by an ACM.

Figure 2.3: MachineChart Design with ACMs

An ASM is a state machine with Mealy semantics [100]; i.e., state changes are initiated by *incoming events*, which may or may not process changes to auxiliary state variables during the transition, and finally resulting in an *outgoing action*. The ACM encapsulates the ASM with a standard interface procedure consisting of an input parameter to accept the incoming event, and an output parameter to manage the output action.² The engine uses the action to make decisions on the flow of its own agenda and possibly to issue events to other components of the SUD. In theory, the state machines of the specification are to be transformed into ASMs which work with an engine and other so-called *concrete* design components to satisfy the specific semantics of the specification language. This way ACMs are “pluggable”; i.e., the ACMs can be placed within different concrete designs which correctly model the specification semantics with different levels of constraint. A specification includes a description of the state changes under an assumed communications semantics. This is mapped to an engine-ACM pair and any other necessary “concrete” design features needed to correctly represent the temporal requirements. For example:

²Recent developments support a flexible, user defined procedural interface with ACMs. As such data transfers are allowed. In this report the simple event/action interface is assumed.

1. Statecharts use Mealy state machines with an underlying broadcast communication semantics, therefore the state machines would map directly to ASMs and the engine-ACM combination, along with other "concrete" design features, assures that the semantics of the Statecharts specification are not violated.
2. JSD defines event flows by drawings similar to Data Flow Diagrams which are called *System Specification Diagrams* (SSDs). These are supported by *structured text* which defines the semantics of the communications illustrated in the SSDs. In this case, the event flows would have to be modelled by communicating state machines with some formal communication semantics before they could be mapped to a MachineChart design using ACMs.
3. Booch's theory (which is very similar to Buhr's) [14] is to define transitions by Mealy machines while the semantics of the machines are described by *timing diagrams*. The result supports a direct mapping to ASMs, and the information of the timing diagrams is designed into the engine-ACM and "concrete" features.

In summary, ACMs offer support for the design of strongly traceable behaviour specifications. This is particularly apparent when state transition descriptions of behaviour are specified via communicating state machines. Each state machine would then map to an engine or engine-ACM pair. An engine alone is required if the state machine semantics can be directly related to a flow chart. This occurs when a sequence of steps is initiated by an event, and can complete without any other incoming events or without being influenced by the history of state information. An engine-ACM pair is required when the initial state cannot be returned to, without interaction from some other system or environmental components, or when state history affects the next state.

The structural decomposition performed at the specification level must consider the direct mapping of behaviour requirements during a specification to design transformation process. An appropriate technique, known as *Entity Modelling* is discussed in the next section. It is an approach to structural decomposition which is used in JSD and supported by the Object-oriented community [10]. It is con-

sidered to be a good technique for decomposing a reactive system into interacting components so as to minimize the effects of requirement changes on the system structure [14].

2.2 Specification Models and Methods

2.2.1 Entity Modelling

A model is an abstract representation of something in the real world. Entity modelling provides a structured view of what is happening in the SUD, and is the decomposition technique used by:

1. STATEMATE which decomposes the system into *modules* which perform *activities* according to a *behaviour* described by one, in a collection of, interacting (communicating) statecharts [58].
2. Buhr who decomposes by *places* which do *work* in (relative or relational) *time* [23].
3. The object-oriented development community who decompose by *objects* supporting different types of *relationships* [106, 10, 110].
4. Those studying knowledge representations of software requirements who support describing entities in the domain of their relationships (discourse and events) [15].

The technique for entity modelling begins with finding the system *entities* from the original, customer supplied specification, and by making *early design decisions*. Booch refers to this process as *entity abstraction*; i.e., entities of the problem domain are given a “useful model” which might introduce solution-specific entities. He refers to this as *process abstraction*, however it does involve engineering. In concurrent problems, abstraction, encapsulation, and concurrency must all interact, therefore real-world objects need support of “objects of invention” which serve as mechanisms to provide some higher level behaviour [14]. Thus, reactive system specification

requires the addition of entities to support high level (externally-visible) behaviours so as to provide a means of defining logical and sensible (useful) models including the recognizable real-world entities.

Consider a solution space where all system knowledge is distributed versus one where the system knowledge is centralized. The behaviour models of interacting components in the first case will include communications between all entities; i.e., each must inform and question all others before initiating critical processing. In the second case, a scheduler entity would grant or deny permission to execute critical processing. These behaviour models are significantly different, but reflect the same externally-visible behaviour.

A system design is not fully constrained by these specification structures. There are several options available for the encapsulation/modularization of the entities in the design model. Also, the communications defined by the specification are stated as requirements which can be constrained by a design as long as the fundamental semantics are not violated. For example, a specification for an asynchronous message transmission requires no more than that the sender eventually executes its next step. The design might represent this as a procedure call, an asynchronous communications, or even a synchronous communication. All are correct if it can be proven that the sender is never permanently blocked by the interaction.

Finding entities will be difficult for the programmer accustomed to looking for functionality in a system, however the approach is well documented in the JSD literature [112, 91], in Buhr's design method [23], as well as in the OORA/OOD community [106, 10], and studies indicate that it is achievable [99].

What are entities? Sutcliffe describes entities as:

... real-world objects with which the system will interact. Entities are composed of actions which either effect or respond to changes in the real-world. [112]

Booch describes objects as constituents of the model of the *problem space*.

... an object is an entity that has state (some value) and can be characterized by the operations that it undergoes and initiates. [13]

How are entities identified? Sutcliffe (and older material of Booch) propose searching out nouns and noun phrases in the specification. Although this seems tedious, Rosson's studies show that finding entities can become quite natural, and eventually can be performed simply from reading an informal specification [99]. In their own words, Booch suggests the analysts:

... recognize the major actors, agents, and servers in the problem space plus their role in our model of reality. Usually, the objects we identify in this step derive from the nouns we use in describing the problem space. [13]

In more recent material, Booch supports the proposals of Shlaer and Mellor [105], Ross [98], and Coad and Yourdon [35] which together define the following list of possible entities: (1) tangible things, (2) people roles, (3) places, (4) events, (5) organizations, (6) concepts, and (7) devices. In the case of concurrent systems, this list is updated to include engineered entities inherent in the specification of reactive systems [14].

Sutcliffe recommends the analysts:

... list things which happen in the system as simple statements, phrases and sentences. The approach is to list the process, identify phrases with nouns in common, and then examine them for the objects they identify as potential entities.

It is Berard's opinion that entity recognition requires consideration of a component and what it can do: as opposed to a component that does one particular thing; i.e., it requires:

... a mindset closer to a hardware mindset than a conventional software functional mindset where functionality is still important, but it is now localized within objects and within descriptions of interactions among objects.

Another issue which makes entity analysis difficult is that it requires a complete specification because it must generate a complete model. Thus, there is a problem in

finding and understanding requirements which are spread over several sources (e.g. requirements documents, standards documents, experienced people, and existing software or prototypes).

... It is hard to make the decisions that must be made to write such a document (a complete and precise requirements document). We often do not know how to make those decisions until we can play with the system. Only when we have built a similar system before is it easy to determine the requirements in advance. It is worth doing, but it is not easy. [86]

The quote taken from a paper by D.L. Parnas is very appropriate for the proposed analysis technique. Real-world modelling is hard to do without a preconceived understanding, or an attainable, pre-implementation understanding of the real-world. This technique will not necessarily be easy to follow. It will require complete structural and behaviour modelling which means that there is a need for early design considerations. It may be necessary to constrain the original requirements to achieve such a model, and it may be necessary to animate (prototype) the model for experts and users to make comments.

As an example, this section introduces the classical Readers/Writers problem assuming the following customer supplied requirements specification (as it is stated in [20]):

... A data structure in primary or secondary memory is accessed by a number of actors (readers and writers), some of which only need to read its contents and others to update it. In general, reading and writing can occur at any time, although only one writer at a time should be allowed to write. While writing, all other writing and reading actors must be denied access. However, when no one is writing, any number of reading actors may be allowed access.

From this specification, *phrase listing* is performed to find the system entities: i.e., (1) a data structure (resource) is accessed, (2) the resource is accessed by readers and writers. (3) readers and writers can be scheduled at any time as long as there is only ever one writer at a time. (4) the scheduling denies access to readers when

a writer is active, and (5) the scheduling allows multiple readers. The entities are chosen by examining nouns and noun phrases: they are: (1) the readers, (2) the writers, and (3) the resource.

2.2.2 STATEMATE and Statecharts

STATEMATE is a pictorially-based tool for the specification of reactive system requirements. The tool does not support any particular methodology, but suggests an approach similar to Hatley's method of real-time software specification [58]. In STATEMATE, a module-chart notation is provided for the preliminary SUD decomposition. Entity modelling would be a very appropriate technique to be applied when defining the system *modules* just as it would be useful as a decomposition technique to start the Hatley method and notation. If a module is equivalent to an entity (as defined earlier), then the module has a set of functions which it can perform. To specify these functions, the tool provides another notational language called activity-charts. Activity-charts are analogous to the Data Flow Diagrams (DFDs) used in Hatley's method [59]. Finally, the temporal requirements are specified by a third notation called statecharts. A statechart schedules the *activities* of its module with respect to other activities and inter-entity communication of *events*. In Hatley's approach Control Flow Diagrams (CFDs) illustrate the interactions between entities, and Control Specifications (CSPEC's) schedule the activities (the DFD processes).

Statecharts are finite state machines, and could be fit into an ACM if the appropriate engine and *concrete* surroundings can be determined to correctly represent the specified behaviour. However, the semantics of a statecharts are quite restrictive when considering the specification of reactive systems. Statecharts support a broadcast communications paradigm. When an event arrives at one of these state machines, it is assumed that all activities are prepared to respond immediately to the event. If the machine is not able to accept the event in its current state, then the event is discarded; i.e., events are transient. These semantics suit hard real-time system specifications where a static schedule will be used in the implementation to guarantee a *predictable* execution. In these systems, it is imperative that all ac-

tivities complete within their deadlines, and that events are not lost. Therefore a specification written in statecharts will act as a tool to prepare this static schedule. If a schedule can be defined to meet all the timing constraints of functional requirements, then the statechart specification can be used to verify that there are no temporal errors in the specification. Verification of state machine-based specifications is discussed later in Sections 2.4 and 3.4.

2.2.2.1 Statechart Properties

Statecharts were developed (prior to STATEMATE) as a tool for specifying event triggered state descriptions, and the interactions between, entities in a reactive system. Their goal was to develop a “modular, hierarchical, and well-structured” approach which solves the problems of state explosion by separating the combined system description into a set of communicating state machines [55]. The authors defined three properties required for their solution:

1. The approach must introduce the concepts of independence and concurrency to support the notion of a set of communicating state machines in place of a single complex system state description.
2. The approach should support generalized state transitions.
3. The approach must offer the ability to cluster, and alternatively refine, state descriptions.

Each of these properties is outlined below.

Independence and Concurrency — The concept of separating a system state description into a set of communicating state descriptions has been used by the protocol specification community for several years. This approach is formal if the state machine model is supported by a strict semantic definition for the communications; i.e., rules to define the temporal properties of state machine communications. In the protocol specification community, one of the most commonly used *process models*³

³The process model is the communications model: i.e., communicating state machines and a supporting event transfer semantics.

defines processes which interact asynchronously through simplex FIFO channels. An example which uses this model is illustrated in Figure 3.20. The semantics of the model are quite simple; output events are queued in the appropriate FIFO channel, and input events are taken from the front of the appropriate channel. It is assumed that only one action (an event output, or an event receive) occurs at each execution step. This definition allows for a combined state description to be produced, and in doing so temporal errors are detected [118].

With statecharts, the process model has state machines interacting with broadcast communications. When an event is output, all state machines which can transition from their current state (because of this event) do so immediately and simultaneously. All transitions take zero time, and therefore the output events of these transitions affect the system immediately. These semantics can cause a machine to oscillate between states; i.e., statecharts can become unstable under this definition, and therefore specific semantic rules have been included to examine state changes as a sequence of *micro steps* instead of as immediate [56]. As mentioned earlier, these complex semantics suit systems which can guarantee that all entities are ready and waiting for all incoming events at all times; i.e., systems with strict, predictable, and therefore static schedules. Because of the semantic definition, statecharts can also be analysed for temporal errors as was done with the protocols specification model.

Generalized Transitions — Transition labels in statecharts are very flexible, and include the following components: (1) an input event, (2) a condition for transition, and (3) an output *action*. Every transition must have at least one input event or condition; i.e., there are no spontaneous transitions. An input event is a broadcast event issued by another entity as part of its last action (i.e., the result of some other transition). The condition for transition is a boolean expression which tests either auxiliary state variables of the statechart or one of a list of acceptable system state expressions. These expressions, listed in Table 2.1, allow a statechart to examine the state of other, concurrent statecharts or of associated activities. The *action* is any combination (including a null set) of an output event, a state variable update, or a command which affects an activity; e.g., to initiate (**start(x)**), or terminate

	EVENTS	CONDITIONS	ACTIONS
in statechart	<i>entered(state)</i> <i>exited(state)</i>	<i>in(state)</i>	
connecting statechart to activities	<i>started(activity)</i> <i>stopped(activity)</i>	<i>active(activity)</i> <i>hanging(activity)</i>	<i>start(activity)</i> <i>stop(activity)</i> <i>suspend(activity)</i> <i>resume(activity)</i>
data and conditions	<i>read(data)</i> <i>written(data)</i> <i>true(condition)</i> <i>false(condition)</i>	<i>data=expression</i> <i>data<expression</i> <i>data>expression</i> ...	<i>data:=expression</i> <i>make_true(condition)</i> <i>make_false(condition)</i>

Table 2.1: Special events, conditions, and actions of statechart transitions.

(**stop(x)**) an activity **x**.

Clustering and Refinement — To illustrate hierarchy, a statechart *state* can encapsulate any other valid statechart; therefore transitions can originate and terminate at any level of the statechart, each with its own implication. For example, a transition on event *c* in Figure 2.4, implies that all states of state **A** (which includes **B** and **C**) transition to state **D** when event *c* has been offered to the system. Transitions which cross encapsulation boundaries show explicitly the source or destination of the transition. If a transition stops on a boundary, then the default entry state becomes the destination state. The default state is shown by a small arrow as illustrated in Figure 2.5, or by a history icon which is shown in two modes in Figure 2.6. In the first case, the destination state is the last active state of the particular statechart, while the H^* icon indicates that the last active state of the lowest level of hierarchy is the default state. Therefore in the first case states **B** or **C** are the potential default states while in the second case one of states **C**, **D**, or **E** is the default state.

There are two weaknesses in this approach to the definition of soft real-time reactive systems: (1) the process model is restrictive, and, as a consequence, (2) when used as part of a complete specification (structural, functional, and tempo-

Figure 2.4: Illustration of default exits in statecharts

Figure 2.5: Illustration of Default Entries in statecharts

Figure 2.6: Examples of statechart history notation

ral). statecharts become schedulers for activities; and therefore, the two distinctly separate models (statecharts and activity-charts) become intertwined. For example, the queuing requirements of an asynchronous event transfer are specified by a queue module with *enqueue* and *dequeue* activities. These are then activated by statecharts which start the *enqueue* activity as the action of one transition, and then initiate a transition when the *enqueue* activity is no longer *active* (This is determined by the condition **active(enqueue).**)

To translate statechart specifications to MachineChart designs, both statecharts and activity-charts must be understood in combination. The state machine features must be abstracted out for the ACM, and the control flow features abstracted out for the definition of the engine. These issues are studied in [39].

2.2.3 Jackson System Development

JSD uses entity analysis as part of its methodology for developing software. The process of specification, design, and implementation is covered by the single JSD methodology. However, this approach is typical in that it does not trace behavioural requirements through the development cycle in any formal way. At the beginning of the development cycle, temporal requirements are specified in English text, and never formalized into a logical or mathematically based model. The structural, temporal, and functional requirements are spread over three different notations: (1) SSDs (system specification diagrams) which are like DFDs, (2) structured text which gives semantic meaning to the SSDs, and (3) PSDs (process structure diagrams) which are like flowchart (structure chart) descriptions of the sequential requirements.

One interesting feature of the methodology is *networking*. The unavoidable, externally coupled behavioural requirements (I/O) are left out of the system design until after the basic system interactions have been defined. In the *networking* stage the I/O dependent requirements are added on top of the existing system design; i.e., after the SSDs, structure text, and PSDs have been defined. This is similar to the layered development process used by the communications protocol community (ISO) which is discussed in the next section.

2.2.4 Protocol Specifications

All of the ISO protocol specification techniques provide a formal method for specifying behaviours. Some are state machine based (e.g. SDL [102] and ESTELLE [19]), and others are mathematically based (e.g. LOTOS [12] which is based on CCS [80]). Some modelling techniques offer strictly synchronous communications as found in CSP [61], while others offer only asynchronous communications. All, like statecharts, can be formally analysed for temporal errors using reachability analysis (which is discussed in Section 3.4), temporal logic, or through interpretation of unambiguous, operational semantics. Two features of specification which have evolved from this field of research and which are also of interest to this thesis are: (1) the layered approach to specification, and (2) the unconstrained, multiple interleaving of behaviour patterns which can be specified in LOTOS.

2.2.4.1 Layered Protocols

As discussed in the previous section, a goal of the behaviour specification model is to support hierarchies. This has been researched extensively in the development of communications systems. It is called the *layered* approach to system development, and is actually based on the decomposition of functionality. However, because the functionality is in terms of communicating entities, the result is that protocols (two-way behaviour specifications) are defined hierarchically. “N-layer” protocols define the interactions needed between peer entities at layer N to accomplish the functional requirement allocated to that layer. The behaviours of any given layer are initiated by outputs of the layers above and below, while these are, in turn, initiated by inputs coming from their own surrounding layers.

Each layer performs a function called a *service*; e.g., the lowest layer is the Physical Layer, and its purpose is to transmit data across the physical media. This layer offers services to the Data Link Layer through interface functions called *service access points*. The Data Link Layer and the Network Layer (its immediate user) perform most of the functionality needed to connect, and correctly transfer messages from one network node to another. Higher layers include, at the top level, the application layer which offers services to the application programs which must

communicate across a network.

2.2.4.2 LOTOS

LOTOS is an ISO standard specification language. It is not based on state machines, but rather on CCS (Calculus of Communicating Systems) and CSP (Communicating Sequential Processes). Systems are broken down into entities which can communicate synchronously, and the language can represent all of the structural, functional, and behavioural requirements simultaneously. This is considered to be an advantage; i.e., all of the requirements are in a single descriptive notation. Another feature of this language which is of interest is that it imposes few constraints on the behaviour pattern other than what is specified. The semantics of the process model do not impose any assumptions on the order of event arrival over the order in which they are sent, therefore, randomly generated events are also randomly received.

2.2.5 Summary

All of the specifications models discussed in this section are suitable for use given a structural decomposition defined through entity analysis. Each also supports a layered and thus incremental development of a specification. However there are drawbacks, in terms of the objectives of this thesis, in each method.

STATEMATE and Statecharts — As communicating finite state machine descriptions, statecharts could be mapped directly to ACMs in a design, and they are verifiable. However, the underlying broadcast communications paradigm of the statechart process model has been described as a problem for reactive system specifications with soft timing constraints. The broadcast semantics are overly restrictive and force the analyst to add modules and activities to specify fundamental properties of the interactions like synchronous and asynchronous communications, or event queueing. Therefore a specification with soft constraints requires both the understanding and coordination of two models: (1) statecharts, and (2) activity-charts. This method is optimized for the strict timing constraints of hard real-time systems.

Jackson System Development — The behavioural requirements of a system are

specified in JSD by defining the operational semantics of the data flow-like system structure diagrams. These semantics are specified informally in *structured text*. Therefore, this method also requires two different notations to completely specify the behavioural requirements. The result is informal; i.e., it is not open for formal verification. Although translating a JSD specification to a MachineChart design could be defined systematically, it would not be rigorous; i.e., it would be very difficult to prove strong traceability between the two development phases.

ESTELLE — ESTELLE is a Pascal-like formal description language which specifies the state changes of communicating finite state machines as well as the functional operations associated with the entity. Therefore all requirements of an entity are defined by a single notation which could have a strongly traceable relations to ACMs. However, it is constrained in terms of the communication paradigms which can be expressed. It assumes that all interactions are via a non-blocking send and a blocking receive with unbounded FIFO queuing. Each entity is limited to a single input queue, and therefore event ordering is constrained to the order in which they are received. The application behaviour is not separated from the communications, in that, the behaviour must deal with *unspecified receptions* (unexpected incoming events) instead of dealing only with those events currently of interest.

SDL — SDL looks very much like flow-charts with both behavioural and functional requirements being specified by the single notation. Again this is a formal description language which would allow for direct mappings to ACMs. The channels can be described as entities, and therefore, although it is slightly cumbersome, this language can support the description of alternative communication paradigms, and individual event orderings. However, it assumes, as in ESTELLE, that there is a single input queue for each entity, and thus the ordering of all events at the destination entity is constrained to the order of their arrival, and the application behaviour, once again, depends on the communication semantics.

LOTOS — LOTOS is another formal description language, however it is not based on finite state machine descriptions. The specifications typically follow a *resource oriented* style where the behaviour of each entity is defined within a LOTOS process.

and the system behaviour is expressed by combining the individual behaviours using algebraic operators [115]. Behaviours are constrained to synchronous interactions, however, asynchronous message passing can be specified by including a queue entity in the model: each process synchronizes with the queue entity to send or receive an event. (This is similar to adding a queue activity in STATEMATE.) In LOTOS, events are provided by the *environment* to all waiting processes. These events are then consumed by the waiting processes, and are no longer available to other processes. Once again, the behaviour descriptions of each process are dependent on this underlying communications model.

This review of existing models and methods prompts the following “wish list” of properties appropriate for a process model for communicating state machines. These features support the thesis goal for strongly traceable behaviour requirements between the specification and design phases of reactive system software development.

Entity Analysis — The process model should allow for the description of communicating *entities*.

Multiple Communication Paradigms — It should offer capabilities for describing synchronous, asynchronous, and broadcast communications without the introduction of support entities.

Multiple Individual Event Orderings — It should be possible to order individual event types; e.g., by an assigned priority attribute, or by arrival time.

Communicating Finite State Machines — These interactions are well defined by communicating state machines.

Verifiable Behaviour Specifications — The process model should have specific operational semantics leading to unambiguous behaviour models which can be verified using standard techniques like reachability analysis (cref. § 2.4 and § 3.4).

Layered Behaviour Descriptions — The state machines should allow for layered behaviour descriptions.

Unconstrained Event Orderings — The process model should not impose restrictions on the arrival order (or processing order) of independent event types. The entities should not be limited to *input-enabled* semantics;⁴ but rather, events should be held for the entity and accepted or screened according to the behaviour description.

Single Notation — The model should allow for high level functional requirements to be defined within the same structural decomposition and notation as the behavioural requirements.

Mapping to ACMs — The specifications written using this model should map to MachineChart designs using ACMs; i.e., they should support a *plug-in* capability for the application-behaviour. Given an explicit separation of application-behaviour and communication semantics, the application-behaviour can be transferred “as-is” to an ACM, while the communication semantics are designed to suit the implementation domain.

2.3 Knowledge Based Software Engineering

Knowledge based software engineering is historically better known as automatic programming. The beginnings of the research field is compiler technology where abstract high-level programming languages are automatically transformed into executable programs. The ultimate goal is the automatic transformation of informal specifications to correct and efficient programs. As this is not yet an achievable goal, the focus is typically one layer in the following hierarchical list of transformation problems: (1) compilation [97], (2) transformation of existing programs to efficient programs [37, 43], (3) transformation of specifications to implementations [6, 45, 107, 89], and (4) the transformation of informal requirements to formal specifications [44, 6, 96, 95]. The work at level 1 is extensive and relatively mature as is the work at level 2 in particular for algorithmic sequential programs. At level 3,

⁴Entities are input-enabled if they react to all incoming events immediately as in statecharts and I/O automata [75].

much of the work has involved the development of high-level, *wide-spectrum* specification languages which support a broad domain of abstract concepts without forcing the analyst to specify an implementation. At level 4, where there is currently much interest, the research is attempting to find ways to reduce the amount of details that the analyst must consider in parallel.

The advantages of transformational programming tools are that they theoretically support the development of reliable software through correctness preserving transformations. They support documentation efforts, and allow for the storage of the process of development which can then be used to re-run a development process on a modified, and thus new initial representation [51]. The drawbacks are that the tools are often domain specific and thus are applicable only within a small range of problems. The output programs are efficient, but very often illegible, and modifications made manually to the output can be lost when a transformation process, which was logged during the initial software development, is re-applied to a modified specification [8]. As a result many experts focus on development assistance instead of automatic generation [51, 6, 117]. The assistants consult the user for domain specific information and thus support broader domains. They are amenable to the immediate introduction of incremental improvements in all areas of automation.

Although deductive synthesis and program verification are two important areas of research in this field, this section focuses on transformational approaches to software engineering [96]. Transformational techniques are usually correctness preserving; they transform an abstract level input to a correct, concrete, and generally more efficient view. Transformations are applied either procedurally or through knowledge in a library of transformations. Two types of transformations are: (1) *vertical* which replace an abstraction at one level with one at a lower level, and (2) *lateral* where equivalence relations are applied to alter a structure at the same level of abstraction. Input languages which allow both transformation types are known as *wide spectrum languages*.

The first attempts at automatic programming show lateral transformations applied to existing and correct programs to improve the efficiency of the program. When problems arose in controlling the application of transformations, the trend

moved from procedural transformations to libraries where the user would make the difficult decisions with assistance in applying the transformation. As examples, this section discusses two general transformation systems: (1) SAFE/TI, and (2) PSI, and two general programming environments: (1) PSD, and (2) PA.

2.3.1 General Transformation Systems

SAFE/TI — SAFE (Specification Acquisition From Experts) and TI (Transformational Implementation) are two projects from the ISI (Information Science Institute). The SAFE project is concerned with the synthesis of formal specifications from informal ones, while in the TI project, formal specifications are transformed into efficient programs [89].

The motivation for SAFE is the difficulty in writing and understanding the large amount of detail required for a complete formal specification [5]. A mechanism has been developed to resolve ambiguities between partial specifications, and to refine and modify existing specification structures [4]. The input to SAFE is a parenthesized natural language description of the problem and its solution [6]. A mechanical analysis approach (like the mechanical methods described in Section 2.2.1) is used to convert the description to *event descriptors* during the *linguistic phase*. *Objects* are found from nouns and *sets* from plurals. *Events* are extracted from verbs and *conditionals*, *conjunctions* and *loops* from verbs associated with *sets*. The next stage of SAFE is the *planning stage* where the overall behaviour is determined and recorded as a sequence of event descriptors. Next, these are refined into *actions* (where actions take on the same definition as JSD actions [112]), and the fine details are resolved by symbolic evaluation; the result is an operational specification in AP3 [95]. SAFE consults the user when ambiguities cannot be resolved automatically.

TI is an independent project to transform specifications written in GIST to implementations through a continuous, sequential process of applying transformations which: (1) replace specification constructs with algorithmic ones (vertical transformations), or (2) simplify constructs (lateral transformations) [74]. The transformations are selected, by the user, from an extendable catalogue and applied automatically by the system. The catalogue entries describe correct implementations for

the specification constructs and possible optimizations, while the correctness of an extension (a user added transformation) is the responsibility of the user [45]. The process is documented to allow it to be re-run, however, the transformation to a target implementation language is completely automatic.

More recently, the research group has focussed on the starting specification language: GIST, through: (1) GLITTER, a user directed tool which assists in the manipulation of GIST specifications to satisfy particular TI transformations [45], and (2) the construction of GIST specifications via *parallel elaborations* [44].

PSI — PSI is a system providing assistance in the synthesis of efficient programs. It is constructed as a set of *expert groups*: the acquisition group, the model builder, and the synthesis group [89]. The acquisition group uses an English parser and interpreter, a trace expert, a discourse expert, and a domain expert to acquire the specification from the user. The results are program fragments which are input to the model builder. The model builder generates an abstract implementation in a high-level language with assistance from the user only when necessary. The synthesis group employs a coding expert, PECOS, and an efficiency expert, LIBRA, to generate the final program output [7]. PECOS is an extensible knowledge base of transformations for symbolic programming. It develops program trees through successive refinement: i.e., it generates multiple solutions where each path from the tree root to a leaf is one program solution. User advice, heuristics, and LIBRA are used to prune the tree and thus minimize the search [68].

This work has more recently been used in the knowledge based programming environment called CHI (and the equivalent commercial product called REFINE) [107]. CHI uses the wide-spectrum language V and shared databases (as opposed to autonomous expert groups) to give the user a more readable and consistent interface. Again the transformations reflect knowledge of the high level specifications and programming knowledge.

2.3.2 General Programming Environments

PDS – Program Development System — PDS supports the definition, testing, and maintenance of large programs and program families being maintained

in synchrony [31, 96, 89]. The emphasis is on program derivation from abstract specifications by nontrivial transformations. A program is defined as a collection of modules for which PDS checks the consistency of the interface specifications. A module may have several descendants all of which are considered to be refinements of a single original specification. A goal of this project is to support a multilevel view by providing appropriate structures and the means for maintaining these different versions under modification. A database of program constituents give automatic and incremental re-derivation [32].

This system also provides specification tools, analysis tools and execution tools. The transformations are not stored in a catalogue, but rather, they are defined by the user and represent design decisions which are encapsulated by modules. The rules are represented *schematically*; i.e., with a syntactic input pattern, an optional semantic predicate for its applicability (contingencies for use), and a syntactic output pattern. They introduce the user's nomenclature, implement the abstract implications of the specification, and derive the implementations of an abstract program. These rules are applied in repeated passes so as to refine the implementation from the input specification.

PA – The Programmer's Apprentice — This environment, being developed at MIT automates (assists at) several levels of development: e.g., analysis, synthesis, modification, specification, verification, and documentation. It acts as a "junior partner and critic" [117] keeping track of details and assisting in documentation, debugging, and maintenance. There are three main areas of assistance which are found in: (1) the Requirements Apprentice, (2) the Design Apprentice, and (3) KBEmacs which automatically implements a program after the user has selected appropriate algorithmic program fragments and instantiated them to correspond to the current application [95, 117]. All three supply expert knowledge and common structures for their domain. The latter are referred to as cliches and are intended to promote reuse. Therefore the cliches of KBEmacs are typically programming constructs (e.g., search algorithms). The cliches of the Design Apprentice represent high-level design decisions from which the tool automatically concludes on low-level decisions. For the Requirements Apprentice the cliché library is a declarative database of information

relevant to requirements in general and to the domains of interest. It is however, the user who must make the difficult decisions (choosing the cliches) needed to develop the specification, design, or program.

In terms of verification, when given a partial program the apprentice can complete the description using a knowledge base of programming plans (knowledge on the structure of programs and cliches: e.g., loops and data structures). On the other hand, given a program, it can analyse the structures and recognize the programming plans within it. It can reason about plans by: (1) verifying their existence in a concrete program, (2) finding discrepancies between the expected and actual use of the plan, and (3) evaluating the effects of modifications on an existing program. The low-level plan structures provide a means of program analysis, while the cliches (a component of program plans) support program synthesis.

The cliches do not represent correctness preserving transformations but rather a means to build programs from well known and understood program plans. The assistance is available to, but not forced on, the developer.

2.3.3 Summary

The trend, as discussed through example above, has been to build programming *assistants* starting with the lowest level of auto-programming (translations to more efficient programs) and, as the research progresses, each of the more abstract levels are added to the tool (e.g., specification to implementation, and requirements to specification transformations). Therefore, in general, the current focus of this community is on the development of formal specifications from informal statements of the requirements.

The problem being considered by this thesis is at the mid-level of automatic programming: i.e., the issues of translating formal and assumed correct specifications, to design, however, the problem domain is new to this field of research. Whereas the study of efficient implementations for concurrent programming has a reasonably extensive background (e.g., semaphores, monitors, inter-process message passing, and distributed locking algorithms [76]), the typical process of designing reactive system software with soft timing constraints is still relatively creative and ad hoc.

The lack of standards for design alternatives (i.e., the wide range of real-time support provided by the different target implementation languages), and the relative youth of concurrent programming dictates lower levels of knowledge and experience in the development of concurrent system designs [8].

Transformational programming replaces abstract structures with concrete, conventional constructs according to standard design *plans*. The concept of *process plans* is strongly emphasized in the literature on knowledge-based software engineering. The Kestrel report [51] mentions "programming process" as a key feature. They claim that currently the process of development is informal and largely undocumented because the interest lies only with the end product. Feather [44] states that the "construction process" is worthy of explicit formalism and support. Many of the existing tools produce a description of the development process which is still largely influenced by the designer who acts as decision maker during the development. The purpose of these scripts is to support the common goal for maintainable software where modifications can be reflected by re-application of the same design process. This process is considered to be well known by the programmers.

The habits of programmers (their *programming plans*) for the design of sequential programs are studied in [108]. This paper indicates that experienced programmers select accepted and valid, approaches and algorithms over non-conventional alternatives. Fickas [45] implies that correct programming is relatively easy while efficient programming is key.⁵ However, for reactive system designs the following questions arise:

1. How clearly are the design processes for concurrent systems understood?
2. Can these processes be automated to produce efficient designs?
3. How much experience exists for developing these systems?
4. Are there known standard design approaches (or do the various languages each support a different strategy)?

⁵This paper considers a reactive system specification, however, it is viewed only as a closed system (i.e., the environment is included in the solution), and solved by an event driven simulation design.

5. How do “non-standard” techniques effect the planning for system development?

This thesis is motivated by the assumption that both correct and efficient designs are still difficult to achieve for reactive systems, and therefore research on the development of such designs is lagging that of sequential algorithmic designs. There is still much work to do at the level of specification to design transformation [8].

The main goal, when placed in context with this material, is a framework which: (1) catalogues clear and concise alternative potential solutions (i.e., *plans* called *design fragments*) to reactive system software designs, (2) a method for designing with design fragments, and (3) a process for selecting and verifying the use and interconnection of design fragments. The design fragments are design alternatives for the communications within a given allocation of entities to actors:⁶ i.e., they define formal relations between the specification and design alternatives which can be proven rigorously. Such a framework would catalogue potential solutions much like *schemas* in PDS [89], Harandi’s *templates* [53], or the *cliches* of the Programmers Apprentice [117] except that the alternatives depend on features other than efficiency and correctness such as constraints from hardware and real-time support in the target language. Sets of alternatives of correctness preserving transformations would be limited in size (at least in the early stages of this research) by the high level of domain (application) specific properties in reactive system design. However an approach used in the AFFIRM environment, where users are shown how to prove properties of design choices for algebraic data-type specifications [89], is an interesting alternative for reactive system design.

A “wish-list” of properties for the *design fragments framework*, taken from this study on transformational programming, is given in Section 4.2.1 as part of the introduction to the design process.

⁶An actor is a concurrent component of the design.

2.4 Formal Validation Techniques

Formal validation techniques for concurrent software design can be used to provide *complete traceability* between a formal specification and its design. There are two fundamental techniques: (1) to show conformance by proving that the specification, expressed as a set of property assertions, is satisfied by the design [52, 84, 77, 34, 50, 3], and (2) to show that the design is a *reduction* of the specification by proving an equivalence relation between the specification and design (e.g., observational equivalence) [80].⁷

Both approaches can be used as a compositional technique where the proofs are performed incrementally as the design is being constructed, or as a non-compositional technique where the specification and design are both complete before the validation process begins. This thesis is interested in non-compositional techniques for evaluating assertions, and illustrating equivalence between behaviours. Non-compositional techniques are considered, not because the design must be complete, but rather because the technique is applicable regardless of the state of the design because of the independence of design fragments; i.e., the assumption that the final design, and thus the remaining design (completed or not), conforms to the specification given that all contingencies are satisfied.

Validation via Property Assertions: The current trend for validating a design against a specification, defined as a set of assertions, uses *model checking* [34]. In this technique branching time temporal logic statements are used to represent the specification as a set of safety (invariant) and liveness (progression) assertions. These statements are verified against a graph of all reachable system states which is generated automatically for finite-state concurrent system designs: e.g., designs expressed in CSP [61, 34], or ESTELLE [19, 50].⁸ Although the verification has a

⁷Reduction of behaviour is similar in notion to *satisfaction* in VDM where, for sequential programs, the design satisfies the specification if the pre-condition is weakened and the post-condition strengthened with respect to the higher-level specification [67]. VDM is used by Ledru [72], to *refine* a terminating statechart specification.

⁸Section 3.4 examines a procedure for developing a graph of reachable system states from a finite state machine description: *reachability analysis*.

complexity linear to both the size of the specification and the size of the reachable state graph [34], the graph itself explodes exponentially with the size of the system, and is not necessarily finite [50].

Methods of partial evaluation are used to reduce this problem in some cases. Here, the graph is generated “on the fly”, and the verification procedure is stopped as soon as a contradiction is encountered. However, to prove that a statement is true, the whole graph is required. Partial ordering is used to reduce the size of the state graph by eliminating interleavings which have known consequences to the system behaviour. Such “known” consequences can be difficult to define. Designs can be layered such that behaviours are decomposed, and thus simplified in terms of verification, but they are, henceforth, constrained by the interfaces introduced by the decomposition.

This area of research still holds a lot of promise. A great deal of work is being done to reduce state graph sizes, and to optimize the processing involved in the validation algorithms. Recent reports [40] express optimism based on compression algorithms, and large RAM disks now available on moderately priced workstations.

Validation via Equivalence Proofs: Proving that one behaviour is a *reduction* of another also requires the combined system behaviour (e.g., as defined by a graph of reachable states). In this case, reachable state descriptions of both the specification and the design are required, and the two behaviour patterns are compared to show that one is equivalent to the other based on some definition of equivalence. For example, the design might be a constrained version of the specification, taking one of a set of possible paths through the reachable state description of the specification. In this case, the observed behaviour is equivalent thus giving *observational equivalence*.

In terms of specification contingencies, only those segments of the system behaviour including the specification segment being designed, are required for the proof. However, there is no obvious mechanical way to extract this behaviour without first defining the entire system behaviour. Therefore, performance issues in this technique suffer from the same problems as above: i.e., state explosion.

Chapter 3

Specification of Software Requirements

This chapter covers four aspects of the specification technique developed for designing reactive systems with *strong traceability*.

Concepts — A discussion of the conceptual ideas behind EBD specifications is given in Section 3.1. Here, the goal is to illustrate how and why the models and methods (defined later in the chapter) fit into the overall development process for strong traceability.

Model — The specific details of *Entity Behaviour Descriptions* are given in Section 3.2. This discussion includes: (1) an architectural overview of the communications model underlying the specification technique which explains how the application behaviour is decoupled from the communication semantics, (2) the primitive components and functions which make up an EBD, (3) the syntax of EBDs, (4) the general semantics of EBDs, (5) the precise semantics of EBD specifications, and (6) examples of EBD specifications including the Readers/Writers problem specification used extensively throughout this thesis.

Methods — A brief discussion on deriving EBD specifications is provided in Section 3.3. The methods suggested are based on *typical* decomposition tech-

niques for dividing a system into subsystems and further decomposition of subsystems into entities using the *slice* analysis techniques of Vigder and Buhr [115, 25, 26]. Hatley's methods of real-time system specification are used to define the behavioural requirements by communicating EBDs [59].

Analysis — EBD specifications can be verified using *reachability analysis* [121], however their definition allows for extensions which reduce the state space of the analysis when process pools (collections of like EBDs) are present in the specification (as is the case of the Readers/Writers problem). This extended reachability analysis process is described in Section 3.4.

3.1 Specification in the Design Framework

The EBD specification technique is influenced by two major factors: (1) the domain of the problem to be solved by this thesis (i.e., reactive system software design), and (2) the proposed design process for *strong traceability*.

Results from Problem Domain Study: The problem domain is large concurrent, reactive system design. Studying the state of the art in reactive system specification and design has highlighted several goals for the EBD specification technique (cref. § 2). The results are as follows.

Finite State Machines — Many specification techniques for reactive systems are based on finite state machine descriptions [55], [42], [65], [23]. As such, state machine descriptions of behaviour are considered to be a reasonable choice. EBDs present state machine specifications of behavioural requirements.

Entity-Based Structure — Entity analysis is an accepted approach to decomposing a system for specification and design, therefore EBDs represent *communicating entities*.

Multiple Communications Paradigms and Event Orderings — EBDs have the flexibility to support the following (typically implementable) interprocess

communication paradigms: (1) broadcast, (2) asynchronous, and (3) synchronous, and the following event orderings: (1) FAIR, (2) FIFO, (3) LIFO, and (4) priority, without the introduction of new entities (e.g., as queue entities are required in LOTOS).

Single Notation for Both Functional and Behavioural Requirements —

EBDs support both *flow chart* transitions, (based on internal decisions) and *state machine* transitions (based on external, event driven decisions), without the introduction of a new supporting notation (like STATEMATE activity-charts). This satisfies the goal of providing structure for the functional requirements within the behavioural specifications.

Hierarchical Behaviour Descriptions — Large design issues promote a need for hierarchical (layered) EBD specifications, and methods for dealing with *subsystems* and interface descriptions at subsystem boundaries.

Event Screening as a Function of the Application Behaviour — The underlying communications model of an EBD specification minimizes the constraints that it imposes on event ordering; i.e., the specification, rather than the communications model, defines the required ordering for accepting incoming messages, and must screen or discard incoming messages as not applicable to the current state. For example, as mentioned in the background chapter, randomly generated messages are not constrained, by the model, to be received in any particular order (e.g., by order of arrival).

Results of Design Process:

Allocation — The allocation step of the development process is key to “large sense traceability” (i.e., tracing completeness of the design). It requires that EBDs of the specification be allocated wholly to an actor in the design. Several EBDs can be allocated to a single actor, but an EBD is never decomposed across actor boundaries at the allocation phase. As such, it is important to define the entities of the specification in a manner which would allow them to be mapped to a single design object. Entity analysis is a suitable process

for establishing entities in the specification and its design, and is therefore used by the methods defined for establishing an EBD specification. An EBD specification defines the requirements of a set of communicating entities.

Designing with Design Fragments — The ACM component of MachineCharts is used to capture the application behaviour explicitly in a design,¹ while design fragments implement the semantics of the communications. The EBD specification technique has an established understanding of the separation of application-specific details from the generic semantics of the communications.

Selecting Design Fragments — The selection of design fragments supports the concepts of reusability of the specification. Different choices result in products with the same behavioural description, but potentially varied properties. As such, the EBD specification technique is quite plain in nature. It supports the description of behaviour, but does not include any features which would allow for the description of product-domain-specific properties such as performance, actor priorities, or criticalness and timing. Such properties are difficult to qualify; their subjectivity is a function of policy or personal bias.

Proving Completeness — As mentioned under “Allocation”, a schema for completeness is defined by the specification and allocation of EBDs to actors. The separation of application-specific behaviour details from the semantics of the communications is used to prove completeness of a design. When the application behaviour is mapped directly into an ACM in the design, the components which remain to be satisfied are the individual interactions. The hierarchy of the specification is used to mechanically define a schema where the events (which must be satisfied by the design through application of design fragments) define the leaf nodes of a tree structure (cref. § 5.2).

Proving Correctness — Design fragments define formal relations between independent, non-overlapping, generic specification segments and partial MachineChart designs. Independence, in terms of the design of such segments, is

¹MachineCharts are not critical to the design approach presented here.

guaranteed by the design fragments, and formal relations are made possible by EBDs which have a formal representation (i.e., state machines) with precise operational semantics.

The result is the EBD specification model defined in the next section. The model is not necessarily complete,² but is complete with respect to the behaviour descriptions: i.e., complete in terms of how conformance is to be shown between a specification and its design by the process proposed in this thesis. The model is also not restricted to this design method. Its simple foundation makes it a useful tool for specifying concurrent systems in general.

²For example as mentioned above, EBDs do not describe properties such as performance, actor priorities, and criticalness. The description is also incomplete in terms of their potential for analysis.

3.2 EBDs –

A Model for Communicating Entities

Chapter 2 reviewed several existing models for specifying reactive system behaviour. In conclusion, a state machine model is chosen as the basis for *Entity Behaviour Descriptions* (EBDs): the specification technique defined in this thesis. The *abstract behaviour* of a concurrent system can be expressed in this manner. This behaviour specification consists of two elements.

Application Behaviour — The application-specific behaviour defines the conditions (e.g., the event receptions), and actions (e.g., event sends) of the state transitions.

Communications Semantics — The communications semantics describe how the events are transferred from one machine to another: e.g., an asynchronous send.

Two of the existing state machine based communications models introduced in Chapter 2 are: (1) the statechart model, and (2) the ISO model with asynchronous communications. The statechart model employs communicating state machines for the application-specific behaviour, and broadcast communications to define the operational semantics. The ISO model also uses communicating state machines, but supports the communications operationally with simplex, FIFO, point-to-point channels. *Entity Behaviour Descriptions* (EBDs) are communicating state machines which allow the analyst to express three communication paradigms (asynchronous, synchronous, and transient broadcast) with a variety of event orderings (e.g., *don't care* ordering, and FIFO ordering). The precise operational semantics are defined by an underlying communications model (a meta-model) which explicitly separates the description of the application-specific behaviour from the semantics of the communications.

This section defines the EBD specification model as follows. Section 3.2.1 provides a brief overview of the meta-model on which EBD specifications are interpreted. This model is referred to throughout the remaining text in the chapter.

Section 3.2.2 defines the elements which constitute an EBD, and Section 3.2.3 defines the precise operational semantics of an EBD specification. Section 3.2.4 provides some example specifications, and Section 3.2.5 discusses related research.

3.2.1 Architectural Description

EBD specifications are based on a communications model which explicitly separates the application-specific behaviour from the communication semantics. This separation is described by a meta-model consisting of meta-level EBDs, and *Entity Interfaces* (EIs) as shown in Figure 3.1. Every EBD is a communicating state machine description of the abstract behaviour of an entity. In the meta-model, each EBD is associated with a peer EI which serves the EBD to transfer messages to the appropriate destination EI, and consequently the destination EBD. The EIs manage the different communications paradigms and event orderings, expressed by the EBD, according to strict operational semantics.

The decision to include EIs to describe the operational semantics of the communications model was affected by the use of entity analysis. That is, the communications model itself can be viewed as a reactive system, and as such it might be configured by EBDs which interact with EIs who carry out the message transfer according to the instructions of the EBD. Although the abstract behaviour is described by the EBD, the meta-level EBD is responsible for only the application-specific behaviour while the EI is responsible for the communication semantics.

In this model, there is a client-server relation between the EBD (the client) and its EI (the server). The EBD sends messages to the EI when it wishes to receive an event, and then waits for the EI to respond. Similarly, the EBD sends messages to the EI to send events, and again waits for the EI to respond before proceeding. The EIs interact by sending asynchronous events which are queued first-in-first-out (FIFO), according to their arrival time, at the interface to the receiving EI (i.e., all incoming events are placed in one queue as shown in Figure 3.2). It is the functionality inside the EI which buffers and retrieves events according to their communication semantics and the ordering specified by the EBD.

All events (except for transient broadcast) are saved by the EI in its internal

Figure 3.1: Communications Meta-Model

buffers. Events are, therefore, deferred until they are acceptable (and received) by the EBD. No assumption is made to discard events which arrive at unexpected times (i.e., those referred to as *unspecified receptions* in the ISO model). The EBD must *admit* (possibly in a particular order), *reject* (which is the default for transient broadcast events which are received by the EI, and not currently requested by the EBD), and *screen* the incoming events. This description constitutes the application behaviour: i.e., for an event reception, the EBD defines which messages are valid for a given state (called the *satisfiable event set*); and, for an event send, which messages are to be issued. Figure 3.2 illustrates the view of event queueing as seen by the EBD and as compared to that of the EI. The EI sees a single FIFO queue of events, and the EBD sees multiple queues with unrestricted orderings.

The EBD also defines the required communication semantics (which are carried out by the EI). For an event reception the EBD defines: (1) the source of each message in the satisfiable event set, (2) its communications paradigm, (3) the order in which messages are received, and (4) if there are any data- or time-dependencies on accepting a message or a satisfiable event set. For an event send the EBD defines:

Figure 3.2: EBD and EI Views of Event Queuing

(1) the destination of each message to be sent, (2) its communications paradigm, (3) the order in which the messages are to be sent, and (4) any *attributes* to be associated with an event (e.g., the sender identifier and the arrival time are attributes assigned to all events by the sender and receiver EIs respectively).

This chapter provides details of the EBD state/transition modelling technique, and uses this to define the operational semantics of the communications model by developing a precise description for the meta-model.

3.2.2 Entity Behaviour Descriptions

EBDs are the state/transition description technique used to represent the interactions of entities in a reactive system. Like statecharts they support state clustering and refinement, have an underlying communications semantics, and a communications model for showing concurrency. EBDs have four primitives: (1) *operations*, (2) *conditions*, (3) *events*, and (4) *directives* which can be used to describe four primitive functions: (1) *perform operation*, (2) *evaluate condition*, (3) *send event*, and (4) *receive event*. Unlike statecharts, these primitive functions are assumed to require a finite amount of time to accomplish; there is no notion of instantaneous and simultaneous primitive functions occurring within an EBD. This is analogous to the ISO asynchronous model which also assumes that only one primitive function

(e.g., send event or receive event) occurs at any given time.

EBDs execute sequences of primitive functions: e.g., *evaluate conditions*; *receive event*; *perform operation*; *send event*; *perform operation*; and so on. It is the *operation* primitive which allows for flowchart (data-based) and state machine (event-based) transitions to be modelled by the same notation. With statecharts, the *actions*: **start**(*activity*) and **stop**(*activity*), indicate the execution of a functional requirement specified within an activity-chart. With EBDs these activities (one type of *operation*) are shown directly on the EBD. The statechart condition: **active**(*activity*), is defined by the EBD semantics for *operations*.

This section begins with definitions of the EBD primitives and the primitive functions. EBD states and transitions are then defined in terms of the primitive functions which they can represent. This is followed by a description of the interactions between the EBDs and the EIs (meta-entities) of the communications model which results in a meta-model of an EBD specification.

3.2.2.1 EBD Primitives

In this section each of the four EBD primitives is defined.

Operation – An operation requires a finite amount of time to execute. For example, operations are used to: (1) represent functional requirements; e.g., *GenerateReport*, and (2) to assign values to auxiliary state variables (based on the execution of a functional requirement, or based on messages and message data received as a result of an interaction). The only strict requirement of an operation is that, once started, it will eventually complete. As such, state *clustering* is supported by EBD specifications. Any segment of an EBD which can be shown to eventually complete, can be *collapsed* into an operation; hence the term *collapsed behaviour*.

Condition – Conditions are boolean expressions (logical conjunctions disjunctions, and negations) of “simple boolean expressions.”

1. Values bound during the execution of an operation can be used to state a simple boolean expression; e.g. $x > 1$ where x is an integer variable assigned a value during the execution of some operation.

2. Values bound in message data can be used to state a simple boolean expression.
3. A boolean function can be used to state a simple boolean expression. The function can evaluate and return additional parameters if necessary; e.g., *empty_queue(Q)* tests a queue state, and *queue_top(Q.top)* tests the queue state and returns the top element of the queue, if the queue is not empty. In this second example, the condition is false if the queue is empty.

Event - Entities communicate via events. Asynchronous and synchronous events define: (1) the source or destination entity, (2) the communications paradigm, and (3) the message in the following formats.

event send:

destination_entity communications_paradigm message

event receive:

source_entity communications_paradigm message

Transient broadcast events define: (1) the communications paradigm, and (2) the message using the following formats:

event send: *message communications_paradigm*

event receive: *communications_paradigm message*

Source and Destination Entities -- The source and destination entities are defined by: (1) a specific entity identifier, (2) an unspecified entity descriptor, or (3) a variable entity descriptor.

specific entity identifier - An entity name is used to define the source or destination entity explicitly.

unspecified entity descriptor — An unspecified entity descriptor is characterized by an underscore (“_”) character at the beginning of the representative label; e.g.: *_Printer - File(FilcId)* will send the asynchronous

Communications Paradigm	Send an Event (used with a destination entity)	Receive an Event (used with a source entity)
Asynchronous	–	+
Synchronous	!	?
Transient Broadcast	<	>

Table 3.1: Syntax Used to Declare the Various Communications Paradigms

File(FileId) message³ to any entity accepting this message.⁴ The accepting entity can receive the message from an explicit sender (e.g., *Sender + File(FileId)*), or from an unspecified sender (e.g., *_Sender + File(FileId)*).

variable entity descriptor — A variable entity descriptor is bound (using the *set()* directive defined below) to the appropriate entity identifier as the result of an interaction. It is used in combination with unspecified entity descriptors as will be shown.

Communications Paradigms — The symbols used to define send and receive events for asynchronous,⁵ synchronous,⁶ and transient broadcast interactions are illustrated in Table 3.1.

Messages — A message can be a simple message type or a message type and associated message data. For example, *Sensor + Data(DataPtr)* defines an asynchronous event sent by the *Sensor* entity with message type *Data* and message data *DataPtr*.

³The “ ” character in this event indicates that it is an asynchronous message as defined in Table 3.1

⁴The underscore character is used in the same manner in the Prolog programming language to identify a variable term

⁵This syntax has been taken from ISO protocol specification community which uses it to represent asynchronous message passing.

⁶This syntax is taken from the CSP and LOTOS communities where it is used to define process synchronization.

Events have associated *attributes*. For example, given that each entity has a unique identifier, the *sender identifier (SID)* is an attribute associated with every event sent, and is assigned its value by the sender EI. The *receiver identifier (RID)* is an attribute associated with every event sent, and is also established by the sender EI. The *arrival time (T)* is associated with every send event, but here the value is assigned by the receiver EI. *Priority* is an attribute assigned by the sender (*SP*) or receiver EBD (*RP*). By default, the priority of a send event is the same as the priority of the sending entity which is, by default, the same for all entities. The priority of an event to be received is, by default, the same for all events in a satisfiable event set. A user defined attribute is included to allow for extensions.

Directives — Directives provide additional information about how events are to be exchanged between entities: i.e., the information is in addition to that which is embedded in the event itself as the source or destination entity and the communications paradigm. In the meta-model, where events are exchanged between EIs through a single FIFO queue interface at each EI, these directives generally describe how events are re-ordered by the EI before being passed from the EI to its peer EBD as a result of a request to receive an incoming event. Directives are applied to individual events to be sent or received, to a list of events to be sent, or the *satisfiable event set* (cref. § 3.2.1 and page 67), and are interpreted by the EIs.

Some directives depend on attributes, while others define dependencies on message data, or timeout limitations. Still others are used to define variable entity descriptors. The EBD model assumes a default set of directives which provide FIFO queueing of all unique events, and FAIR selection over events in a satisfiable event set, as highlighted below.

Directives for individual send events:

set(var=RID) — This directive is used to set the EI variable *var* when an event is sent to an unspecified receiver whose identity is required for interactions which follow. In this situation, the sender EI chooses a specific receiver (*RID*) from one of the possible receivers (i.e., those which can receive this event). This directive is not used in the default model. It must be specified explicitly.

priority(SP) — This directive assigns a priority attribute with value *SP* to the event. The value of *SP* can be statically or dynamically defined. The default value for *SP* is *medium*, where the possible values are: *very low*, *low*, *medium*, *high*, and *very high*.⁷ For a static definition, one of the above values is written into the directive. Values defined by operations are used to dynamically assign a priority value. This attribute is transferred to the receiver EI as defined in Section 3.2.3.

Directive for a set of send events:

order(func) — This directive tells the EI how to order a set of events. By default, events are sent by the EI in the same order in which they are listed in a transition (cref. page 66: *transition consequence*). The predefined values of *func* are: FIFO (the default, defined by the left-to-right ordering of their written representation), LIFO (defined by the right-to-left ordering of their written representation), FAIR (which is a don't care ordering), and priority(SP) where the priority attribute assigned to each event to be sent is used to determine the order. User defined functions, expressed in terms of predefined, or user defined attributes, are also allowed.

Directives for events to be received:

set(var=SID) — This directive is used to set the EI variable *var* when an event is received from an unspecified sender whose identity is required for interactions which follow. In this situation, the receiver EI uses the *SID* attribute associated with the event being matched to a receive event request from the meta-level EBD. This directive is not used in the default model. It must be specified explicitly.

priority(RP) — This directive assigns a priority attribute with value *RP* to the event reception. The default value for *RP* is *medium*, where the possible values are: *very low*, *low*, *medium*, *high*, and *very high*. For a static definition, one

⁷The set of values for priority levels given here are for illustration only. Integer values are equally possible, and in fact, more flexible.

of the above values is written into the directive. Values defined by operations are used to dynamically assign a priority value. The attribute is used with an *order(func)* directive for a satisfiable event set.

order(func) — This directive tells the EI how to order a *unique event* for transferring a message to this EBD in response to a receive request. Each event in a satisfiable event set (i.e., from one transition) defines a *unique event* therefore, an event with an unspecified sender is a *unique event*. The predefined values of *func* are: FIFO (the default, defined using the *T* attribute), LIFO (also defined by the *T* attribute), FAIR (which is a don't care ordering), and priority(SP) where the priority attribute assigned to each event by its sender is used to determine the order (i.e., the priority attribute assigned by the sender is used to order the events at the receiver EI). User defined functions, expressed in terms of predefined, or user defined attributes, are also allowed.

reset — This directive is used to reset (clear) the contents of an EI buffer after an event reception is requested by the EBD, i.e., only the last occurrence of the event is returned to the EBD, and all earlier occurrences are discarded. A binary semaphore is specified using asynchronous send events (to signal the semaphore) and asynchronous receive events with the *reset* directive (to wait on a semaphore).

when(rel(data,value)) — This directive is used to select a message based on message data (as opposed to an attribute). The EI is told to select a message for return to the EBD if the message data, *data*, relates to *value* according to the binary operator *rel*. For example, *rel* might be the equivalence relation giving: *when(msgdata=expected)*. The term, *value*, can be statically defined (e.g., the integer: 2), or dynamically defined using a variable set by an EBD operation (e.g., *expected*).

Directives for satisfiable event sets:

order(func) — This directive tells the EI how to select from the different unique events in a satisfiable event set. The predefined values of *func* are: FIFO

(defined using the *T* attribute), LIFO (also defined by the *T* attribute), FAIR (the default, which states the any event requested infinitely often is selected eventually), priority (SP) where the priority attribute assigned to each event by its sender, is used to determine the order, and priority (RP) where the priority attribute assigned to each unique event, by the receiver, is used to determine the order. User defined functions, expressed in terms of predefined, or user defined attributes, are also allowed.

timeout(*t*) — Timeouts are a notion of the communications model that define the condition where at least *t* time units have elapsed since the start of the (meta-level) EBD-EI interaction. The timeout directive is used on a *timeout transition*; i.e., there can be at most one transition from a given state with the *timeout(*t*)* directive, and that transition is taken in the case where a *timeout* is returned by the EI. Some example *timeout transitions* are given later in this section, and a full description of this feature is given with the operational semantics of the communications model.

3.2.2.2 EBD Primitive Functions

This section discusses the primitive functions of EBDs.

perform operation — An operation represents a requirement involving a finite amount of execution time, and therefore *performing* an operation, is the act of starting and completing the requirement. This can be an activity or function to be evaluated; or, by the hierarchical nature of EBDs, it can be any finite sequence of primitive functions (cref. page 61: States).

evaluate condition — A condition expression is evaluated as true or false.

send event — This function represents the act of sending a message from one entity to one or more other entities, according to the semantics of the communications paradigm defined in the event and any associated EI directives.

receive event — This function represents the act of receiving a message from one entity by another according to the semantics of the communications paradigm

defined in the event and any associated EI directives.

3.2.2.3 EBD States and Transitions

States — States can represent operation primitive functions, or a *null* state. If an operation is associated with a state, it is assumed to be ready to begin execution as soon as the state machine enters the corresponding state, and a transition out of the current state cannot be made until the operation is complete. *Hierarchical EBDs* are achieved by defining an operation as a sequence of primitive functions known to eventually complete (where the sequence of primitive functions is described using EBD states and transitions). Such sequences are called *collapsed behaviours* [16]. A *null state* is used as a placeholder between one transition and a second which requires an event reception. These states are typically labelled *wait* or *null* states to illustrate that no operation is being performed, and that the EBD must await a reply from the EI in order to transition from the state.

Transitions — Transitions can represent any of the primitive functions. Therefore an operation can be performed during a transition, conditions can be evaluated, and events can be sent or received. They are written in a form similar to statechart transitions, and are read left to right so as to define the sequence of the primitive operations.

transition antecedent/transition operation/ transition consequence

transition antecedent — A transition antecedent can result in conditions being evaluated and events being received according to their communications semantics and the associated EI directives. They are written as shown below, and according to the rules which follow.

event reception{directives}[condition]

1. Directives are listed and separated by commas, e.g.,
{order(FIFO), priority(medium)}.
2. Unspecified sender receptions cannot overlap with other specific entity receptions; i.e., a given message type matches with exactly one unique event. For

Figure 3.3: EBD segment illustrating transitions from a null state.

example, if an EBD contains the unique event: *_Sender + File(FileId)*, it cannot have a separate unique event: *S1 + File(FileId)*. Any number of transitions can have the same unique event as part of their transition antecedent.

3. From a null state, every transition must include an event reception or the timeout directive with a non-zero timeout parameter. No spontaneous transitions can occur in EBDs, and a timeout is not considered to be spontaneous. An example is shown in Figure 3.3; it illustrates the concept of a *timeout transition* introduced earlier.
4. The completion of an operation is considered to be an acceptable cause for a transition (as it is in statecharts). EBDs evaluate transitions which do not depend on events before those requiring events. If such a transition can be satisfied, then it is taken without considering those requiring events. As such, there can be only one transition out of an operation state whose antecedent is based solely on the completion of the operation because this transition always evaluates to true. These transitions are identified by empty antecedents (as illustrated in Figures 3.3 and 3.4); and, if defined, then they are the only transition ever taken from the state. They illustrate a direct flow in a flowchart model.
5. Transitions from states which are based only on conditions defined by the outcome of the operation, are allowed, and are analogous to decision points

Figure 3.4: EBD segment illustrating a direct flowchart transition.

Figure 3.5: EBD segment illustrating alternative flowchart transitions.

in a flowchart model (see Figure 3.5). The conditions must cover all possible outcomes of the operation, and that which evaluates to true is the one which is taken. This set of transitions must be deterministic.

6. An operation state can transition as a result of an interaction between entities, therefore events can be used in the antecedents of these transitions. An example is shown in Figure 3.6. When no directives are specified for a event, the default is applied. In this thesis, each unique event is ordered FIFO by default, based on the arrival time; e.g., *reader + rqst* and *writer + rqst* are the two unique events in Figure 3.6.
7. Some transitions depend on message data. When this dependency is described using conditions, the event is provided by the EI before the conditions can be evaluated. Because the EI has responded to a request for an event, that event cannot be returned to the EI, and therefore the transitions of this event must be deterministic (i.e., the conditions must cover all possible data values) as

Figure 3.6: EBD segment illustrating event driven transitions.

Figure 3.7: EBD segment illustrating use of message data.

illustrated by example in Figure 3.7.

When the data dependency is explained to the EI through a directive, the EI does not respond to the EBD until the condition is satisfied.⁸ An example is shown in Figure 3.8.

8. The events which trigger transitions from a state can be ordered using the *order* and *priority* directives. For example, Figure 3.9 illustrates a state with a complex ordering criteria. It shows a directive associated to the satisfiable event set

⁸This type of behaviour can be implemented explicitly by at least the following languages. MLog [69], SR [2], and Concurrent C [48].

Figure 3.8: EBD segment illustrating a message data directive.

by using a line to connect the state to the directive (i.e., $order(priority(RP))$).⁹ The directives assigned to individual events are applied first, therefore: (1) $msg1$ is ordered FIFO (i.e., oldest to newest arrival), (2) $msg2$ is ordered LIFO, and (3) $msg3$ is ordered according to the priority attributes assigned by its sender, $s3$. The satisfiable event set is ordered according to the priorities assigned by the receiver (i.e., this EBD). Therefore, if there is already an outstanding $msg2$ and $msg3$, then a “fair” choice is made between them, and the *first* message of the selected type is returned, where the *first* is defined by the ordering on the individual events. If one or the other of $msg2$ or $msg3$ is already buffered in the EI, then the *first* is returned to the EBD. If there are neither any $msg2$ or $msg3$ events being buffered by the EI, then the *first* (the oldest) $msg1$ is returned to the EBD. Otherwise the first to arrive of any $msg1$, $msg2$, or $msg3$ is returned to the EBD.¹⁰

Therefore transitions can:

1. describe internal decisions if based on values assigned during the operation state preceding the transition (flowchart model),
2. describe event-based decisions if dependent only on the receipt of an event and possibly values of message data (interacting entity model), and

⁹A timeout directive is also associated with the satisfiable event set, but it is shown on a transition as illustrated in Figure 3.3.

¹⁰There are no simultaneous arrivals because all events are queued, FIFO and in a single queue at the EI-EI interfaces (see Figures 3.1 and 3.2).

Figure 3.9: EBD segment illustrating the ordering and priority concepts.

3. describe mixed internal/event-based decisions if both values set by an operation and the receipt of an event are required for the transition.

transition operation — An operation can occur between the transition antecedent and transition consequence. Again this means that any requirement known to complete in a finite amount of time can be shown to occur during a transition. This includes sequences of primitive functions, and thus it extends the use of EBDs to describe hierarchical (layered) behaviours. An example transition operation is the update of auxiliary state variables which depend on the preceding transition antecedent.

transition consequence — A transition consequence is either empty, or a conjunction of send event functions which are interpreted by individual EI directives, or by a group directive using the following format. The parentheses are optional when a single event is to be sent, and the directives are listed and separated by commas.

(send event{EI directives}, ...){group EI directive}

When no directives are provided, the default is applied. For this thesis, the default is to send each event at *medium* priority (i.e., as the individual EI directive), and in the order in which they are listed (i.e., as the group directive). As a result, transient broadcast and asynchronous event sends are delayed by an indeterminate time if they are listed after a synchronous event send.

3.2.2.4 Properties of an EBD State Change

EBD states can have any number of outgoing transitions as long as a deterministic choice can eventually be made as to which transition should be taken. This choice begins by evaluating the transition antecedents. Conditions based on values assigned during an operation (this includes the true condition that an operation is complete) are evaluated first. If any of these transition conditions evaluate to true, then at most one of these transitions has no other components in its antecedent. In this case, the correct transition has been determined (as was described earlier). Otherwise, these transitions become part of what is called the *satisfiable transition set*. This name implies that all elements of this set mark potential transitions, one of which will be taken when the necessary interactions with the EI are complete.

EBDs must define which events are considered at a state change. By showing an event in the antecedent of an outgoing transition, the EBD is implying that this event is of interest to the corresponding state. Alternatively, not including an event in the set of transition antecedents from a given state, implies that the event is *screened* from this part of the behaviour, and deferred until it can be used. The analyst must understand this feature, and specify precisely when events are acceptable, screened, or discarded.

From the perspective of the meta-model, the EBD either: (1) initiates a transition on its own because it has found one whose antecedent is satisfied without an incoming receive event requirement, or (2) interacts with the EI. When interacting with the EI, the EBD sends the EI a meta-message which contains: (1) the events of the satisfiable transitions (*satisfiable event set*), (2) the EI directives of each event, and (3) the EI directive for the satisfiable event set. The default for individual EI directives is to receive each unique event FIFO, while for the satisfiable event set, the default is to make a fair choice from the elements of this set with an infinite timeout period (i.e., wait forever).

Assuming that the EBD eventually receives an event from the EI: this event is used to decide which one of the set of satisfiable transitions is fully satisfied (this might require the evaluation of conditions based on message data). Once all transition operations are performed, the transition consequence is processed. If the

transition consequence invokes the sending of events, the EBD sends the EI a meta-message containing these events, their *individual* directives, and the *group* directive. When the EBD and EI have completed all necessary interactions to send the events, the transition is complete and the EBD enters the destination state.

3.2.3 EBD Semantics

Using the meta-model for EBD specifications, the semantics of the communications model are defined. This section defines the semantics of: (1) the meta-EBDs (i.e., the semantics of the application behaviour component of an EBD specification), and (2) the EIs (i.e., the semantics of the communications defined by an EBD specification).

3.2.3.1 Meta-EBD Semantics

In previous discussions, the scenario interactions of the EBDs and EIs have been described. This section assigns names to the meta-messages, and uses them to build an EBD model of an EBD (i.e., a meta-model). As such, the semantics of EBDs are defined.

RequestNext(SET, SET_DIRECTIVES) — This meta-message is sent to the EI whenever the EBD has determined that it requires an incoming event to transition. The message data, *SET*, is a list of events and their individual directives; i.e., the 2-tuples, *re*:<*event*, *directives*>. These are the events which can initiate transitions from the current state, and for which the conditions are satisfied; i.e. *SET* is the satisfiable event set. *SET_DIRECTIVES* are the EI directives associated with the satisfiable event set. The EI is expected to reply with a message (including message data) which either matches one of these events or indicates a timeout. This event has been selected, in accordance to the EI directives, from the buffered (or yet to arrive) *system events*.¹¹

DeliverEBD(E) — This meta-message is sent from the EI to the EBD when a system event, *E*, or *timeout* has been selected as the correct response to the

¹¹System events are the application-specific events, as opposed to the events of the meta-model.

previous *Request.Next* event. *E* is one of: *br(m)* for broadcast events, *sync(m)* for synchronous events, or *async(m)* for asynchronous events, where *m* is the message and possibly message data, or *timeout*.

SendEvent(Es, Es_DIRECTIVE) — This meta-message is sent from an EBD to its EI when an outgoing system event (or list of events) is to be issued. Each element of the list, *Es*, is a 2-tuple: *se*:<event, directives>. *Es_DIRECTIVE* is the EI directive associated with the group of events, *Es*.

Complete — This meta-message is sent from the EI to the EBD when the outgoing system events have been successfully transmitted according to the semantics of the communication paradigm for each event (i.e., synchronous, asynchronous, or transient broadcast), and the EI directives.

Using these interactions the meta-EBD is specified in the EBD model as follows, and as shown in Figure 3.10.

1. Assuming that an EBD begins in some state, all specified *operations* for that state must be performed. This is shown as state 1 (*perform operation state*) in Figure 3.10, but actually represents both operation and null states.
2. The EBD must then determine if it can transition without receiving an incoming event; i.e., if there is an event which is *condition_only_satisfiable*.
3. If a transition can be taken (without an incoming event) then the transition operation of the corresponding transition must be performed. This is shown as a transition from state 1 to state 3 on Figure 3.10.
4. Otherwise, the EBD sends a *Request.Next(SET.SET_DIRECTIVES)* message to the EI (i.e., if an incoming event is required), and proceeds to wait for a *DeliverEBD(E)* message as shown by the transition from state 1 to the *waiting for event* state (state 2), and the transition from there to state 3 in Figure 3.10.
5. When the transition operations are complete, the events of the transition consequence are sent to the EI in the message *SendEvent(Es, Es_DIRECTIVE)*.

Figure 3.10: EBD of an EBD Meta-entity

For an empty transition consequence, an empty list is sent for *Es*, in which case, *Es.DIRECTIVE* is ignored by the EI.

6. After sending the *SendEvent(Es,Es_DIRECTIVE)* message, the EBD is in state 4 (*wait for transmission*) until it receives a *Complete* message from the EI. At this point, the EBD is able to begin its next operation state.

3.2.3.2 Entity Interface Semantics

The previous sections have given some detail on the EI's behaviour. This section introduces the meta-message sent between EIs, and uses this (and the messages going between the EBDs and EIs) to build an EBD model of an EI.

InteractOn(E,ATT) - This meta-message is sent from one EI to the EI coupled with the destination entity specified by the event. *E* is one of: *sync(m)*, *async(m)*, *br(m)*, or *match* which represent synchronous, asynchronous and broadcast message communications, or the sending of a *match* message to

indicate when entity synchronization (required for the synchronous communications paradigm) has occurred. *ATT* includes the attributes assigned by the sender EI; i.e., (*SID*, *SP*). For unspecified receivers, the EI must select an appropriate destination entity fairly, and define the EI variable, *RID*, which is used by the *set()* directive.

The EI meta-entity starts in an initial state where all of its buffers are empty and no interactions have occurred with either its EBD or any other EI. This is shown in Figure 3.11 as state 1 (*init*). This figure is referenced throughout the description of the EI event model, and is supported by Table A.1 which describes each state in detail, Tables A.2 and A.3 which elaborate the transition labels of Figure 3.11, and Tables A.4 and A.5 describe the transition operations introduced in Tables A.2 and A.3, including those where the EI directives are interpreted and applied. Briefly, in state 2 *wait event*, the EI is waiting the arrival of a system event which matches one of the events in the current satisfiable event set. In state 3 *wait match*, all EI buffers are empty, and the EI is awaiting another EI for acknowledgment on a synchronous interaction. State 5, *wait match or request* is similar to state 3 except that the EI buffers are not empty. In state 4, *wait request*, the EI buffers are building, and there is no outstanding receive request from the EBD.

STATE 1 —

The *init* state is a null state where the EI waits for interactions with the EBD or any other EI. There are three input events which can trigger a transition from this state:

1. *EBD + RequestNext(SET, SET_DIRECTIVES)* — This event causes the EI to transition to another null state (state 2, *wait event*) where it waits for a system event which matches one of the events in *SET*. This transition is labelled as **ClearRequest** on Figure 3.11.
2. *EBD + SendEvent(Es, Es_DIRECTIVE)* — The result of this event depends on the contents of *Es* and *Es_DIRECTIVE*.
 - (a) If *Es* is an empty list, then the EI transitions along a self-loop which sends a *Complete* message back to the EBD. This is shown as the

Figure 3.11: EBD for the EI

SendNull transition from state 1.

- (b) If Es is not empty, then the *send_in_progress* boolean is set to true, and Es and $Es_DIRECTIVE$ are used to define an algorithm (*SendSteps(i)*) for sending the events in the list Es (see Table A.4). This is shown as the **SendRequest** transition.
- (c) As long as the *send_in_progress* boolean is true, one event send (the next according to the *SendSteps(i)* algorithm shown in Table A.2, and defined in Table A.4) is carried out causing the EI to issue an *InteractOn(E, ATT)* message to the appropriate EI(s). A synchronous event send causes the EI to transition to state 3 (*wait match*) to wait for synchronization with the receiving EBD. As a result, any event sends remaining in the *SendSteps(i)* algorithm are delayed by the requirement for synchronization. The transitions: **BrInProgress**, **AsyncInProgress**, and **SyncInProgress** represent the event sends.
- (d) When the *SendSteps(i)* algorithm is completed, the *Complete* message is returned to the EBD, and the *send_in_progress* boolean is set to false. This is achieved by the **SendComplete** transition.

3. $\neg EI + InteractOn(E, ATT)$ — From the initial state there are two transitions which can occur when an event is received from any EI.

- (a) If the event E is a broadcast event, then the self-loop transition **Br** is taken. This transition performs no operations and has no consequence.
- (b) If the event is a synchronous or an asynchronous event then it is buffered as described in Table A.5. The EI transitions to state 4 *wait request* to indicate that its buffers are no longer empty. This transition is given the label **Buffer**.

STATE 2 —

In state 2 either the buffers are empty, or it is known that none of the buffered events match the events in the currently outstanding *satisfiable event set*. It is possible to timeout in this state, and the timeout parameter is exactly that which

was received in the *RequestNext(SET.SET_DIRECTIVES)* sent by the EBD. That is, the EI sends a *DeliverEBD(timeout)* message to the EBD if the time indicated in the original *SET_DIRECTIVES* parameter has elapsed. If the buffers are empty after the timeout, then the **Timeout** transition takes place and the EI returns to the *init* state. If the buffers are not empty, then the **TimeoutNotEmptyQ** transition is taken which puts the EI in the state indicating non-empty buffers; i.e., state 4. The only event which can be received by the EI in this state (state 2) is a $_EI + InteractOn(E, ATT)$ event. This event controls three different transitions:

1. If the event received does not match an event in the satisfiable event set; then:
 - (1) if it was a transient broadcast event, then the event is discarded, and (2) if it was a synchronous or asynchronous event, then the event is buffered. This is a self-loop transition labelled **NoMatch**.
2. If the received event matches one of the satisfiable event set, and the buffers are empty, then the **MatchEmptyQ** transition is taken and the EI returns to the *init* state. During this transition, the EBD is sent a *DeliverEBD(E)* message; and, if this has resulted in a synchronous event interaction, then a *SID – InteractOn(match, (ID, medium))* event is sent by the EI, where *ID* is the identifier of this EI, and *SID* is an attribute of the received event.
3. If the received event matches one of the satisfiable event set, and the queues are not empty, then the **MatchNotEmptyQ** transition is taken and the EI moves into the *wait request* state. During this transition, the EBD is sent a *DeliverEBD(E)* message; and, if this has resulted in a synchronous event interaction, then the EI sends a *SID – InteractOn(match, (ID, medium))* event.

STATE 3 —

At state 3 (*wait match* where EI buffers are empty), it is possible for the EI to receive any $_EI + InteractOn(E, ATT)$ event.

1. If the event *E* is a transient broadcast event, then the self-loop transition **Br** is taken. This label has been used again because it is defined by the same transition antecedent, operation, and consequence.

2. If the event E is a *match* message, then the **MatchReturn** transition is taken back to the *init* state.
3. If the event E is a synchronous or asynchronous event, then the **Buffer** transition is used to move to state 5 *wait match or request*. In state 5, the buffers are not empty.

STATE 5 —

State 5 is similar to state 3. A **MatchReturn** transition moves the EI from state 5 to state 4 (instead of 3 to 1). A **Br** or **Buffer** transition self-loops at state 5. Therefore state 5 is equivalent to state 3 except that buffered events exist in state 5 where none exist in state 3.

STATE 4 —

State 4 (*wait request*) is the non-empty buffer version of state 1, and it handles incoming events as follows:

1. $EBD + RequestNext(SET, SET_DIRECTIVES)$ — This event causes the EI to examine its buffers for a system event which matches one in the satisfiable set. The algorithm for examining the buffers is defined by $SET_DIRECTIVES$ (for the collection of buffers), and by the directives associated with each event. It is carried out by the $check_all_for_match(E)$ condition function defined in Table A.5. By default, the EI uses a “fair” selection policy for the buffers defined by the satisfiable event set, and takes the earliest arrival from that buffer.
 - (a) If none of the buffered events can be matched to an event in the satisfiable event set then, since all buffers have been examined, it is now only necessary to test newly arriving events for a match. Therefore the **RequestNoMatch** transition is taken to move the EI into state 2 where the first match, yet to arrive, is returned to the EBD.
 - (b) If a match (or set of matches) is found, then one is chosen to be sent to the EBD in a $DeliverEBD(E)$ message by $check_all_for_match(E)$. If this is a synchronous event, then $SID-InteractOn(match, (ID, medium))$

is sent. If the buffers are now empty, then the **RequestMatchEmptyQ** transition is taken to return the EI to state 1 (*init*). Otherwise, the self-loop transition, **RequestMatchNotEmptyQ**, is taken.

2. $EBD + SendEvent(Es, Es_DIRECTIVE)$ — The result of this event depends on the contents of *Es* and *Es_DIRECTIVE*, and the same types of transitions are taken here as in state 1; i.e., **SendNull**, **SendRequest**, **BrInProgress**, **AsyncInProgress**, **SyncInProgress**, and **SendComplete**. **SyncInProgress** moves the EI into state 5, since this is the state where waiting for synchronization occurs when the buffers are non-empty.
3. $_EI + InteractOn(E, ATT)$ — From this state transient broadcast events are discarded, and therefore a self-loop is drawn for the **Br** transition (as in state 1). The **Buffer** transition, indicating that a synchronous or asynchronous event has been received is now a self-loop; i.e., where state 1 transitioned to state 4 on this event, state 4 remains in state 4 but applies the same transition operation and consequence.

3.2.4 Examples of Communicating Entity Specifications

This chapter has defined EBDs and their underlying communications model for use in the specification of interacting entities in a reactive system. This section presents seven example specifications.

Example 1 — This first example shows two interacting system entities that communicate with each other, and each with a different external entity (entities which are part of the reactive environment as opposed to the system under development). The internal entities are named *partner1* and *partner2* and they communicate with *env1* and *env2* respectively. The requirement for this example is to have each synchronize with the environment, perform some operation (a functional requirement), and then synchronize with each other, before returning to receive the next message from the environment. This specification is analogous to having process threads merge in Petri-nets. The result is

Figure 3.12: Example 1 - Communicating Entities

shown in Figure 3.12. Each *partner* entity is initially in a null state waiting to receive a message from the environment. On receipt of such a message, each transitions into an operation state where the functional requirement is satisfied. The two entities make their next transitions when they are able to synchronize on the *merge* message. Each performs more processing in *phase2* states before returning to their initial states.

Example 2 — The next example illustrates how to specify an Ada-like rendezvous using EBDs. In this example there is a *user* entity and a *display* entity. The *display* entity is a server which starts in a null state *wait request* as shown in Figure 3.13. To transition out of this state, the *display* must receive a *request(data)* message from *user*. To show that these two entities rendezvous, *user* is placed in a null state *wait* while the *display* entity performs the *copy* operation. When the *copy* is complete, so is the rendezvous, therefore the two entities are shown to synchronize at this point, while each transitions to a new state. The *user* returns to its initial state, and *display* does some post-processing operation before returning to its initial state.

Example 3 — The third example illustrates how to specify a blocking send, and non-blocking receive communication between two entities. The sender entity is quite simple, and is named *sender* in Figure 3.14. This entity is initially in

Figure 3.13: Example 2 – An Ada-like Rendezvous

Figure 3.14: Example 3 – Blocking Send and Non-blocking Receive

an operation state. When the operation is complete it must send some *result* to *receiver* in the message *msg*. A synchronous send is a blocking send. On the other side, *receiver* starts in the null state *ready*. From here, *receiver* can transition to the state *process msg* or *background*. If a *msg* message is available, then the first transition is taken, otherwise the zero valued timeout parameter causes the latter transition to be taken. Both of these destination states represent operation states which are left when the operations are complete.

Figure 3.15: Example 4 - Binary Semaphore Interaction

Figure 3.16: Example 5 - Interrupt Servicing

Example 4 — Figure 3.15 illustrates a binary semaphore type of interaction using the *reset* directive. In the example, when the first *semaphore* message arrives, the entity transitions from state *Wait* to state *Process*. While in the *Process* state, *semaphore* events can arrive from *env*. When the entity has returned to the *Wait* state, if any number of *semaphore* events have arrived in the interim, then the transition to the *Process* state is taken again, and the *semaphore* events are cleared; i.e., the EI buffer for the *semaphore* event is emptied.

Example 5 — To describe event polling, a timeout transition is used as shown in Figure 3.16. An interrupt is specified by dividing the operation state into an indexable set of indivisible instructions. The *pc* variable indexes into the operation, and is advanced if the timeout occurs before the arrival of the

Figure 3.17: Example 6 – Illustration of the *order* and *priority* directives

polled_event. This is analogous to a program counter for an operation which is divided into non-divisible, non-interruptible segments. The counter is incremented prior to the execution of each non-divisible operation element, and as such, it is available when the interrupt service routine completes.

Example 6 — Figure 3.17 illustrates the *priority()*, *order()*, and *set()* directives. *sender1* and *sender2* are similar entities which request a service of the *server* entity after an initial input from an external *_user1* or *super_user* entity. *Sender1* represents the handler for any external user, where the user's identity is used to define the priority of the service request; i.e., *id_prior* is determined

in *Relate Id to Priority* using the *id* data of the *start* message. *Sender2* is a handler for a super-user: one with high priority over the service. The *server* entity accepts the *service* requests from *sender1* and *sender2* using an unspecified sender descriptor (*_sender*), and orders the acceptance of these events according to the priority levels assigned by the senders. The *sender_id* variable is used to save the identifier of the sender of the received message, and used to return the *complete* message.

The *server* entity also accepts events from *sender3*. These events are of a lower priority (*low*), than those from *sender1* and *sender2* (*high*). The events in the satisfiable event set of the *Wait* state for *server* are ordered according to these priority levels; i.e., the *service* messages are accepted over *request* messages, with the highest priority *service* message (as defined by its sender), taken first.

Example 7 – The Readers/Writers Problem — The EBD specification for the Readers/Writers problem is introduced in this example. The original informal specification, and the EBD specification are shown together in Figure 3.18. A discussion on how this specification is developed from the original informal description is given in [16]. For this discussion, it is sufficient to state that readers and writers are identified as the system entities, and their interactions can be described using asynchronous communications as follows. The readers and writers eventually transition from their *init* states, sending a request to access the resource to the scheduler entity, and subsequently enter a null state to wait for a grant acknowledgment. On receipt of the grant message from the scheduler, readers begin the reading process while writers write. When these operations are complete each sends an appropriate notification indicating that the resource is being released. The scheduler entity starts in its *init* state defining a system with no active readers or writers (the *number of readers* auxiliary counter variable *C* has the value 0). Permission to access the resource is granted to readers if no outstanding write requests exist (scheduler states 1 and 2). If a writer requests access to the resource and the scheduler is in state 1, then permission is granted. In either scheduler states 1 and 2 the selection of a read request over a write request is made fairly such that

2

2511-3 1x4 PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

every request is eventually acknowledged (e.g., a FIFO acceptance of requests is considered fair). When a write request is accepted by the scheduler in state 2, a transition is made to scheduler state 3 where the system is waiting for all active readers to relinquish the resource before granting write access permission to the writer. The reader and writer requests are accepted with unspecified senders, and the *set()* EI directive is used to save the most recently acknowledged reader or writer identifier (in the EI variables: *r* and *w*) such that the grant acknowledgments can be returned appropriately. The counter, *C*, cannot take on values less than 1 while in states 2 or 3, therefore the conditions shown cover all possible values. This specification is used throughout the thesis to illustrate all aspects of the development process.

3.2.5 Related Research

This section discusses work found using concepts similar to the EI. Related work on reactive system specification is given in Chapter 2.

The concept of the EI is similar to Minsky and Rozenshtein's *Laws*. Laws define the protocols for interacting objects in an object oriented system [101, 81], or a distributed system [82]. The protocol defines the communication semantics, the precedence relations of events, and the interconnections of entities: i.e., which entities can communicate with which events, and how events are forwarded from one entity to another (e.g., to define super-classes in an object-oriented system). The application (the object) communicates with the Law via two-way asynchronous message passing (however, no queuing is required in an OO implementation). The law passes the selected, appropriate messages, via *deliver* events to the application where the functional requirements are carried out. Conversely, any required message passing to other objects go through the Law, and are thus subject to its rules. The Law is globally defined, but can deal with individual objects using attributes and other stored parameters.

EBDs are a specification technique which define the abstract behaviour of a system, of which the communications semantics are managed by a meta-level EI through a client/server interface. Laws are an implementation tool which define

Informal Specification:

A data structure in primary or secondary memory is accessed by a number of readers and writers. In general, reading and writing can occur at any time, although only one writer at a time should be allowed to write. While writing, all other readers and writers must be denied access to the memory, however, when there are no writers, any number of readers are allowed.

EBD Specification:

Figure 3.18: EBD Specification for the Readers and Writers Problem

and manage the communication semantics for application behaviours. The law is written in Prolog and used at run-time to maintain a consistent, and hopefully, correct protocol for the object interactions. The objects (the application behaviours) run concurrent to the Law, responding to *deliver* events sent from the Law, and issuing *send* events to the Law for transmission to another object. The objects are input enabled, and the precedence relations are defined as a protocol in the Law.

Unlike EBDs there is no notion of screening and queueing incoming events for the application; each is dealt with as it arrives, and out of order events are managed like *unspecified receptions* are in the ISO protocol model.

Other parallel object-oriented languages (e.g., PROCOL [114], and those referenced by Minsky and Rozenshtein) have constructs similar to laws to define the communication semantics and precedence relations of messages. Unlike laws, these constructs are local to each object (whereas a law is a global description: i.e., applicable to all objects unless constrained by attribute values).

In [104], channels are included explicitly in their process modelling technique which is used to specify distributed processes with asynchronous communications. The inclusion of channels allows them to specify the channel based on a *channel grammar* without affecting the specified behaviours of the senders and receivers. Therefore, to some degree, the communication semantics are managed by the definition of the channel grammar, however, they are constrained to the asynchronous message passing. Again, this approach is not used to specify system behaviour, but rather to model a network of distributed processes so as to study and implement systems under a variety of channel types (e.g., where some messages can overtake others in a channel). The technique deals only with asynchronous messages, and does not discuss the behaviours of the processes in the system. The process model is similar to the ISO model with processes and channels, however, the channels are not restricted to simplex FIFO connections between processes.

3.3 Methods for Developing EBD Models

This section discusses methods for developing EBD models from large, informal, concurrent system specifications. The work here, describes typical approaches in terms of EBDs and *designing with design fragments*. This section is divided into two main topics: (1) decomposing large systems into more manageable *subsystems* related by interface specifications, and (2) decomposing subsystems (or small systems) into entities; and, from there, defining the EBD models.

3.3.1 Analysing and Decomposing Large Systems

Large systems present interesting issues to any design process. The increased functionality and multiprocessor environments complicate the analysis and system development efforts. One common approach to managing this problem is to decompose a system into subsystems based on the functional requirements, and define interfaces to which the behaviours of the subsystems are hence constrained. For example, the ISO layered model for distributed communications, separates this system into functional layers which interact through *service access points* assigned to the lower levels, and used by the layer immediately above [124]. As such, the behaviour at any given level is of a more manageable size. A similar approach is suggested for EBD specification and design.

Because the EBD structure and the design structure are consistent (i.e., one or more entities in the specification are allocated wholly to actors in the design), the decomposition of a system into subsystems is easiest performed at the specification stage; and, as such, provides a foundation for the independent design of subsystems. The key to this modelling technique is *collapsed behaviours* (cref. Page 61) which are behaviour descriptions encapsulated within an EBD state. The behaviours at subsystem interfaces are collapsed into states, and used as *black boxes* during the specification of the subsystem entities. Therefore, it is critical to decompose the system into subsystems, and define abstractions (labels) for the collapsed interface behaviours, before the subsystem behaviours are generated.

At the design phase, these collapsed behaviours must: (1) exhibit procedural semantics relative to the subsystem entities expressed in terms of these behaviours, or (2) be expanded to show all incoming events before design of the subsystem is started. (Some design fragments are contingent on patterns with no incoming events.) It is possible to define a *peer* entity which buffers the interface behaviour from the subsystem entities, and thus provides a procedural view for the subsystem entities. The *peer* behaviour represents the collapsed behaviour which is initiated by a subsystem entity, and returns to the subsystem entity once the interface behaviour is complete.

For example, if the subsystems are communicating across a network supported

by remote procedure calls (RPC), then the collapsed behaviour on the caller side includes the call and return semantics of the RPC. (RPC semantics could also be expressed directly using an EBD wait state, and designed using an *RPC* design fragment.) On the receiver side, either: (1) the call accept, perform operation, and call return are expressed by a state reached by receiving the incoming call and left when sending the return, or (2) a collapsed behaviour which calls a *peer* entity to execute the receive/operate/return semantics within a procedural format.

This section presents a typical (assumed) process for developing large systems integrated with EBD modelling and designing with design fragments. Although the process is iterative, it is presented as a series of steps.

Step 1. – The first step to developing large system software is to analyse (read and absorb) the informal requirements along with any existing documentation from prototypes or similar systems, interface documents, and other literature as necessary [86].

Step 2. – The next step is to define the system level entities (cref. §2.2.1) and present a *context* diagram for the system [59].

Step 3. – The third step is to begin to analyse the functional and behavioural requirements of the system entities. This step adds detail to step 1 by viewing the system requirements in terms of entities; i.e., what each entity must accomplish, and the dependencies (and interactions) between entities which enable them to achieve their goals.

Step 4. – Here, the development environment is considered. Generally the designers are working with an embedded system of N processors, where N is some integer, possibly shared memory, and a particular development language (for the bulk of the system code). These target domain issues have an effect on the design process. If, for example, the environment supports a distributed operating system, giving transparent message passing primitives across processor boards, then the development could move directly to the EBD modelling stage from here (Step 9). In this case, the interface procedures are not specialized, but rather expressed using simple EBD patterns such as a single synchronous interaction, or RPC semantics.

Therefore, design with design fragments can be performed directly, and the system behaviour description is less constrained; i.e., the behaviour is not expressed strictly because of limitations defined by the development environment. Here, subsystem decomposition is not performed. The system behaviour is considered as a whole.

On the other hand, each processor board will be subject to interfaces defined during the system design process as discussed in steps 5 through 8. Subsystems are typically allocated to a single processor, although one or more may be assigned to the same processor. In any case, the development and design process is simplified through a *divide and conquer* approach where subsystems are defined and managed independently given a set of interface procedures on which to build the subsystem behaviours as discussed below.

Step 5. – At this stage the design team has a reasonable understanding of the system entities and their functional requirements, some idea of the required entity interactions, and an appreciation for the development environment. As such, loading estimates, external coupling, functional cohesion etc. (i.e., experience and some creativity) are used to allocate the system entities (or functionality) to separate processors. This allocation defines the *subsystems* which are to be modelled by EBDs.

Step 6. – Given a suggested decomposition, the developers are now able to begin analysis of the *threads* (also called scenarios or slices [115]) of the system [25, 26]. That is, given a particular external stimulus, the flow of control through entities, possibly functions, and across processor boards can be defined. This analysis leads to an understanding of the inter-board communications; and, if provided, the shared memory requirements. This is the *process abstraction* alluded to by Booch [14] (cref. § 2.2.1); i.e., the processors and their functionality are placed within a useful model for satisfying the overall requirements. This analysis can lead to *peer* entities which hide the details of a inter-processor communications from the subsystems.

Step 7. – The developers are now responsible for committing to a means of distributed interaction which satisfies the requirements determined in Step 6. For example, remote procedure calls might be used, shared memory, interrupt signals, distributed semaphores, or some combination of these or other distributed interac-

Figure 3.19: Interfacing to Distributed Entities in a Large Design

tion techniques. As such, an interface is developed between the subsystems and the “engineered” entity responsible for satisfying this mechanism of distributed interaction; i.e., the peer. This layering is illustrated in Figure 3.19. The figure shows how the peer entity is layered on top of the hardware or distributed operating system to provide the mechanisms with which subsystems communicate. These mechanisms are the collapsed behaviours used in the EBD models of the subsystems. The subsystems are shown as being layered on top of the peer entities. When peer entities are not needed (e.g., because the subsystems behaviours are expressed directly using remote procedure calls), the subsystems are layered directly on top of the distributed operating system; i.e., the *peer* layer is removed.

Because the peer is being defined during the specification process as a general means of interaction between processors, it may be difficult to commit to low level details (i.e., specific event transfers across these interfaces) at this stage in the development.

Step 8. – Before starting the EBD modelling for the processors, it is important to define the services offered to the subsystems by the peer entity. Although the specific interactions may not yet be defined, it is important to develop a service interface between the subsystems and the *peer*. These services will establish a logical external interface for the subsystems in the form of collapsed behaviours which once named,

can be used consistently within the EBDs; e.g., *read_shared_data*, *write_shared_data*, *send_event*, and *receive_event*). This allows the EBD modelling of the subsystems to continue in parallel with the modelling of the peer.

Step 9. – The developers are now ready to begin EBD modelling of the system (from step 3), or of the subsystems (including the peer entity). Here entity, and thread analysis are again used to develop the EBD models of subsystems. An approach similar to Hatley and Pirbhai's method [59] is recommended [16, 55].

3.3.2 Developing EBD Models

As suggested above, Hatley and Pirbhai's method for developing real-time systems is recommended for developing EBD models of a subsystem [59, 55] which is structured using entity analysis. A discussion which is specific to EBD modelling is given in [16].

3.4 Analysis of EBD Specifications

Reachability analysis is a transition-based validation technique for verifying the entity, (called a process in this discussion), interactions of systems defined by a set of communicating finite state machines (FSMs). The result of this verification procedure is a tree of global system states which: (1) define the currently active process state of each process in the system, (2) define the contents of the communication channels, and (3) are related (parent node to child node) by the procedure of affecting the parent state with a single *perturbation* [121]. The definition of perturbation is dependent on the communications model and the communication semantics on which the technique is fitted and applied, but it basically represents the result of a single interaction on the current system state. The possible interactions are defined by the contents of the communication channels for the parent state and the semantics of the communication paradigms. The *events* which are present in these channels are consumed and generated by processes when an FSM transition is taken. Several unique perturbations can be applied to a single parent node thus giving a

one-to-many parent-child relationship; i.e., a tree. The algorithm was originally defined in [121] for the process model of communicating processes specifying an ISO communications protocol [118, 120]. From it many other transition-based techniques have been defined and used to analyse the properties of communicating processes in a variety of communication models when the process interactions are defined by communicating FSMs. EBD modelling supports this type of analysis.

The methods are not usually designed to manage examples where any number of a *pool* of clients may arrive for service at any time.¹² For example, the Readers/Writers problem introduced in Section 3.2.4 contains pools of reader and writer clients. Verifying systems with unbounded pools of process behaviours is normally approached by reducing the problem to one which is fixed in size; i.e., where there are exactly k clients, and k is some integer. The validation technique is applied to the static configuration to yield a set of composite system states for the represented case. It can, however, be difficult to ensure that the system representation is large enough to exercise all possible execution paths of the combined system behaviour. This is because some of the transitions of the individual process FSMs are only taken as a result of having more process pool instances in the validation configuration. On the other hand, if the number of processes in the pool is too large, then the reachability tree will be unnecessarily large because each pool instance is examined as a unique individual in the set of communicating system processes. Therefore, each added pool instance adds one more process to the composite reachability state descriptions, as well as an extra set of communicated events.

What is desired is a *generalized reachability tree* for systems that include pools of equivalent behaviours. The generalization should be representative of all process pool sizes, and offer the same level of temporal analysis as the non-generalized transition-based verification techniques. For example, Browne [17] uses induction to generalize a temporal logic statement which holds for one size of network to one which holds true for any size network. The technique is performed by finding

¹²The term *pool* is being used to define a group of processes with equivalent behaviours all of which can be specified by the same FSM. This is often also referred to as a set of multiple instances of a behaviour.

a bisimulation between networks of different sizes, and then proving the relation through induction. It is a complex theory with a sound mathematical and logical basis which generates a *generalized temporal expression* of the system behaviour. Garg [46] examines the problem of unbounded process pools in distributed systems which are configured in specific symmetric topologies. Although there are several similarities between his work and what will be presented as in this section as a solution to the problem, there are also several distinct differences.

1. In Garg's paper each topology (star, bus, and ring) defines a new process model and a different procedure for analysing the problem. Our solution will look at only one process model which can represent any process configuration; it is not restricted to symmetric connections.
2. Our model will allow asynchronous process interactions as well as the synchronous communications offered by Garg.
3. Because of his representation, Garg loses the ability to comment on the levels of concurrency within a process pool, in particular, the distinction of mutually exclusive process states.
4. Our reachability procedure will terminate if a finite data space of the process FSMs is assumed; Garg's is not guaranteed to terminate.
5. For the same reason that Garg's procedure may not terminate, it is also, at times, less general; i.e., we are able to combine more states of the reachability tree and still produce a tree which allows for the same level of temporal analysis.
6. Using our approach, it is possible to determine the minimum size process pool required to find the general (for all sized pools) reachability tree: i.e., the starting size for Garg's inductive proofs.

This section will present a validation technique which is a hybrid of reachability analysis and regular expression syntax. The combination of these two concepts provides a means for validating systems of communicating processes where (1) the

interactions are defined by communicating finite state machines (EBDs), and (2) the systems include pools of the same behaviour descriptions. The approach starts with a given initial state and generates a complete interaction domain of the system by considering all ways in which the initial and subsequent states can be perturbed according to the specified process model. Regular expression syntax is used to express the representation (*zero*, *one*, *zero or more*, or *one or more*) of pool processes in a given process state and events communicated by the pool processes. The uniqueness of an individual pool instance is generalized into an expression which defines the state of the pool as a single component of a composite reachability state; i.e., the state of the pool is described by the active process states of the pool instances using regular expression syntax. Communicated events are generalized together to represent the outstanding events in a communication channel. The generalization of outstanding events is possible because of the semantics of the process model being analysed.

Together, the result is a reachability tree which describes the system execution paths in the general case (i.e., one which is representative of all possible configurations). Along with the general representation of each reachability state, it is possible to determine the minimum number of pool processes required to execute all paths of the combined system behaviour. This is the size of the system configuration from which induction-based approaches would start their proof that adding one more process to the pool size would not affect the resulting reachability tree. This *minimum pool size* can be used to apply other validation techniques with a fixed size system configuration with the confidence that all execution paths are reachable. The resulting tree will also provide the analyst with enough information to comment on the level of concurrency within a process pool. In particular, it will be apparent if and when only one process of a pool is allowed to be active in a given process state; this is a property similar to *boundedness* in Petri-nets [90].

Another result of the verification procedure is the capability to produce and analyse what Liu [73] refers to as *global behaviour expressions*. Liu generates similar expressions using a different transition-based validation technique called *graph merging*. His paper discusses how these expressions are used in analysing temporal

properties such as deadlock as well as timing properties such as critical path processing and missing deadlines. The approach inserts delay symbols for the processing involved in the process interactions which define the relative amount of processing involved in a given interaction, and allows execution paths to be analysed in terms of their timing properties. The results of our reachability analysis procedure could be extended to include such timing analysis.

This section starts with a brief review of the communications model used in Zafiropulo and West's [118, 120, 121] description of reachability analysis, compares it to EBD models, and describes the updates required to manage the EBD representation. The addition of regular syntax to manage process pools is discussed in Section 3.4.2 along with a detailed definition of the extended reachability process. Section 3.4.3 uses the Readers/Writers problem as an example to illustrate the process. A discussion of the general versus non-general case is given in Section 3.4.4 where the relationships are demonstrated and the analytical capabilities of the general result are proven informally. This section also assesses the properties of the reachability procedure such as termination, completeness, and information made available to the user which is in addition to the fundamental temporal analysis.

3.4.1 Representation Model

Zafiropulo and West's work focuses on the design and verification of communication protocols where process interactions are specified formally by finite state machines (FSMs). These descriptions are restricted to strongly connected digraphs where the nodes define states of the process (p-states) and the edges are transitions caused by incoming or outgoing events, or non-events. Inter-process communications are performed across simplex, perfect FIFO channels with one channel for each process-to-process connection. This model is shown in Figure 3.20 [118, 121, 120]. The verification procedure applied to this model (referred to as the *classical* model in this thesis) is *reachability analysis* which begins with an initial composite system state (s-state) (e.g., P1 in p-state 0 and P2 in p-state 0, and an empty set of communication channels). To generate a "next" s-state a p-state transition is taken by one of the interacting processes. All possible transitions of the process FSMs

Figure 3.20: Classical Reachability Analysis Process Model Example

are explored; each generating a different *child* s-state of the initial s-state. This is applied recursively to each s-state which has not already been explored, thus all possible interaction sequences are generated, and the process terminates when there are no unexplored p-state transitions in the process FSMs.

The verification technique being presented in this thesis is based on reachability analysis, however, the EBD model is being analysed. EBD states and transitions, like the classical model, are assumed to take some finite amount of time to occur, and therefore each process is capable of executing a sequence of state/transition steps. During the reachability procedure, a new s-state is generated as a result of a one process taking a transition from one p-state to another. Because the process model is queuing events, simultaneous process transitions would have to be on unrelated event communications and therefore two steps in the reachability procedure would result in the same final s-state. For a process to transition, it is not necessary that it receives an event. Process states are assumed to take time, and therefore they can represent ongoing operations or execution of some functional requirement. Completion of such a requirement can cause a transition, and these are shown as events with only the message component (no process identifier or communication

paradigm indicator).¹³ In the classical model these are shown as *i* transitions which represent *internal events*.

When a state machine is ready to take a transition from one p-state to another, there are two possibilities. First, the machine might be able to make a transition on an internal event. If this is the case, the transition is taken and any outgoing events resulting from the transition are placed in the correct destination queues. The second way that a state machine is able to transition, is to select from the queues an event which will cause a transition from the current p-state. When reachability analysis is applied to this model, each s-state of the reachability tree will generate a new s-state for as many transitions as there are events existing and applicable for the current p-states. Any interleaving of synchronous or asynchronous events is possible, assuming that these events are acceptable in the current p-state.

Another feature of EBD models which will be used in this discussion are *auxiliary variables*. The state machines will be supported by counters to describe the number of *active* processes within a process pool. Unbounded pools with unbounded resources do not make interesting problems; there is always enough resource to go around. Unbounded pools with bounded resources are of interest. This section looks at resources which can be bounded by integer values; e.g., 1 file, or 6 tokens. Resource bounds appear in the process transitions as conditions. For example, from a given p-state, the same incoming event will cause a transition to one p-state if the count *C* equals the bound *max*, and will transition to a different p-state if $C < \text{max}$. In the first case, the condition $[C = \text{max}]$ is added to the description of the transition, and in the second the condition $[C < \text{max}]$ is added to the transition. The general format for writing transitions is:

incoming event[*condition list*]/*auxiliary variable updates*/*outgoing event list*

As an example, the specification for the Readers/Writers problem is given in Figure 3.21.¹⁴ Readers and writers define two separate process pools which make re-

¹³Note, that this is a different notation from that defined in Section 3.2.

¹⁴This version is slightly different from that given in Figure 3.18.

EBD Specification:

Figure 3.21: Readers/Writers Specification for Verification

quests to a scheduler process for permission to access the shared resource (*new* and *spon_1* are internal events). Permission is granted to readers if no outstanding write requests exist (scheduler p-states 1 and 2). If a writer requests access to the resource and the scheduler is in its own p-state 1, then permission is granted. If the scheduler is in p-state 2, then a transition is made to scheduler p-state 3 where the system waits for all active readers to release the resource before the writer is granted access.

Auxiliary variables, used as counters to describe the state of an unbounded process pool, will provide the reachability procedure with the information needed to evaluate the minimum number of process instances required to execute all paths of the combined behaviour (see Section 3.4.2). For example, in the Readers/Writers problem, a count of the number of active readers is maintained. Because the interest of this work lies in the understanding of unbounded pool processes, only pool counter auxiliary variables are discussed (although others are allowed, and would not change the theory). To simplify matters, conditions based on auxiliary variables are limited to: (1) equals, (2) greater than, and (3) less than comparisons between a pool counter and some other value. For example, in the Readers/Writers problem a notification of reading complete (event $(\neg r + rf)$) is managed one way if only one reader ($C = 1$) is active, and another if more than one ($C > 1$) readers are active as shown in the scheduler EBD of Figure 3.21. Updates to the various pool counters of a process are expressed in the form $C' = C \text{ op } N$, where op is one of $\{+, -\}$; N is a constant and C represents a pool counter; e.g., in the Readers/Writers example, all updates are either $C' = C + 1$ or $C' = C - 1$ to count one more or one less active reader respectively.

3.4.2 Reachability Analysis with Process Pools

Regular expression syntax is used to describe the p-states of, and events sent to or from a process pool. The s-state space is described by a set of p-states followed by lists of events which have been sent to the corresponding process, and the possible range of values of any auxiliary, pool counter variables. For example, given the EBDs of Figure 3.21 for the Readers/Writers problem, the initial reachability s-state is described by:

s-state 0:

scheduler : $[1] : []$

writers : $[1^*] : [new^*]$

readers : $[1^*] : [new^*]$

reader count variable range : $[0, 0]$

This defines the s-state where the scheduler is *idle*, and there are any number, including zero, of readers and writers in their *idle* p-states (1^*), with the possibility that any number, including zero, of readers or writers may begin to request access to the scheduled resource (*new*¹⁵). It also shows that the range of possible values for the reader count in this s-state is equivalent to exactly zero. The meaning of a value range is discussed later.

For process pools, the s-state syntax has been expanded to show different pool instances in the same or different p-states for a given composite s-state.

n — is used to say that *exactly one*, $\{1\}$,¹⁵ process is in p-state n .

n^* — is used to say that *zero or more*, $\{*\}$, processes may be in p-state n .

n^+ — is used to say that *one or more*, $\{+\}$, processes are in p-state n .

The same syntax applies to the events in the event lists as was seen in the example initial s-state where the writer event list contained *zero or more new* events.

The method of solving the reachability tree with this procedure involves moving one transition at a time as with the classical model. The steps involved in generating the new s-state in either case are:

1. The incoming event is removed from the corresponding communication channel. This establishes a *less one* relation between the parent and child reachability s-states with regards to this event. This relation is referenced in later discussions.
2. The process is taken out of the previous p-state and placed in the correct new p-state. Again a *less one* relation is established between the parent and child s-states, but this time with respect to the p-state which is being removed. At the same time an *add one* relation is established for the new p-state.
3. The outgoing events are added to the correct communications channels, giving an *add one* relation for the event, between the parent and child s-states.

¹⁵The parenthetical term, $\{1\}$, along with $\{0\}$, $\{*\}$, and $\{+\}$, will be used as short forms throughout this work for *exactly one*, *zero*, *zero or more*, and *one or more* respectively.

Rule Number	p-state or event representation in an s-state	operation	Next representation for p-state or event
1	n	<i>less one</i>	<i>empty set</i>
2	n^+	<i>less one</i>	n^+
3	n^+	<i>less one</i>	n^+
4	n	<i>add one</i>	n^+
5	n^+	<i>add one</i>	n^+
6	n^+	<i>add one</i>	n^+
7	<i>empty set</i>	<i>add one</i>	n

Table 3.2: Transformations on pool states and events

4. The auxiliary variables are updated accordingly.

When pool processes are being represented by regular syntax expressions, the *less one* and *add one* updates affect the expressions as shown in Table 3.2.

3.4.2.1 S-state Matching Rules

The most significant difference between the reachability analysis procedure of this thesis and the classical approach is in the rules used to match the newly generated s-state with an existing s-state. The matching rule of Zafropulo and West requires an exact match between the existing and generated s-states. This is illustrated in Figure 3.22 where, for example, s-state 2 generates one s-state which is equivalent to s-state 3 and a new s-state labelled as 4. In our method, a match exists when the s-states are the same, or the information in the two s-states can be *generalized* such that both s-states are still represented and no p-state or channel information is lost. Therefore, matching s-state descriptions contain all the same elements, but an element of one state can be *subsumed* by the same, more generally described element of the other. The generalization relations are outlined in the following sections.

Figure 3.22: Classical Reachability Analysis Matching Process

3.4.2.2 Exactly one {1} subsumed by one or more {+}

Any p-state showing *exactly one* process can be generalized by the p-state *one or more*. Similarly any single event can be generalized to represent *one or more* of the same. For example, s-state 1 and the newly generated s-state shown below would generalize to s-state 1'.

s-state 1:	generated s-state:	s-state 1'
$p0: [3]: [(-p1 + e1)]$	$p0: [3]: [(-p1 + e1)^+]$	$p0: [3]: [(-p1 + e1)^+]$
$p1: [1^-, 2]: [(p0 + e2)^*]$	$p1: [1^-, 2^+]: [(p0 + e2)^*]$	$p1: [1^-, 2^+]: [(p0 + e2)^*]$
$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$

This type of update can occur when a process (e.g., $p1$) receives an event ($(p0 + e2)$) which moves one of the $p1$ pool processes from p-state 1 to p-state 2, and consequently sends an outgoing event ($(p0 - e1)$). The information in the generated s-state ($(-p1 + e1)^+$ and 2^+) subsumes the information in the original s-state and appears in the resultant s-state 1'.

The label s-state 1' was used to indicate that the previous s-state 1 is no longer the most general representation, and that it is replaced. The reachability

procedure, given later in this section, indicates how such a change affects the other s-states.

There are two main situations where this generalization rule is applied:

1. An unbounded number of pool processes transition from one p-state (e.g., p-state 1) to another (e.g., p-state 2). At the same time, each might send an asynchronous event to some receiving process (e.g., this exact p-state transition is taken by reader processes in the Readers/Writers example). As a result of the first application of this transition, a single occurrence of p-state 1 and the sent event are seen in the s-state. Every subsequent, equivalent transition taken by other pool processes changes the s-state by increasing the number of processes in the p-state and the number of events in the receiver's queue by one. Since the pools are considered to be unbounded, this generalization rule permits a finite reachability tree (indicating the potential for an infinite number of processes in the *one or more* p-state) instead of an infinite tree.
2. The receiver of these pool process events accepts and acknowledges the events. This is done, for instance, by the scheduler process in the Readers/Writers example when the read requests ($_r+rr$) are acknowledged by grant read events ($r-gr$). This type of transition is common in cases where access to a shared resource is requested by the pool processes. If the resource is bounded, then it is also common for an auxiliary counter to be used to help with the specification as in the Readers/Writers example where the count is incremented with each grant read event. In general the first occurrence of this transition creates a new s-state with the single occurrence of the acknowledgment event. Similar s-states (which match according to this generalization rule) are generated when the receiver transitions along a self-loop. Just as the self-loop transition is used to reduce the size of the process finite state machine, this generalization rule will compress the reachability tree.

The generated s-states include the same p-state and event information, but have different *counter* values for the pool process states. The pool process counters qualify the $\{*\}$ or $\{+\}$ expressions. The variables are updated ac-

according to the behaviour descriptions until the most general expression for the given s-state is obtained. The lower bound of the counter range defines the minimum number of pool processes required to establish this s-state (when the upper bound is non-zero). The upper bound of the range defines the minimum number of processes of the pool required to define the generalized s-state.

For example, in an expanded (uncompressed) version of the reachability tree, a series of s-states might be developed where the only difference from one to the next is the value of an auxiliary counter (see Figure 3.23). With this matching rule, s-state 1 will generate the same s-state 2 as in the expanded version. When s-state 2 is examined, a new s-state is generated with all elements the same and the count value equal to 2 instead of 1. As counters are defined by their range of values (e.g., $[1, 2]$), all of the information found in these two states can now be represented by a single state. If the expanded version had 4 states, the compressed version still has only two states with a range of $[1, 2]$ as shown in Figure 3.23. The range stops increasing as soon as the s-state stops generalizing. For example, in the expanded case of Figure 3.23, the third and fourth s-states are exactly equivalent, therefore the third count was not needed to exercise all distinct behaviours from this s-state, nor to generate the fully generalized representation of the s-state.

This step causes the pool instances to lose their individuality; i.e., there is no way to distinguish between any two different instances if their existence is grouped into a single symbolic representation. However, the system is unaffected by non-specific labelling of pool instances. For example, switching the labels of two readers does not effect the execution flow. The combination of events $((rr1), (rr2))$ is equivalent to $((rr2), (rr1))$ in terms of the system execution, and therefore is no more informative than (rr^+) . To allow for this generalization, specifications are constrained such that processes receiving events from a pool are not dependent on message data to select a transition (see Section 3.4.4.1).

Although individuality is lost, the ability to recognize mutually exclusive process states is kept by recognizing the distinction between *exactly one* and *one or more*. In terms of generating new s-states, the uniqueness falls out of the conditional

Figure 3.23: Expanded versus Compressed Reachability Trees

transitions where conditions are based on the pool counters. When counters are used correctly, there is no loss in the ability to comment on the level of concurrency of the pool instances as a group; e.g., *zero* versus *exactly one* versus *one or more* versus *zero or more*.

3.4.2.3 One or more {+} subsumed by zero or more {*}

Any p-state or event with a *one or more* entry in the s-state can be generalized to *zero or more*; e.g.,

s-state 2:	generated s-state:	s-state 2'
p0: [3]: [(-p1 + e1)+]	p0: [3]: [(-p1 + e1)*]	p0: [3]: [(-p1 + e1)*]
p1: [1*, 2+]: [(p0 + e2)]	p1: [1*, 2+]: [(p0 + e2)+]	p1: [1*, 2+]: [(p0 + e2)+]
p1 count: [1, 2]	p1 count: [0, 1]	p1 count: [0, 2]

In this example, a self-loop transition on event $(\neg p1 + e1)$ by $p0$ (which according to Rule 2 changes $(\neg p1 + e1)^+$ to $(\neg p1 + e1)^*$) has not changed the p-state of $p0$ (it remains at 3) nor the elements of the event list of $p0$. The event $(p0 + e2)$ has been added to the s-state components for $p1$ by Rule 4. The same elements exist in the original s-state and the generated s-state. The $(\neg p1 + e1)^*$ event of the generated state subsumes the old $(\neg p1 + e1)^+$, as does $(p0 + e2)^+$ over $(p0 + e2)$ giving the matched s-state 2'.

The example shows the counter values decreasing; i.e., $[1, 2]$ became $[0, 1]$ for the generated state, and the full range in the final *generalized* state was shown as $[0, 2]$. This generalization rule is most beneficial for compressing the reachability tree when: (1) several pool processes are taking the same transition which removes the initial p-state from the state list and the consumed event from the event list of the s-state description, and (2) when pool processes are being "counted down". The previous rule (*one* subsumed by *one or more*) would keep two states when "counting up" pool processes: the zero state and the *one or more* state. When *one or more* is subsumed by *zero or more*, the counting down process is compressed into a single state with ranges starting at zero.

These states can be interesting if the $\{*\}$'s should really be specialized to strictly zero; i.e., the state can become over-generalized, allowing all combinations of all p-states and events. A state might, in fact, be representing two distinct cases: (1) the *zero* case, and (2) the *one or more* case. To determine if a separate zero-case state is needed, *zero-value invariants* are required. This is a necessary step for allowing the reachability tree to turn back onto the initial s-states of the global system description. The system execution is expected to reach stages where the paths loop back to s-states defined earlier. Without this step, the verification procedure will generate s-states which provide no useful information. These s-states are strictly self-looping, and usually imply that "all possible combinations of p-states and events are possible"; i.e., everything related to the pool processes can be *zero or more*. The information required to avoid these situations is in the zero value invariants, which provide an invariant definition for the system when a process pool count reaches zero. For the Readers/Writers example the following statements are always true

when there are no readers in the system.

1. The scheduler must be in scheduler p-states 1 or 4.
2. There are no outstanding $(s + gr)$ or *spon_l* events.
3. There are no readers in reader p-state 3.

For the zero value invariant this information is re-stated as shown in Figure 3.21; e.g.,

1. The scheduler must be in scheduler p-states 1 or 4.
2. The scheduler may be accepting $(_r + rr)$, $(_w + wr)$, or $(_w + wf)$ events.
3. Writers are in writer p-states 1, 2, or 3.
4. Writers may be accepting *new*, $(s + gw)$, or *spon_l* events.
5. Readers are in reader p-states 1 or 2.
6. Readers are accepting only *new* events.

The application of these invariants is illustrated in the next section where this step will allow the procedure to determine (for example) that when there are no readers actively using the resource, then there are no readers in their own p-state 3.

The invariant clauses are necessary statements if any real information is to be produced from the system by the procedure, however, it might be difficult to obtain these expressions. An automated reachability procedure does provide reasonably quick feedback on small systems from which it is possible to determine the zero value invariant. The over generalized states are those which cover all pool process states with *zero or more* active pool instances, and all pool process outgoing events are represented as *zero or more*. These states are typically found at the root of the generalized reachability tree, and are often the indicators needed to decide on the zero-value invariant specifications. This approach to defining invariants is somewhat similar to that taken by Dijkstra in [38] where regular expression syntax is used to define a process and channel expression for a ring topology of processes. Here, a

token is used to characterize the process with access rights and regular expression syntax is used to allow an unspecified number of equivalent processes to exist in the ring network. Dijkstra develops the invariant of the global behaviour from a simple known initial behaviour description by iteratively proving that the global behaviour expression is closed under known transitions.

3.4.2.4 Zero {0} subsumed by zero or more {*}

Any p-state or event with a *zero* entry can be generalized to *zero or more*, however this generalization rule will not be applied because it can lead to ambiguities in the matching process. For example, consider the sequence of s-states starting with **s-state 3**, and applying a transition which moves a **p1** pool process from p-state 1 to p-state 2 by consuming an *i1* event and generating a $(p0 - e1)$ event. The result is **s-state 4** as shown below.

s-state 3:	generated s-state:	s-state 4
$p0: [3]: []$	$p0: [3]: [(-p1 + e1)]$	$p0: [3]: [(-p1 + e1)]$
$p1: [1^-]: [i1^-]$	$p1: [1^-, 2]: [i1^-]$	$p1: [1^-, 2]: [i1^-]$
$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$

If this same transition is applicable to **s-state 4**, then it *self-generalizes* to **s-state 4'**.

s-state 4:	generated s-state:	s-state 4'
$p0: [3]: [(-p1 + e1)]$	$p0: [3]: [(-p1 + e1)^+]$	$p0: [3]: [(-p1 + e1)^+]$
$p1: [1^-, 2]: [i1^-]$	$p1: [1^-, 2^+]: [i1^-]$	$p1: [1^-, 2^+]: [i1^-]$
$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$

Finally, if **p0** now self-loops by consuming a $(-p1 + e1)$ event, then the result matches both **s-state 4'** and, when *zero* is subsumed by *zero or more*, **s-state 3**.

s-state 4':	generated s-state:
$p0: [3]: [(-p1 + e1)^+]$	$p0: [3]: [(-p1 + e1)^*]$
$p1: [1^-, 2^+]: [i1^-]$	$p1: [1^-, 2^*]: [i1^-]$
$p1 \text{ count}: [0, 0]$	$p1 \text{ count}: [0, 0]$

There are three choices for the match: (1) match the oldest s-state (e.g., **s-state 3**), (2) match the youngest s-state which is equivalent to ignoring the generalization rule and would create **s-state 4''**), or (3) matching both and collapsing the reachability tree to a single s-state representation. In the first case, the matching eliminates the potential to determine the p-state transition used in the original generation from the parent s-state to this s-state. For the second option this relation is defined by all elements of the s-state which are in the child and not in the parent, and vice versa, therefore there is no loss in continuity (For example, from **s-state 3** to **s-state 4''**, $(-p1 + e1)$, **p1** p-state 2 and internal event *i2* have been added. Therefore, it should be possible to determine which **p1** transition from p-state 1 to p-state 2 was applied.) The third option leads to the same ambiguity as the first as well as a potentially complex restructuring of the reachability tree if the generalizations are passed on to the child s-states, and other s-state-to-s-state relations are collapsed as a result. This is an interesting property of the generalization rule which is discussed again in Section 3.4.4.

3.4.2.5 Counter Ranges

Before describing the reachability procedure, there are several issues involving the process counters which must be considered. When there are self-loop transitions which generate the *count-up* and *count-down* s-states, there must be a way of deciding when to stop applying the transition, and move on to another. For example, if an infinite number of read requests can exist, then the scheduler can take its self-loop transition at p-state 2 infinitely often. The reachability procedure must, therefore decide how many times the transition is required before no new information about the s-state is obtained. In the Readers/Writers example, a range of $[1, 2]$ is needed to define (by 1) the lower bound of the counter (which defines the minimum number of readers required to reach this s-state), and (by 2) the upper bound of the counter when all paths leading to this s-state have been examined. However, a problem arises if there is a transition from this self-looping p-state to another p-state which requires a count value of 10, for example, for one of its own transitions (see Figure 3.24). If the $[1, 2]$ counter range is directly transferred to the

Figure 3.24: Counter Value Example

new s-state description which places the scheduler in p-state 3, then the condition $C' > 10$ (on the transition leading to p-state 5 in Figure 3.24) cannot be satisfied, and therefore this transition is never taken. These problems motivate the following notation for counter ranges:

- $[m, n]$ — This representation defines the specific range from integer value m to n .
- $[y, n^+]$ — In this case y is an appropriate lower limit representation and n^+ indicates that the counter values increase indefinitely, but that the s-state will reach a counter upper bound of $(n + 1)$ as a result of traversing all transitions leading to this s-state once. Therefore the $[1, 2]$ range required by the Readers/Writers example, is defined by $[1, 1^+]$.
- $[m^-, z]$ — Here the z represents any appropriate upper limit representation, and m^- indicates that the counter can take on any value down to zero. The counter value can decrease to as little as $(m - 1)$ as a result of one execution of each of the transitions leading to this s-state.

As a result of this notation, transitions result in the ranges shown in Table 3.3 where C is the counter variable name and R is a resource bound being compared to the current counter value. Rows 1 to 12 describe the transitions which generate new s-states, and rows 13 to 24 describe transitions which *self-generalize* the current

Figure 3.25: Multiple Conditioned Self-loops

s-state. This table looks at what happens to strict limits (m or n), or count-down and count-up limits (m^- , or n^+). It assumes: (1) the conditions for the transition can be satisfied by the existing current value range of the s-state, and (2) that self-generalization transition updates are applied only if the s-state is a generalizing match of its parent. Note that for an exact match, the counter range is left as found in the parent thus realizing that the general form can be established without this extra application of the transition. With respect to the 'OR'ed representations, the upper limits take the higher representation (where 3^+ is considered to be higher than 2^+), and the lower limits take the smaller representation.

The table shows that the self-generalizing transitions numbered 21a and 23b constrain one of the limits to a fixed value (R). Because transitions are applied to s-states sequentially, and overwrite the range values when a self-generalization occurs, it is important that these loop transitions be examined first, otherwise the ranges might be needlessly constrained. For example, Figure 3.25 illustrates a set of possible self-loop, self-generalizing transitions. The transition at p-state I with the condition $[C' < 3]$ will constrain the upper limit of the counter value range according to update 21a; i.e., giving a range with an upper bound of 3. The other, with condition $[C' > 2]$, leaves the range unconstrained on the upper limit according to update 20; i.e., the upper limit is 3^+ for this transition. The s-states generated as the children this s-state must be provided with the least constrained range (e.g., with the upper limit 3^+) to ensure that all subsequent potential transitions are satisfiable.

	transition	Lower limit		Upper Limit	
	condition/update	m^-	m	n	n^+
	s-state $\Rightarrow$ s-state				
1	/	m^-	m	n	n^+
2	$/ C = C + 1$	$(m + 1)^-$	$(m + 1)$	$(n + 1)$	$(n + 1)^+$
3	$/ C = C - 1$	$(m - 1)^-$	$(m - 1)$	$(n - 1)$	$(n - 1)^+$
4	$C = R/$	R	R	R	R
5	$C > R/$	$R + 1$	$R + 1$	n	n^+ OR R^+
6	$C < R/$	R^- OR m^-	m	$R - 1$	$R - 1$
7	$C = R/C = C + 1$	$R + 1$	$R + 1$	$R + 1$	$R + 1$
8	$C > R/C = C + 1$	$R + 2$	$R + 2$	$n + 1$	$(n + 1)^+$ OR $(R + 1)^+$
9	$C < R/C = C + 1$	$(R + 1)^-$ OR $(m + 1)^-$	$(m + 1)$	R	R
10	$C = R/C = C - 1$	$R - 1$	$R - 1$	$R - 1$	$R - 1$
11	$C > R/C = C - 1$	R	R	$(n - 1)$	$(n - 1)^+$ OR $(R - 1)^+$
12	$C < R/C = C - 1$	$(R - 1)^-$ OR m^-	$(m - 1)$	$R - 2$	$R - 2$
	self-generalizing				
13	/	m^-	m	n	n^+
14	$/ C = C + 1$	m^-	m	n^+	$(n + 1)^+$
15	$/ C = C - 1$	$(m - 1)^-$	m^-	n	n^+
16	$C = R/$	m^- OR $(R + 1)^-$	m	n	n^+ OR $(R - 1)^+$
17	$C > R/$	m^-	m	n	n^+ OR R^+
18	$C < R/$	R^- OR m^-	m	n	n^+
19	$C = R/C = C + 1$	$(R + 1)^-$ OR m^-	m	n OR $(n + 1)$	$(n + 1)^+$ OR R^+
20	$C > R/C = C + 1$	m^-	m	n^+	$(n + 1)^+$ OR $(R + 1)^+$
21a	$C < R/C = C + 1$	R^- OR m^-	m	$(R > n): R$	$(R > n): (n + 1)^+$
21b	$C < R/C = C + 1$	R^- OR m^-	m	$(R \leq n): n$	$(R \leq n): n^+$
22	$C = R/C = C - 1$	$(m - 1)^-$ OR R^-	m OR $(m - 1)$	n	n^+ OR $(R - 1)^+$
23a	$C > R/C = C - 1$	$(R \geq m): m^-$	$(R \geq m): m$	n	n^+ OR R^+
23b	$C > R/C = C - 1$	$(R < m): (m - 1)^-$	$(R < m): R$	n	n^+ OR R^+
24	$C < R/C = C - 1$	$(R - 1)^-$ OR $(m - 1)^-$	m^-	n	n^+

Table 3.3: Counter Range Updates for Given Transition Conditions and Updates

3.4.2.6 Reachability Procedure

With this background, the reachability process can now be defined by the following set of recursive steps which are illustrated in more detail in the example section to follow:

Step 1: Select an s-state for which all possible next s-states are to be evaluated;

e.g.:

s-state 0:

scheduler : [1] : []

writers : [1*] : [new*]

readers : [1*] : [new*]

reader count variable range : [0.0]

Step 2: Select a process or process pool of this s-state (e.g., readers).

Step 3: Select a transition for which all conditions are satisfied: i.e., the incoming event is present, at least one process is defined to be in the start p-state of the transition, and the conditions are satisfied by some sub-range of the corresponding counter variable. In the example, the readers are all ready to transition on the arrival of a *new* event since all are active in p-state 1. Therefore the transition which is chosen is the one which transitions one reader from its p-state 1 to its p-state 2. This transition will *consume* one *new* event and generate one ($r + gr$) event which is sent to the scheduler process. (This transition is shown in the EBDs of Figure 3.21.)

When working with an s-state with self-generalizing p-state transitions, order the selection of these transitions such that: (1) the counter value ranges become less constrained on each transition (see Section 3.4.2.5), and (2) *exactly one* events, {1}, are consumed after the {*} and {+} events. This second ordering rule does not affect the correctness of the resulting reachability tree, but rather the consistency of the results. The example used in Section 3.4.2.4 will be used again to explain this ordering criteria. If **s-state 4** is initially perturbed by the transition in which process **p0** self-loops (consuming the $(-p1 + e1)$ event), the result would be an immediate match with **s-state 3**. Therefore a parent to child relationship is created from **s-state 4** to **s-state 3**. In the example of Section 3.4.2.4, this relation did not exist explicitly, but rather implicitly through the self-generalization which created **s-state 4''**; i.e., **s-state 4** is a parent to itself. The relation pointing back to **s-state 3** as a child is not wrong; it is valid under the condition that there

is *exactly one* ($p0 + e1$) event in the $p0$ event queue. To generate a reachability tree which is consistent in this regard, the transitions should be ordered such that the more general rules (e.g., Rule 2 over Rule 1) are used to create the parent to child relationships. This is achieved by taking transitions consuming events which already have general representation ($\{*\}$ or $\{+\}$) before ones needing a $\{1\}$ event (since this has the potential to change to a $\{*\}$ or $\{+\}$ event).

Step 4: Generate the s-state for this transition by updating the p-state (or p-states for synchronous event transitions where two processes transition on the same step), the event lists, and the counter values. In the example, the following s-state is generated.

s-state A:

scheduler : $[1] : [(_r + rr)]$

writers : $[1^*] : [new^*]$

readers : $[1^*, 2] : [new^*]$

reader count variable range : $[0, 0]$

Consuming one *new* event and moving one reader out of its p-state 1 each requires the application of Rule 3 in Table 3.2. Generating a $(_r + rr)$ and moving one reader into its p-state 2 each applies Rule 7, and the result is **s-state A** above.

Step 5: If the counter variable range includes the value 0 (or m^-), then the generated s-state is analysed according to the zero-value invariant to see if $\{*\}$ components should be removed. If any components are removed, then this s-state becomes quite different from the "count-down" s-state from which it was generated. Therefore, two s-states will now represent the count-down process; i.e., the "count-down" s-state with a counter range lower bound of 1, and the s-state produced by this step which has a counter range of $[0, 0]$.

Step 6: The generated s-state is compared to all existing s-states to see if a *match* exists. If an equivalent s-state is found (ignoring the counter values) then the source s-state is linked to the previously generated s-state as its parent. If the generated s-state can be matched through the generalization rules with a previously found s-

Figure 3.26: Generalized Matching Process

state, then the general version is put in place of both and given the number (label) of the existing s-state (noted as primed s-states in this work). Therefore, the link from the source s-state to the replacement s-state is added to the system description. This is illustrated in Figure 3.26.

Step 7: The effect of this generalization must be recursively applied to the child s-states of the updated s-state. The link from a parent to a child is known to be true by some mechanism, and the transition which caused the link is distinguishable by the components which exist in one and not the other (see Section 3.4.2.4), therefore the generalizations are not applied upward; i.e., generalizations in a child s-state are not reflected in its parent. Generalizing the child s-states causes a “ripple down” effect of the update which will be illustrated by example in the next section. A description of how a generalization at one s-state affects its children is given below where the two generalization rules are shown to give two possible situations for the “ripple” generalizations. In the following description a *component* is either a p-state or event which is part of the s-state description.

1. Parent component has *exactly one* {1} subsumed by *one or more* {+} :

- (a) {0} zero for same child component — Rule 1 had been applied when generating the child s-state. Now, Rule 2 is applicable, and the component in the child s-state should appear as *zero or more*

$\{*\}$.

- (b) $\{1\}$ *exactly one* for same child component — No change had been applied when generating the child s-state. No change is still applicable and the component in the child s-state should appear as *one or more* $\{+\}$.
- (c) $\{+\}$ *one or more* for same child component — Rule 4 had been applied when generating the child s-state. Now, Rule 5 is applicable, and the component in the child s-state should appear as *one or more* $\{+\}$.
- (d) $\{*\}$ *zero or more* for same child component — This is not possible in a single application of a transition. Therefore, the child has self-generalized to this representation through at least one *add one* (e.g., Rule 4), and one *less one* (e.g., Rule 2) operation on this component. The component in the child s-state is left as *zero or more* $\{*\}$.

2. Parent component has *one or more* $\{+\}$ subsumed by *zero or more* $\{*\}$:

- (a) $\{0\}$ *zero* for same child component — This is not possible.
- (b) $\{1\}$ *exactly one* for same child component — This is not possible.
- (c) $\{+\}$ *one or more* for same child component — Either, no change had been applied when generating the child s-state or Rule 5 had been applied. If no change is still applicable, the component should appear as *zero or more*. If Rule 5 was applied, then Rule 6 now applies and again the component in the child s-state should appear as *zero or more* $\{*\}$.
- (d) $\{*\}$ *zero or more* for same child component — Rule 2 had been applied when generating the child s-state. Now, Rule 3 applies and the component in the child s-state should appear as *zero or more* $\{*\}$.

When the “ripple down” generalizations are made, the parent s-states are compared to the child s-states and the components which might have been generalized in the

Parent	Child			
+	0	1	+	*
Reason	Case 1(a)	Case 1(b)	No change or case 1(c)	Less one or case 1(d)
Final Child	*	+	+	*
*	0	1	+	*
Reason	Impossible	Impossible	Case 2(c)	No change or case 2(d)
Final Child	N/A	N/A	*	*

Table 3.4: Ripple Matching Requirements

To "ripple" updates down to child s-states:

If the component is in both s-states **Then**

If the representations are the same **Then**
make no changes

Else replace the child component with the most
general representation (see Table 3.5)

EndIf

Else Insert a "Parent representation less one"
representation into the child s-state

EndIf

Figure 3.27: Algorithm For Ripple Generalizations

parent (i.e., the {*} and {+} components) are updated as described in Table 3.4. The table shows, for a given parent representation and child representation, the possible reasons for the combination, and then the final representation of the corresponding component in the child s-state. An algorithm for achieving these results is given in Figure 3.27.

Step 8: If the new s-state is a generalization of the s-state being analysed (a *self-generalization* because of a "count-up" or "count-down" of pool processes), then the counter value ranges should be set to cover the largest range of values as defined in Table 3.3.

Step 9: Steps 3 to 8 are repeated until all possible transitions are taken.

First Component	Second Component	General Component
1	1	1
1	+	+
1	*	*
+	1	+
+	+	+
+	*	*
*	1	*
*	+	*
*	*	*

Table 3.5: Outcome of Generalizing Two Components

Step 10: Steps 2 to 9 are repeated for each process of the s-state.

Step 11: Steps 1 to 10 are repeated for each s-state in the reachability tree.

Step 12: The procedure terminates when all s-states have been explored.

3.4.3 An Example

At this point the Readers/Writers example will be used to illustrate the verification process. **S-state 0** which was defined in a previous section, is repeated below.

s-state 0:

scheduler : [1] : []

writers : [1*] : [new*]

readers : [1*] : [new*]

reader count variable range : [0, 0]

The first step of the reachability procedure is to perturb this initial s-state by having a writer accept a *new* event. This step yields the following s-state:

s-state 1:

scheduler : [1] : $[(-w + wr)]$

writers : $[1^*, 2]$: $[new^*]$

readers : $[1^*]$: $[new^*]$

reader count variable range : $[0, 0]$

This transition places one writer in its p-state 2 and the outgoing event ($s - wr$) is added to the list of event arrivals for the scheduler as $(-w + wr)$. The transition illustrates that removing one event (new) from *zero or more* leaves again *zero or more* (update Rule 3 of Table 3.2).

The other possible event transition from **s-state 0** is for a reader to accept a *new* event. This leads to the s-state description:

s-state 2:

scheduler : [1] : $[(-r + rr)]$

writers : $[1^*]$: $[new^*]$

readers : $[1^*, 2]$: $[new^*]$

reader count variable range : $[0, 0]$

This completes the transitions from **s-state 0**. Exploring **s-state 1** results in s-states produced by the receipt of any one of the three events: (1) $(-w + wr)$ for the scheduler, (2) new for a writer, and (3) new for a reader.

1. **scheduler** $(-w + wr) \Rightarrow$

s-state 3:

scheduler : [4] : []

writers : $[1^*, 2]$: $[new^*, (s + gw)]$

readers : $[1^*]$: $[new^*]$

reader count variable range : $[0, 0]$

Rule 1 is applied to the $(_w + wr)$ event and the scheduler p-state 1. The transition outputs the event $(w - gw)$, which now appears in the writer event list, and places the scheduler in p-state 4. Both of these actions require the application of Rule 7.

2. **writer new** $\Rightarrow$

s-state 1':

scheduler : [1] : $[(_w + wr)^+]$

writers : $[1^*, 2^+]$: $[new^*]$

readers : $[1^*]$: $[new^*]$

reader count variable range : [0,0]

This is a more general representation of p-state 1 (i.e., it matches **s-state 1** according to our generalizing match rules), and therefore replaces the old version. The generalization must be reflected in the younger s-states of the reachability tree. To do this, the child s-states of **s-state 1** must be examined to ensure that the relationship between each component of each s-state is still valid. The only child of **s-state 1** is **s-state 3**. Here, the parent includes the component $(_w + wr)^+$, while the expression for this component in the child s-state is an *empty set*. According to Table 3.2, an occurrence of generalization case 1(a) has been found. To satisfy this relation, the child must define this component as $(_w + wr)^+$ *less one*, which is $(_w + wr)^*$ according to Rule 2 and the algorithm of Figure 3.27. The result of filtering the generalization through to **s-state 3** is:

s-state 3':

scheduler : [4] : $[(_w + wr)^*]$

writers : $[1^*, 2^+]$: $[new^*, (s + gw)]$

readers : $[1^*]$: $[new^*]$

reader count variable range : [0,0]

It is also necessary to ripple these generalizations through to the child s-states of those s-states which have just been generalized, however, **s-state 3** does not have any children as yet.

3. **reader new** $\Rightarrow$

s-state 4:

scheduler : [1] : $[(-w + wr)^+, (-r + rr)]$

writers : $[1^*, 2^+]$: $[new^*]$

readers : $[1^*, 2]$: $[new^*]$

reader count variable range : $[0, 0]$

This s-state is generated using the generalized description of **s-state 1**.

The analysis eventually builds the tree shown in Figure 3.28 where the s-state descriptions are given in Appendix B. The tree shows that **s-state 10** is generated from **s-state 6** when the writer gives up access to the resource and issues a $(s - wf)$ event: giving the following:

s-state 10:

scheduler : [4] : $[(-w + wr)^*, (-w + wf)]$

writers : $[1^*, 2^*]$: $[new^*]$

readers : $[1^*]$: $[new^*]$

reader count variable range : $[0, 0]$

From this s-state, the $(-w + wr)$ event is queued (it cannot be applied to this s-state), and therefore it does not require the generation of a new reachability analysis s-state. The interesting event is $(-w + wf)$. The s-state generated by the transition on this event is a more general representation of **s-state 1** because both the $(-w + wr)$ event and the writer p-state 2 are now represented as *zero or more* which subsumes *one or more*.

The path through s-states: 0,1,3,6,10,1 (see Figure 3.28), shows the system running when only writers are present. Similarly the path through s-states: 0,2,5,9,14,2, represents the system when only readers are present. **S-state 14** is generated like

Figure 3.28: Readers/Writers generalized reachability tree

s-state 2 generates **s-state A**
when $(_r + rr)$ is consumed.

No match is found for **s-state A** therefore
s-state A becomes the new **s-state 5**.

s-state 2:
scheduler: [1] : $[(_r + rr)^+]$
writers: [1*] : [new*]
readers: [1*, 2+] : [new*]
reader count variable range: [0, 0]

s-state 2:
scheduler: [1] : $[(_r + rr)^+]$
writers: [1*] : [new*]
readers: [1*, 2] : [new*]
reader count variable range: [0, 0]

↓ $(_r + rr)$

⇒

↓ $(_r + rr)$

s-state A:
scheduler: [2] : $[(_r + rr)^+]$
writers: [1*] : [new*]
readers: [1*, 2+] : [new*, (s + gr)]
reader count variable range: [1, 1]

s-state 5:
scheduler: [2] : $[(_r + rr)^+]$
writers: [1*] : [new*]
readers: [1*, 2+] : [new*, (s + gr)]
reader count variable range: [1, 1]

Figure 3.29: S-state 2 initially generates this new s-state 5

s-state 10; i.e., when a reader finishes with the resource. Along this path, the reader count variable is used. For example, in the self-generalized **s-state 2**, the acceptance of event $(_r + rr)$ by the scheduler results in **s-state 5** as shown in Figure 3.29. The transition taken by the scheduler process, takes the scheduler out

s-state 5 generates **s-state 8**
when $(\neg r + rr)$ is consumed.

S-state 8 matches **s-state 5** with
 $(s + gr)$ subsumed by $(s + gr)^+$, therefore
s-state 8 becomes the new **s-state 5'**.

s-state 5:
scheduler: $[2] : [(\neg r + rr)^+]$
writers: $[1^+] : [new^+]$
readers: $[1^+, 2^+] : [new^+, (s + gr)]$
reader count variable range: $[1, 1]$

$(\neg r + rr)$

$\Rightarrow$

s-state 5':
scheduler: $[2] : [(\neg r + rr)^+]$
writers: $[1^+] : [new^+]$
readers: $[1^+, 2^+] : [new^+, (s + gr)^+]$
reader count variable range: $[1, 1^+]$

s-state 8:
scheduler: $[2] : [(\neg r + rr)^+]$
writers: $[1^+] : [new^+]$
readers: $[1^+, 2^+] : [new^+, (s + gr)^+]$
reader count variable range: $[2, 2]$

Figure 3.30: S-state 5 self generalizes to a new version of s-state 5

of its p-state 1, and puts it in its p-state 2. A $(s + gr)$ event is issued to the reader, and the reader count is incremented from $[0, 0]$ to $[1, 1]$ according to the $(C' = C + 1)$ output condition (see update 2 of Table 3.3). When **s-state 5** accepts the same event transition, $(\neg r + rr)$, it adds the $+$ extension to the $(s + gr)$ event, and the value range of the reader count is also updated (to $[2, 2]$) as illustrated in Figure 3.30. However, since this is a self-generalizing, self-loop transition, the counter range is affected by entry 14 of Table 3.3 which changes $[1, 1]$ to $[1, 1^+]$. In a count-up state, the values define the range from the minimum number required to reach the s-state to the number required to produce the most general description which covers all transitions leading to this s-state.

Eventually, the first time **s-state 14** is generated (when a reader relinquishes access to the resource) the result is:

s-state 14:

scheduler : $[2] : [(\neg r + rr)^+, (\neg r + rf)]$
writers : $[1^+] : [new^+]$
readers : $[1^+, 2^+, 3^+] : [new^+, (s + gr)^+, spon_1^+]$

reader count variable range : $[1, 1^+]$

The events listed in this s-state are ordered such that the $(_r + rf)$ event (a $\{1\}$ representation) is applied to the s-state after all of the other events have been examined. Readers accepting the *spn_1* event will self-generalize **s-state 14** by generalizing $(_r + rf)$ to $(_r + rf)^+$ (see Figure 3.31).

It is not apparent in this case that a transition from **s-state 14**, made on the event $(_r + rf)$, will lead to **s-state 2**. Here, the *zero-value invariant* of the reader count variable is used to eliminate $\{*\}$ p-states which need to be *specialized* out of the description. From **s-state 14** a transition on the event $(_r + rf)$ is possible when the count variable is 1, and a different transition is taken if the value is greater than one. Therefore the value range must be used to determine which transitions are possible. This is illustrated in Figure 3.32. When the counter range is restricted to $[1, 1]$, the applicable transition decrements the counter according to update 10 of Table 3.3 giving $[0, 0]$. The invariant statements are used to eliminate any p-states or events with the $\{*\}$ option that are not reasonable when the value reaches zero. This s-state, once pruned by the invariant clause, matches **s-state 2**, and thus this transition is made. For values greater than 1, the range is initially $[2, 2^+]$. According to update 11 of Table 3.3, this becomes $[1, 1^+]$ when the applicable $C > R$ ($C' = C - 1$) update is applied as shown. The resultant s-state matches **s-state 14**; i.e., the s-state self-generalizes. According to rule 23a, $[1, 1^+]$ remains the same when the bound ($R = 1$) is greater than, or equal to, the lower limit, 1, and a decrement operation is to be applied to the counter.

The final result shows that one reader and one writer are required to reach the most general s-state descriptions of the combined behaviour, however two readers are required to cover all transitions. It also indicates that only one writer may ever be active (in **s-state 3**) at any one time, but that the parallelism for readers is unbounded when no writer access requests are made.

S-state 14 generates **s-state C**
when (*spon_1*) is consumed

S-state C matches **s-state 14** with
(*_r + rf*) subsumed by (*_r + rf*)⁺ giving
s-state 14

s-state 14:
scheduler: [2] : [(*_r + rr*)^{*}, (*_r + rf*)]
writers: [1^{*}] : [*new*^{*}]
readers: [1^{*}, 2^{*}, 3^{*}] : [*new*^{*}, (*s + gr*)^{*}, *spon_1*^{*}]
reader count variable range: [1, 1⁺]

s-state 14':
scheduler: [2] : [(*_r + rr*)^{*}, (*_r + rf*)⁺]
writers: [1^{*}] : [*new*^{*}]
readers: [1^{*}, 2^{*}, 3^{*}] : [*new*^{*}, (*s + gr*)^{*}, *spon_1*^{*}]
reader count variable range: [1, 1⁺]

↓ *spon_1*

⇒

s-state C:
scheduler: [2] : [(*_r + rr*)^{*}, (*_r + rf*)⁺]
writers: [1^{*}] : [*new*^{*}]
readers: [1^{*}, 2^{*}, 3^{*}] : [*new*^{*}, (*s + gr*)^{*}, *spon_1*^{*}]
reader count variable range: [1, 1⁺]

Figure 3.31: S-state 14 self-generalizes with s-state C

3.4.4 Evaluation

3.4.4.1 Generalization Properties

There are two features to this *generalized* reachability procedure which result from the generalization rules (see Section 3.4.2.1). The first property is identified by the loss of individuality of pool process instances, and this defines the difference between this method and all those which require static definitions for pool processes. The second feature is compressed s-states which are built using the first property; i.e., by using queues containing any number, *one or more*, of events sent from various pool instances to compress “count-up” and “count-down” reachability s-state sequences. This section will examine these properties and prove informally that they do not invalidate the reachability analysis.

Property 1: Lost Individuality — This method collapses all pool instances into a single representation, and therefore takes away the ability to distinguish an event source. As a result, message data associated with events sent by a pool instance, cannot be used in the receiver’s specification to evaluate conditions of a

S-state 14' generates **s-state A** when the counter range is $[1, 1]$
 and **s-state B** when the counter range is $[2, 2^+]$.
 The *zero value invariant* is applied to **s-state A** to give **s-state A'**

Figure 3.32: S-state 14 initially generates s-states A and B as shown

transition. This is the only constraint added by the generalized p-state and event representations.

Informal proof — If data is not used to change the control perspective, then the pool instances are identical. The process model was defined in Section 3.2 to include a separate FIFO queue for each event. Events are defined by their source process, the communications paradigm being used, and the message being sent. Collapsing all pools into one representation means that there is no way to distinguish the source process, and therefore there can only be one queue for each distinct messages when dealing with pool instances. If the sender cannot be recognized, and data is not used in the transition, the specification must be independent of the FIFO ordering within this queue. Therefore, as long as one pool instance, in the correct p-state, is advanced by any subsequent interactions with this receiver, then the system as

S-state A' matches the self-generalized **s-state 2'** with
 $(\neg r + rr)^+$ subsumed by $(\neg r + rr)^*$ giving **s-state 2''**
S-state B matches **s-state 14'**.

Figure 3.33: A matches s-state 2, and B matches s-state 14''

a whole is consistently represented by the s-state information. The order in which events are sent by independent processes is not necessarily the same as their arrival order, and, furthermore, there is no effect on the state of the system when identifiers are switched on identical processes.

The reachability procedure counts numbers of instances not individuals, and therefore the generalization can be viewed as collapsing the pool instances into a single process which generates multiple events.

Property 2: Compressed S-states — This method compresses series of s-states which show: (1) a series of equivalent pool instance transitions which build up the event queues of the receiving processes, and (2) a series of event receives which are characterized by self-loop transitions of the receiving process (and generally update counter values as a result). There are no analytical losses as a result of compressing “count-up” and “count-down” s-state sequences.

Informal proof — Within an uncompressed sequence of s-states, p-states and

events might be represented differently (i.e., $\{1\}$, $\{+\}$, or $\{*\}$). However, the specific elements exist in all s-states. In the compressed versions, the counter ranges are used to reflect the different levels of concurrency by defining the upper and lower bounds, while the individual s-state elements take on the representation appropriate for that range. To achieve the correct results, the procedure requires added information and this comes in the form of: (1) *zero value invariants*, and (2) the use of counters to define conditions for transitions.

Zero-value invariants are needed to distinguish between a "count-down" series and a new s-state. They are used to remove $\{*\}$ p-states and events which are known to be strictly *zero* when counters reach the value zero (see Section 3.4.2.3). The range conditions must unambiguously define the response to every possible counter value. If transitions are ordered such that the least constrained value range is transferred to a child, then the child can either widen an unconstrained range or restrict it to a lower/higher value for the lower and upper bounds of the range respectively; i.e., insert a known limit. As such, the least constrained counter range is passed on to subsequent child s-states.

3.4.4.2 Properties of the Reachability Procedure

This section discusses five properties of the reachability procedure.

Termination — In classical reachability analysis, it is possible to create a situation where the verification procedure will not terminate. This occurs when the unbounded, process pairwise, communication channels grow at a faster rate than the messages can be consumed. Since all communications between processes are FIFO ordered in the same channel, each message send becomes a new entry in the channel, and thus the contents can grow without bound. In our procedure, each event is represented by a single term in the system state descriptions. When messages of the same type are continuously incoming on a channel, they are represented by a single n^+ or n^- term. With finite state machines, bounded data spaces, and finite state and event descriptions (three different representations for each type), there must be a finite set of possible combined, system state descriptions. Therefore, the process

must eventually terminate.¹⁶ Unbounded data spaces are possible if auxiliary variables (other than the pool counters) are allowed to grow without bound since they would also be components of the composite system state descriptions.

Resultant Tree Size — Compressing “count-up” and “count-down” states corresponds directly to a reduction in the size of the reachability tree. Since no information is lost, the power of reachability analysis is maintained; i.e., the ability to detect temporal errors is not hindered by the technique because all distinct execution paths are recognizable from the generalized reachability tree.

The tree size is not compressed when the match: *zero or more* subsumes *zero*, could be made. As discussed in Section 3.4.2.4, this can lead to a complex tree compression and inexplicable s-state to s-state transitions. It would be a worthwhile exercise to examine the impact of applying this matching rule during the reachability procedure. Given the current reachability procedure, a second pass of s-state matching can be performed on the reachability tree such that *zero or more* is made to subsume *zero*. In the Readers/Writers example, the tree is reduced, as shown in Figure 3.34. The s-state labels are equivalent to those in Figure 3.28, and are followed by a list of the s-states which are subsumed by that s-state. This result can be related back to the scheduler specification; i.e., s-state 4 is equivalent to scheduler p-state 1 (*init*), s-state 17 is equivalent to p-state 2 (*readers*), s-state 18 is equivalent to p-state 3 (*remove readers*), and s-states 7, 11, and 15 combine to represent p-state 4 (*writer*). The p-states of the readers are shown to be relatively inconsequential in terms of the system behaviour in this design, except when a writer is introduced. When a writer exists, readers cannot enter their own p-state 3. The execution of a writer generates distinct reachability s-states implying that their behaviour is significant within the system; i.e., the execution through the writer p-state loop restricts the system concurrency significantly (to a single writer). This result also shows that there is no deadlock in the system, and that starvation is eliminated by a fair choice of execution paths from s-state 4 (or scheduler p-state 1).

Minimum Pool Sizes — The counter range upper bound offers the ability to

¹⁶This reason for termination is equivalent to the non-FIFO channel option on the VALIRA reachability tool where like events are generalized in the channels [116].

Figure 3.34: Minimized Reachability Tree for Readers/Writers

determine the minimum pool size required to produce a fully general tree; i.e., one where the size is not changed by the addition of one more pool instance. This value allows the analyst to define a fixed configuration for the system and thus apply other existing validation techniques which require static configurations. It also defines the starting point for an inductive proof of system properties; as required by Garg [46] and Browne [17] in some aspects of their work. For the Readers/Writers example, the reader count indicated that there must be two readers to execute all behaviours, and from the reachability tree, which indicates that writers are mutually exclusive (see below), it can be determined that a single writer will suffice.

Levels of Concurrency — The generalized representation also offers an analyst the ability to comment on the levels of concurrency within a pool of processes directly from the composite system state descriptions. The pool process state descriptions are defined as a single complex components of the composite system states. From these descriptions comments can be made on the levels of concurrency within a process pool. For example, mutually exclusive states are shown by process states which never generalize from an *exactly one* representation. The remaining allowed process states of pool instances might characterize waiting states for the pool instances which are not in this mutually exclusive state. The writer p-state 3 is mutually exclusive, and since these processes execute a deterministic sequence (p-state 1; p-

state 2; p-state 3) the writing operation is mutually exclusive. Petri Nets provide a similar property, referred to as *boundedness*, described by the number of tokens in any given place at the same time [90].

Completeness — The results of this reachability analysis are dependent on the completeness of the specification; i.e., an incomplete specification produces an inconclusive result. For example, consider the *2 of 3* problem where each process of a pool is trying to acquire access to 2 out of a collection of 3 resources. A specification which includes deadlock is given in Figure 3.35. The pool instances, *U1s*, are quite simple, they make two requests (*req1* and *req2*), use the resources, and then release each resource (*rel1* and *rel2*). A three p-state scheduler, *U2*, has been defined. Scheduler p-state 1 is the initial, “no outstanding requests” state, p-state 2 is the “1-3 outstanding requests state”, and p-state 3 is the “more than 3 outstanding requests state”. A counter, *C*, is used to count the number of resource requests. The *rel1* messages are not expected to be in the queues when ($0 < C < 1$), and the range ($C > 1$) is covered by the self-loop transition at scheduler p-state 2, and the transition leaving scheduler p-state 3 for p-state 2. The *rel2* messages can arrive whenever ($C > 0$), and this range is covered by a self-loop transition at p-state 2, the transition from p-state 3 to p-state 2, and the transition which takes the scheduler out of p-state 2 and back to p-state 1. Arrivals of *req1* and *req2* are independent of the value of *C*, and therefore there must be transitions which cover all possible values. The transitions which are conditional on ($C < 3$) are relatively intuitive; accept a request if two or fewer requests have previously been received. To complete the specification, there must also be a transition to cover the range ($C > 2$), and the one shown simply states that scheduler moves to p-state 3 where requests are queued until a resource is released.

Without this last transition (i.e., leading to scheduler p-state 3), the reachability procedure would not be able to find the deadlock condition, but would rather indicate that the solution is good when there are anywhere from 1 to 3 outstanding requests. The specification on the bounds of the resource must be complete to force the reachability procedure to examine the system execution at the bound, and at least one step beyond the bound. With this transition, the procedure can deter-

Figure 3.35: Deadlock Specification for the 2 of 3 Problem

mine that three requests coming from three different pool instances (i.e., there are 3 requests and no instances in pool p-state 3) will cause a deadlock in the system. For a static configuration and the classical reachability procedure, the deadlock is found when the configuration includes three or more pool processes. The classic case depends on having a sufficiently complete configuration as opposed to a complete specification.

Chapter 4

Designing within the Design Fragments Framework

This chapter discusses the two phases of design in the *Design Fragments* framework: (1) allocation, and (2) designing with design fragments. The separation of application-specific details from communication semantics inherent in EBD specifications is used in direct relation with these two steps. The application-behaviour is captured by the ACMs defined by the allocation, and the communication semantics are satisfied by correct application of design fragments. Together, they provide strong traceability of the behavioural requirements in the design. Allocation provides large sense traceability (i.e., completeness), and design fragments provide small sense traceability (i.e., correctness at the level of specification segments).

Allocation — Allocation is the first step in the design process. Here, EBDs are assigned to actors in the design. This phase in the development process is covered in Section 4.1 under headings for concept (which includes a discussion on the decomposition of work for designing large systems), models, and methods.

Design Fragments — After allocation, design fragments are used to complete the design of the behavioural requirements. The design fragments satisfy the requirements of the interactions between entities within a given set of actors. If all associated *contingencies* are proven, then they ensure that there is conformance between the specification and design. This is discussed in

Section 4.2 which presents the material as follows. Section 4.2.1 describes the relationship between the design fragments framework and current research in knowledge-based software engineering. Section 4.2.2 introduces the concepts behind, and motivation for, design fragments while sections 4.2.3, 4.2.4, and 4.2.5 describes design fragments and their application in detail. Section 4.2.6 presents an initial population for the *design fragments framework*, and finally, Section 4.2.7 discusses a potential design environment. Validation of the design fragment *contingencies* is discussed in Chapter 5.

4.1 Allocation

4.1.1 Concepts

The allocation step is fundamental to *strong traceability*. EBDs are assigned to actors such that they are not decomposed across actor boundaries, however any number of EBDs can be co-allocated to the same actor as follows:

All EBDs are assigned to an actor:

$$\forall x(EBD(x) \rightarrow (\exists a(actor(a) \cap (x \in a))))$$

No EBD is assigned to more than one actor:

$$\forall x(EBD(x) \rightarrow \neg(\exists a \exists b(actor(a) \cap actor(b) \cap (x \in a) \cap (x \in b))))$$

The allocation step preserves the specification as development moves from requirements to design. The EBDs and their allocation to actors are used to establish the root of a schema for proving completeness; i.e., all interactions defined by the EBDs must be satisfied by the design, therefore all interactions required of an actor must be satisfied by the design. Chapter 5 discusses how the EBDs and their allocation are used to create such a schema against which design decisions are compared for proof of completeness.

For large systems, allocation is performed separately on each subsystem, however each can still represent a large design problem which must be decomposed for work distribution (c.f. § 3.3). In this discussion, the term *top-level partition* is used to describe the behaviour defined by an EBD after all collapsed behaviours have been abstracted out of the description. For example, in the Readers/Writers problem of Figure 3.18, the reader *idle* and *read* states represent collapsed behaviours: both interact with a user through (for example) a terminal. The top-level partition of a reader collapses these behaviours into single states (as shown in the EBD of the example), and does not interpret them as being different from any sequential operation. Designers may wish to group segments of EBDs which could be referred to as collapsed behaviours (i.e., they neither deadlock nor diverge, and have satisfiable event sets which are disjoint from the top-level partition) before or during this decomposition step. Thus given a set of communicating EBDs, the possible distribution of work is as follows:

1. Each collapsed behaviour can be designed independent of the top-level partition, where the top-level partition is defined by the EBD assuming that all operations (including collapsed behaviours) are finite sequential requirements, needing appropriate placeholders. In some cases these are re-usable behaviours, and in all cases the design is expanded onto the actor design which includes the encapsulating top-level partition(s). Once designed, these *services* represent *large design fragments*. Like a macro expansion, they fit into the subsystem designs as additions to the actors which contain this behaviour. The behaviour of the service could: (1) be defined by a procedure which is included and called (or macro expanded) within the actor, (2) require a rendezvous with another actor which would again present a procedural interface to the subsystem, or (3) require the addition of multiple inter-actor interactions with a new actor. This third alternative may require that the specification be refined and the allocation to include this new actor, such that, the schema for proving completeness can continue to be generated mechanically.
2. Each operation representing a sequential functional requirement can be designed independent of the behaviour in which it is contained. This may again

represent a re-usable design.

3. The designs of actors with co-allocated EBDs can also be worked on independently.
4. The tasking model of the subsystem can be developed independently. It is expected that these models will include ten or fewer interacting actors (as suggested by [23]).¹ This can be further decomposed if the subsystem actors can be separated into disjoint sets.

4.1.2 Models

Information derived from the allocation step is placed in tabular form for the purpose of documentation.² As such there is a direct link from the software specification documents containing the EBDs, and the software design documents.³ These tables initialize the proof of completeness of the design as discussed in Chapter 5.

Specifically, each actor of the design is represented by an *Actor Table*. There is a precise relation between the EBDs allocated to an actor and the information in an Actor Table. An example Actor Table is shown in Figure 4.1. The actor name is \$reader\$. It satisfies the requirements of the *reader* EBD in the Readers/Writers problem, and symbolizes the corresponding ACM of the design by \$rdr\$. The tables illustrate all static labels by encapsulating a mnemonic in \$ symbols, while properties yet to be defined are characterized by mnemonics encapsulated by ? symbols⁴. This type of notation was used by Heninger to cross-reference parameters in a tabular approach to complex system specification [60]. Heninger used these symbolic

¹This assumes inclusion of only those behaviours which interact within the subsystem, as opposed to externally coupled behaviours which are expected to be viewed as collapsed behaviours relative to the top-level partition. The assumption is based on industry experience with MachineCharts designs.

²Other approaches to maintaining this design decision could be used.

³The design steps described by this thesis represent preliminary design: i.e., the software structure and relations for interaction.

⁴Such properties are called *lemmas* in this work.

ACTOR TABLE

Actor: \$reader\$

EBDs: *reader*

ACMs: \$rdcr\$

Basic Model: Design Fragment for Figure 4.11 is used.

Top Level Partitions	Comments
?READER?	
Layered Partitions	Comments
?IDLE?	
?READ?	
Comments	
The design consists of a pool of \$reader\$ actors.	

Figure 4.1: Actor Table Example

notations to distinguish and correctly forward a reviewer to more detailed aspects of the specification. All lemmas must be demonstrated to complete the design process.⁵ therefore the ? symbols are used as an indicator that the property is assumed in the current table and demonstrated elsewhere.

An actor table defines the lemmas of an actor which must be proven to ensure a complete and correct design. These include: (1) the ACMs, (2) the *basic model* which is discussed in Sections 4.2.3, 4.2.4, and 4.2.5, and sets up the actors engine⁶ and interface, and uses the ACMs and interconnections between actors to carry out the abstract behaviours of the encapsulated EBDs, (3) the top-level partition(s) (one per EBD allocated to the actor), and (4) the hierarchically *layered partitions* which represent the collapsed behaviours. (The functional requirements are not being addressed in this discussion.)

Given that the first step to design is the allocation of EBDs to actors, the Actor Tables can be established at the start of the design process to collectively define

⁵Recall that the term *demonstrate* is used to refer to any level of proof (formal or informal), or assumed satisfaction of a contingency. The design can be complete without being formally proven correct. The demonstration of contingencies is discussed in Section 5.1.

⁶The actor engine is its mainline program; it encapsulates the actor's *agenda* [23].

the requirements of a complete design. Once an actor is named, and the EBDs are assigned to the actor, most of the remaining components of an Actor Table can be generated automatically from the specification. For example,

1. ACMs are represented by the consonants of the EBD names. They are encapsulated within \$ symbols to indicate that they are labels to be used in the design.
2. The basic model must be designed, and therefore cannot be determined automatically. The designer must complete this field with a reference to a design. This is discussed in more detail in Section 4.2.7.
3. Each *partition* is a requirement of that actor, and the top-level partitions can be symbolized by the EBD names in upper case letters encapsulated by ? symbols. The layered partitions are symbolized by the names that appear on the EBD in upper case letters encapsulated by ? symbols. In the Readers/Writers example, both the *idle* state and the *read* state define collapsed behaviours where interactions occur between the user and a terminal interface [16]. All states, except wait states, are assumed to contain a collapsed behaviour; and, as such, are entered in the tables as layered partitions.

The Actor Table has several locations for comments. This feature is prevalent in TTCN; an ISO accepted standard documentation technique for protocol specifications [41, 92]. Whereas most of an Actor Table can be generated automatically, the comment sections are free-form areas in which additional comments can be added by the designer.

Actor Tables are used along with other specialized tables to document the design process as discussed in Sections 4.2.7 and 5.2.

4.1.3 Methods

Allocation is a creative step. It depends on target domain issues such as those suggested below.

Implementation Language -- Is there support for concurrent elements, or does the implementation language force an all-to-one allocation?

Performance -- Is the system timing so critical that a static *a priori* defined schedule is required meaning that an all-to-one allocation is also required? Are the functions of certain entities more critical than others? Can priority assignments be used to solve issues of timing for critical functions? Will bottlenecks arise under certain configurations?

Hardware -- Is this a multiprocessor environment? Hardware considerations were discussed as a key part of the decomposition of the system for specification in Section 3.3.

Load Balancing --- Given a multiprocessor environment, is each processor being utilized appropriately? Is the work load reasonably distributed? Does the distribution avoid potential bottlenecks? Is throughput acceptable?

Operating System -- Can the underlying operating system be used to provide concurrency? Can it be used to hide inter-processor communications?

Experience -- Does experience dictate a certain distribution; e.g., without studying loading properties?

Prototypes -- Has a prototype been produced successfully? What was learned about performance (e.g., bottlenecks) from a prototype?

Reuse -- Do portions of this system exist as a result of earlier projects?

4.2 Designing with Design Fragments

4.2.1 Relation to Current Work in Knowledge-Based Software Engineering

The *design fragments framework* presents precise alternative potential solutions for the specified communications under different entity to actor allocations. As such it presents design *plans* for reactive systems (cref. § 2.3). Each alternative is reasoned before being placed into the framework providing the user with a set of formal relations between specification and design. The relations are contingent on a given set of proofs, and might constrain a specification (e.g., in terms of performance). Constraints (when apparent) are defined as properties of the design fragment. Therefore the framework is a catalogue of potential solutions much like schemas in PDS [32], Harandi's templates [53], and the cliches of the Programmers Apprentice [117] except that the alternatives depend on features other than program efficiency, such as constraints from hardware and real-time support in the target language. Design fragments represent a factoring of the communications semantics from the application behaviour, and provide alternative solutions for the communications.

The framework is a design assistant where specification segments are pattern matched to yield potential design fragments, and the assistance is found in the form of the proofs required to ensure the validity of the design choice. It is a transformational approach as opposed to the building concept of the Programmer's Apprentice. Instead of correctness preserving transformations, the framework provides transformations and the means to prove the correctness in a given application and partial design. This approach is used by the AFFIRM system (cref. § 2.3). The design fragments are expected to provide helpful insight into incorrect or over-constraining specifications; and, as such, will be used in an iterative specification and design process. In this sense, the method is used to *build* a design.

This section concludes by listing several common goals of transformational programming defined in terms of the *design fragments framework*.

Safe, Verified, and Reliable Software — Transformational techniques are as-

sumed to help in the development of *good* software. Here *strongly traceable* behaviour requirements are key. The framework helps the design process through the definition of design properties and the requirements needed to establish strict and correct design *plans* for reactive system software.

Correct Transformations — Where as automatic systems (e.g., as discussed by Fickas [45]) check preconditions before applying a transformation, here the user is given the *contingencies* existing for a particular transformation.

Design Catalogues — Here the goal is the same: i.e., to produce a sufficiently populated (and correct) library of design fragments.

Extensible Framework — This goal is quite straight-forward. The user should be able to extend the framework by the addition of new potential solutions.

Documentation of Process — The design fragments are labelled such that the process of design can be logged for maintenance and for re-design after modifications.

Measure effects of a transform — The user defines the effect caused by a transform in relation to the properties defined by the chosen design fragment, or those defined by the designer through analysis or opinion.

Schematic (Pattern) Replacement Rules — Design fragments provide partial designs for independent, non-overlapping specification segments. The segments covered by the design fragment are matched to a segment of the specification, and *contingencies* define the rules under which the design is correct.

Descendance Relationships — Design fragments can emulate the exact behaviour of the specification, or represent a descendance relation where a subset of the behaviour paths are executable under the design. That is, given a reachability tree for a system specification, the design fragment can either: (1) define a implementation with the same reachable states, or (2) define an implementation which covers a subset of paths from one reachable state to another. This subset represents a constrained, but valid behaviour for the system implementation.

Performance -- This framework approach assumes that there are too many domain dependent factors involved in the development of reactive systems for automatic translation, therefore the performance is in the hands of the designers and thus depends on experience. The development process for this framework attempts to include only sensible design fragments with such properties highlighted. Properties can be added by the designers as necessary.

Broad Domains for Source and Target — The framework supports three communications paradigms and multiple queueing disciplines for reactive system specification, and should support several design alternatives.

4.2.2 Design Fragments – A Conceptual View

The design fragments describe implementations of the EI, where by definition, the EBDs are encapsulated by ACMs. Typically, fragments implement a specific type of communications semantics (such as asynchronous communications) or a specific queueing discipline (such as FIFO), but not necessarily as an exact emulation of the requirements model. This is discussed by way of an example.

Consider the problem of two EBDs, allocated to separate actors, that communicate via FIFO asynchronous events. The semantics of this asynchronous event transfer are expressed by the following properties: (1) a send, once initiated, must complete eventually; (2) the event being sent must be made available for reception; (3) an event available for reception remains in this state until it is received; (4) an event which is available for reception, and which is part of the satisfiable set of the receiver, must be used to complete an initiated reception when it is the first applicable event to have arrived (i.e., FIFO queueing); (5) an event which is part of the satisfiable set, but not available for reception, must block the receiver from processing along the path where this reception is defined by this transition antecedent. Note however, that there is no insistence that the event be received.

Figure 4.2 shows several MachineChart designs containing fragments that could support event transmission without violating these properties. In some cases, a property is guaranteed implicitly; i.e., it is a property of the design fragment. In

Figure 4.2: Designs Illustrating Fragments For Asynchronous Event Transfer

other cases, the guarantee would have to be proven explicitly.

The design fragments use *messages* to transfer events between actors; i.e., an event is the unit of communications in the requirements model, and a message is the unit of communications between actors in the design model. The design fragments are presented below.

Jab — A simple “jab” implements the requirements of a sender EI for an asynchronous message transfer between two actors (properties 1 and 2 described above), and can be used to transfer an event from an EBD (within an ACM) in sender actor to an EI implementation in a receiver actor. The guarantee of return from the send is implicit in a jab, and the message is made available by this finger. The gate at the receiver will queue the incoming message FIFO (satisfying property 3), and the waiting place on the gate can be used to open and close the gate according to the satisfiable set (satisfying property 4). Property 5 is satisfied partially by the gate which represents a blocking receive, and because this event is alone in its satisfiable event set. As such, a FIFO selection is made on the message (and thus event) in this example.

Courier — A *courier* actor could pick up messages from the sender actor using

a rendezvous (a 2-way synchronous message transfer), and deliver it to the receiver actor using another rendezvous. The EI implementations at either end would have to coordinate pick-up and reception of the message on behalf of the ACMs (EBDs). This mechanism implicitly guarantees an eventual return from the send (satisfying property 1) because the sender actor internally buffers the messages to be picked up by the courier actor (this is the standard practice for courier actors). The output of the courier makes the message available for reception (satisfying property 2), and a receiver design like that described above completes the requirements for the properties of an asynchronous send (properties 3 through 5); i.e., the courier does not alter the event ordering.

Buffer — A *buffer* actor could be used by the sender as a message mailbox: that is, the sender actor would use a rendezvous to deposit a message (event) in the mailbox (for which the sender actor is implicitly guaranteed to complete). The receiver actor would be expected to pick-up messages (to be received) from the buffer actor using a rendezvous. Again, assuming an unbounded or non-full buffer, the send is implicitly guaranteed to terminate (property 1), and message is available (and kept available) for reception at the buffer (satisfying properties 2 and 3). The receiver will satisfy properties 4 and 5 if the buffered message(s) represent the satisfiable event set, and the receiver goes to the buffer to retrieve a message whenever a corresponding request is made by the ACM. This assumes that the FIFO event ordering is maintained by the buffer.

Rendezvous — The sender actor could rendezvous directly with the receiver actor. There are no implicit guarantees of property 1 because the synchronous interaction is not implicitly guaranteed to return. Otherwise the properties (2 through 5) are satisfied by design components like those in the “jab” solution.

Each solution has certain characteristics which make it more, or less, appealing to a given application and implementation domain. The first design fragment is essentially an emulation of the requirement semantics. While this is a valid design, it is not necessarily the best for the simple reason that it requires buffering of

messages at the receiver actor --- in real systems buffers are not infinite. It is also not realizable in the Ada programming language.

The second and third design fragments use simple Ada rendezvous and additional actors to implement a form of asynchronous event transfer based on synchronous messages; however, they still require some form of buffering and they introduce extra actors.

The fourth design fragment uses only synchronous messages (Ada rendezvous), thus it can avoid all buffering of messages, but it requires the most effort to prove that it does not violate the asynchronous communications semantics of the requirements model (i.e., that the design conforms to the requirements).

The design fragments for EBDs co-allocated to the same actor vary from (1) a basic emulation of the concurrency through internal queues and an engine that constantly moves events from one queue to another, to (2) a series of scheduled procedure calls. The former approach is much easier to construct and to verify (with respect to the semantics of the requirements model), however, the latter could be much more efficient.

4.2.3 Design Fragments – A Separation of Concerns

The design process, which uses the design fragments framework, is based on the fundamental assumption that the application-behaviour of an individual EBD is represented explicitly in the design. A direct mapping of these requirements supports strong traceability, and the *ACM* design component of the MachineCharts design notation is used for this purpose. The design fragments satisfy the requirements of the EIs which, according to the meta-model of Section 3.2, interact with the EBDs to execute the *send* and *receive* primitive operations. Therefore, the requirements of these operations must be satisfied by design fragments which interact with the basic *ACM* design. These primitive operations have different semantics depending on the specified communication paradigms and the queueing disciplines defined by the events and EI directives. They are viewed, in the design phase, in terms of five basic design features: (1) the *ACMs*, (2) the interface to an *ACM*, (3) the actor engine, (4) the actor interface, and (5) the interconnections between

actors.

When each EBD is allocated to its own actor, these components map directly to the EBD meta-model of Figures 3.1 and 3.2 as follows: (1) the ACMs map to the application-behaviour defined by the meta-level EBD, (2) the ACM interface maps to the client/server interface between the EBDs and EIs, (3) the actor engine maps to the functionality of the EI, (4) the actor interface maps to the EI buffers, and (5) the interconnections map to the EI-to-EI interactions. For this thesis, the first two components of an actor's design are fixed: i.e., the ACMs and their interface. The other three components are candidates for design using design fragments. Each component can be considered separately, but as such is dependent on assumed design properties for the other two components. For example, if a "rendezvous" solution is chosen as the design fragment to connect two actors that communicate asynchronously (as presented in the previous section), then this push finger can be *plugged* into a design if there is: (1) a unique placeholder in the definition of the sender's engine for the send operation, and (2) a unique placeholder in the receiver's engine for the receive operation. The send must be performed according to the behaviour defined by the sender EBD, and the receive must be performed according to the behaviour of the receiver EBD. In summary, a direct "plug-in", design fragment to connect these two actors requires actor engines and interfaces which emulate the specification semantics.

As another example, a satisfiable event set of unique events which requires a "FAIR" selection policy, can be implemented by a single FIFO gate design fragment. Thus, the actor's interface has a single entry point for each event that is part of this satisfiable event set. As such, each event interaction for this set must have the same implementation because, for example, a jab finger and a push finger cannot use the same gate as their destination. All of the incoming fingers must have the same implementation semantics. Therefore, a design fragment which implements a portion of an actor's interface requires interconnections which *match* the gate design. These logically separable design elements are contingent on one another; i.e., each design fragment includes *design contingencies* relating to the necessary, coordinating design elements.

Figure 4.3: Three designed components of an actor

The three components: the engine, the interface, and the interconnections, are all contingent on one another with respect to their individual designs. The design process defined by this thesis reduces the complexity of *matching* design components by restricting the variations allowed on two of the components; allowing only one to vary directly. Figure 4.3 illustrates the design components as emulations of the EBD meta-model.

The engine — To emulate the EI functionality, the engine interacts with the ACM through an event/action interface defined in Section 4.2.4.1, and defines the agenda of the actor. The agenda includes: (1) pushing the ACM, (2) pushing operation buttons, (3) sending inter-actor events in the order defined by the EI directives; i.e., it carries out the functionality of the *SendSteps(i)* algorithm defined in Table A.4.⁷ and (4) receiving an inter-actor event by pulling the gates (defined by the interface) according to the corresponding satisfiable event set and the ordering defined for that set.⁸ Because multiple entities can be co-allocated, the agenda must also establishes the relative placement of operations and *intra-actor communications*; i.e., where, in the agenda, each communication primitive is defined to occur.

⁷Later in this section, events are said to be sent to other actors from the *send_placeholders* of the actor's engine.

⁸This gate or select bar pull operation is later referred to as the *receive_place*.

The Interface — The interface establishes the gates of an actor, and satisfies the required ordering of unique events.⁹ The gates represent the buffers of the EI; thus, they establish the EBD view of event queueing (see Figure 3.2).

The Interconnections — The interconnections satisfy the requirements of the *inter-actor communications*, therefore, they represent the EI-to-EI interactions for actor-to-actor communications. When designed independently of the other components, they can be plugged into an engine's agenda on the sender side (at a `send_placeholder`), and the matching gate of the receiver (at a `receive_placeholder`).

To limit the complexity of having independent design fragments for each of the three components, two of the components are restricted, and only one is allowed to vary directly. The interconnections represent the EI-to-EI interactions, and are the most intuitively changed. They represent the smallest, and simplest pieces for design. As individuals, they are not application-dependent, and yet they are the most dependent on the paradigms supported by the implementation environment (particularly the implementation language). Thus, the proposed method allows direct variation of interconnections by design fragments. However, the potential for variation in the other two components must not be precluded by this decision. For example, the method should include design alternatives which affect one or more of the other components; e.g., the actor interface, in which case the interconnection and the interface must be *matched*. For example, if an asynchronous interaction is implemented by a rendezvous, the gate must perform the two-way synchronization required for a rendezvous. If several interactions are combined into a single interconnection with added message data, then the gate must be able to accept the incoming data: the engine must be able to interpret this data to resolve which event has arrived.

A design fragment which directly varies an interconnection requires a default, understood design for the actor interface and engine (referred to here as a *basic model*). This (semi-fixed) design supports a direct "plug-in" capability for design fragments which represent an emulation of the inter-actor communications. The default is strictly maintained in the situation where only the interconnections are

⁹The gates are later called the *receive_placeholders*.

affected by the design alternative. Otherwise, the design fragment must define the manipulation of the default design which guarantees the correct *matching* of the three components; and, at the same time, maintains the “plug-in” capabilities for other design fragments. The default design is defined in terms of placeholders into which interconnections are “plugged”, and with which the design fragment can force the matching of the three design components through a mechanical manipulation of the known, default design. This approach does not preclude design fragments which vary the interface or actor engine. In fact, because all intra-actor interactions are allocated to the engine, the only way to have alternative designs for co-allocated behaviours is to allow for various engine designs. However, the engine and interface must, at all times, present the placeholders required for actor interconnections, and define the placeholders of all characteristics which might be manipulated by any other design fragment. The rest of this section (i.e., 4.2) is devoted to the details of a design fragments approach to design using a basic model. The information is presented as follows. This subsection continues with a high level, conceptual view of a basic model. It is followed, in Section 4.2.4, by a detailed description of ACMs and their interface. Given this information, the requirements a one basic model are discussed. Once a basic model is established, the details of design fragments are covered (Section 4.2.5).

It is possible to vary one or any combination of these components directly, however, it is felt that starting with the interconnection provides the most intuitive basis, and the most flexible design process. Thus, the design methodology of this thesis presents a default design for both the actor engine and interface, and refers to the combined structure as the *basic model*. The design fragments either: (1) build up a basic model for the actor (in the case of specification segments with intra-actor communications), or (2) include interconnections between actors (for specification segments with inter-actor communications). They can manipulate placeholders required by the basic model (e.g., making them redundant), however, it is a property of the design fragment to ensure that any such changes are valid; i.e., ensuring that the required *matching* is satisfied, and that placeholder contents are defined or re-defined (see Section 4.2.5). The idea of a *basic model* and matching design fragments

Figure 4.4: First Example Basic Model and Design Fragment Set

is discussed below, at a conceptual level, by way of examples.

Example Basic Model

FIFO or FAIR queueing of unique events, and FAIR or receiver defined priority queueing for satisfiable event sets:

The basic model in question covers the default queueing disciplines of the EBD specification technique, and is illustrated in Figure 4.4. Here, the basic model defines a separate gate for every unique event involved in an inter-actor communication; and, thus supports FIFO (and FAIR) queueing of these events. To get a FAIR selection policy for the satisfiable event sets, the engine pulls the gates using a choice bar connected to all gates, which are opened and closed using boolean parameters corresponding to the satisfiable event sets.¹⁰ To get a priority based selection policy the gates of the same priority are connected using choice bars which are pulled, in order, by conditional pull fingers; i.e., ones which return false if no events are queued at the gates. The design fragments represent non-overlapping specification segments with any of the three communications paradigms specified. This example is discussed in detail in Section 4.2.4 (i.e., in terms of placeholders), and throughout the rest of the thesis.

¹⁰Satisfiable event sets with both inter- and intra-actor communications are discussed Section 5.1.6 and C.1.2.

Figure 4.5: Second Example Basic Model and Design Fragment Set

Example Basic Model

FIFO or FAIR queueing of unique events, and FIFO or FAIR queueing for satisfiable event sets:

The components of this design strategy are illustrated in Figure 4.5. Here, the basic model defines a separate gate for every satisfiable event set with inter-actor communications; and, thus supports FIFO (or FAIR) queueing of all the events. Because all events are received at the same gate, the design fragments associated with this basic model must implement all of the communications paradigms which can be specified, using the same paradigm (e.g., all design fragments implement asynchronous communications). Alternatively, each design fragment implements the requirements of one or more states (thus covering all events in a satisfiable event set, and forcing all events in the set to have the same implementation semantics.) The concepts of individual, non-overlapping specification segments which are used to define design fragments is discussed in detail in Section 4.2.5.

Example Basic Model

Sender defined priority queueing of unique events, and FAIR, or sender defined priority queueing for satisfiable event sets:

The components of this design strategy are illustrated in Figure 4.6. Here, the basic model defines a separate gate for every priority level. The engine uses conditional

Figure 4.6: Third Example Basic Model and Design Fragment Set

pull fingers to order the selection of events: i.e., pulling the highest priority gate first and so on. Because different events are received at the same gate, the design fragments associated with this basic model must again implement all of the communications paradigms which can be specified, using the same paradigm. The design fragments must also define the required decision making on event sends, when the sender priority level is dynamically defined.

Example Basic Model

Sender defined priority queueing of unique events, and receiver defined priority queueing for satisfiable event sets:

The components of this design strategy are illustrated in Figure 4.7. Here, the basic model defines a separate gate for every message and priority level combination. The engine uses conditional pull fingers to order the selection of events: i.e., pulling the highest priority gate (or choice bar) first and so on where, in this case, the priority is defined by the receiver. The design fragments must also define the required decision making on event sends when the sender priority level is dynamically defined.

It should also be mentioned that MachineCharts are an extendable notation: i.e., annotations or new symbology can be added to represent the different queueing techniques. Some examples are given in Figure 4.8. In the top example, a sender-based priority queue is represented by a gate with a *PI* symbol by one corner (where *l* is an integer, and represents the priority level). A receiver-based priority pull is

Figure 4.7: Fourth Example Basic Model and Design Fragment Set

illustrated below this. Here, the gates attached to the choice bar are numbered such that the number defines which gate is tested first for any waiting entries. A LIFO queue could be illustrated by a pull finger which goes through a gate, and shows the tail external to the actor. In the final example, the queuing required by the *when()* directive is illustrated by annotating the pull finger with the necessary relation. These notations are most useful when the features they describe are directly supported by the implementation language.

Approaches other than that described here (i.e., using the concept of a *basic model*) are possible, but would require similar support for *matching* the associated components to reconcile any outstanding *design contingencies*. For the proposed approach, if the designer chooses an implementation for the engine and interface which does not satisfy the requirements of the chosen basic model, then the specification can be refined such that there is such a correlation, before the design fragments are applied. For example, if the actor interface groups two or more gates together, when a separate gate is required for each unique event by the design fragments, then the specification is re-written (refined) with the same events combined into a single unique event. This is illustrated in Figure 4.9. At the top of the diagram, is the interface for the Readers/Writer specification (Figure 3.18) when a separate gate is required for every unique event, and all gates are connected to a single choice bar. If, however, the read request (*rr*) and write request (*wr*) gates are combined

Gates with queueing based on sender defined priorities

Gates selected according to receiver defined priorities

Illustrating LIFO queueing

Figure 4.8: MachineChart Notations for Alternate Queueing Disciplines

into one at the interface (as shown on the right), then the specification is re-written to match this interface (as shown at the bottom of Figure 4.9) before the design is completed using design fragments which expect a separate gate for every unique event.

Refinement of the specification is not required if a design fragment is defined to cover a specification segment including the requirements seen by all three of the design components. In the above example, the design fragment would have to consider the entire EBD (thus it considers interactions, states, which is where satisfiable sets and unique events are defined, and the transitions, which are carried out by

Figure 4.9: Refining a Specification to Match a Basic Model Design

EBD and EI interactions: i.e., details of the interconnections, the interface, and engine), and would present a design solution for all transition antecedents including the events which are effectively grouped into a single unique event.

4.2.4 Design Fragments – The Setup

This section discusses the design of the ACM and its interface, and the requirements of the basic model (i.e., corresponding to that of Figure 4.4) after which the details of the set of matching design fragments can be explained in Section 4.2.5.

4.2.4.1 The ACM

The ACM and its environment are fixed designs in this approach. The ACM component encapsulates the application-specific behaviour defined by an EBD. It uses data of the actor to evaluate conditions and to define satisfiable event sets, and it provides information through a fixed event/action interface as follows.

The ACM – Conditions

To correctly evaluate conditions, the ACM accesses a global parameter store in the actor where all of the necessary data is stored.

The ACM – Satisfiable Event Set

To define the satisfiable event set, a set of global boolean parameters are maintained by the actor, and assigned values by the ACM. Each boolean represents the satisfiability of every unique event to be received. The values are *true* if there exists an outstanding receive-event primitive.

The ACM – Event/Action Interface

The event/action interface between engines and ACMs¹¹ must allow for the progression of the behaviour descriptions. An ACM must: (1) evaluate conditions, and thus condition parameters are made accessible to the ACM as described above, (2) initiate operations, (3) initiate an event send, and (4) initiate an event reception for a given satisfiable event set which is defined by shared boolean parameters as

¹¹An engine invokes the ACM by passing it an *event*, to which the ACM responds with an *action*.

Events sent to the ACM:

<i>init</i>	Initialize the EBD: transition out of the ASM start state.
<i>completed(OP)</i>	The requirements of operation, <i>OP</i> , have been satisfied.
<i>received(E)</i>	Satisfiable event, <i>E</i> , has been received.
<i>next.action</i>	The requirements of the last <i>send(E)</i> action have been satisfied.

Actions returned by the ACM:

<i>send(E)</i>	Send all events in list <i>E</i> .
<i>start(OP)</i>	Start operations <i>OP</i> .
<i>receive</i>	Receive an event from the satisfiable event set.

Figure 4.10: Initial Event/Action Interface for Agenda Design

discussed above. The actual primitive functions (i.e., perform operation, send event, and receive event) are all indivisible, and can therefore be implemented outside of the ACM if appropriate events and actions are defined such that the actor engine can carry out the required behaviour.

For the sake of modularity, coupling, and cohesion, operations (representing functional requirements) are implemented outside of the ACM as individual buttons called by engines in response to a corresponding action declared by the ACM. This design requires a *start(op)* action and *completed(op)* event in the engine-ACM interface.

To initiate a send event primitive function, a *send(E)* action is introduced, and a *next.action* event is sent to the ACM on completion of the send requirements. To initiate a receive event primitive function, a *receive* action is returned by the ACM, and the satisfiable event set is defined by the set of corresponding boolean parameters. In response, the ACM is sent a *received(E)* event.

The event/action interface is completely defined in Figure 4.10. The events are sent to the ACM to trigger a response: an *action*. In the case of an *init* event, the ACM might: (1) start a transition operation (*start(OP)*) which would, in turn, require a *completed(OP)* event, (2) send a system event (*send(E)*) which would eventually result in a *next.action* event, (3) start a state operation which is managed in

the same manner as the transition operation, or (4) receive a system event (*receive*) which eventually (assuming that there is no deadlock in the specification) results in a *received(E)* event where *E* is a member of the current satisfiable event set. In the case of a *received(E)* event, the ACM might evaluate a condition based on message data before issuing one of the actions listed for the *init* event. In the case of a *completed(OP)* or *next_action* event the ACM might evaluate conditions before issuing a *receive* action, or one of the other actions listed as a response to the *init* event.

4.2.4.2 The Basic Model

The basic model partially satisfies the properties of the EI. It covers all of the requirements of the actor engine and interface, but not the interconnections between actors. For the requirements of the engine, the basic model must define a suitable default agenda for the actor which: (1) uses the encapsulated ACMs to determine the behaviour, (2) pushes the operation buttons for the functional requirements in response to a *start(OP)* action, (3) initiates inter-actor send events in response to corresponding *send(E)* actions, (4) initiates inter-actor receive events in response to corresponding *receive* actions, and (5) indicates where operations and intra-actor interactions are occurring. For the requirements of the interface, the basic model must define a unique gate for every unique event defined by the encapsulated EBDs. This is the first basic model discussed in Section 4.2.3: whose components are shown in Figure 4.4.

The interactions between EBDs are satisfied by: (1) *inter-actor* design fragments which are plugged into the basic model design (*IPC*).¹² or (2) *intra-actor* design fragments which establish portions of the basic model for actors with co-allocated EBDs by overriding the default design defined by the basic model (*SEQ*).¹³ A basic model must provide placeholders at which the design fragments can be invoked as the result of a request from the EBD. This request represents the initial action of the specification segment covered by the design fragment. As discussed in Section 4.2.3,

¹²IPC is a common abbreviation for inter-process communications, and is used here to represent inter-actor communications.

¹³The SEQ abbreviation represents the sequential behaviour of co-allocated EBDs.

the placeholders must be consistent with the design fragments used to design the interactions between EBDs. For this basic model (as introduced in Section 4.2.3 and Figure 4.4) there is: (1) a *send_placeholder* for every *send(E)* action which can be returned by an ACM and that corresponds to an inter-actor communication, (2) a *receive_placeholder* for a corresponding inter-actor *receive* event which can be initiated by the encapsulated EBD, (3) an *action_placeholder* for every *start(OP)* action, and every *send(E)* or *receive* action associated with intra-actor communications, and (4) an *interaction_placeholder* for every intra-actor interaction. The *send_* and *receive_placeholder*s are used by the *IPC* design fragments which satisfy the requirements of inter-actor communications (i.e., the interactions). The *action_* and *interaction_placeholder*s are used by the *SEQ* design fragments which are applied when designing co-allocated behaviours.

Send Event Requirement: If an EBD has reached a send primitive operation (and thus supplied the appropriate *send(E)* action), then this send must be executed eventually by the basic model of the actor. The location of this invocation is called the *send_placeholder* for inter-actor communications, or an *action_placeholder* for intra-actor communications.

Receive Event Requirement: If an EBD has reached a receive primitive operation (and thus supplied the appropriate *receive* action), then this receive must be executed eventually by the basic model of the actor. For inter-actor communications, the location of this invocation is called the *receive_place*, while the location of each event reception is called the *receive_placeholder*. The *receive_place* represents the implementation of the queueing, and acceptance of the events in a satisfiable event set (i.e., it is the pull fingers in Figures 4.4 to 4.7). The *receive_placeholder* represents the implementation of each uniquely receivable event (i.e., the gates of Figures 4.4 to 4.7). For intra-actor communications, the *receive* action is represented by an *action_placeholder*.

The design fragments define the MCL (MachineChart Language [23]) code to be expanded at the appropriate placeholder. For example, an actor sending a message using a jab finger places the MCL code: *push msg* (*msg* will have been bound to

the application-specific term)¹⁴, at the corresponding *send_placeholder*. The actor receiving a message through a gate, opened by pulling on a *choice* bar (i.e., *pull choice*) at the corresponding *receive_place*, completes the semantics of the interaction by immediately closing the gate and pushing the ACM with the *received(E)* event at the corresponding *receive_placeholder*. The returned action is processed by the actor's engine; possibly defining another placeholder.

The placeholders are part of the actor engines; e.g., the push and pull operations initiated by the EBD encapsulated within the ACM are carried out by the engine using the design fragments. The design fragments assume that the basic model: (1) uses the ACMs of all EBDs allocated to the actor, (2) includes or references buttons/boxes for all of the functional requirements defined by the EBDs allocated to the actors, (3) supports the progression of the behaviours by providing unique placeholders for every unique send- or receive-event primitive operation, and (4) satisfies the queueing properties defined by the EBD specified events and their corresponding EI directives.

An example basic model design is shown in Figure 4.11. It satisfies these requirements for an EBD specified with the default EI directives, and one which is allocated to its own actor. In Figure 4.11 the engine responds to the actions returned by the ACM as follows. When the action *send(E)* is returned, the engine initiates, in order, the send primitive operation within the *send alternatives case* for each event in the list *E*. Therefore, the case statements are the *send_placeholders*. The contents of this case statement (e.g., for event *e1*: *when e1 => push e1_gate*), are properties of the chosen *inter-actor* design fragments as discussed in Section 4.2.5. This MCL code is placed at the appropriate *send_placeholder*, therefore for a "jab" solution, the case statement would be: *when E => push E_gate*. The *pull choice* MCL statement is performed when the ACM returns a *receive* action. It represents the *receive_place* for every satisfiable event set (i.e., where the "FAIR" selection policy is satisfied). Unique gates (one per unique event) represent the *receive_placeholders* for each unique event, and the position at which these events are queued FIFO. The

¹⁴MCL works with a MachineChart diagram to minimize redundancy. Therefore the jab semantics are implied by the diagram, and push is an MCL term used with all fingers.

```

Engine MCL:
engine basic_model is
event : (init.next_action.completed(OP).received(E));
action: (send(E).start(OP).receive);
begin
  event = init; push ACM;
  loop
    case action is
      when send(E) => for each element in list E
        - This is referred to as: send alternative case.
        case event is
          when e1 => push e1_gate; - send_placeholder for e1
          ...
          when others => ;
        end case;
      end for;
      event = next_action; push ACM;
    when receive => - This is referred to as: receive_place.
      pull choice;
    when start(OP) => case OP is
      - This is referred to as: operation case.
      when op[i] => push op[i]; - action_placeholder
      when others => ;
    end case;
    event = completed(OP); push ACM;
  end case;
end loop;
end basic_model;

```

Figure 4.11: Solution to Satisfy Assumed Basic Model Design

statement *pull choice*, implements a *wait OR* (a selective accept in Ada) with all of the open gates crossing the *choice* bar. Gates are opened and closed according to the boolean value of the waiting place which, are defined by the boolean parameters representing the satisfiable event set. This finger is also attached to a timeout component whose timeout parameter is defined by the ACM.

When more than one EBD is allocated to an actor, the requirements of a basic model have deeper implications. First, the engine of the basic model uses **several** ACMs to satisfy the requirements. Design fragments for specification segments with co-allocated behaviours define parts of the basic model for the actor. Thus, a significant part of proving these *intra-actor* design fragments is establishing that they satisfy the requirements of the selected basic model. Second, the model must consider the progress of all EBDs sharing the same actor and engine.

With co-allocations, it is possible that the entities not involved in an outstanding *inter-actor* communication might still have behaviour requirements which can progress (at the time that an interaction is designed to take place: i.e. at the `send` or `receive_placeholder`). This is not the case for a one-to-one allocation of entities to actors because each entity is capable of executing only one behaviour pattern at a time. The semantics of the EBD and EI interactions are based on client-server relations where the EBD always waits for acknowledgment from the EI before continuing with the next primitive operation. This waiting must not block potentially executable behaviour (by definition of the `send` and `receive` event requirements of the basic model). Therefore, once initiated, an inter-actor `send/receive` must complete if any other EBD allocated to the same actor is, at that time, capable of executing any further primitive operations.

This property can be satisfied by the inter-actor design fragments (e.g., non-blocking sends), or by the system specification (i.e., application-specific proofs guaranteeing non-blocking semantics). For example, if an asynchronous inter-actor `send` is required, and the `jab` design fragment is used, then the `send` is guaranteed to complete. The inter-actor design fragments for a `receive` operation often depend on the sender to guarantee completion (because they are blocking `receive` alternatives), and therefore most inter-actor design fragments will require application-specific proofs (to illustrate that the corresponding `send` requirement is guaranteed to occur eventually). Design fragments for co-allocated specification segments are discussed in detail in Section 4.2.5.2. The next section gives details on design fragments for inter- and intra-actor communications.

4.2.5 Design Fragments — A Detailed View

Design fragments offer potential alternative solutions for specification segments, and either: (1) define a basic model, or (2) are fit into an actor's basic model at appropriate placeholders with possible consequences at other placeholders (e.g., some placeholders become redundant). In the first case, the design fragments are called *actor-level* fragments, and in the second case, they are referred to as *segment-level* fragments. The designer selects potential solutions by matching generic specification patterns associated with the fragments to the requirements found in a particular specification. As such, the design fragments framework includes potential solutions for the following.

Individual Primitive Operations – In these segment-level fragments, the send and receive primitive operations are considered at the entity level. Each design fragment includes (at least) the design contingency that the corresponding and outstanding receiver or sender properties of the interaction be satisfied. Therefore entities can be designed independent of the other components of the system and plugged together in such a way that the outstanding contingencies are satisfied (*matched*). The design process is being approached entity by entity.

Individual Interactions – In these segment-level fragments, the sending and receiving of an event are considered together to build the design, interaction by interaction.

Patterns of Interaction – Standard sequences of primitive operations (e.g., a client/server scenario) are considered to define representative design fragments with reduced proof requirements. Certain patterns are shown to have properties for which the design fragment is guaranteed to satisfy some requirements, therefore the proofs required by the designer are simplified by restricting them to the application-specific requirements. These segment-level fragments are fit into the basic model at the starting send_ or receive_placeholder. The remaining placeholders defined by the specification segment become redundant and

are removed. These fragments also include states, and thus the requirements of the satisfiable event sets, and specific send-event directives.

Collapsed Behaviours – Collapsed behaviours are known to be deterministic: i.e., they are guaranteed to complete eventually. This notion can be used to simplify the proofs of certain design alternatives. These behaviours also represent a form of modularity within the specification which should be allowed to remain separate if the designer should choose a similar structural breakdown. For example, collapsed behaviours are often externally coupled. These behaviours generally have a higher probability for modification which is one typical criteria for modularity. The corresponding design fragments are segment-level fragments.

EBDs – Basic model designs cover the whole EBD or a collection of co-allocated EBDs. Design fragments for EBDs therefore, establish basic models for actors; and as such, are called actor-level fragments. For example, the design shown in Figure 4.11 is a basic model design fragment for a single EBD, with any arbitrary behaviour description, which is allocated to its own actor.

The design fragments satisfy the semantics of the specification segment as defined by the meta-model of Section 3.2, where the semantics of a single event interaction are detailed in terms of the design features in Tables 4.1, 4.2, 4.3 for asynchronous, synchronous, and transient broadcast events respectively. These tables include parenthetical comments with terminology from the event/action interface of the ACM described in Section 4.2.4 and detailed in Figure 4.10.

The choice of a fragment depends on several factors, but primarily on the allocation of EBDs (as encapsulated ACMs) to actors:

- if two (or more) EBDs are allocated to different actors, then *inter-actor fragments* (IPC) are used.
- if two (or more) EBDs are co-allocated (allocated to the same actor), then *intra-actor fragments* (SEQ) are used;

Asynchronous Send	Asynchronous Receive
<ol style="list-style-type: none"> 1. Once initiated (by the ACM action <i>send(E)</i>), an asynchronous send must eventually complete, and either: (1) continue with any outstanding sends defined by the <i>send(E)</i> action, or (2) allow the ACM to sequence to the <i>next</i> primitive operation by means of the ACM event <i>next_action</i>. 2. The event being sent must be made available for reception in accordance with the corresponding EI directives. 	<ol style="list-style-type: none"> 1. When an event is made available for reception, it must be kept in this state until it is received by the ACM. 2. Events which are available for reception and part of the satisfiable event set must be considered for response to a <i>receive</i> action initiated by the ACM in accordance with the associated EI directives. 3. Events which are part of the satisfiable event set, but which are unavailable for reception, must block the ACM from taking the corresponding transition. This does not constrain another satisfiable event from being received.

Table 4.1: Semantics of Asynchronous Interactions in Design Terms

As mentioned earlier, the solutions are often contingent on other design decisions (e.g., the design fragment used for the matching end of an interaction, and the design of the actor's engine and interface), and/or features of the specification which are application-specific. These *contingencies* are defined from *lemmas*¹⁵ associated with a design fragment. As design fragments are created, lemmas are defined to represent the properties of the specification which cannot be guaranteed implicitly by the design fragment and the required properties of the coordinating design components. Contingencies express the lemmas in terms of the components of the design fragment, and must be proven satisfiable to ensure that the fragment conforms with the specification.

¹⁵Lemmas are propositions which, for the purpose of a design proof, are assumed at the level that they are introduced, and demonstrated eventually

Synchronous Send	Synchronous Receive
<ol style="list-style-type: none"> 1. Once initiated (by the ACM action <i>send(E)</i>), a synchronous send is eventually completed after the event has been received by the destination ACM. At this point, the design must either: (1) continue with any outstanding sends defined by the <i>send(E)</i> action, or (2) allow the ACM to sequence to the <i>next</i> primitive operation by means of the ACM event <i>next_action</i>. 2. The event being sent must be made available for reception in accordance with its EI directives. 	<ol style="list-style-type: none"> 1. When an event is made available for reception, it must be kept in this state until it is received by the ACM. 2. Events which are available for reception and part of the satisfiable event set must be given considered for response to a <i>receive</i> action initiated by the ACM in accordance with the associated EI directives. 3. Once received, the sender must be released eventually. 4. Events which are part of the satisfiable event set, but which are unavailable for reception, must block the ACM from taking the corresponding transition. This does not constrain another satisfiable event from being received.

Table 4.2: Semantics of Synchronous Interactions in Design Terms

Lemmas can be established by comparing a generic version of a specification segment with a corresponding and explicit EBD representation of the design alternative. For example, Figure 4.12 shows a specification segment with an asynchronous send beside a specification segment which represents the "rendezvous" design solution discussed in the previous section. That is, if the (parameter and functionless) rendezvous design fragment were translated directly to an EBD specification segment, it would appear as shown on the right-hand side of Figure 4.12. Lemmas state the properties of the specification segment which are not implicitly satisfied by the design segment. In the example, the semantics of the specification state that when the sender entity has completed the operation at state *I*, it transitions to state *J* sending out the asynchronous event *rr-msg*. The semantics of the design are that when the sender entity has completed the operation at state *I*, it transitions to state

Transient Broadcast Send	Transient Broadcast Receive
<ol style="list-style-type: none"> 1. Once initiated (by the ACM action <i>send(E)</i>), a transient broadcast send must eventually complete, and either: <ol style="list-style-type: none"> (1) continue with any outstanding sends defined by the <i>send(E)</i> action, or (2) allow the ACM to sequence to the <i>next</i> primitive operation by means of the ACM event <i>next_action</i>. 2. The event being sent must be made available for reception at all channels where reception is potential in accordance with its EI directives. 	<ol style="list-style-type: none"> 1. When an event is made available for reception it is received if: (1) there is currently an outstanding ACM <i>receive</i> action, and (2) the event is part of the satisfiable event set. In all other cases, the event is discarded. 2. Events which are part of the satisfiable event set, but which are unavailable for reception, must block the ACM from taking the corresponding transition. This does not constrain another satisfiable event from being received.

Table 4.3: Semantics of Transient Broadcast Interactions in Design Terms

Figure 4.12: Defining Contingencies by Equating Visible Behaviours

J when it and *rx* synchronize on *msg*. The transition is guaranteed for the specified asynchronous event send, but not for the designed synchronous event send where there is an added constraint of synchronization. Thus the design fragment is based on a lemma which requires the transition (the event send) to eventually complete.

The contingency expresses this lemma in terms of the design, and provides a means to prove conformance between the specification and the design. In the example, both transitions occur if *rx* is always ready to receive *msg*, when *msg* has been sent. More generally, both specifications transition if *rx* is guaranteed to eventually receive *msg*. If *rx* eventually receives *msg*, then a synchronization will eventually occur, and the transition is eventually taken. Therefore, under this condition, the design alternative is applicable. The design represents a constrained version of the specified behaviour: one which requires reception of *msg*. This design alternative is dependent on the application behaviour; and, as such includes a *specification contingency*.¹⁶

Design contingencies are defined to ensure conformance with the requirements of the basic model design concept. They express the properties of the fragment which are required for correct implementation, but which cannot be mechanically bound given the specification segment which is represented. For example, if a design fragment covers only the sending side of an interaction, then a design contingency exists to state that the receiver must be a compatible match for this design. Other properties which are not specified (e.g., timing and criticalness) are defined subjectively as *Performance Issues*.

The framework is populated by: (1) defining a potential solution for some recognizable specification pattern including the placement of the design fragment in terms of basic model, placeholders, and the manipulation of placeholders, (2) proving this relative to the semantics of the meta-model and the definitions of Table 4.1, 4.2, and 4.3,¹⁷ (3) defining the design- and application-specific (specification) contingencies from the required coordinating design and the lemmas introduced in (2) above, and (4) assessing and highlighting the inherent properties of the fragment which might be of interest to the designer (labelled *Performance Issues* in this thesis). IPC' and

¹⁶The design is a *reduction* of the specification (cref. § 2.4). In general, the reduction is not local to the specification segments, but rather a refinement of the system behaviour. Therefore, the contingency can be proven only under consideration of the entire system behaviour; e.g., through analysis of a graph of reachable states for the system specification.

¹⁷This proof is done once before the design fragment is established. As such, this proof can be assumed when the design fragment is used correctly and with contingencies satisfied.

SEQ design fragments are illustrated through example in the sections which follow.

4.2.5.1 Inter-Actor (IPC) Design Fragments

The first example considers an inter-actor design fragment for an asynchronous interaction (labelled DF-AsyncInter-IPC-1), and is created as follows.

1. Defining a Potential Solution

Asynchronous Interaction – DF-AsyncInter-IPC-1. An asynchronous interaction is characterized by a transition in the sender entity which matches:

any antecedent/any transition operation/receiver-message

and a corresponding transition (with the same message) in the receiver entity: i.e., matching:

sender+message/any transition operation/any transition consequence

An appropriate (inter-actor) design fragment for this interaction combination is shown in Figure 4.13. It is labelled DF-AsyncInter-IPC-1 to identify it as: Design Fragment for Asynchronous Interaction of Communicating Actors Number One, and it represents the “jab” design fragment proposed in the previous section.

Figure 4.13 provides an illustration of the generic specification segment for the design fragment. The designer matches these with segments of the EBDs in the application. (A double hyphen (--) in the generic specification matches any term in the application.) If a match exists then the designer adds the fragment to the existing design and should prove (demonstrate) the contingencies associated with the fragment. In this example there are no specification contingencies, and no design contingencies when added to a basic model design.¹⁸

¹⁸Recall that a basic model is an engine and interface design which emulates the semantics of an EI with unique send_ and receive_placeholders (cref § 4.2.3 and Figure 4.4).

DF-AsyncInter-IPC-1

Performance Issues:

1. "msg" is buffered at "rx". Delays or losses are possible if buffer limitations are stressed.
2. Otherwise, "msg" is guaranteed to complete.

Figure 4.13: Design Fragment for an Asynchronous Interaction

The design fragment consists of those portions drawn using solid lines and the associated *MachineChart Language* (MCL). In this example, the MCL for the finger is *push msg* which completes the design of the *send_placeholder*. The term, *msg* is one of the generic terms of the specification segments, therefore it will be bound to the corresponding term used in the specific application. All gates in MachineCharts are defined

by *critical* (in rendezvous) and *post visit* (post rendezvous) sections. Any gate which accepts a jab, must have a null critical section (by definition: i.e., there is no rendezvous with a jab). Therefore, the gate associated with this event (i.e., the *receive_placeholder* for the event) is defined by a null critical section with a post visit sequence which informs the ACM of the event reception to trigger a return action which is handled by the engine of the basic model. The variable, *event*, is read by the ACM after the ACM is *pushed* (called) by the gate; and, in this example, the event is *received(msg)*.

The dashed lines of the MachineChart drawing in Figure 4.13 (e.g., the actor and ACM) are components which are defined by the *basic model* (see Section 4.2.4).

2. Proving a Potential Solution Relative to the EBD Semantics

Proof of DF-AsyncInter-IPC-1. To prove that this design fragment is applicable, all of the requirements of an asynchronous send and an asynchronous receive must be considered. An informal proof was presented in Section 4.2.2 for the “jab” implementation where it is shown that all properties are satisfied: i.e., the proof was completed without introducing any lemmas when a basic model implementation is provided for the actor’s engine and interface. Q.E.D.

3. Defining Design and Specification Contingencies

Because all properties of an asynchronous send were shown to be satisfied by the design in step 2, there are no lemmas and thus no specification contingencies to associate with this design fragment. The design fragment is to be plugged into a basic model design.

4. Assessing the Properties of the Design Fragment

In general, the sender side of this design fragment is guaranteed to complete. However, the design requires message buffering, and therefore delays or losses are possible if buffer limitations are stressed. This is

stated as a *Performance Issue* for this design fragment.

A second example will define DF-AsyncInter-IPC-2 which represents the rendezvous solution presented in the previous section.

1. Defining a Potential Solution

Asynchronous Interaction – DF-AsyncInter-IPC-2. The design fragment for the rendezvous solution to an asynchronous interaction is shown in Figure 4.14. The generic specification segment is the same as in DF-AsyncInter-IPC-1, however, the MachineChart shows a push finger as opposed to a jab finger. This push finger is plugged into the basic model of the sender actor at its corresponding send_placeholder, and into the receiver actor at the corresponding receive_placeholder.

2. Proving a Potential Solution Relative to the EBD Semantics

Proof of DF-AsyncInter-IPC-2. To prove that this design fragment is applicable, it is necessary to determine the conditions under which the semantics of the specification segment to be designed, and the semantics of the design are equivalent. This example was discussed in the introduction to this section. The comparison of the specification segment and the design specification found that the transition is not implicitly guaranteed to occur in the design, but is in the original specification.

For this example, the same conclusion can be derived by comparing the requirements of an asynchronous interaction with those of a synchronous interaction as they are defined in Tables 4.1 and 4.2. This proof was presented informally in Section 4.2.2 where it was shown that the first property of the asynchronous send (requiring the send to complete eventually) is not implicitly guaranteed by this design fragment. All of the other properties (2 through 5) are satisfied when the fragment is used with a basic model. Therefore this design fragment introduces a lemma for property 1; i.e., the solution is valid if the associated contingency can be proven. Q.E.D.

DF-AsyncInter-IPC-2

Specification Segments

Spec. Contingencies:

1. "tx+msg" must be received eventually by "rx".

Design Fragment

Design Contingencies:

1. Requires the basic model defined in 4.2.4.

tx @send_placeholder for tx-msg:
push msg;

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg);
push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

Figure 4.14: Second Design Fragment for an Asynchronous Interaction

3. Defining Design and Specification Contingencies

To satisfy the lemma introduced above, the receiver must always eventually receive the message. Therefore the designer should show that the message, once sent, is received eventually by the receiver. The generic description of the corresponding specification contingency is: *tx+msg* must be received eventually by *rx*. This design fragment is also depen-

dent on a basic model design for the actor's engine and interface.

4. Assessing the Properties of the Design Fragment

Senders are delayed awaiting the rendezvous. This is described generically by the statements: (1) *tx* blocks until *rx* acknowledges that the message has been received, and (2) *tx* is delayed by *rx*.

A third example will define DF-AsyncInter-IPC-3 which represents the courier solution presented in the previous section. This example illustrates the importance of capturing states in a specification segment (and thus the satisfiable event set requirements) as well as transitions when an actor's interface design is altered from its default.

1. Defining a Potential Solution

Asynchronous Interaction – DF-AsyncInter-IPC-3. A design fragment for a courier solution to an asynchronous interaction is shown in Figure 4.15. If the design fragment satisfied the same specification segment as DF-AsyncInter-IPC-1 and DF-AsyncInter-IPC-2, then the contingencies and properties would be equivalent to those of DF-AsyncInter-IPC-2. In fact, nothing is gained by adding the courier task in this case. Rather, a courier is typically used when the sender entity acts as a server for relatively simple requests, and/or forwards requests from one source to several entities analogously to a dispatcher. As such, the entity can await a request, service the request, and return to await the next request including a push from the courier to pass on a previously identified, asynchronous message send. The example specification is as shown in Figure 4.15 and again in Figure 4.16. Because this segment covers the properties of an entire EBD, it could also be referenced by, for example, DF-ServerEBD-1; i.e., multiple labels are used for the same fragment to support the search for fragments when starting with alternative views (asynchronous message send versus a server or dispatcher EBD). The entity is shown to be waiting on a set of incoming events. Each event reception initiates a transition which is guaranteed to eventually return

DF-AsyncInter-IPC-3

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. Requires the basic model defined in 4.2.4.

tx @send_placeholder for rx-msg:
insert(msg.msg_buffer): get_msg := true;

tx@receive_place for wait state:
include get_msg in the choice frame..

add get_msg gate:
begin critical section
 msg := dequeue_head(msg_buffer);
end critical section;
if is_empty_queue(msg_buffer) then
 get_msg := false;
end get_msg;

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical section, and post visit statements:
 event = received(msg);
 push ACM;

courier task:
engine is
begin
 loop
 push get_msg;
 push msg;
 end loop;
end courier.

Performance Issues:

1. "rx" must receive "msg" and release "courier" often enough to avoid buffer overflow at "msg_buffer"
2. size of "msg_buffer" is dependant on the rate of reception of "msg" by "rx".
3. "rx-msg" can be delayed by any of the "N" paths leading from "wait state".

Figure 4.15: Third Design Fragment for an Asynchronous Interaction

to the initial *wait state*. The transition whose consequence, an asynchronous event send, is to be designed using a courier actor, is shown separate from the other loops. In order to guarantee correct ordering of events between *tx* and *rx*, *rx* must not receive any other events from *tx*.

The design alternative adds a courier actor to pick up *msg* from *tx* and transport it to *rx*. This design requires the addition of the *get_msg* gate to the *tx* actor. The gate is to be pulled whenever *tx* is effectively in *wait state* and if there is a *msg* message waiting to be sent to *rx*. The *rx-msg* send_placeholder is used to buffer *msg*, and open the *get_msg* gate by setting the *get_msg* waiting place to *true*. The receive_place for the *wait state* satisfiable event set is altered to include the *get_msg* gate in its choice frame. The *get_msg* gate is defined to access the head of *msg_buffer* and return *msg* as outgoing data in the reply to *courier*. The *rx* actor is typical of all actors receiving asynchronous messages as discussed so far.

2. Proving a Potential Solution Relative to the EBD Semantics

Proof of DF-AsyncInter-IPC-3. In this specification segment *Wait State* has *N* (where *N* is some integer) self-looping paths, one of which explicitly matches a transition with an asynchronous message send (i.e., *any receive event/--/rx-msg*). In the solution, the *msg* message is buffered at the *get_msg* gate of the *tx* entity before the entity continues back to *Wait State* where the *get_msg* message is now part of the satisfiable event set. The original specification and the equivalent specification for the design solution are shown in Figure 4.16. The design change is the original transition such that the message buffering is done internally to *tx*, and the *get_msg* gate is opened. The event send is not performed until an alternative transition, initiated by the *courier?get_msg* event, is taken. During this transition, *msg* is dequeued, and sent to *courier* in a reply event, *reply_msg(msg)*. This transition illustrates the rendezvous semantics of the *get_msg* gate shown in the design.

Original Specification:

any receive event/--/optional outgoing asynchronous communication (not to rx)

Equivalent Specification
for Design with Courier:

any receive event/--/
any outgoing asynchronous
communication (not to rx)

any receive event/--,
insert(msg,msg_buffer,get_msg := true/
(matching the transition
antecedent and transition operation of tl
separated transition shown above)

courier?get_msg[get_msg = true]/
dequeue_head(msg_buffer),
evaluate get_msg/courier!reply_msg(msg)

Figure 4.16: Behaviour of a Solution uses a Courier

The specified send is completed by this design as long as the internal buffer at *get_msg* is large enough to hold all requests, or if there is a guarantee that the courier will pick up the messages such that the designed buffer will not overflow (property 1 of an asynchronous send as defined by Table 4.1). The message is made available by the courier at its outgoing finger in the first instance of the event send: i.e., the courier picks up the message at *get_msg* and provides the message to the *msg* gate of the receiver (property 2). Subsequent event sends are made available by the courier if the receiver eventually receives the couriered message, and thus releases the courier allowing it to pick up and deliver the next message buffered at *get_msg*. These messages are otherwise available at *msg_buffer* (unless losses occur because of overflow). Therefore the rate of reception by the receiver entity affects the required buffer size at *get_msg*. The components of the receive are equivalent to DF-

AsyncInter-IPC'-2 which satisfies properties 1 to 3 of the asynchronous receive.

This design fragment encompasses a state (*Wait State*), and thus the requirements of the satisfiable event set. Therefore, the corresponding *receive_place* must be altered along with the definition of the *send_placeholder* for the asynchronous send. In this case, the *send_placeholder* buffered *msg*, and set the boolean *get_msg*, which is used to open and close the gate, to *true*. A new gate is added (i.e., *get_msg*), and the *choice* bar of the *wait state* *receive_place* is extended to include the *get_msg* gate. The basic model is manipulated by the design fragment, and it is the definition and contingencies of the design fragment which must establish the terms under which the changes are valid. As designed, the *receive_place* still provides a fair selection of the events in the original satisfiable event set. Q.E.D.

3. Defining Design and Specification Contingencies

There are no contingencies other than the requirement for a basic model. The concerns over the *msg_buffer* size are properties of the design fragment just as the queue at the *msg* gate of DF-AsyncInter-IPC'-1 was a performance issue for that design fragment.

4. Assessing the Properties of the Design Fragment

In order to avoid buffer overflow at *msg_buffer*, the receiver must receive the message *msg* often enough. The size of *msg_buffer* is therefore, dependent on the rate of reception of *msg* by the receiver. It is also possible that the sending of *msg* is delayed by the functionality of the other self-looping transitions. These issues are stated in terms of the design for the design fragment as shown in Figure 4.15.

The final example for IPC' Design fragments considers a pattern of interaction which represents a client/server scenario. It is called DF-ClientServer-IPC'-1.

1. Defining a Potential Solution

The client/server scenario in question is characterized by the following specification patterns:

In entity B:

send asynchronous event i to entity A / wait state / receive asynchronous event j from entity A

In entity A:

receive asynchronous event i from entity B / perform operation / send asynchronous event j to entity B

Here, *perform operation* must complete eventually; i.e., one entity (the client) is waiting while the other (the server) executes an operation guaranteed to eventually complete. The proposed design fragment, DF-ClientServer-IPC-1 is shown in Figure 4.17.

In DF-ClientServer-IPC-1, the interactions are designed using a rendezvous implementation. The client initiates a rendezvous with the server during which the operations of the server are performed. For this design fragment, the interaction is satisfied by the MCL statement (*push request_ack*) made by the client, and the MCL statements within the critical section of the *request_ack* gate of the server. (Note that *push ACM* instantiates the value of the variable *action*, however, in this example the value is defined by the specification segment, and thus known, before it is supplied by the ACM.)

A reply is issued only after the operation requirements are satisfied. The reply is implicit in the *end* statement of the critical section of gate *request_ack*, which releases the client to continue its processing. Once released, the client has effectively received the *ack* message from the server, and this event is passed on to the ACM.

DF-ClientServer-IPC-1

Specification Segments

Spec. Contingencies:

1. "wait_ack" must be a null operation with satisfiable set: $\{server+ack\}$
2. "operation" must complete eventually.

Design Fragment

Design Contingencies:

None.

client @send_placeholder for server-request:
 push request_ack;
 event = next_action; push ACM;
 event = received(ack);
 push ACM;

server @receive_placeholder for client-request:
 server gate request_ack is:
 be in critical section
 event = received(request);
 push ACM;
 push operation; -- button not shown
 event = completed(operation);
 push ACM; -- absorb client-ack action
 end;
 event = next_action; push ACM;
 end.

Figure 4.17: Inter-actor Client/Server Design Fragment

3

1010a-1 PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

2. Proving a Potential Solution Relative to the EBD Semantics

An informal proof of this solution is as follows. In this potential solution, the interactions are designed using a rendezvous implementation: i.e., entity B initiates a rendezvous with entity A during which the operations of A are performed. A reply is issued only after the operation requirements are satisfied. For asynchronous messages the message send, initiated by entity B, is guaranteed to eventually complete since it does so at the end of the rendezvous. B waits for an acknowledgment of completion of the service which is satisfied by the reply of the rendezvous. Since this is the only event in the satisfiable set, it represents a fair selection. The arrival time is deterministic, therefore the queuing requirements are satisfied, and entity B is blocked until this reply is received.

The receiver model (entity A) matches the standard receiver design shown in several of the design fragments already (e.g., DF-AsyncInter-IPC-1) with one exception being that the critical section now contains calls to execute the operations of the scenario. The receiver's send operation is satisfied by the reply which is guaranteed to complete because it's receiver (the client) is waiting for this interaction. Q.E.D.

3. Defining Design and Specification Contingencies

To match the pattern of interaction defined for this design fragment, the client entity must be in a wait state after the initial request event and before the final reply event. For this wait state, there can be only one satisfiable event; i.e., the reply event. The operations performed by the server must complete eventually. These requirements are listed as specification contingencies because operations in a design fragment can be bound to rather complex specification segments;¹⁹ and, as such, do not necessarily define trivial pattern matching. Otherwise, no lemmas are introduced.

¹⁹ Any segment, or disjunction of segments which complete eventually can be bound to an operation of a design fragment.

4. Assessing the Properties of the Design Fragment

This design represents an explicit emulation of the requirements given that the pattern matching is correct; and therefore, no properties are defined for this fragment.

Example Design Using IPC Design Fragments: Given the design fragments defined thus far, it is possible to construct the design for the Readers/Writers problem shown in Figure 4.19. The Readers/Writers specification is repeated as Figure 4.18. The design was developed with constraints to rendezvous communications, using DF-ClientServer-IPC-1 and DF-AsyncInter-IPC-2 as follows:

1. The reader sequence ($s\text{-}rr; \textit{wait grant}; s\text{+}gr$) and scheduler sequence ($_r\text{+}rr\{set(r=SID)\}; C' = C + 1; r\text{-}gr$) are matched with the client/server design fragment DF-ClientServer-IPC-1 of C.2.
2. The event send, $s\text{-}rf$, made by the reader, and the receive, $_r\text{+}rf$, made by the scheduler, are matched with design fragment DF-AsyncInter-IPC-2.
3. The writer sequence ($s\text{-}wr; \textit{wait grant}; s\text{+}gw$) is matched with the client side of DF-ClientServer-IPC-1. The corresponding pattern for the server is found in the scheduler specification. In this case, the matching sequences are: (1) $_w\text{+}wr\{set(w=SID)\} // w\text{-}gw$, or (2) $_w\text{+}wr\{set(w=SID); \textit{the behaviour leading from scheduler state 2 to scheduler state 4 which includes state 3}\}; w\text{-}gw$ (see Figure 3.18). That is, there are two different and disjoint possibilities for the operation to be performed by the server (the scheduler): (1) an empty operation which occurs when the scheduler transitions directly from state 1 to state 4, or (2) the behaviour between scheduler states 2 and 4 through 3 where readers with access to the resource are allowed to complete, without accepting new readers until the writer has been serviced.
4. The event send, $s\text{-}wf$, made by the writer and the corresponding receive, $_w\text{+}wf$, made by the scheduler are matched with design fragment DF-AsyncInter-IPC-2.

Informal Specification:

A data structure in primary or secondary memory is accessed by a number of readers and writers. In general, reading and writing can occur at any time, although only one writer at a time should be allowed to write. While writing, all other readers and writers must be denied access to the memory, however, when there are no writers, any number of readers are allowed.

EBD Specification:

Figure 4.18: Readers/Writers Specification

To complete the design process, the contingencies defined by the design fragments must be proven.

For Design Decision 1 (using DF-ClientServer-IPC-1)

wait_grant must be a wait state with satisfiable set: *scheduler+gr* - This contingency is confirmed intuitively. Q.E.D.

Figure 4.19: Readers/Writers Design using Synchronous Communications

$C'' = C + 1$ **must complete eventually** — This contingency is also intuitively satisfied. Q.E.D.

For Design Decision 2 (using DF-AsyncInter-IPC-2)

$\text{reader} + rf$ **must be received eventually by scheduler** — This is a much more difficult proof since it requires an analysis of the specification to ensure that the event reception occurs given that the send is initiated. It can be argued as follows. (1) Once $s-rf$ has been issued, the reader actor is blocked (by the rendezvous) until the message is received by the scheduler. (2) As the reader must have issued and been granted a read request, eventually the scheduler must receive the $\text{r} + rf$ event (there is an assumption of correct behaviour for the EBD). (3) Once the receive is completed the rendezvous terminates, thus the send has completed and the message was received. Q.E.D.

For Design Decision 3 (using DF-ClientServer-IPC-1)

wait_grant **must be a wait state with satisfiable set: scheduler + gw** — This contingency is confirmed intuitively. Q.E.D.

$[\text{null} \mid \text{scheduler behaviour from state 2 to 4 through 3 must complete eventually}]$ — The empty operation (*null*) will complete eventually, therefore this disjunct is proven. For the second disjunct, it is necessary to prove that the behaviour can

neither deadlock nor diverge: i.e., take a path which does not lead eventually to scheduler state 4.

If the specification has been verified for temporal errors, then the designer can assume that there is no deadlock. The designer can also state, by inspection, that the behaviour does not diverge. Q.E.D.

For Design Decision 4 (using DF-AsyncInter-IPC-2)

writer+wf must be received eventually by scheduler — This argument is similar to that presented for the reader finished events. Q.E.D.

The resultant design is shown in Figure 4.19.

4.2.5.2 Intra-actor (SEQ) Design Fragments

SEQ design fragments are used for designing the combined behaviours of co-allocated EBDs. They are generally more complex than the IPC design fragments because they deal with cooperating behaviours of EBDs allocated to the same actor; and, as such, they must be proven to satisfy the requirements of a basic model (cref. § 4.2.4).

Every set of co-allocated EBDs are designed at the *actor-level* (i.e., in terms of each and every EBD allocated to the actor), by a single actor-level SEQ design fragment which establishes the format of the actor's engine design as a whole, and satisfies the requirements of a basic model. These can require complex proofs, but the format is still considered to be a design fragment because the designer is still assisted in proving the underlying design decision. The actor-level fragments define complete designs for all intra-actor communications of the co-allocated EBDs; i.e., there is a default solution for every interaction of the co-allocated EBDs.

At the level of specification segments, (which are generally distinguishable services defined by interacting EBDs, and guaranteed to complete), *segment-level* SEQ design fragments can be used, and fit into the actor-level, basic model design. These design fragments replace the default design of the basic model (i.e., defined by an *actor-level* SEQ design fragment) with an equivalent alternative (if the contingencies can be satisfied). Like their inter-actor counterparts, these segment-level SEQ design

fragments are positioned at appropriate placeholders, and can affect the implementation at other placeholders. In this case, however, the default implementation of the overall design is replaced, as opposed to being defined.

All SEQ design fragments provide placeholders for every *action* which can be returned by an encapsulated ACM (see Figure 4.10) called *action_placeholders*, and every known interaction called *interaction_placeholders*. As such, the segment-level SEQ design fragments are plugged into actor-level SEQ fragments at the appropriate placeholder. Each fragment must have a default implementation to be used if not otherwise altered by the inclusion of another segment-level design fragment. Each must also define placeholders for all of the actions and interactions covered by the fragment to allow for the addition of still other segment-level design fragments. This ensures the independence of SEQ design fragments. The segment-level design fragment must indicate how the default implementation is affected by its own inclusion; i.e., what is replaced, and what is to be removed when redundancies arise. This concept is described in more detail throughout this section by example.

SEQ design fragments must satisfy the requirements of a basic model to support the addition of inter-actor (IPC) design fragments. They, therefore require proof of the existence of *send_placeholders* and *receive_placeholders*. To ensure the independence of all design fragments, every inter-actor communication must be proven to complete eventually, or it must be shown that all co-allocated behaviours are blocked when a single outstanding interaction is initiated (cref. § 4.2.3). This guarantees that each design fragment conforms to the specification within itself, and is therefore independent of design fragments which perform communications prior to, or after the specification segment being considered by each design fragment. When a segment-level SEQ design fragment is being used, then it is necessary to prove eventual completion of the segment which generally requires the same proofs; i.e., that every inter-actor communication completes eventually.

This section introduces DF-Inter-SEQ-1, an actor-level SEQ design fragment; others are discussed in C.1.2.

1. Defining a Potential Solution

The first actor-level, potential solution for a many-to-one allocation

builds a design by abstracting a single correct schedule (order) for the requirements of the co-allocated EBDs. This can be achieved by: (1) abstracting a schedule from the reachable state graph (cref. 2.4 and 3.4), or (2) recursively considering individual, *satisfiable* interactions.²⁰ In doing so, a fair choice can be made, and one complete scenario of the combined behaviour is scheduled *a priori*. The schedule is used to define the agenda (or a portion of the agenda) for the actor's engine. This design fragment is illustrated in Figure 4.20.

The interactions between co-allocated EBDs, which are defined by the schedule, represent both the *send* and *receive* action_placeholders, and the corresponding interaction_placeholders. The default implementation for each interaction has the following properties. (1) The implementation of an interaction is to push the ACMs involved, sending the events that will advance their state accordingly. Although the returned information is known *a priori* by the schedule, it may be needed by segment-level design fragments. At the end of an actor's design, if it is found that the information returned by the ACMs is redundant (and not used), then the statements which advance the ACMs can be removed (e.g., by commenting out these statements). (2) The implementation of a *start(OP)* action is a direct push of the corresponding *OP* button (where this is the assumed design for operations within a basic model). Again, the ACMs are advanced accordingly using the *completed(OP)* event.

2. Proving a Potential Solution Relative to the EBD Semantics

This design fragment includes a single engine which executes a provably correct static schedule of the co-allocated behaviour by calling the operation buttons and initiating the inter-actor communications in a specific and correct order. When the ACM actions are required for segment-level design fragments inserted into this actor design, the ACMs are advanced accordingly as shown in C.2. This actor-level design fragment is strictly

²⁰This represents a *random walk* through the reachability states [119], and does not require a complete reachability tree.

DF-Inter-SEQ-1

Spec. Contingencies:

1. For every incoming message, "rxmsg":
Unless the entire agenda of this actor depends on "rxmsg" to initiate its processing (or processing loop), then "rxmsg" must be sent eventually when the actor is blocked on the corresponding receive_placeholder.
2. For every outgoing synchronous event: "txmsg":
"txmsg" must be received eventually by "rx".

Design Contingencies:

1. "actor" must satisfy the requirements of a basic model: specifying all send_, receive_, action_, and interaction_placeholders.
2. "behaviour engine" must define a fair static schedule for the combined behaviour of "ebd1", "ebd2", ...

Performance Issues:

1. MCL to advance ACMs through their behaviours can be removed if, when the actor's design is complete, all actions returned by the ACMs are redundant information: i.e., already pre-determined by the design.
2. "behaviour engine" defines priorities and preference of events and entities in "actor".

Specification Segments

Any co-allocated behaviour where the behaviour of all co-allocated EBDs are considered: i.e., this is an actor-level SEQ design fragment.

"behaviour engine" is a static schedule for the combined behaviours of all EBDs allocated to "actor"

MCL at each interaction_placeholder
(= receive and send action_placeholders respectively):
event=received(rxmsg); push receiver_ACM;
event= next_action; push sender_ACM;

MCL at each start(OP) action_placeholder:
push OP;
event= completed(OP); push ACM;

Figure 4.20: Actor-level SEQ Design Fragment

application dependent.

Conformance of the behavioural requirements is a result of correctly abstracting a schedule for the operations and inter-actor communications of the co-allocated EBDs. Q.E.D.

3. Defining Design and Specification Contingencies

The correct use of this fragment depends on one design contingency; i.e., a fair and correct schedule. Since the `send_` and `receive_placeholders`, and the `action_` and `interaction_placeholders` are strictly defined by the schedule, the only remaining required proofs are the specification contingencies; i.e., that no inter-actor communication blocks the progression of the co-allocated behaviour.

4. Assessing the Properties of the Design Fragment When used to establish an actor's overall engine design, this design alternative constrains the behaviour to a single scenario, and thus may not represent a typical execution of the reactive system. The schedule defines priorities and preferences of events and entities co-allocated to the actor.

Example Design Using SEQ Design Fragments: If all of the EBDs of the Readers/Writers problem are allocated to a single actor, and the satisfiable interactions are examined, then one correct schedule is as follows: (1) interact on a read request,²¹ (2) interact on a grant read, (3) read,²² (4) interact on a reader finished notification, (5) interact on a write request, (6) interact on a grant write, (7) write, (8) interact on a writer finished notification, and (9) repeat (1) to (8) forever. This schedule is correct if shown to be fair (i.e., "fair" is the default selection policy for a satisfiable event set). Fairness, in this case, is application-specific, however alternating between the two functional requirements seems fair in terms of a simulation of the system, and in view of the reachability tree of Figure 3.28. An argument that readers and writers occur equally often, and thus alternating the functions is a valid approximation, is a property of this simulation. This design is illustrated in Figure 4.21. Once a schedule is in place, and because the information returned by the ACMs is not used in this example, the implementations to advance the ACMs through the schedule are removed, giving the agenda defined in Figure 4.22. There are no external interactions, and no low level SEQ fragments have been included,

²¹This represents the `rr` `interaction_placeholder`, the `send(s-rr)` `action_placeholder`, and `receive` `action_placeholder` for the satisfiable event set containing `(.r+rr|.w+wr)`.

²²This represents the `start(read)` `action_placeholder`.

```

package engine is
begin
  event = init; push reader.ACM; push writer.ACM; push scheduler.ACM;
  loop
    - send(s-rr) action_placeholder, receive action_placeholder for (.r+rr|.w+wr),
    - rr interaction_placeholder
    event = next_action; push reader.ACM;
    event = received(rr); push scheduler.ACM;
    - receive action_placeholder for (s+gr), send(r-gr) action_placeholder,
    - gr interaction_placeholder
    event = next_action; push scheduler.ACM;
    event = received(gr); push reader.ACM;
    - start.OP(read) action_placeholder
    read;
    event = next_action; push reader.ACM;
    - send(s-rf) action_placeholder, receive action_placeholder for (.r+rf|.w+wr),
    - rf interaction_placeholder
    event = next_action; push reader.ACM;
    event = received(rf); push scheduler.ACM;

    - send(s-wr) action_placeholder, receive action_placeholder for (.r+rr|.w+wr),
    - wr interaction_placeholder
    event = next_action; push writer.ACM;
    event = received(wr); push scheduler.ACM;
    - receive action_placeholder for (s+gw), send(w-gw) action_placeholder,
    - gw interaction_placeholder
    event = next_action; push scheduler.ACM;
    event = received(gw); push writer.ACM;
    - start.OP(write) action_placeholder
    write;
    event = next_action; push writer.ACM;
    - send(s-wf) action_placeholder, receive action_placeholder for (.w+wf),
    - wf interaction_placeholder
    event = next_action; push writer.ACM;
    event = received(wf); push scheduler.ACM;
  end loop;
end.

```

Figure 4.21: Readers/Writers Simulation using DF-Inter-SEQ-1

therefore there are no specification contingencies to consider, and no placeholders to specify.

A Second Example: Consider the Readers/Writer design implemented by the static schedule of Figure 4.22, but change the writer behaviour such that the transition from state 1 to state 2 is dependent on the receipt of the event, *start* from an external entity *user*; i.e., the transition changes from $//s-wr$ to $user?start//s-wr$. Every call to operation *write* is now preceded by an inter-actor receive operation. Using the static schedule design fragment (DF-Inter-SEQ-1) the interaction is added at a position which defines the immediate processing of the interaction; i.e., the sched-

```

package engine is
begin
  loop
    read;
    write;
  end loop;
end.

```

Figure 4.22: Readers/Writers Simulation Without Placeholders

ule is: *loop read; receive_placeholder for user?start;write end loop* (when action_ and interaction_placeholders are suppressed).

To prove that this is a correct design, the design must show that the progression of the behaviour is not blocked by the *start* interconnection: i.e., either: (1) the reader and scheduler are always blocked when a *start* event is expected from *user*, or (2) that the *start* messages will not block the progress of the actor indefinitely. In this case neither property can be proven; i.e., the reader and scheduler interactions can continue without any input from the writer, and the *start* message comes from an unreliable source. Therefore the design satisfies the specification only under the assumption that the *start* messages will arrive infinitely often (i.e., the *start* messages arrive deterministically).

In this example, the designer has been forced to understand the limitations of this design. The analysis of the specification contingencies illustrates the dependency on the actions of an external source.

4.2.6 Design Fragments Framework

Appendix C provides an initial population for the *Design Fragments Framework* based on the basic model for the default queueing disciplines of EBDs. Several examples are included to illustrate the use of design fragments; and, as a result, show that the proof process for contingencies can be quite complex.

4.2.7 Method for Designing with Design Fragments

A method which generates strongly traceable designs must capture and account for each design decision and the corresponding proof of correctness. This is new concept for design: i.e., current design methods account for the result and possibly (as in the case of knowledge-based tools such as the assistants) the process of building a design. As well as capturing the partial design steps, there is a need to assist the designer in: (1) the selection of design fragments, and (2) the proof of design and specification contingencies. The examples of C.1 show fairly complex proofs from rather simple descriptions of the contingencies: i.e., "behaviour must complete eventually", "design must be coupled with a synchronous receiver", or "requires guaranteed reception". A method for design must help the designer to recognize the contingencies in terms of the particular application, and offer assistance for difficult proofs. The design method presented in this thesis acknowledges several related issues:

Organization - There is a need to manage the information held in the design fragments framework, the EBD specifications, and the design and proof process.

Selection - There is a need to assist the designer in selecting design fragments to be applied in the design process.

Proving Contingencies - Proving contingencies can be a complex problem. In particular, specification contingencies can be proven formally, only through analysis of the reachable system states (cref. § 2.4 and § 3.4).

Scaling - There is a need to manage large systems requiring many actors.

This section presents some ideas on automating the development process: and, as such, addresses some aspects of the first three issues presented above. (Chapter 5 looks at assisting the designer in proving: (1) contingencies (i.e., correctness), and (2) completeness which is particularly important for large system designs.) A tool to support this design process might, for example, consist of the following components:

The Framework — The framework itself would be a component of the tool; i.e., as a library of design fragments. The fragments would contain the same in-

formation as given in Figures 4.13, 4.14, 4.15, and C.2.

A Framework Browser — A library *browser* could be defined to assist the designer in finding the potential solutions applicable to a particular problem (specification segment). The search could begin by finding a pattern within the specification segments of the design fragments, and narrowed according to characteristics such as the semantics of the implementation (e.g., with Ada, the solutions must be in terms of synchronous interactions), or performance (e.g., avoiding added overhead of extra actors such as the courier or buffer actors in DF-AsyncInter-IPC-3 and DF-AsyncRx-IPC-3).

The EBD Specification — The tool would support graphical entry of the EBD specification subject to design via design fragments. Therefore, much like the STATEMATE tool, the specification model could be entered into the tool's database.²³

Allocation — The tool would allow EBDs in the specification to be allocated to actors in the design.

Design — The tool could, given a particular design fragment identifier, set up a *Build Table* (described in the next paragraph) in which the application-specific terms of the specification are bound, by the designer, to the generic labels of the design fragment. For example, using both the EBD specification model and a *design window*, the designer could use a mouse device to select and bind the design fragment to the actual specification segment. Once this binding is completed, the tool would adjust the design and specification contingencies such that they are described in terms of the application.

Reasoning System — For contingencies which are not verified by inspection (i.e., the designer wishes to pursue a more detailed proof), a proof or reasoning

²³A tool for entering EBD specification; with the default queueing disciplines (i.e., directives are not supported) is being implemented as an addition to the TimeBench designer's workbench [24]. The tool also supports the allocation step of the design process [30].

system could be used to assist in the proof process. Within a tool, the discourse between the designer and a reasoning system could be logged for future reference along with ties between the set of descriptions provided by the user and the *Build Table* associated with the particular design fragment.

Completeness Checker — The tool would perform an automated check for completion: i.e., illustrating that all specified requirements are covered by the design.

There are three main features to the design phase which focuses on designing a system through consideration of independent specification segments (i.e., designing with design fragments): (1) the selection and application of design fragments, (2) the proof or reasoning of contingencies, and (3) the documentation of design decisions and proofs in *Build Tables*.²⁴ Every design decision (i.e., choice of a design fragment) is described and proven correct (or demonstrated) within a Build Table. Table 4.4 illustrates a *generic* Build Table for the client/server design fragment DF-ClientServer-IPC-1 defined in Section 4.2.5.1. The table is identified by the design fragment and a *design step* which is an identifier for this design decision within the design of a complete system (Build Table Reference). The table describes the specification segment of the design fragment linearly using the generic, instantiable terms found in the pictorial version of the design fragment (i.e., as given in Figure 4.17). A semi-colon (;) is used to delineate the start and/or completion of a state or transition operation (as opposed to the boundaries of a bubble or the slashes (/) of a transition). Space is provided for binding these terms to the application (i.e., the right-hand column of the "Application Bindings" section of the table), and this requires input from the designer (preferably automated to some degree: e.g., using a mouse device as mentioned earlier). Before the bindings are established, the contingencies section can be described in generic terms as found in the design fragments of Section 4.2.5 and Appendix C. Once these bindings are established, the contingencies section of the table describes the required proofs at a low level

²⁴Allocation and completeness are other design issues which are not addressed in this discussion.

Build Table Reference:**BUILD TABLE: DF-ClientServer-IPC-1**

Generic Specification Segment	
client: server-request;wait_ack;server+ack server: client+request;operation;client-ack	
Application Bindings	
client server request wait_ack ack operation MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. wait_ack must be a wait state with satisfiable set: {server+ack}	
2. operation must complete eventually	
Properties and Comments	

Table 4.4: Generic Build Table for DF-ClientServer-IPC-1

of granularity, and in terms of the application.²⁵ The "Properties and Comments" section allows free-form entry of comments relating to the properties, not already assigned to a design fragment as a Performance Issue; i.e., other domain related issues considered relevant by the designer. Neither the *Design Contingencies*, nor *Performance Issues* sections are included in this example. Sections which have no entry are, generally, not shown.

Table 4.5 shows the DF-ClientServer-IPC-1 Build Table bound to a specification segment of the Readers/Writers problem. Here, readers are the client, and the

²⁵Carrying the bindings from one section of the table to another is easily achieved by a spreadsheet tool such as LOTUS 1-2-3 [49].

Build Table Reference: 1 of Readers/Writers Design**BUILD TABLE: DF-ClientServer-IPC-1**

Generic Specification Segment	
client: server-request;wait_ack;server+ack	
server: client+request;operation;client-ack	
Application Bindings	
client	_reader
server	scheduler
request	rr
wait_ack	wait_grant
ack	gr
operation	[C'=C+1]
MachineChart Design Fragment	Figure 5.3
Specification Contingencies	
Contingency	Proof
1. wait_grant must be a wait state with satisfiable set: {scheduler+gr}	confirmed
2. [C'=C+1] must complete eventually	confirmed
Properties and Comments	

Table 4.5: Build Table for Design Decision in Readers/Writers Design

scheduler is the server. The service is to increment the reader counter, C , and to give resource access to the reader by returning with a grant read (gr) message.

Every design fragment in the framework has its own generic version of a Build Table. When a design is being developed, and a design fragment is selected, the user retrieves a copy of the generic table. The generic terms of the design fragment are bound, by the user, to the application-specific terms of the corresponding specification segment in the EBD model. To demonstrate the contingencies, the designer is then faced with application-specific descriptions of the design and specification contingencies: e.g., Table 4.5 includes the statements: (1) *wait_grant* must be a wait state with satisfiable set: {*scheduler+gr*}, and (2) [$C'=C+1$] must complete even-

tually. An automated system could determine that a sequential MCL description defines an operation which will complete eventually (proof 2), and ask only for verification that the *wait_grant* state is a wait state with a single exit event *scheduler+gr* (proof 1). However done, Table 4.5 requires confirmation of both contingencies. Since proofs 1 and 2 can be verified by inspection in this example, the user simply confirms the statements and the proof is complete. This is achieved by making a note in the *proof* column of the contingencies section that the statement is valid (e.g. by using the word *confirmed* as shown).

Some proofs can be satisfied by inspection (as done above), however this is not always the case. Chapter 5 discusses proof and reasoning techniques for typical contingencies.

Chapter 5

Validation

This chapter discusses two fundamental attributes of *strong traceability*: correctness and completeness. Validating correctness in a design is achieved by proving the contingencies associated with the design fragments used to develop the design. This provides for traceability in the small; i.e., illustrating the correct implementation of each selected specification segment. Validating correctness is covered in Section 5.1. This section includes the first examples which use Build Tables in the design process in sections 5.1.7 and 5.1.8. Proving completeness illustrates traceability in the large: every requirement in the specification is shown to be satisfied by the design. This is discussed in Section 5.2.

5.1 Validating Correctness

Validating correctness is defined as the process of demonstrating all contingencies which are expressed in the Build Tables of a given design. The application-behaviour is present in the design through the ACM which encapsulates the meta-level EBD. Furthermore, each design fragment satisfies several requirements of the communications semantics and the required basic model design concept. Therefore, much of the conformance proof is embedded in this design method. The contingencies represent only those portions of the design which cannot be proven through a mechanical binding of the specification and the design. They are either: (1) related to the specification, and represent a different (but equivalent) view, or a constraint on the behaviour: *specification contingencies*, or (2) are related to the basic model design approach, and represent the *matching* (cref. § 4.2.3) required to satisfy this concept: *design contingencies*.

This section reviews the two types of contingencies (*specification* and *design*), and presents algorithms for demonstrating the contingencies of a design. *Properties* are also discussed. This section also looks at some *typical* contingencies which were encountered while working on this thesis. The presentation illustrates the scope of contingencies, and thus the level of demonstration required for validation (or verification; i.e., formal proof). These contingencies are used to explore two approaches to validation: (1) using formal methods and tools as introduced in Section 2.4, and (2) using reasoning techniques which have been defined informally to illustrate the potential for some level of expert assistance in demonstrating contingencies without formal proof.

5.1.1 Validating Specification Contingencies

Specification contingencies arise when design fragments constrain the specified behaviour to a behaviour which is conforming by nature of the application; i.e., they describe the conditions under which a particular application-behaviour will conform to the design fragment being used. An example is the synchronous rendezvous which is given as a potential solution for asynchronous transfers (DF-AsyncInter-IPC-2).

Here, the specification of an asynchronous transfer is implemented as a synchronous transfer. This is correct if, in the particular application, the receiver is guaranteed to eventually receive, and thus terminate the interaction. The specification contingency associated with this design fragment ensures that this is the case, and instructs the designer to prove that there is conformance in this way.

Specification contingencies require analysis of the behaviour described by the EBDs. They define the conditions under which a design alternative provides communications which conform to the specification. They provide the designer with the means to show how different communication semantics can be used to produce an *equivalent* behaviour: e.g., the designed behaviour is a *reduction* of the specified behaviour (c.f. § 2.4). Some specification contingencies represent safety properties; for example, explicitly showing groupings of events in satisfiable event sets, or applying specific selection policies as FAIR. Others represent *liveness* properties; e.g., that an interaction eventually occurs, or that a behaviour segment eventually completes. In either case, the proofs can be illustrated formally because the designer is proving that there is conformance (a formal relation) between two formally defined representations (EBD segments): (1) the original specification segment, and (2) an EBD representation of the design alternative (see Section 4.2.5). Therefore, specification contingencies can be proven mechanically (although this may be difficult because a reachable state graph of the system is required as discussed in Sections 4.2.5 and 2.4).

Specification contingencies are independent of one another. Design fragments are placed into the framework only after they have been shown to conform to the specification, independently, but under a set of contingencies. Here, independence is with respect to the behaviour. Design fragments, and thus the specification contingencies, assume that the rest of the design (i.e., that which is outside of the specification segment in question) conforms to the specified behaviour. Therefore, the proof of each specification contingency is independent of the next. A side effect of specification contingencies is to emphasize places in the specification which are over constraining; e.g., synchronization is specified, but the pattern of interaction establishes the synchronization regardless of which of the three types of communi-

cations are specified for the interactions.

5.1.1.1 An Algorithm for Demonstrating Specification Contingencies

Algorithm:

For every Specification Contingency in every Build Table
 Demonstrate the contingency

This algorithm illustrates the independence of each contingency: i.e., that each specification contingency can be demonstrated under the assumption the rest of the design conforms to the specification as was discussed above, and in Sections 2.4 and 4.2.5. There is no required ordering of specification contingencies (within a table or between tables), nor is it necessary to wait for the design to be completed to consider the contingencies. Each contingency must be covered, but the work of validating specification contingencies is performed as the designer sees fit.

5.1.2 Validating Design Contingencies

Design contingencies ensure correct *matching* of the design components (cref § 4.2.3). Matching is required when: (1) the properties of the fragment cannot be mechanically bound to the requirements of the basic model: e.g., the placeholders must be defined and positioned by the designer in some SEQ design fragments such as DF-Inter-SEQ-1, which defines a static schedule for co-allocated EBDs (see Section 4.2.5.2), or (2) a design fragment covers only a part (i.e., the send or receive part) of an interaction. In the first case, the demonstrations are typically mechanical (i.e., done by inspection). As with specification contingencies, these contingencies are independent of one another, and do not impose any ordering for their demonstration. In the second case, the design contingencies themselves can be used to resolve the contingencies once the design is complete.

For example, if DF-AsyncTx-IPC-2 (see C.2) is used to design an asynchronous message send, then this establishes a design contingency: *tx+msg* must be designed as a synchronous receive of *rx*. To prove these design contingencies, a *match* proof statement (which is placed in the proof column of the Build Table like the *confirmed*

statement shown in Figure 4.5), defines the context of this design segment, and that of the required matching design segment. Using the example above, the *match* proof statement establishes the following facts about this design decision: (1) *rx-msg* is a synchronous send of *tx*, and (2) *tx+msg* must be a synchronous receive of *rx*. If the receive event is designed correctly and also proven by the *match* statement, then it will generate the following facts: (1) *tx+msg* is a synchronous receive of *rx*, and (2) *rx-msg* must be a synchronous send of *tx*. These statements prove that the design contingencies are satisfied. This matching process is best left to the end of the proof process.

5.1.2.1 An Algorithm for Demonstrating Design Contingencies

Algorithm:

For every Design Contingency in every Build Table

 If *matching* is required across actors

 Then, add the *match* statements to a *match sheet*

 Else, demonstrate the contingency

Resolve the outstanding *matching* requirements using the *match sheet*; i.e.,

For every *X must be Y* statement

 Find the matching *X is a Y* statement.

This algorithm introduces a *match sheet* on which to record and resolve the *matching* requirements defined between actors; i.e., the matching of “*must be*” and “*is a*” statements.

5.1.3 Properties and Performance Issues

Properties and *performance issues* arise when the design might change the expected behaviour; e.g.: (1) the specifications assume unbounded buffers and this is not possible in a real implementation, or (2) in terms of timing such as delays which affect the entities as a result of the design decision. They do not represent require-

ments for conformance, but rather the cost of having conformance with a particular solution. Properties and performance issues can present some inter-relationships; i.e., they are not necessarily independent. For example, a delay introduced by one design decision might propagate to other segments of the specification. They are demonstrated using the application-behaviour and information outside of the specified behaviour such as personal experience, prototyping, and expected arrival distributions of external events. Typically, properties and performance issues cover *safety* requirements; i.e., invariants such as a requirement for a number of queued elements to be within a bound, or that delays imposed by blocking interactions fall in a reasonable bound.

5.1.3.1 An Algorithm for Demonstrating Properties and Performance Issues

Algorithm:

For every Property or Performance Issue in every Build Table
 Demonstrate/consider the effect in relation to other
 properties as necessary.

5.1.4 Typical Contingencies

This section presents some contingencies, properties, and performance issues (collectively called *issues*), which were encountered during the process of writing this thesis. They illustrate the scope of the application-, design- and domain-dependent issues expressed in design fragments, and hint at the level of demonstration required to illustrate correctness (which is the focus of the next two sections; i.e., Section 5.1.5 and 5.1.6). A list of *typical contingencies* and *typical properties* is given below: all but the first three are segment-level level issues, as opposed to actor-level issues resulting from SEQ design fragments. Entries 4 through 6 illustrate liveness properties required by specification contingencies. Entry 7 considers the performance of different design alternatives, and entries 8 through 11 are safety properties: 8 – 10 are specification contingencies, and 11 is a performance issue.

1. **Placeholders** – To demonstrate that the application of a SEQ design fragment satisfies the requirements of the basic model by introducing all required placeholders.
2. **Fair Schedules** – To demonstrate that a fair schedule is defined for a set of co-allocated EBDs.
3. **Combination Receive Places** – To demonstrate that a receive place for both inter- and intra-actor interactions makes a fair selection from the satisfiable events.
4. **Eventual Completion: Operation** – To demonstrate that a specification segment can be collapsed into an operation, thus showing that the defined operation completes eventually.
5. **Eventual Completion: Interaction** – To demonstrate that an interaction completes by showing: (1) that the receiver will eventually receive the event, or (2) that the sender will eventually send the event.
6. **Immediate Reception** – To demonstrate that a receiver is ready and waiting for an event.
7. **Reasonable Performance** – To demonstrate reasonable performance of eventual completions.
8. **Satisfiable Event Set Groupings** – To demonstrate properties of satisfiable event sets; e.g., single member sets, disjoint sets, or sets with consistent common members.
9. **Re-entrancy** – To demonstrate that an operation (including collapsed behaviours which represent services) can be re-entrant; i.e., that they are accessed mutually exclusively.
10. **Separable Behaviours** – To demonstrate that the incoming messages involved in a collapsible behaviour form a disjoint set from those of the EBD as a whole.

11. **Queueing Limitations** - To demonstrate that the limitations of queueing implementations will not cause losses or unreasonable delays.

The next two sections consider techniques for demonstrating these issues. Section 5.1.5 examines the use of mechanical, formal techniques, while Section 5.1.6 introduces techniques based on intuitive reasoning. It is expected that contingencies will, in general, be supported by informal demonstrations: i.e., arguments like those suggested in Section 5.1.6, or simple statements of assumed conformance. Regardless of the level of demonstration, the design process provides strong traceability in terms of: (1) the application-behaviour encapsulated by an ACM, (2) the basic model concept, (3) the proof process performed during the definition of design fragments, and (4) the explicit listing of features which might result in a violation of conformance between the specification and the design (i.e., the contingencies). Given formal proofs, the design will correctly conform to the specification. Given informal proofs, the contingencies represent a closed set of conditions for failure, and are useful for defining tests to increase the level of confidence in a design.

5.1.5 Demonstrating Contingencies: Formal Techniques

Liveness Issues — Liveness issues (i.e., Eventual Completion: Operation (entry 4 of the "typical contingency" list presented in Section 5.1.4), Eventual Completion: Interaction (5), and Immediate Reception (6)) are instances of specification contingencies. They illustrate that a specification can be implemented with alternate communication semantics without violating the required externally visible behaviour: i.e., to show that the specified and designed behaviours are fundamentally equivalent for the given application. Such proofs can be carried out with the help of proof tools which perform static analysis (e.g., using VALIRA a reachability analysis tool [116]), or tools for model checking (e.g., XESAR [50], or VEDA 2[40]).

Model checking tools can be used to look for paths which contradict the contingency. (Note that in model checking, once a contradiction is found, the process completes. To prove that an assertion is guaranteed requires analysis of a complete reachable state graph as discussed in Section 2.4.) For example, for a contingency of the type *Eventual Completion: Interaction*, the tool can be used to look for paths

where a sent event is never received. If there is such case, there is a chance that it will be found before the entire reachable state graph is searched. Otherwise (i.e., if a contingency requiring eventual reception is satisfied), the search will be exhaustive.

For example, the Readers/Writers design of Section 4.2.5.1 uses the client-server design fragment DF-ClientServer-IPC-1, and requires a proof showing that readers eventually complete their operation once a writer has been recognized by the scheduler. A model checking proof tool can be used to show that, in all cases, readers eventually send finished notifications after they are granted access to the resource.

A separate design example of 4.2.5.2 which uses a static schedule for co-allocated readers, writers and a scheduler, has a contingency that *start* messages eventually arrive from a user to invoke the writers. A tool could be used to show that the system blocks indefinitely awaiting these external events. The designer must, in this case, use his own knowledge about the expected rate of these events to validate the contingency; i.e., this behaviour is not fully specified.

Reasonable Performance (7) — Some static analysis methods allow for inclusion of timing for functional requirements [73, 90, 63, 64, 65, 83]. These tools (mainly scheduling and performance modelling tools) might be of particular interest since many changes to the communication semantics which result in contingencies based on liveness properties, affect the expected timing and delays of entities.

Satisfiable Event Set Groupings (8) — Several properties of the satisfiable event set can be analysed formally because often the semantics of the specification are not being altered in any way. In general, only the syntax is changing; for example, grouping events under a single event name with an unspecified sender. Therefore tools could be developed to show equivalence between the two variations of the specification; and, as such resolve these ambiguities allowed in EBD specifications.

Such a tool could use the following algorithm:

1. Start with a set of the set of all incoming events; e.g., $\{\{rr, wr, rf, wf\}\}$ would represent the incoming events of the scheduler in the Readers/Writers problem.
2. For every state in the EBD, split the set into the satisfiable and non-satisfiable sets. This gives the satisfiable event sets of every state.

3. Define the common satisfiable event groupings for the entire EBD by iteratively decomposing existing sets, but never combining sets based on the satisfiable and non-satisfiable sets of each state.

As an example of defining the common satisfiable event groupings within an EBD, the analysis of the scheduler EBD of the Readers/Writers problem is considered. The set appears as follows after analysing state 1: $\{\{rr,wr\},\{rf,wf\}\}$. Here, the requests are satisfiable events and the finished notifications are not. This new set of sets is used as input to the decomposition problem at state 2. After analysing state 2 the set is as follows: $\{\{rr,wr\},\{rf\},\{wf\}\}$. Note that the *rf* event is not added to the set containing the requests: the sets are never increased in size. However, the *rf* event is satisfiable while the *wf* event is not yet satisfiable, therefore these two events must be separated into different sets. The remaining two states (3 and 4) do not change this last result.

The result shows that the request events are always satisfiable at the same time; i.e., they represent a common grouping. The individual state sets can be used to determine if events are always alone in the satisfiable event sets of an EBD, or to determine if events are in more than one satisfiable event set

Re-entrancy (9) — Re-entrancy can be proven formally in two ways: (1) by showing that the EBDs initiating an operation are all co-allocated, and that priorities are not used to interrupt an operation, and (2) by showing that an operation has a single source for its initiation, and that this source is specified to wait while the operation occurs.

Separable Behaviours (10) — Behaviours can be proven to be separable by analysing the satisfiable event sets as discussed above. This proof requires the designer to define the set of incoming events for an EBD at the level which abstracts out the collapsed behaviour, and the set of incoming events for the collapsed behaviour. The union of these sets must be the null set.

5.1.6 Demonstrating Contingencies: Intuitive Reasoning

Placeholders (1) — Given a SEQ design fragment (e.g., a static schedule for

co-allocated EBDs), placeholders are demonstrated by inspection. Each required placeholder is defined as a design contingency; and, as such, each is demonstrated independent of the others, based on the design.

Fair Schedules (2) — If the DF-Inter-SEQ-1 fragment is used, a fair schedule must be defined, by the designer, for the behaviour of the co-allocated EBDs. If the schedule is produced using formal techniques, then it can be *confirmed* intuitively. Otherwise, one approach to demonstrating its validity is to show that the schedule represents a valid path through the reachable state graph. This is again, a formal proof. Without the help of a reachable state graph, this demonstration is difficult to do: the designer must evaluate the coverage of the schedule (i.e., all functional requirements should be covered by the schedule).

Combination Receive Places (3) — With some SEQ design fragments, the designer must demonstrate that the receive_places are fair for satisfiable event sets with both inter- and intra-actor interactions. Here, the demonstration is typically dependent on the design process: i.e., the design process works to demonstrate fairness. For example, the designer could argue fairness as was done for the Producer/Consumer design of C.1.2.2 where it was shown that the behaviour of the co-allocated EBDs would always eventually block on the inter-actor interactions.

As with the receive_places of strictly inter-actor interactions, the schedules to receive these satisfiable sets (with both inter and intra-actor events), could be added as elements of the design fragments framework. For example, the following ideas might evolve into design fragments. (1) The inter-actor receive_placeholders are scheduled when all intra-actor behaviours are otherwise blocked. (This was used in the Producer/Consumer example design.). (2) The event queue could be checked for a match with the intra-actor events of the satisfiable set, before a conditional receive is made on the inter-actor elements. If the conditional receive fails, the action is returned to the end of the event queue. This puts the intra-actor events at a higher priority, and fairness must be considered. It also defines a new type of receive_placeholder; i.e., a conditional receive_placeholder. Conditional receive_placeholders would require matches with non-conditional sends, while conditional send_placeholders would require conditional send IPC design fragments which

are matched with non-conditional receive design fragments. (3) The design suggestion above could be exchanged such that inter-actor messages are tried before the intra-actor messages.

Eventual Completion: Operation (4) — Operations states in a design fragments can be associated with: (1) specification segments, (2) disjunctions of specification segments, or (3) specification segments appended or prefixed by specific behaviour patterns. As such, some design fragments define new collapsed behaviours. Eventual completion is demonstrated if the specification segment neither diverges, deadlocks, nor livelocks. Since it is assumed that the specification has been verified for deadlock and livelock as a whole, the designer will not be expected to review these properties. Therefore, what is left in this case, is to show that the new, collapsed behaviour does not diverge; i.e., does not lead along a path other than that of the specification segment under consideration. The behaviour being bound to the specification segment of the design fragment must always, and eventually exit via the exit path bound to the specification segment of the design fragment. The performance will usually be a consideration in these cases; i.e., the delays associated with the operation are assessed.

To demonstrate that a behaviour pattern (which can be a disjunction of behaviours) eventually completes, the designer shows that: (1) it (or each disjunct) cannot diverge (or leave) the behaviour by any path other than the defined exit path, and (2) that there is a guarantee of completion of the interactions which must take place for the behaviour to progress. In some cases this second requirement can be demonstrated by the notion that there is no deadlock in the system, however there will be cases where a specific path through the combined behaviour is not taken unless another entity is active or continuously active. For example, in the Readers/Writers problem, it may be necessary to show that there is an infinite supply of readers or writers for some patterns to be shown to complete eventually (or within a reasonable worst case delay).

Showing eventual completion becomes even more difficult when collapsed behaviours cross EBD boundaries as with SEQ design fragments because the behaviour being assessed for divergence is the combined behaviour of the participating EBDs.

Reasonable Performance (7) — This topic is far too complex to cover sufficiently in this thesis, however, a few points will be highlighted. Reasonable performance of collapsed operations is often dependent on deterministic operation lengths and wait periods for sporadic or periodic incoming events. This assumes that the performance properties of a collapsed behaviour (e.g., the delays and bottlenecks) are contingent on lengthy operations or wait times. This is an oversimplified view, but will help to indicate where further study is required which could possibly result in a re-design: i.e., conversion to a design which, for example, focuses on concurrency instead of fewer actors.

To assess the delays of a collapsed operation, each of the functional operations is examined in turn. Each should require a finite and deterministic worst case time from which the overall worst case time can be determined. Wait periods on incoming events are deterministic for periodic events, and thus the delays can be assessed at some level. Wait periods on sporadic events are generally non-deterministic, and require assumptions about their nature. For example, there is the Readers/Writers design of Section 4.2.5.2 (where the static schedule of Figure 4.22 is updated to include an external interaction with the *user* entity). The design is based on the assumption that the *user!start* events arrive frequently enough to avoid excessive delay to the readers which are independent of the writers when the resource has not been allocated to a writer.

Eventual Completion: Interaction (5) — This demonstration technique is similar to the model checking approach; i.e., it is goal directed, and considers reachable states. The suggested process for demonstrating eventual reception of an event is as follows. As a first step, the user would define the set of states of the receiver for which the event is satisfiable. If this set contains every state of the receiver EBD, then the proof is completed trivially; i.e., the event is received infinitely often. Otherwise, the user defines the set of states (not including those in the set where the event is satisfiable) from which the original set of states can be reached. If the transitions from these states require interaction with the sender of the event in question, then these interactions must have already occurred (by the fact that they must also conform to the specification), and therefore the satisfiable states are

reachable, and the argument completed. This approach is analogous to a backwards chaining proof of the reachability state graph: i.e., starting with the final system state, the analysis works back to find the previous system states where it is hoped that the expected reachability properties are intuitively understood. If however, the transitions are based on interactions with other EBDs, then the argument consists of defending the reachability of the states in the satisfiable set. The use of the design fragment is defended by: (1) the idea that deadlock does not exist in the system, (2) the reliability of external sources, and (3) the determinism of operations.

As with receivers, demonstrating that a send eventually occurs is potentially very difficult. The same type of arguments are needed: i.e., to show to the states from which the event is sent are reachable either infinitely often, or given the current state of the receiver, or given the current arguable state of the system.

Immediate Reception (6) — This is another specification contingency which can be very difficult to demonstrate without the aid of a reachable state graph. The analysis is similar to showing that a receiver will eventually receive, however this is much more constrained: i.e., the receiver must be in the receiving state (with no other satisfiable events) when the event send is initiated.

5.1.7 Example Design – 1: Readers/Writers

This section will repeat the design of the Readers/Writers problem focusing on the client/server design fragment to achieve the result shown in Figure 4.19 (and repeated in Figure 5.2). The Readers/Writers specification is repeated in Figure 5.1. Here, the tabular design approach will be used with reasoned arguments to demonstrate the validity of the contingencies.

Step 1: Using DF-ClientServer-IPC-1 between Readers and Scheduler

The first step in this design example is to apply the DF-ClientServer-IPC-1 design fragment to the interactions between the reader and scheduler which include the read request and the grant read acknowledgment. The designer would retrieve a copy of the generic Build Table for this design fragment (as was shown in Table 4.4). A reference number is assigned to the build step (e.g. reference number 1)

Informal Specification:

A data structure in primary or secondary memory is accessed by a number of readers and writers. In general, reading and writing can occur at any time, although only one writer at a time should be allowed to write. While writing, all other readers and writers must be denied access to the memory, however, when there are no writers, any number of readers are allowed.

EBD Specification:

Figure 5.1: Readers/Writer Specification

to distinguish this build step within a design. The designer would then complete the application bindings resulting in the instantiated Build Table of Table 4.5 (repeated here as Table 5.1. As a result of this binding, the design fragment is as shown in Figure 5.3: i.e., this is an instantiated, application version, of the design fragment. A reference to such an illustration can be added in the bindings section of the Build Table. (In an automated system, this reference and illustration could be generated

Figure 5.2: Readers/Writers Design Solution

Build Table Reference: 1 of Readers/Writers Design

BUILD TABLE: DF-ClientServer-IPC-1

Generic Specification Segment	
client: server-request;wait_ack;server+ack server: client+request;operation;client-ack	
Application Bindings	
client	<i>_reader</i>
server	<i>scheduler</i>
request	<i>rr</i>
wait_ack	<i>wait_grant</i>
ack	<i>gr</i>
operation	$[C'=C+1]$
MachineChart Design Fragment	Figure 5.3
Specification Contingencies	
Contingency	Proof
1. <i>wait_grant</i> must be a wait state with satisfiable set: $\{scheduler+gr\}$	confirmed
2. $[C'=C+1]$ must complete eventually	confirmed
Properties and Comments	

Table 5.1: Build Table for Design Decision in Readers/Writers Design

Figure 5.3: Client/Server Design Fragment Bound for Readers/Writers Design

automatically, or on request.)

The specification contingency section of the table is instantiated according to the contents of the application bindings section of the table; i.e., the generic names found in this section take on the same values as defined in the bindings section. In this example, the first specification contingency appears as a statement: “*wait_grant* must be a wait state with satisfiable set: $[scheduler+gr]$ ”. The second specification contingency appears as the following statement: “ $C'=C+1$ must complete eventually”. On the “Proof” side of each contingency the designer can either: (1) confirm the statement, or (2) establish a link to a detailed proof of the statement. For this example, both statements are intuitively true, and therefore each is confirmed without further demonstration.

Step 2: Using DF-ClientServer-IPC-1 between Writers and Scheduler

The second design step uses the same design fragment for the similar writer and scheduler interaction. Table 5.2 shows the table for this design step after the bindings have been completed. In this example, the operation of the server, *operation*, is bound to a reference, *?WriterFree?*. The reference is encapsulated by “?” symbols to indicate that the value is incomplete at this level, and is referenced to a more detailed description by the *WriterFree* label. The operation bound to the term *operation*, is a disjunction of two behaviours: (1) the null behaviour of the direct transition from scheduler state 1 to scheduler state 4 (i.e., the transition operation of the $_w+wrset(w=SID)//w-gw$ transition), and (2) the behaviour between scheduler state 2 and scheduler state 4 through state 3 shown as *behave(2)* in Figure 5.4. The *?WriterFree?* reference in the Build Table leads to the description of the disjunction using EBD syntax shown in Figure 5.4.

Build Table Reference: 2 of Readers/Writers Design**BUILD TABLE: DF-ClientServer-IPC-1 (Inter-actor client/server)**

Generic Specification Segment	
client: server-request;wait_ack;server+ack server: client+request;operation;client-ack	
Application Bindings	
client server request wait_ack ack operation MachineChart Design Fragment	_writer scheduler wr wait_grant gw ? WriterFree?
Specification Contingencies	
Contingency	Proof
1. <i>wait_grant</i> must be a wait state with satisfiable set: { <i>scheduler+gw</i> }	Confirmed
2. ? <i>WriterFree</i> ? must complete eventually	?? <i>WriterFree</i> ??
Properties and Comments	

Table 5.2: Build Table for Second Design Decision in Readers/Writers Design

As in step 1, the first contingency can be confirmed visually, however, in this case the designer may wish to provide a more detailed examination (proof) of the second contingency. This process is illustrated in Figure 5.5.

In Figure 5.5 the contingency statement in the Build Table is shown at the top of the page under the reference label ?? *WriterFree*?? (note that double question marks around the label *WriterFree* are used to reference the demonstration). Using the ideas for reasoning eventual completion given in the previous section, the first step is to decompose the disjunct, and handle each case separately. As a new proof concept is applied, the proof statements have been indented. This form is similar to the trace facilities of Prolog debuggers [93]. The proof has been decomposed

?WriterFree?

WriterFree is an operation defined by the disjunction of: (1) a null transition operation labelled and referred to as [], and (2) the following behaviour labelled and referred to as behave(2).

Figure 5.4: Description Referenced by ?WriterFree? label

??WriterFree??

?WriterFree? must complete eventually

-> [] must complete eventually

-> behave[2] must complete eventually

-> behave[2] must not diverge

-> behave[2] must be deterministic

-> _r+rf must be deterministic

-> reader is a reliable source in state 3

-> reader is guaranteed to reach state 3

□ reader state 3 is always reachable

□ reader is in state 3 if scheduler is in its own state 3

Proof

prove

confirmed

prove

confirmed

prove

prove

confirmed

prove

statement:

depends on initiation of readers

confirmed

Figure 5.5: Proof of ??WriterFree?? as Collapsible Operation

into two sub-proofs: (1) “[] must complete eventually”, and (2) “*behave[2]* must complete eventually”.¹ The first statement can be confirmed visually as shown, while the second statement may require further consideration.

To reason about the eventual completion, the previous section suggests that the designer prove: (1) that the behaviour does not diverge, and (2) that the behaviour is deterministic (i.e., expected to occur). These two sub-proofs are shown in Figure 5.5. The first statement is confirmed by the designer by inspection, while the second statement is considered in more detail. At this point, the suggestions for reasoning performance considers if the incoming events of this behaviour are deterministic. The only incoming event in this example is *_r+rf*, and therefore the statement of proof appears as: “*_r+rf* must be deterministic”.

The designer might again look for assistance in confirming this statement. The discussion on reasoning performance suggests that incoming events are deterministic if the source is reliable when in the sending state (e.g. “*reader* is a reliable event source when in state 3”), and that reaching the sending state is guaranteed because: (1) the state is reached infinitely often (“*reader* state 3 is always reachable”), or (2) the state is guaranteed by the state of the receiver (“*reader* will be in state 3 if *scheduler* is in state 3”). These disjuncts are shown using the box symbol, commonly used to signify a choice, at the appropriate indentation level. The designer confirms that the reader is reliable when in state 3, and that the readers are in state 3 when the scheduler is in state 3. The designer has used knowledge about the expected behaviour to terminate the reasoning process. Thus the contingency has been argued, and a record of the reasoning is maintained.

Step 3: Using DF-AsyncTx-IPC-2 for *s-rf*

The next design decision is given in Table 5.3. Here, the DF-AsyncTx-IPC-2 design fragment, which implements a rendezvous send, is being applied to the *scheduler-rf* event sent by the reader. There are two contingencies associated with this design fragment: (1) the specification contingency that the event must be received eventually (e.g. once initiated, *_reader+rf* must be received eventually by *sched-*

¹ *behave[2]* is the label assigned in Figure 5.4 for the compound behaviour pattern between scheduler states 2 and 4 through state 3.

Build Table Reference: 3 of Readers/Writers Design**BUILD TABLE: DF-AsyncTx-IPC-2 (Rendezvous sender)**

Generic Specification Segment	
tx: rx-msg	
Application Bindings	
tx	<i>_reader</i>
rx	<i>scheduler</i>
msg	<i>rf</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>_reader+rf</i> must be received eventually by <i>scheduler</i>	?BT3ref?
Design Contingencies	
Contingency	Proof
1. <i>_reader+rf</i> must be designed as a synchronous receive of <i>scheduler</i>	match
Performance Issues	
Issue	Comment
1. <i>_reader</i> blocks until <i>scheduler</i> acknowledges that <i>rf</i> has been received.	OK
2. <i>_reader</i> is delayed by <i>scheduler</i> .	OK
Properties and Comments	
1. <i>scheduler-rf</i> is a synchronous send of <i>_reader</i>	

Table 5.3: Build Table for Third Design Decision in Readers/Writers Design

uler), and (2) the design contingency requiring a synchronous receiver to match this synchronous send (e.g. *_reader+rf* must be designed as a synchronous receive of *scheduler*).

For this design fragment, the demonstration of the specification contingency is referenced by the label ?BT3ref? (i.e., Build Table 3 reference). The argument for validity of this contingency is shown in Figure 5.6. It follows the suggested approach of the previous section for illustrating eventual reception of an event. Table 5.3 shows

<i>?BT3ref?</i>	
	Proof
<i>_reader+rf</i> must be received by <i>scheduler</i>	prove
-> <i>scheduler</i> receives <i>rf</i> in states 2 and 3.	
-> These states are reachable from state 1 when <i>reader+rr</i> is received and <i>reader-gr</i> is sent.	
-> <i>reader</i> is in state 3 when sending <i>rf</i> .	
-> <i>reader</i> reaches state 3 by receiving <i>scheduler+gr</i> .	
-> Conclude that <i>reader</i> and <i>scheduler</i> are interacting: i.e., <i>scheduler</i> is in states 2 or 3 when <i>reader</i> is in state 3.	
-> Conclude that <i>_reader+rf</i> is received.	confirmed

Figure 5.6: Proof of *BT3ref* Contingency of Readers/Writers Design

a proof for the design contingency as the term *match*. This is used to establish two statements about the design fragment: (1) *scheduler-rf* is a synchronous send of *reader*, and (2) *_reader+rf* must be designed as a synchronous receive of *scheduler*. These statements can be *matched* with the appropriate design decisions defined by another Build Table in the system design; i.e., that which involves the receipt of the *rf* event. These statements define fundamental constraints to the specification which are yet to be resolved by both the sender and receiver entities.

Step 4: Using DF-AsyncRx-IPC-2 for *_r+rf*

The fourth design decision is described in the Build Table of Table 5.4. Here, DF-AsyncRx-IPC-2 is used to design the asynchronous receive *_reader+rf* made by the scheduler as a rendezvous receive. There are no specification contingencies for this design fragment because the semantics of the receive are equivalent for the asynchronous design and the synchronous design, however, the sender must be implemented by a synchronous sender such that there is compatibility at the end of the

Build Table Reference: 4 of Readers/Writers Design**BUILD TABLE: DF-AsyncRx-IPC-2 (Rendezvous receiver)**

Generic Specification Segment	
rx: tx+msg	
Application Bindings	
rx	<i>scheduler</i>
tx	<i>_reader</i>
msg	<i>rf</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>scheduler-rf</i> must be designed as a synchronous send of <i>_reader</i>	match
Performance Issues	
Issue	Comment
1. <i>_reader</i> blocks until <i>scheduler</i> acknowledges that <i>rf</i> has been received.	OK
2. <i>_reader</i> is delayed by <i>scheduler</i> .	OK
Properties and Comments	
1. <i>_reader+rf</i> is a synchronous receive of <i>scheduler</i>	

Table 5.4: Build Table for Fourth Design Decision in Readers/Writers Design

rendezvous. If the *match* option is used to prove this design contingency, then the statements associated with this design fragment are: (1) *_reader+rf* **is a** synchronous receive of *scheduler*, and (2) *scheduler-rf* **must be** designed as a synchronous send of *_reader*. These design statements match those of the last design step: i.e., the **must be** requirements of one design fragment are satisfied by the **is a** statements of the other.

Step 5: Using DF-AsyncInter-IPC-2 for the *wf* interaction

The final design decision uses DF-AsyncInter-IPC-2, which is a rendezvous interaction equivalent to the combination of DF-AsyncTx-IPC-2 and DF-AsyncRx-IPC-2 used separately in Build Tables 3 and 4. Here, there are no design contingencies:

Build Table Reference: 5 of Readers/Writers Design**BUILD TABLE: DF-AsyncInter-IPC-2 (Rendezvous interaction)**

Generic Specification Segment	
tx: rx-msg	
rx: tx+msg	
Application Bindings	
tx	<i>writer</i>
rx	<i>scheduler</i>
msg	<i>uf</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>writer+uf</i> must be received eventually by <i>scheduler</i>	confirmed
Performance Issues	
Issue	Comment
1. <i>writer</i> blocks until <i>scheduler</i> acknowledges that <i>uf</i> has been received	OK
2. <i>writer</i> is delayed by <i>scheduler</i> .	OK
Properties and Comments	

Table 5.5: Build Table for Fourth Design Decision in Readers/Writers Design

the match is defined by the design fragment, however, changing an asynchronous interaction to a synchronous interaction does set a specification contingency. The reasoning of this contingency would be very similar to that shown in Figure 5.6, and as such has been *confirmed* without providing a detailed proof in this example.

This example illustrates how the proof process can be guided by: (1) binding the generic terms of the design fragment to the application, giving application-specific contingencies, and (2) using suggested approaches for reasoning contingencies. It is hoped that use of this design process will provide insight into typical approaches for reasoning contingencies. In an automated system, the user could select an approach to proving a contingency. One example solution is to use a menu assigned to the

contingency which allows the designer to select: (1) *confirmed*, (2) to generate a reference to a proof with assistance, or (3) to generate a reference to a statement of proof. This idea, of documenting with forwarding references, suits a hypertext environment [28].

This proof process can now be compared to the ad hoc arguments of Section 3.6. The set of Build Tables and associated references do require more paper, however it formally establishes the bindings of the specification to the design fragments and presents systematic documentation to support maintenance and review efforts.

5.1.8 Example Design – 2: Producer/Consumer

In this section, the Producer/Consumer design which is discussed in C.1.2.2, is repeated and completed using the Build Table approach. The specification is introduced in Figure C.2, and is repeated as Figure 5.7. In this design, the producer and buffer EBDs of Figure 5.7 are allocated to the same actor, while the consumer EBD is allocated to its own actor. The resultant partial MachineCharts design is shown in Figure 5.8. The completed design is given, in MachineChart form, in Figure 5.9.

DF-AsyncInter-IPC-1 is used to design the asynchronous interactions, without message data, between the buffer and the consumer, and DF-AsyncInter-IPC-4 is used when data is passed along with the event. These design steps are shown in the Build Tables of Tables 5.6, 5.7, 5.8, and 5.9. There are no specification contingencies associated with these design steps because each uses a design fragment which strictly emulates the specification. DF-AsyncInter-IPC-4 is contingent on data type compatibility between the actual and formal parameter of the message data.

DF-Prim-SEQ-1 (Build Tables 5.10 and 5.11) is the design fragment for co-allocated EBDs which sets up an event simulation solution. It is to be used for the overall design of the producer/buffer actor. In this design there are no other segment-level (as opposed to actor-level) SEQ designs to add to this overall design. The Applications Bindings section of the Build Table requires a list of all of the EBDs allocated to the actor being defined by this design fragment. The specification contingencies associated with this design fragment require that each send and receive

Figure 5.7: Producer/Consumer Specification

of an inter-actor event does not block the progression of the combined behaviour, while the design contingencies require fair receive_placeholders for every inter-actor interaction. (This design fragment defines the send_placeholders to exist when an inter-actor send reaches the head of the event queue.) Because asynchronous sends are non-blocking by definition, the proofs of completion of these interactions are implicit, and therefore they do not appear in the contingencies section of this Build Table as shown in Tables 5.10 and 5.11.

The receive_placeholders exist when the *receive* actions reach the head of the queue when the inter-actor receives are independent of the intra-actor receives. However, in this example, this is not the case. If the satisfiable event sets of the buffer are examined (there is only one since there is only one buffer state), it is

Figure 5.8: Producer/Consumer Design after initial EBD to actor allocation

Figure 5.9: Producer/Consumer Design

Build Table Reference: 1 of Producer/Consumer Design**BUILD TABLE: DF-AsyncInter-IPC-1 (Job interaction)**

Generic Specification Segment	
tx: rx-m: g	
rx: tx+msg	
Application Bindings	
tx	<i>consumer</i>
rx	<i>buffer</i>
msg	<i>RqstFull</i>
MachineChart Design Fragment	
Performance Issues	
Issue	Comment
1. <i>RqstFull</i> is buffered at <i>buffer</i> . Delays or losses are possible if buffer limitations are stressed	OK
2. Otherwise, <i>RqstFull</i> is guaranteed to complete.	OK
Properties and Comments	

Table 5.6: Build Table for First Design Decision in Producer/Consumer Design

found that both inter and intra-actor events are satisfiable at the same time (e.g., $\{\{\text{Return}, \text{RqstEmpty}, \text{RqstFull}, \text{PutBuff}\}\}$). For the producer, each satisfiable set contains a single event, (e.g., $\{\{\text{init}\}, \{\text{pack}\}, \{\text{Reply}\}\}$), each of which is an intra-actor event.

Proving that fair receive placeholders exist requires the definition of a schedule for combined inter- and intra-actor receives. This is shown in Figure 5.10. The schedule is defined such that inter-actor interactions are made when all co-allocated EBDs are otherwise blocked. Therefore, the designer must prove that this situation arises deterministically. The proof is strictly application-dependent, and does not map to a typical proof process other than to examine the reachability of the system behaviour. The reasoning for this contingency is given by the simple statement that the producer continues to produce elements until the buffer is full, and continues to produce whenever a buffer slot is made available. This results in the schedule

Build Table Reference: 2 of Producer/Consumer Design**BUILD TABLE: DF-AsyncInter-IPC-4 (Jab with message data)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>buffer</i>
rx	<i>consumer</i>
msg	<i>Full</i>
actual_param	<i>buff</i>
formal_param	<i>c_ptr</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
<i>buff</i> and <i>c_ptr</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>Full</i> is buffered at <i>consumer</i> . Delays or losses are possible if buffer limitations are stressed.	OK
2. Otherwise, <i>Full</i> is guaranteed to complete.	OK
Properties and Comments	

Table 5.7: Build Table for Second Design Decision in Producer/Consumer Design

defined in C.1.2.2 where the producer initially fills the buffer, the consumer and producer alternate using the same buffer slot, and when the producer has reached its stop state, the consumer retrieves the rest of the buffered information.

Proving that the receive placeholders are non-blocking requires a proof that *consumer* sends *buffer-RqstFull* and *buffer-Return* when the producer/buffer actor is blocked. Taking an approach similar to that defined in Section 5.1.6, the arguments for these contingencies use dependencies on the send and receive states to analyse the combined behaviour. The analysis assumes that there is no deadlock in the

Build Table Reference: 3 of Producer/Consumer Design**BUILD TABLE: DF-AsyncInter-IPC-4 (Jab with message data)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>consumer</i>
rx	<i>buffer</i>
msg	<i>Return</i>
actual_param	<i>c_ptr</i>
formal_param	<i>buff</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
<i>c_ptr</i> and <i>buff</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>Return</i> is buffered at <i>buffer</i> . Delays or losses are possible if buffer limitations are stressed.	OK
2. Otherwise, <i>Return</i> is guaranteed to complete.	OK
Properties and Comments	

Table 5.8: Build Table for Third Design Decision in Producer/Consumer Design specification, and is shown in Figure 5.11.

5.2 Validating Completeness

Completeness is validated by showing that all of the requirements defined by the EBD specification are satisfied by the design. Given that every EBD has been allocated to one (and only one) actor, completeness proofs can be decomposed into the completeness proofs of the individual actors. Thus the allocation step establishes

Build Table Reference: 4 of Producer/Consumer Design**BUILD TABLE: DF-AsyncInter-IPC-1 (Jab interaction)**

Generic Specification Segment	
tx: rx-msg	
rx: tx+msg	
Application Bindings	
tx	<i>buffer</i>
rx	<i>consumer</i>
msg	<i>ack</i>
MachineChart Design Fragment	
Performance Issues	
Issue	Comment
1 <i>ack</i> is buffered at <i>consumer</i> . Delays or losses are possible if buffer limitations are stressed	OK
2 Otherwise, <i>ack</i> is guaranteed to complete.	OK
Properties and Comments	

Table 5.9: Build Table for Fourth Design Decision in Producer/Consumer Design

a starting point for a schema to be used to illustrate completeness. Figure 5.12 illustrates this level of decomposition. At the top of the structure is a specification for a system (or subsystem in a large design problem). This is decomposed into the actors which will be used to design a software solution. This decomposition step is validated mechanically; i.e., the allocation rules of Section 4.1.1 must be satisfied by the decomposition to ensure completeness.

The allocation of EBDs to actors is documented using Actor Tables (cref. § 4.1.2). There must be an Actor Table for every actor introduced by the allocation step, and every EBD of the specification must be covered by exactly one Actor Table. An example Actor Table is given in Table 5.12.

Given a complete and correct allocation of EBDs to actors, completeness and consistency validation of the remaining design decisions is performed mechanically using tables; i.e., Actor Tables, Build Tables, and *Partition Tables*. This section

Build Table Reference: 5 of Producer/Consumer Design (page 1 of 2)

BUILD TABLE: DF-Prim-SEQ-1 (Simulation engine)

Generic Specification Segment	
any co-allocated, actor-level, behaviour	
Application Bindings	
actor list co-allocated EBDs list events received from external EBDs list events sent to external EBDs list any satisfiable event sets with both internally generated and externally supplied events MachineChart Design Fragment	<i>prod_buff</i> <i>producer, buffer</i> <i>consumer+RqstFull,</i> <i>consumer+Return</i> <i>consumer-Full, consumer-ack</i> <i>{consumer+RqstFull,</i> <i>consumer+Return,</i> <i>producer+RqstEmpty,</i> <i>producer+PutBuff}</i> Figure 5.8
Specification Contingencies	
Contingency	Proof
1. <i>consumer+RqstFull</i> must not block the progression of the combined behaviour.	?consumer_sends?
2. <i>consumer+Return</i> must not block the progression of the combined behaviour.	?consumer_sends'

Table 5.10: Build Table for Fourth Design Decision in Producer/Consumer Design

begins with a description of Partition Tables which are mechanically created from Actor Tables (which are, for the most part, mechanically created from EBDs). This is followed by a description of how all of these tables are related to one another, and how they are used to validate completeness.

5.2.1 Partition Tables

Partition Tables document the decomposition of *top-level* and *layered* partitions initially defined by Actor Tables. A top-level partition is one which abstracts out collapsed behaviours as layered partitions. Layered partitions, when expanded, in-

Build Table Reference: 5 of Producer/Consumer Design (page 2 of 2)

BUILD TABLE: DF-Prim-SEQ-1 (Simulation engine)

Design Contingencies	
Contingency	Proof
1. <i>consumer+RqstFull</i> requires a fair receive_placeholder	?producer_buffer?
2. <i>consumer+Return</i> requires a fair receive_placeholder	?producer_buffer?
3. <i>consumer-Full</i> requires a fair send_placeholder	sent when this action reaches the head of the event queue.
4. <i>consumer-ack</i> requires a fair send_placeholder	sent when this action reaches the head of the event queue.
5. Receive_place for { <i>consumer+RqstFull</i> , <i>consumer+Return</i> , <i>producer+RqstEmpty</i> , <i>producer+PutBuff</i> }	?producer_buffer?
must consider the fair selection of its elements	
Performance Issues	
Issue	Comment
This design alternative requires a lot of overhead to manage the event queue	OK
Properties and Comments	

Table 5.11: Build Table for Fourth Design Decision in Producer/Consumer Design

clude their own top-level partition, and possibly other layered partitions. The latter occurs when the level of hierarchy of collapsed behaviours is greater than one.²

A Partition Table for a top-level partition lists the interactions (by message name) which define the behavioural requirements of the partition as shown in Figures 5.13 and 5.14. For example, the top-level Partition Table, associated with the *reader* in the Readers/Writers problem, would list the following interactions: ?rr?, ?gr?, and ?rf? as shown in Figure 5.14 (with references to the Build Tables of Section 5.1.7). If a message is designed using IPC design fragments, then a correct design has only one Build Table reference for that message with two exceptions: (1)

²In this thesis, it is assumed that all operation states (except wait states) include collapsed behaviours; and, as such, they are represent layered partitions.

?producer_buffer?

-> Must provide fair schedule for receiving:
 {{Return.RqstEmpty,RqstFull.PutBuff}}

Proof**Statement**

Schedule intra-actor receives when *buffer* and *producer*
 are otherwise blocked.

Schedule inter-actor receives when they reach head of event queue.
Producer and *buffer* block when the buffer is full: i.e.,
 the producer continues to produce elements until the buffer is full,
 and continues to produce whenever a buffer slot is made available.

Figure 5.10: Proof of ?producer_buffer? as a Statement

ACTOR TABLE

Actor: \$reader\$

EBDs: *reader*

ACMs: \$rdr\$

Basic Model: Design Fragment for Figure 4.11 is used.

Top Level Partitions	Comments
?READER?	
Layered Partitions	Comments
?IDLE?	
?READ?	
Comments	
The design consists of a pool of \$reader\$ actors.	

Table 5.12: Actor Table Example

?consumer_sends?	Proof
<i>buffer-RqstFull</i> and/or <i>buffer-Return</i> must be sent by <i>consumer</i> when <i>producer</i> and <i>buffer</i> are blocked on the corresponding receive_placeholders.	prove
-> <i>consumer</i> sends <i>buffer-RqstFull</i> when <i>init</i> is received in the start state or from state 3 after receiving a <i>buffer+ack</i> event	Statement: Assume <i>init</i> is received at system startup.
-> <i>buffer+ack</i> depends on <i>buffer-Return</i>	
-> <i>buffer-Return</i> is asynchronous. it depends on the <i>consume</i> operation	Statement: Assume <i>consume</i> executes in a finite time with a tolerable worst case.
-> <i>consume</i> depends on <i>buffer+Full(buff)</i>	
-> <i>buffer</i> accepts <i>consumer+RqstFull</i> when the buffer is full	Statement: Defined by the schedule for inter-actor receives.
-> <i>buffer</i> issues <i>consumer-Full</i> given a request.	Statement: <i>consumer-Full</i> implies <i>buffer-Return</i> . and thus <i>buffer-RqstFull</i> .

Figure 5.11: Proving that Sends are Initiated by the Consumer

Figure 5.12: Initial Decomposition of Specification

PARTITION TABLE Top-Level Partition**Partition:****Parent:**

Message	Build Table Reference

Figure 5.13: Unbound Top-Level Partition Table

PARTITION TABLE - Top-Level Partition**Partition:** ?READER?**Parent:** \$reader\$

Message	Build Table Reference
?rr?	1 of Readers/Writers Design
?gr?	1 of Readers/Writers Design
?rf?	3 of Readers/Writers Design

Figure 5.14: Top-Level Partition Table for ?READER?

a message whose event defines an unspecified sender, and (2) a message whose event defines an unspecified receiver. If unspecified senders and receivers are the case, then there are as many Build Table references as there are potential senders and receivers.³ SEQ design fragments can overlap (cref § 4.2.5.2). In this case, the Build Table references of the Partition Table should indicate which fragments are applied to a message, and the order in which they are applied. Although there is apparent redundancy for the design of this message in the Build Tables, the Partition Table resolves the ambiguity.

The Partition Table for a layered partition is similar to an Actor Table in that it includes a section for the top-level partition (of which there can only be one entry

³If the design includes a pool of actors which represent the unspecified sender or receiver, then only one entry is required since this design applies to all actor instances.

PARTITION TABLE Layered Partition**Partition:****Parent:**

Top-Level Partition	Comments
Layered Partitions	Comments
Comments	

Figure 5.15: Unbound Layered Partition Table

in this case), and any layered partitions which are defined by collapsed behaviours within a layered partition.⁴ This is illustrated in Figure 5.15, and 5.16. Each partition defined here references another top-level or layered Partition Table (as is the case for the partitions of an Actor Table).

In Figure 5.16, the ?READ? partition is illustrated. It is a layered partition of the \$reader\$ actor and ?READER? top-level partition. The top-level of this partition is also referred to as ?READ?. Some example layered partitions, ?START_MENU? and ?READ_MENU? are shown for the ?READ? partition. The tables are used to develop a tree schema for the behaviour requirements starting at the root with the specification, which is broken down, through the allocation step, to actors, which are decomposed into partitions, which eventually define the messages of the system as leaves. Figure 5.17 illustrates this concept using the Readers/Writers example: extended here, (e.g., by the ?START_MENU? and ?READ_MENU? partitions) for illustration of the completeness problem only. Note that the leaves of the tree (i.e., the messages) are always related to top-level Partition Tables.

A partition represents a potentially re-usable design. For example, the top-level

⁴The EBD modelling technique allows for hierarchical definitions (collapsed behaviours) within any state or transition operation. Since collapsed behaviours must complete eventually, the hierarchy must be finite.

PARTITION TABLE Layered Partition**Partition:** ?READ?**Parent:** \$reader\$?READER?

Top-Level Partition	Comments
?READ?	
Layered Partitions	Comments
?START_MENU?	
?READ_MENU?	
Comments	

Figure 5.16: Unbound Layered Partition Table

partition, ?READER? (with or without the particular solutions for the layered partitions defined in our example; e.g. ?IDLE? and ?READ?), represents a reusable design partition. Two or more EBDs can be defined using the same collapsed behaviour(s); e.g., the ?READER? partition could be used as a collapsed behaviour within several EBDs. Partitions therefore, will not necessarily have unique parents. The parent field is completed by entering: (1) the unique path used to reach the partition, if one exists, or (2) a *don't care* symbol (e.g. X) for a re-usable partition. The paths are defined according to the tree schema; i.e., *actor:top-level partition:layered partition* and so on as appropriate. For example, the ?READER? top-level partition has the parent, \$reader\$, which is the actor name as shown in Figure 5.14. The layered partition ?READ? has the parent, \$reader\$?READER?, which is the actor name followed by the parent partition of ?READ?.

The messages of the top-level Partition Tables represent the leaves of the tree schema. These messages are bound to Build Tables during the design process. Because the design is creative, there is no way, given only the information of the Partition Tables, to mechanically bind these messages to Build Tables. Although the process of developing Actor and Partition Tables is mechanical, the development of Build Tables is dependent on a creative design process.

Figure 5.17: Example Decomposition Schema for Readers/Writers

5.2.2 An Algorithm for Validating Completeness

As stated in the introduction to this section, the validation of completeness can be defined by the requirements of the actors given a correct allocation of EBDs to actors. Section 4.2.3 states the requirements of actors in terms of five design components: (1) the ACMs, (2) the interfaces of the ACMs, (3) the actor engine, (4) the actor interface, and (5) the interconnections. Therefore, completeness can be shown if there are designs for these 5 components. The ACMs and their interfaces are fixed designs, and are assumed to be correct and complete. The actor engine and interface are designed as a *basic model* using actor-level design fragments, and the interconnections are designed by segment-level design fragments.

Using Actor Tables and Partition Tables, a tree schema is defined with two types of leaves: (1) messages of the system specification (as discussed in Section 5.2.1), and (2) basic models which are referenced in Actor Tables. These two leaf types reference design fragments as shown in the Actor and Partition Table examples of this chapter. Some messages are designed as interconnections using IPC design fragments, while others are components of a basic model designed using SEQ design fragments. SEQ design fragments can overlap one another (cref. § 4.2.5.2), and thus there can be more than one SEQ design fragment associated with a particular message as discussed in Section 5.2.1.

The binding of messages and basic models to Build Tables is done manually by the designers, therefore, there is potential for error. The validation of completeness is defined as a consistency check for the Build Tables references given in Actor and top-layer Partition Tables. It is performed by listing all of the requirements (i.e., all of the basic models, and messages defined as leaves of the decomposition schema), and using the Build Tables to mark off the correct and complete design of each of these requirements. The algorithm is given in Figure 5.18. It shows that actor-level design fragments are used to mark off basic model requirements as well as message requirements, while segment-level fragments mark off only message requirements. It also shows that multiple SEQ design fragments can be applied to a particular message, and notes that the order of application must be resolved in these cases. However, there is typically only one IPC design fragment associated with a message.

Algorithm for proving completeness of a design using Actor, Partition, and Build Tables:

1. Define the set, **DF**, of design fragments to be the collection of all design fragments used to design the specification as documented in the Build Tables.
2. Define the set of requirements, **REQ**, to be the set of all messages (event primitives) and basic models defined by the Actor and top-level Partition Tables.
3. Remove any component, **df** $\in$ **DF**.
4. If **df** is an actor-level design fragment then
 - remove the corresponding basic model requirement from **REQ**
 - For every event, **E**, defined by the specification segment(s) of **df**:
 - Find and remove the corresponding requirement **req** $\in$ **REQ**
 - If a matching component cannot be found then
 - If **df** is a **SEQ** design fragment then
 - note that a resolution is required for the design of **E**
 - Else if **df** indicates that there is an unspecified sender or receiver for the message then
 - the design is correct
 - Else there is an error in the design.
5. Repeat steps 3 and 4 until **DF** is an empty set.
6. If **REQ** is an empty set, and no errors were noted, then
 - the design is complete and consistent
 - Else if **REQ** is an empty set then
 - the design is complete, but inconsistent because there are multiple implementations for some interconnection(s).
 - Else the design is incomplete.

Figure 5.18: Algorithm for proving completeness

5.2.3 Example Completeness Proof

This section provides an example of the completeness proof algorithm given above. It uses the Readers/Writers design of Section 5.1.7, and the completeness schema of Figure 5.17. The set of design fragments, **DF**, is defined by Build Tables 1 through 5 of Section 5.1.7; i.e., Tables 5.1, 5.2, 5.3, 5.4, 5.5. The set of requirements are defined by the Actor and top-level Partition Tables as:

1. Actor Table defined basic models: \$reader\$, \$writer\$, \$scheduler\$.
2. Top-level Partition Table \$reader\$:?READER? defined messages: ?rr?, ?gr?, and ?rf?.

3. Top-level Partition Table $\$reader\$$:?READER?:?IDLE? defined messages: none.⁵
4. Top-level Partition Table $\$reader\$$:?READER?:?READ? defined messages: *?quit?*, *?next?*, and *?enter(file)?*.
5. Top-level Partition Table $\$reader\$$:?READER?:?READ?:?START_MENU? defined messages: *?left_arrow?*, *?right_arrow?*, and *?read_request?*.
6. Top-level Partition Table $\$reader\$$:?READER?:?READ?:?READ_MENU? defined messages: *?scroll_fwd?*, and *?scroll_bwd?*.
7. Top-level Partition Table $\$writer\$$:?WRITER? defined messages: *?wr?*, *?gw?*, and *?wf?*.
8. Top-level Partition Table $\$writer\$$:?WRITER?:?IDLE? defined messages: none.
9. Top-level Partition Table $\$writer\$$:?WRITER?:?WRITE? defined messages: none.
10. Top-level Partition Table $\$scheduler\$$:?SCHEDULER? defined messages: *?rr?*, *?wr?*, *?gr?*, *?gw?*, *?rf?*, and *?wf?*.⁶

From Build Table 1, the following events are defined: (1) $r:s-rr$, (2) $s:_r+rr$, (3) $r:s+gr$, and (4) $s:r-gr$. The first and third events mark off $\$reader\$$:?READER?:?rr? and $\$reader\$$:?READER?:?gr? respectively. The second and fourth events mark off **req**: $\$scheduler\$$:?SCHEDULER?:?rr? and $\$scheduler\$$:?SCHEDULER?:?gr? respectively. Build Table 2 marks off the similar request and grant messages of the writer and scheduler actors. Build Table 3 marks off **req**: $\$reader\$$:?READER?:?rf?, while Build Table 4 marks off **req**: $\$scheduler\$$:?SCHEDULER?:?rf?. Build Table 5 marks off both the writer and scheduler requirements for the *?wf?* message.

⁵At this point, the ?IDLE? partition has not yet been defined as either a wait state, a functional operation state, or as a collapsed behaviour. As such, the specification is incomplete.

⁶All of the states of the scheduler EBD are known to be wait states, therefore, they are not defined as layered partitions.

With these tables, it has been shown that all of the requirements of the top-level partitions: \$reader\$:"READER?", \$writer\$:"WRITER?", and \$scheduler\$:"SCHEDULER?", are complete and consistent. The Build Tables for the basic models are missing, as are the Build Tables for the layered partitions within the "READER" partition.

Chapter 6

Example System Design

This chapter will cover portions of a large system design to illustrate the specification, design, and validation process suggested in this thesis. The problem in question is a sonar system requiring signal processing, tracking (which detects potential sources of interference), along with two concurrent displays which also act as input and control stations. The software is to be developed within a configuration of four embedded processors with the software being written in Ada (except the signal processing which is to be written in assembler). There is no direct support by a distributed operating system, but a large shared memory is available along with interrupt capabilities and a shared data bus.

This chapter is subdivided into the following sections: 6.1 which describes the system analysis and development of the initial EBD specification for one of the subsystems, 6.2 which allocates the EBDs defined in 6.1 to actors, and defines a potential break down for assigning the work to a team of designers, and 6.3 which describes the development and validation of the tasking model for the subsystem defined in 6.1. This final section references Appendix D.

6.1 Analysing and Modelling the System

This section is organized according to the development steps defined in Section 3.3.1, and summarized in Figure 6.1.

Analysis and Modelling Steps:

1. **Analyse the Given** - Perform a high-level, introductory review and analysis of the system
2. **Define the System Entities** - Perform entity analysis at the system level.
3. **Initial Analysis of the Functional and Behavioural Requirements** - Analyse the requirements in terms of the system entities.
4. **The Development Environment** - Review the development environment know the development domain issues.
5. **Defining the Subsystems** - Allocate the system entities to the processors of the known distributed framework.
6. **Process Abstraction** - Develop the system design; i.e., place the system entities within a useful model for defining the system behaviour.
7. **Distributed Interaction Commitment** - Design the methods of distributed interaction.
8. **Services for Distributed Interaction** - Define the collapsed behaviours to be used when defining subsystem behaviours with distributed interactions.
9. **EBD Modelling** - Perform EBD modelling for the subsystems.

Figure 6.1: Analysis and Modelling Steps

Step 1. Analyse the Given

Each screen is divided into several display areas as illustrated in Figure 6.2:

1. A large section of the screen is devoted to the display of the spectral data. It is shown as *SPA* (Spectral Data Area) on Figure 6.2. There are several selective formats for this display area. Data which has been detected and is thus monitored by the tracking algorithms, are visually, and appropriately annotated with *track markers*. A cross-hair cursor is provided to allow the operator to obtain detailed information from the signal and track processing on a particular point or track marker on the display. The cross-hair cursor is positioned by a trackball.
2. There is a small area displaying the information corresponding to the position of the cross-hair cursor. This is labelled as *CHC_Info* in Figure 6.2.
3. There is an area which displays other textual data (*TA* in Figure 6.2), and allows editing of signal and track processing parameters, and operator supplied data on tracked signals. This area also supports several different display formats and an alpha-numeric cursor. The cursor is positioned using the arrow keys of a keyboard, which also supports the entry of parameter edits.

Figure 6.2: Display for Example Sonar System

4. There is an area which is static: i.e., the format never changes. Here, information such as the current time is displayed. It is labelled as *STA* (Static display Area) in Figure 6.2.
5. There is an error or warning area (*ERR*) where the operator is notified if an entry, or request is not valid under the current configuration: e.g., a parameter edit is entered out of range.
6. There is an area supporting soft-button entry of operator initiated requests (e.g. to change display formats). This area is managed by a joystick, and labelled *SB* in Figure 6.2.
7. There are also hard buttons on the console for initiating operator requests.

The system context diagram is given in Figure 6.3. It shows the system under

Figure 6.3: Context Diagram for Example Sonar System

development, the sensors, and two operator consoles. The main system-wide *threads* of processing are:

1. **New Sensor Data** -- Data arrives periodically at an interface between the sensors and the system. The data is transformed for display and tracking. After the tracking algorithms are complete, the display is again updated to reflect these results along with the spectral results.
2. **Spectral Display Format Change** - The operator selects (using soft or hard button inputs) from different types of display formats for the spectral data, and (by keystroke edit inputs) the range of the data within a display type. The local display adjusts to these selections.
3. **Cross-Hair Cursor Movement** - The trackball is used to move the cross-hair cursor across the spectral data. The movement is echoed locally, and the associated detailed spectral and tracking data is updated.
4. **Textual Display Format Change** - The operator selects (using soft or hard button inputs) from different types of display formats for the textual data. The local display adjusts to these selections.
5. **Processing Parameter Edit** -- The operator edits a processing parameter which is then used in the next cycle of processing. The new information is

Figure 6.4: Sonar System Design

updated at both display stations as necessary (i.e., if it is part of the selected or static displays of both stations).

6. **Operator Defined Data** – The operator adds information about a tracked signal. The added information is updated at both display stations as necessary (i.e., if it is part of the selected or static displays of both stations).
7. **Operator Initiated Tracking** – The operator requests tracking of a particular signal by placing the cross-hair cursor at the corresponding position and invoking the correct hard or soft button.

Step 2. Define the System Entities

The system entities are: (1) the *signal processor*, (2) the *track processor*, and (3) the two *displays* as shown in Figure 6.4.

Step 3. Initial Analysis of the Functional and Behavioural Requirements

Signal Processor – This entity periodically (behavioural requirement) executes the signal processing algorithms (functional requirements). Signal processing is performed in response to an external stimulus.

Track Processor – This entity periodically (behavioural requirement) executes the track processing algorithms (functional requirements). Track processing is performed in response to the completion of signal processing.

Display Processor - This entity displays the system data in response to: (1) the completion of signal processing, (2) the completion of track processing, and (3) operator entry of system maintained data. This entity displays the cursor positions and display formats chosen by the local operator.

Step 4. The Development Environment

The system is to be developed on a target of four embedded processors and a shared memory. The bulk of the software is to be written in Ada, and there is no operating system support for distributed interaction and shared data access.

Step 5. Allocating the System Entities within the Distributed Framework — Defining the Subsystems

To maintain functional cohesion, each of the system entities is allocated to a separate processor as shown in Figure 6.4.

Step 6. Process Abstraction

Relating back to the list of threads given in step 1, the inter-processor communications and shared data requirements are determined as follows.

1. **New Sensor Data** - The data produced by the signal processor is used by both the track processor and the displays. Therefore this data must be either transferred to these entities or stored in shared, accessible memory. Since this constitutes a very large amount of information, it must be maintained in the shared memory. Both the track processor and displays react in a stimulus/response scenario to the completion of the signal processing, therefore there is a need to support synchronization (timed or event driven) between the signal processor and the other embedded processors.

Similarly, the data produced by the track processor is too bulky to transfer to the displays within a message, therefore, it too is to be stored in the shared memory. The displays must also be synchronized to the completion of the track processing.

2. **Spectral Display Format Changes** - Changes to screen formats are local to a particular display, however, they require sporadic retrieval of system data

from the shared memory (as opposed to periodic retrieval as above).

3. **Cross-Hair Cursor Movement** – Movement of the cross-hair cursor requires local display updates (i.e., the cursor echo), and sporadic retrieval of currently displayed spectral and track information.
1. **Textual Display Format Changes** – See (2) above.
5. **Processing Parameter Edit** – An operator initiated change in processing parameters must affect the next cycle of the corresponding signal or track processing algorithm. This information is a candidate for message data, although it could be transferred through shared memory along with a signal of the change as was the case in (1) above. This thread indicates that there must be support for data transfer from either display to the track or signal processor. Process synchronization (i.e., with blocking sends) is not required in this case: the processing algorithms could reset their parameters before every cycle.

However, the operator has affected system data, and this must be reflected immediately on both displays (if the other screen contains an appropriate display format). Therefore there is a conditional synchronization required between the two displays.
6. **Operator Defined Data** – As above, operator inputs of system data must be reflected immediately on both displays.
7. **Operator Initiated Tracking** – An operator request to affect the tracking of a particular signal affects both displays immediately, and the next tracking cycle as in (5) above.

Step 7. Distributed Interaction Commitment

Step 6 indicates a need for shared memory, message passing, and synchronization. In this step, the designers commit to a particular scheme for inter-processor interactions. For the purpose of this example, the design will commit to a large shared memory area as shown in Figure 6.4, and note that in several instances there is a

single writer (the signal processor or track processor) and multiple readers, otherwise there are multiple writers and readers (for data which can be added or changed by either display).

Display updates can occur periodically or sporadically. To maintain consistency with the response, display updates will be triggered by a signal event. To avoid unnecessary work to re-display all areas of each display, the events will be named to indicate the type of update (e.g., spectral data update, track data update, parameter update etc.).

For this example, the design will commit to protected read and write access of the shared memory, and message passing via queues in shared memory and interrupt triggers.

Step 8. Services for Distributed Access and Interaction

The interactions and shared memory accesses are designed using a client/server model where each subsystem, allocated to a processor, represents the client making calls to the *peer* entity acting as a server (cref. § 3.3.1). The following procedural interface exists between the subsystems and peer:

read(IN:*data_identifier*,OUT:*data*)

write(IN:*data_identifier*,INOUT:*data*)

put_message(IN:*msg*)

get_message(OUT:*msg_list*)

The peer entity is responsible for hiding the shared memory protection mechanism and synchronization mechanisms (signalling, waiting, and queueing) for the events.

Step 9. EBD Modelling

For this example, the display subsystem entities will be modelled. Each display can be broken down into the following entities: (1) the spectral data display area (SPA), (2) the static display area (STA), (3) the textual display area (TA), (4) the error message display area (ERR), (5) the cross-hair cursor information display area (CA), (6) the cross-hair cursor display (CHC), (7) the trackball input handler (TRK), (8)

the soft button display (SB), (9) the joystick cursor display (JC), (10) the joystick input handler (JSTK), (11) the soft button interpreter (SOFT), (12) the alphanumeric cursor display (ANC), (13) the keyboard and hard button input handler (KBD), (14) the keystroke interpreter (KEY), (15) the system event interpreter (EVT), and (16) the display driver (DSPLY). The main processing threads are as follows:

1. **System Event** — A system event is made available to the displays through the distributed access, peer subsystem (i.e., via *get_message*). The event triggers display updates in the static display area, the spectral display area, and/or the textual display area.
2. **Operator Event with Local Effects** — The operator initiates an input which affects the display locally; e.g., the operator moves one of the cursors, or changes a display format.
3. **Operator Event with System-wide Effects** — The operator initiates an input which affects the system data, and must therefore be saved in shared memory, and reflected at both this and the other display station.

These entities, and their relationships are illustrated in Figure 6.5. The corresponding EBDs are shown in Figures 6.6 through 6.21. The diagrams present the top-level behaviours of each entity; a discussion of each is provided below.

SPA (Spectral Data Display Area: Figure 6.6) — In the *start spa* state the initialization of the spectral area display entity is performed; e.g., a default screen format is defined. When this entity receives the subsystem wide *startup* broadcast event, it transitions to the *generate spa* state. Within this operation state, the selected display format is produced. This includes the spectral data, the scales and its corresponding annotation, and the markers defined by the track processing. The peer subsystem is used to read data from the shared memory. Some of the data (e.g. ranges, and marker position details) is saved for use by the CHC entity which determines the information required for display with respect to the cross-hair cursor position and the data it overlaps.

Figure 6.5: Entity Relationships for the Display Subsystem

For the rest of its lifetime, this entity is event driven. Updates to the screen are required when a corresponding system event occurs, or the operator has requested a format change. The unspecified sender, (*_I* for Input), refers to the KEY and SOFT entities. A change to the screen format is given a higher priority to avoid updating a display which is to be removed in the next cycle of the SPA entity. The functionality of states 3 and 4 (*update central data* and

Figure 6.6: SPA EBD

update tracking markers respectively) are included in state 1, but separated out for system data changes as opposed to display format changes to avoid unnecessary access to the shared system memory.

It should be noted that operation states 1, 3 and 4 require a relatively large amount of processing.

STA (Static Display Area: Figure 6.7) — Once initialized, the static display area entity simply responds to system events by updating those segments of the display affected by the event type.

TA (Textual Display Area: Figure 6.8) — The textual area is much like the spectral display (SPA). In this case, the system event might not pertain to the particular display format, and in this situation, it is simply ignored by the *Update Test* operation. Changes to system memory initiated at this operator station, are reflected immediately as a result of an event sent by the keystroke interpreter where the change is established (i.e., *update_TA(type)*). Screen format changes are, again, given a higher priority.

ERR (Error Message Display Area: Figure 6.9) — After initialization, the

Figure 6.7: STA EBD

Figure 6.8: TA EBD

error message display area is toggled from blank to indicating an error in response to the evaluation of a soft or hard button input, or a keyboard edit.

CA (Cross-Hair Cursor Information Display Area: Figure 6.10) --- The cross-hair cursor information display area displays the relevant information of the

Figure 6.9: ERR EBD

Figure 6.10: CA EBD

overlapped spectral and tracking data in response to the movements of the cursor. The data is determined from the information saved by the SPA entity as discussed above. Note, however, that this is not a critical function albeit a time consuming one: i.e., there is apparently room for specification refinement as necessary.

CHC (Cross-Hair Cursor Display: Figure 6.11) — The cross-hair cursor entity displays the cross-hair cursor in response to trackball movements.

TRK (Trackball Input Handler: Figure 6.12) — The trackball entity waits for interrupts from the corresponding external source. The associated data is passed on to the CHC and CA entities.

SB (Soft Button Display: Figure 6.13) — The soft button display entity updates the variable display in response to a corresponding operator request (i.e.,

Figure 6.11: CHC EBD

Figure 6.12: TRK EBD

a soft button function itself). To coordinate a series of inputs, this entity is synchronized with the soft button interpreter entity (SOFT).

JC (Joystick Cursor Display: Figure 6.14) --- The joystick cursor entity displays this cursor in response to joystick movements.

JSTK (Joystick Input Handler: Figure 6.15) --- The joystick input handler entity waits for interrupts from the corresponding external source and retrieves the data from the appropriate location. The data is passed on to the JC entity.

Figure 6.13: SB EBD

Figure 6.14: JC EBD

Figure 6.15: JSTK EBD

SOFT (Soft Button Interpreter: Figure 6.16) — The soft button interpreter entity determines the function associated with an input. For changes to the soft button display, the SB entity is notified. For display format changes, the spectral or textual display area entities are informed. Otherwise, the input affects the system memory, and therefore, the peer entity is used to read and/or write a change to the data, and send out the corresponding system event via *put_message*. The update is reflected at this display station immediately by issuing the *TA-update_TA(type)* event to the textual area entity.

ANC (Alpha-numeric Cursor Display: Figure 6.17) — The alpha-numeric cursor display entity repositions the cursor in response to operator inputs.

KBD (Keyboard and Hard Button Input Handler: Figure 6.18) — The keyboard and hard button input handler entity waits for interrupts from the corresponding external source and retrieves the data from the appropriate location. The data is passed on to the KEY entity.

KEY (Keystroke Interpreter: Figure 6.19) — The keystroke interpreter eval-

Notes: Not_Menu_Reset =
 [not menu reset AND spa change]//(SPA-change_SPA(params),ERR-ok)
 OR
 [not menu reset AND ta change]//(TA-change_TA(params),ERR-ok)
 OR
 [not menu reset AND spectral data change]//(SPA-refresh_SPA(type),
 TA-TA_refresh(type),ERR-ok)
 OR
 [not menu reset AND other chnage]//(TA-refresh_TA(type), ERR-ok)
 OR
 [not menu reset AND local echo]//(DSPLY-display(command), ERR-ok)
 OR
 [not menu reset AND Local error]//(DSPLY-display(command), ERR-err)

Figure 6.16: SOFT EBD

Figure 6.17: ANC EBD

Figure 6.18: KBD EBD

uates the input and acts accordingly. Therefore, a joystick enter is passed to the SOFT entity while an arrow key is passed to the ANC entity. An edit character is echoed and buffered until the return key is depressed. An edit submission, or other function which affects the system memory, is handled as was the case for the SOFT entity. Display format change requests are also managed as in the SOFT entity.

EVT (System Event Interpreter: Figure 6.20) — The system event interpreter interacts with the peer entity through the *get.message* service to affect the display subsystem as a result of system events.

DSPLY (Display Driver: Figure 6.21) — The display driver entity converts display commands coming from the other display entities into commands which the external display hardware can interpret.

6.2 Allocation and Decomposition of Work

Step 10 and 11 defined below follow the discussion of Section 4.1.1.

Step 10. Allocation of EBDs to Actors

SPA Display – Because of the complexity of this display area (which includes display of the spectral data, markers defined by the tracking algorithms, and the data and range annotations), the SPA entity is assigned to its own actor as shown in the Actor Table of Figure 6.22.

Notes: Operator Request =
 [spa change]//(SPA-change_SPA(params),ERR-ok)
 OR
 [ta change]//(TA-change_TA(params),ERR-ok)
 OR
 [spectral data change]//(SPA-SPA_refresh(type),
 TA-refresh_TA(type),ERR-ok)
 OR
 [other chnage]//(TA-refresh_TA(type), ERR-ok)
 OR
 [local echo]//(DSPLY-display(command), ERR-ok)
 OR
 [Local error]//(DSPLY-display(command), ERR-err)
 OR
 [joystick button]//(SOFT-soft_enter, ERR-ok)
 OR
 [anc movement]//(ANC-ANC_move(direction, ERR-ok)

Figure 6.19: KEY EBD

Figure 6.20: EVT EBD

System Events Handler (SEH) – The EVT (system events interpreter) and STA (static display area) entities are allocated to this actor. The static display is not large, and sections can be updated relatively quickly in response to appropriate system events as described in Figure 6.23.

Figure 6.21: DSPLY EBD

ACTOR TABLE**Actor:** \$spa\$ (Spectral Display Area)**EBDs:** SPA**ACMs:** \$spa\$**Basic Model:** Design Fragment of Figure 4.11 is used.

Top-Level Partitions	Comments
?SPA?	Operations require lengthy processing.
Layered Partitions	Comments
SPA:?START SPA?	set default screen
SPA:?GENERATE SPA?	includes functionality of ?UPDATE SPECTRAL DATA? and ?UPDATE TRACKING MARKERS? below.
SPA:?UPDATE SPECTRAL DATA?	
SPA:?UPDATE TRACKING MARKERS?	
Comments	
The operations provide data (e.g. ranges and tracking marker positions and details) used by the CA entity.	

Figure 6.22: SPA Actor Table

TA Display – The textual area displays are also rather complex, and can require a fair amount of processing, therefore the TA entity is allocated to its own actor as shown in Figure 6.24.

Input Interpreter (II) – This actor consists of the KEY, SOFT, SB, ANC, and ERR entities as described in Figure 6.25.

ACTOR TABLE**Actor:** \$seh\$ (System Events Handler)**EBDs:** *EVT***ACMs:** \$evt\$*STA*

\$sta\$

Basic Model: Build Table Reference 1

Top-Level Partitions	Comments
?EVT?	Interacts with peer subsystem through <i>get_message</i>
?STA?	Display requires minimal processing (however, this includes system memory read accesses).
Layered Partitions	Comments
STA:START STA?	Set STA labels.
STA:GENERATE STA?	Present entire display.
STA:UPDATE STA?	Update portion of screen affected by the system event
EVT:START EVT?	
EVT:GET SYSTEM MESSAGES?	Using the services of the peer entity, all events destined for this subsystem are obtained from shared memory.
Comments	

Figure 6.23: SEH Actor Table

CHC Information (CHC Info) – The CA entity is allocated to its own actor as shown in Figure 6.26. There is a great deal of processing required to interpret the cursor position and determine the correct information to place in this display area. This display may not be as critical as the CHC display which echoes the cursor movements; i.e., it may be sufficient to refine the specification such that this function is performed at a lower rate than the function of the CHC echo entity. As such they are separated into individual actors at this point in the design. (Again, there may be room for specification refinement in the CA entity.)

ACTOR TABLE

Actor: \$ta\$ (Textual Area Display)

EBDs: ta

ACMs: \$ta\$

Basic Model: Design Fragment of Figure 4.11 is used.

Top-Level Partitions	Comments
?TA?	Operations require lengthy processing.
Layered Partitions	Comments
TA:START TA?	set default screen
TA:GENERATE TA?	includes functionality of ?UPDATE TEXT?
TA:UPDATE TEXT?	below.
Comments	
The operations provide data (e.g. values of editable parameters, and locations accessible by the alpha-numeric cursor) used by the KBD and KEY, and ANC entities respectively.	

Figure 6.24: TA Actor Table

Operator Input Handler (OIH) – Here, several entities are co-allocated to represent the operator inputs: CHC, TRK, JC, JSTK, and KBD. Each of these entities either accepts an external event from the operator console or performs a minimal, but time critical function for local, visual feedback. This allocation is shown in Figure 6.27.

Display (DISPLAY) – The DSPLY entity is allocated to its own actor as shown in Figure 6.28. It is expected that this processing will be given a higher execution priority over most of the other actors to avoid a bottleneck of display requests.

The sixteen entities of the display subsystem are allocated to seven actors. The corresponding top-level partition tables for the \$seh\$ actor are shown in Figures 6.29 and 6.30. The remaining top-level partition tables are given in D.1. These tables represent the information which can be generated automatically from the EBD specifications and the allocation of EBDs to actors. There are no entries in the *Build Table Reference* columns at this stage in the design process therefore they

ACTOR TABLE**Actor:** \$ii\$ (Input Interpreter)**EBDs:** *key**soft**sb**anc**err***ACMs:** \$key\$

\$soft\$

\$sb\$

\$anc\$

\$err\$

Basic Model: Build Table Reference 16.

Top-Level Partitions	Comments
?KEY? ?SOFT? ?SB? ?ANC? ?ERR?	
Layered Partitions	Comments
KEY?START KEY? KEY?EVALUATE BUTTON EQUIVALENT? SOFT?START SOFT? SOFT?EVALUATE BUTTON EQUIVALENT? SB?START SB? SB?CHANGE SOFT BUTTON MENU? ANC?START ANC? ANC?DEFAULT POSITION? ANC?DETERMINE POSITION? ERR?START ERR? ERR?CLEAR ERROR? ERR?SET ERROR?	Determine hard button or keyboard input and act accordingly. Determine soft button input and act accordingly. Button layout determined here is used by the <i>soft</i> entity. Default ANC position at startup. The data defined here is shared with the <i>ta</i> entity.
Comments	
The operations of this entity can be quite complex requiring read and write access of system memory, and the issuing of system events.	

Figure 6.25: II Actor Table

ACTOR TABLE

Actor: \$chc.info\$ (CHC Information)

EBDs: *ca*

ACMs: \$ca\$

Basic Model: Build Table Reference 15.

Top-Level Partitions	Comments
?CA?	Is it necessary to respond to every <i>trk+chc_pos(x,y)</i> event?
Layered Partitions	Comments
CA:?START CA?	This functions uses data defined by the SPA entity.
CA:?DETERMINE OVERLAP?	
Comments	

Figure 6.26: CHC Info Actor Table

are shown as blank in all cases. The layered partition tables are not provided for this example because the EBDs do not define lower levels of behaviour as yet. The entities defined by data modelling (e.g. for the joystick cursor position and the data shared between the input interpreter actor and the textual area actor) are also not considered at this time.¹

Step 11. Decomposition of Work

1. The System Events Handler actor is a relatively small function and could be allocated to a single designer.
2. The SPA Display actor includes a complex set of functional requirements. Because it shares format and range information with the CHC Information actor, the two should be developed within a cohesive team of developers.
3. The TA Display actor is also very complex in terms of functionality. This actor shares positional and value information with the Input Interpreter actor

¹It is expected that the data modelling stage of developing EBD specifications will come after the analysis of externally triggered threads. Data modelling introduces data stores and protected data required by the system, and local EBD behaviours. It is discussed in [16].

ACTOR TABLE

Actor: \$oih\$ (Operator Input Handler)

EBDs: *chc*

ACMs: \$che\$

trk

\$trk\$

jc

\$jc\$

jstk

\$jstk\$

kbd

\$kbd\$

Basic Model: Build Table Reference 22.

Revision History: See - Build Table Reference 20

Top-Level Partitions	Comments
?CHC? ?TRK? ?JC? ?JSTK? ?KBD?	
Layered Partitions	Comments
CHC: ?START CHC? CHC: ASSEMBLE DISPLAY COM: AND? TRK: ?START TRK? JC: ?START JC? JC: EQUATE TO BUTTON POSITION? JSTK: ?START JSTK? KBD: ?START KBD?	The joystick position determined here is used by the SOFT entity
Comments	
The entities of this actor forward cursor and keystroke inputs to the H actor, and provide immediate visual feedback for the joystick and trackball (cross-hair) cursors.	

Figure 6.27: OIH Actor Table

(i.e., with respect to ANC positioning and edit values). As above, these actors should be developed by a team.

1. The Operator Input Handler actor is behaviourally rather complex: it consists of several small but time critical functions. This actor should be assigned to

ACTOR TABLE**Actor:** \$display\$ (Display)**EBDs:** *DSPLY***ACMs:** \$dsply\$**Basic Model:** Display Fragment of Figure 4.11 is used.

Top-Level Partitions	Comments
?DSPLY?	
Layered Partitions	Comments
DSPLY:?START DSPLY?	The display command is interpreted and used to produce the necessary visual response.
DSPLY:?INTERPRET DISPLAY	
COMMAND?	
Comments	
This actor is expected to be given a high priority.	

Figure 6.28: DISPLAY Actor Table

PARTITION TABLE – Top-Level Partition**Partition:** ?EVT?**Parent:** \$seh\$

Message	Build Table Reference
?update_STA(type)?	
?update_TA(type)?	
?update_SPA(type)?	
?start?	

Figure 6.29: Primitive Partition Table for ?EVT?

PARTITION TABLE – Top-Level Partition**Partition:** ?STA?**Parent:** \$seh\$

Message	Build Table Reference
?update_STA(type)?	
?start?	

Figure 6.30: Primitive Partition Table for ?STA?

Figure 6.31: Tasking Relationships for the Display Subsystem

a small team of one or two developers.

The allocation within the teams defined above depends on the decomposition of the operation states (e.g., by display format for the SPA and TA displays).

6.3 Tasking Model for Example Subsystem

This example will be extended to illustrate the design process for the tasking model of this subsystem. The tasking relationships defined as a result of the allocation step are illustrated in Figure 6.31, while the final design is shown in Figure 6.32. In this section, a discussion is given on the design fragment selection and consequential specification refinement processes, and the validation of contingencies. The presentation might appear to be rather ad hoc. It represents a design process (i.e., a stream of conscience as opposed to a design presentation or walkthrough); and,

Figure 6.32: Design for the Display Subsystem

as such, demonstrates: (1) the independence of separate design issues as they are defined by design fragments, and (2) how a structured and consistent document is obtained regardless of the order of design decisions. This section references the appendix sections D.2 for the final, refined EBDs, D.3 for the final versions of the Actor Tables, D.4 for the final versions of the Partition Tables, and D.5 for the Build Tables.² Finally, the validation of completeness is discussed.

Design Decision 1: SEH Basic Model — The basic model of this actor is designed using the DF-Inter-SEQ-1 design fragment which requires a static schedule for the co-allocated EBDs. This is expressed in the corresponding Actor Table, and described by Build Table 1 on Page 385 of D.5. The synchronization with the *start* system event at each entity is combined into a single receive. The remaining requirements are sequenced as if the functionality of the STA entity was merged with that of the EVT entity. The schedule presented is assumed to be correct, and thus the corresponding design contingencies are set to *confirmed*. This schedule is dependent on the *start* message to enter its processing loop, and therefore the specification contingency is also confirmed.

Design Decision 2: EVT sends *TA-update.TA(type)* — For this event interaction, the alternative design fragments are, (1) the jab, (2) the rendezvous, (3) the buffer, and (4) the courier. The first alternative (the jab) is not an option because it is not supported by the implementation language (Ada). The rendezvous solution depends on the eventual reception by the TA entity. This is defined in the *Performance Issues* section of this design fragment (DF-AsyncInter-IPC-2); i.e., even if eventual reception can be shown, there may be a need to consider any delay of the sender. Since the TA entity is required to perform extensive processing, there is the potential for unreasonable delays of the EVT entity because of the rendezvous. Such delays would affect the response times of other screen area updates to system events. The courier solution (DF-AsyncInter-IPC-3) does not apply because the **Get Systems Messages** state of the EVT entity is not a wait state, but rather one which has the EVT blocked awaiting a response from the peer entity.

²The “Properties and Comments” section of Build Tables is neither used nor shown for this example

For a buffer solution the queuing must be considered (DF-AsyncRx-IPC-3). The processing required at the TA entity in response to an event is substantial. With this alternative design, it is possible that several events arrive, destined for the TA, during this processing. However, the data to be displayed is always retrieved from the shared system memory which defines only the current system state (as opposed to the stages representing the different events). Successive updates would not provide any more information to the on-looking operator. Therefore, if several events of the same type arrive while the TA entity is busy, only one is relevant.

Consideration of the buffer solution has shown that the original specification could be over-constraining. The requirement to update the screens as a result of every event might be unnecessarily strict in this application. After further consideration, updating on the last event is judged to be a reasonable specification of the requirement. The specification is therefore refined; e.g., by (1) adding the *reset* directive to the *update_TA(type)* event to filter out repeated events, or (2) adding a buffer entity which acts to overwrite same events.

The first alternative is illustrated in Figure 6.33. Here, all events accepted by the *Wait Change* state of TA are filtered using the *reset* directive which clears the EI buffers after one event is returned to the EBD. Because these events cause the same type of processing to occur, and again, only the most recent request to update the screen is relevant in terms of the system data, the *reset* alternative is appropriate and reduces unnecessary processing. A design fragment for the specification fragment including the *Wait Change* TA state could be created for this example, but it would have to be very specific to this particular system specification; i.e., one which included all of the requirements of the TA *Wait Change* state and its satisfiable event set. This design fragment defines a buffer (e.g., *TA_Buffer*) as the receive_place for this satisfiable event set, and a gate of *TA_Buffer* (e.g., *put_msg*) as the receive_placeholder for the events: (1) *update_TA(type)*, (2) *refresh_TA(type)*, and (3) *change_TA(params)*. *TA_Buffer* is responsible for managing the requirements of the receive_place which are, to order the events according to the receiver defined priority; i.e., order the events such that the *change_TA(params)* event is given higher priority. This same design fragment could include all of the send event requirements

Figure 6.33: Refined Specification for TA Entity using the Reset Directive

for the sender entities (i.e., KEY, SOFT, and EVT), or be used in combination with an asynchronous send event design fragment such as DF-AsyncTx-IPC-2 which uses a rendezvous send that matches with the *put_msg* gate of the added *TA_Buffer* actor.

For this example, the second alternative is used; i.e., a buffer entity is added to the specification as shown on Page 369, where the *TA_Buffer* entity appears as a component of the finalized EBD set of D.2. Changes are also required and illustrated for KEY, SOFT, EVT, and TA. Here, all of the events in the satisfiable event set with *EVT+update_TA(type)* must go through this buffer; i.e., *_I+refresh_TA(type)* and *_I+change_TA(params)*. This alternative corresponds to a more intuitively reusable design fragment which covers all of the requirements of the new buffer entity. The design fragment, DF-Overwrite-Priority-Buffer-1, is given on Page 362. It defines a generalized version of a buffer which performs the overwrite, and priority queueing required for this example, and represents a design fragment which is both directly reusable, and reusable in the sense that it is easily modified to manage similar buffering requirements as will be shown under Design Decision 4. (Note that for the first alternative solution, which uses the *reset* directive, the designer must recognize the buffering characteristics of the wait state to select and apply a design fragment using a buffer actor for the multiple events.)

Given the refined specification, the TA_Buffer entity is allocated to its own actor. DF-Overwrite-Priority-Buffer-1 is used for its design as shown in Build Table 2 on Page 387 of D.5. The synchronous interaction with TA_Buffer is designed using DF-SyncInter-IPC-3 which covers an explicit rendezvous with message data, and is contingent on matching formal and actual data parameters. This is confirmed by the designer as shown in Build Table 6 on Page 391 of D.5. The asynchronous interactions between the EVT, KEY, or SOFT entity and the TA_Buffer entity are designed using DF-AsyncInter-IPC-5. Each interaction design requires consideration of a specification contingency which states that the messages involved must be received by TA_Buffer eventually. In all cases, the message is always an element of the current satisfiable event set for TA_Buffer, and therefore by definition of a fair selection policy, each message is accepted eventually. This argument is presented as a statement in the *proof* column of each Build Table: i.e., Build Tables 3 through 5 on Pages 388 through 390 of D.5.

In this analysis, the design fragments for the original specification have highlighted a needless constraint in the original specification, and were used to help refine the specification as a result.

Design Decision 3: EVT sends *SPA-update_SPA(type)* — Build Tables 7 through 11 on Pages 392 through 396 of D.5 show that the same arguments used for the TA actor above are applied to the SPA actor to design *SPA-update_SPA(type)* from EVT, *SPA-change_SPA(params)* from II, and *SPA-refresh_SPA(type)* from II. The new SPA EBD is given in D.2, and the corresponding Actor and Partition Tables in D.3 and D.4 respectively. The validation of contingencies is also similar to the presentation given above.

Design Decision 4: TRK sends *CA-trk_pos(x,y)* — The CA entity requires a relatively large amount of processing to update the cross-hair cursor information area of the screen. It would be very difficult to implement this evaluation to perform at the same rate as the fastest motion of the trackball cursor without hindering the performance of other functional requirements for this subsystem. This might suggest that the *CA-trk_pos(x,y)* event can be buffered in an overwrite buffer as above. This refinement is reflected in the CA, TRK, and CHC_Pos_Buffer EBDs of D.2.

and Build Tables 12, 13 and 14 on Pages 397 through 399 of D.5. In this case, DF-Overwrite-Priority-Buffer-1 is modified for use when there are no requirements to queue different events based on some priority scheme. The result is DF-Overwrite-Buffer-1 of Page 362.

Design Decision 5: CHC_Info Basic Model - The CHC_Info actor (encapsulating the CA entity) has a relatively simple behaviour for which a design fragment could be defined giving an appropriate basic model which eliminates interactions with an ACM. A periodic loop might handle the requirements. If such a design fragment is defined, i.e., DF-Periodic-Loop-1 given on Page 363 of C.3, it is found that this fragment is dependent on the guarantee that the period equates to the arrival rate of the events. In this case, the events are buffered at an overwrite buffer, *CHC_Pos_Buffer*, and the arrival rate of events is non-deterministic. After further consideration, the designer decides to investigate a periodic update of this display area further. A periodic display would provide a guaranteed visual response time during rapid movement of the cursor (and could use checks to forego the time consuming evaluation of system data if no movement has occurred during the last update period). After consultation with the analysis team and customer, this idea is adopted and reflected in a refined CA specification as given in on Page 374 of D.2.

A new design fragment is added to the framework. It is similar to DF-Periodic-Loop-1 in that the resultant design looks the same, but the matching specification explicitly defines the generic version of the new CA EBD which has an explicit periodic loop specification. This fragment is also found on Page 364 of C.3. The design for this example is found in Build Table 15 on Page 400 of D.5.

This design fragment covers the behaviour of the entire EBD, and represents an explicit transformation of the specification, therefore, there are no specification contingencies. A design contingency requiring type compatibility between the actual and formal data parameters is confirmed by the designer.

In this design step, a design decision was made, but conformance could not be validated between the corresponding design fragment, and the specification; i.e., with DF-Periodic-Loop-1. As a result, the specification was analysed and refined. In the end, the design conforms to the specification. This is a different refinement process

from that discussed earlier where consideration of design fragments indicated that refinement was possible. Here, a creative design decision, and subsequent application of the corresponding design fragments indicating that the specification required further analysis to achieve conformance which subsequently lead to a refinement.

Design Decision 6: II Basic Model — The basic model of this actor is designed using the DF-Inter-SEQ-1 design fragment which requires a static schedule for the co-allocated EBDs. This is expressed in the corresponding Actor Table and described by Build Table 16 on Page 401 of D.5. The synchronization with the *start* system event at each entity is combined into a single receive. The remaining requirements are sequenced by merging the behaviours into a single sequential representation which branches out according to the type of input processing requested. The designed schedule is assumed to be correct, therefore all design contingencies are confirmed, and the designed schedule is used to confirm the specification contingencies.

Design Decision 7: Any entity sends *DSPLY-display(command)* — This event is issued by most of the entities in the subsystem either at the top-level behaviour or within an operation state. The display updates interact with the display hardware to provide the visual response required by the system. The display driver actor *display* is likely to run at a relatively high priority (as is common among device drivers in embedded systems). As a result, a design decision to use a rendezvous for this asynchronous transfer appears reasonable; i.e., since the display driver is at a higher priority, the remaining lower priority actors would block when the *DSPLY* entity is active anyway. Therefore the rendezvous solution does not appear to adversely affect the performance, and provides a “simple” solution. This decision is illustrated in Build Tables 17, 18 and 19 on Pages 404 through 406 of D.5. The single specification contingency references the proof labelled *?Display-Queue?*. Again, the contingency requires the event to be received eventually by the receiver entity. Because the event is always a member of the current satisfiable event set (after initialization), it is guaranteed to be received eventually.

Design Decision 8: OIH Basic Model — The basic model of this actor is de-

signed using the DF-Inter-SEQ-1 design fragment which requires a static schedule for the co-allocated EBDs. This is expressed in the corresponding Actor Table and described by Build Table 20 on Page 407 of D.5. The synchronization with the *start* system event at each entity is combined into a single receive. The remaining requirements are sequenced by merging the behaviours into a single sequential representation which branches out according to the type of input processing requested. Once again, the schedule is assumed to be correct, and thus all design contingencies are confirmed by the designer. Specification contingencies 1 through 3 are also confirmed using the designed schedule. The 4th contingency is proven using the nature of the CHC_Pos_Buffer entity. This contingency is concerned with the *chc_pos* message and how the corresponding synchronous interaction could block the combined behaviour. The event in question is always a member of the current satisfiable event set for CHC_Pos_Buffer.

Design Decision 9: System *start* event — The EBDs define this as a broadcast receive event. Since this is not supported explicitly by the implementation language, the rendezvous is used. A contingency of the send event primitive is to ensure appropriate startup sequences in light of the fact that these events are sequenced instead of running in parallel, careful thought must be used to define the order in which these actors are invoked; e.g., the DISPLAY actor should be initialized before the TA and SPA actors. Although the sender side of this interaction has not yet been defined in the system or display subsystem specification, the interaction is considered in Build Table 21 on Page 419 of D.5 to highlight this notion of ordering. The sender entity is listed as *system*. The designer has confirmed that a correct ordering of *start* messages can be determined. There is a specification contingency requiring that the *start* message be received eventually by all entities in the system. This is confirmed by inspecting the EBDs.

Design Decision 10: KBD sends *KEY-keypressed(valuc)* — The OIH actor must respond quickly to the incoming input interrupts to reflect updates of the cursor positions, and edit characters. As such, it does not seem reasonable to block the OIH actor in a rendezvous with II or a buffer to send the *keypressed* message. Therefore, a courier solution is selected for this event transfer. DF-AsyncInter-IPC-3 cannot

be used directly because one of the design contingencies requires that the parent entity be allocated to its own actor. Therefore DF-CASE-SEQ+Couriers, which allows one or more couriers to be used with simple co-allocated server behaviours is introduced and used (see Page 365 of Appendix C.3). The basic model for the OIH actor is thus redefined. The original design fragment is rejected and the new design fragment is given in Build Table 22 on Page 411. The rejected build is referenced by the Actor and Partition Tables as shown in Table 6.27 and D.4 respectively.

The design adds the courier receive event to the satisfiable event set which waits for an incoming request (i.e., an interrupt from the keyboard, joystick, or trackball). The processing requires the OIH to return one queued message destined for the KEY entity for each accepted request of the added courier actor. This addition should not produce unacceptable delays in responding to the remaining input types as discussed in the Performance Issues section of the Build Table.

The designer must prove that the synchronous interaction on the *chc_pos* message will not block the combined behaviour of the actor. The proof is given as an argument which states that *CHC_Pos_Buffer* is known to accept this event in all of its states. The design contingency states that the receipt of the *keypressed* message requires a synchronous receiver to match the synchronous send of the courier. DF-AsyncRx-IPC-2 is used for this requirement as shown in Build Table 23 on Page 414 of D.5; i.e., this design decision generates the appropriate *match* statements for the design contingency of Build Table 23.

There are also design contingencies requiring the *external* senders of *trk_int(x,y)*, *jstk_int(x,y)*, and *kbd_int(char)* to match the synchronous receives designed for these events. The send sides of these interactions are not requirements of the subsystem specification given for this example, and therefore the associated design contingencies cannot be validated at this point in the system design.

Design Decision 11: Events sent to externals — DSPLY and EVT both send events to external entities which are not fully defined by the specification given for this example. DSPLY sends the *external-display_command(code)* event while EVT issues events in the collapsed behaviour of the *Get System Messages* state. The collapsed behaviour is not defined as part of this subsystem, and is therefore

not addressed by this example. The event sent by DSPLY is designed using DF-AsyncTx-IPC-3 which defines a rendezvous send of a message with message data. It is covered by Build Table 24 on Page 415 of D.5. The receiver side cannot be designed with the given specification; and therefore, the specification and design contingencies are not validated at this time.

Validating Completeness — To validate the completeness of this subsystem design, each Build Table is checked against the Actor Table and Partition Table hierarchy. In doing this process, it is found that every message listed in the Partition Tables of D.4 is covered by at most one design fragment except where an unspecified sender is defined for a given message, in which case multiple design fragments relate to a single message entry (e.g., the *display(command)* message of the DSPLY entity). The message requirements are found through either: (1) interaction_placeholders of the SEQ design fragments, or (2) direct implementation of an interconnection for IPC design fragments. The Build Table Reference column of every top-level Partition Table in D.4 is completed by the correct Build Table reference according to the completeness validation check. Although the design is complete in terms of coverage, it is not complete in terms of correctness validation; i.e., the Build Tables indicate some unresolved contingencies; namely those which are concerned with external entities.

6.4 Summary

The design given in this chapter illustrates the potential power of the design fragment approach to design. In particular, it shows how a specification is defined and refined by the design process; all the while ensuring conformance between the specification and the design. Two examples of detecting how a specification is refined are given. They are: (1) the applicable design fragments may not produce a suitable design, as such the specification is re-evaluated, and (2) a design decision made by an experienced designer might not conform to the specification therefore the two must be evaluated to see which is more appropriate. In the end, the refined specification must conform to the design. The original specification is still a potentially useful

product; i.e., it can be used to develop a similar product in a different development domain.

In some cases, the basic model of the actors could be greatly simplified by a straightforward and typical sequential program as opposed to one which uses the EBD through the ACM. To ensure conformance, these basic models would have to be developed as design fragments; i.e., the entire EBD is mapped to a generic but properly defined *basic model design fragment*.

This example also illustrates the potential power of the design documentation. The Actor and Partition Tables can be generated automatically given an allocation of EBDs to actors, and every Build Table definition can be kept even if the design is eventually rejected. This way, knowledge gained by the designers is kept, and can be used for alternative designs, or as a reference to evaluate similar design decisions. The ad hoc design process illustrated in the previous section resulted in the consistent and well structured design document shown in Appendix D; which itself, is used to validate design completeness, and illustrate the validation of correctness.

4

PM-131 x4 PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

1.0

28

25

32

22

1.1

20

1.8

1.25

1.4

1.6

Chapter 7

Conclusions

The goal of this thesis research is to develop a design approach for strong traceability of behavioural requirements in reactive systems. The method is to include the software specification, a systematic design approach for translating the specification into a design, and a validation process to illustrate strong traceability between the specification and its design. The thesis suggests that this type of approach is not currently practiced. Current popular trends provide either complete traceability (for correctness preserving transformations using formal relations), or weak traceability (for ad hoc, and creative design methodologies based on informal, trial-and-error relations).

Correctness preserving transformations have the advantage that they are correct, and thus improve the level of predictability. Their disadvantage comes from the limited, and often non-applicable domain knowledge of the transformers, and the complex process of defining a complete, correct, and appropriate specification for transformation. Ad hoc methods are more practical and more popular for the design of reactive systems with soft timing constraints. These methods use the designer's knowledge of the problem and the target domain, to generate a solution which can be implemented directly. However, the validation of the design is through cross-reference tables, prototypes, and case-based testing, most of which is performed manually, and provides only weak traceability between the specification and its design.

The objective of this thesis is to define a design process which considers the advantages of both automatic and informal techniques; i.e. conformance between a specification and its design, and designer expertise on target specific issues. A process is defined which uses formally defined *design fragments* in a systematic, but creative way, such that *strong traceability* is achieved.

The design fragments represent formal relations to the specification under the conditions of the associated contingencies. Much of the conformance proof is completed by the development of the design fragments, which is based on an underlying design for actors called a *basic model*. The contingencies represent only those properties which are not enforced explicitly by binding the design fragment to the specification. Specification contingencies can be proven using formal or informal methods, but in either case provide strong traceability through their definition, and the documentation of their validation. Design contingencies require matching design decisions with the rigorously defined basic model requirements, or with other design decisions; and as such, also contribute to strong traceability. Properties and performance issues are argued proofs based on experience and insight, and are again fully documented to provide conformance traceability.

The EBD specifications are complete only in terms of the behaviour. The domain specific details (e.g., processor loading and process allocation, language constraints, timing and performance) are left to the designer. The designer uses domain knowledge and expertise to creatively, but systematically allocate entities of the specification to actors of the design, and complete the design by selecting and validating design fragments. Properties of the design fragments such as the communication paradigms, the complexity, and the performance are used to guide the designer's choices, while maintaining strong traceability.

As a result the design process produces a design whose behaviour: (1) is traceable with respect to the specification, (2) can be better understood by the designers and reviewers through well-structured and comprehensive documentation, and (3) conforms to the specification when the contingencies are satisfied.

7.1 Contributions

This thesis has defined: (1) a software specification model for reactive systems: *Entity Behaviour Descriptions*, (2) a verification technique for EBD specifications with unbounded process pools accessing bounded resources, (3) a technique for developing and documenting design fragments, (4) a design process which uses design fragments and leads to strong traceability, (5) a validation process to illustrate conformance through the illustration of completeness and correctness of the design, and (6) a documentation process to support strong traceability.

Together, these contributions define a systematic design technique for reactive system software. The separation of application-behaviour from communication semantics in the EBD model is used to structure an actor's design. Actors are defined to be a composition of five design components. Two, the ACM and its interface, are fixed and provide strong traceability of the application-behaviour. The actor's engine and interface are designed to a *basic model* which defines placeholders at which to "plug-in" inter-connections. These components satisfy the required communication semantics.

The concept of a basic model with "plug-in" inter-connections is used to define a technique for developing and documenting *design fragments* which represent independent, formal relations between specification segments and alternative, potential design solutions. Design fragments are *programming plans* suited to the design of communications in reactive systems. The ACMs, basic model, and design fragments satisfy much of the conformance issues between a specification and its design, leaving the designer with only smaller, and generally independent proof requirements (contingencies).

The independent nature of design fragments, and hierarchical format of EBD specifications, simplifies the documentation process. Each design decision is represented by a separate *Actor Table* or *Build Table* for allocation and design fragment decisions respectively. Each event requirement is highlighted within a *Partition Table*, and accessible through a consistently defined tree schema. Strong traceability is supported by a structured, consistent (checkable), and complete (including the validation) documentation process. Each contribution is discussed individually, and

in more detail below.

Contribution 1: Entity Behaviour Descriptions (EBDs)

For software specification, the goal is to define a model for specifying *inter-* and *intra-*entity behaviour via relatively simple, yet flexible, state machines defined with an explicit understanding of, and formally specified operational semantics to support, the separation of application-behaviour from the semantics of the communications. This was achieved through the definition of EBDs. They were developed in consideration of several properties highlighted by the background study. Based on their similarities to accepted specification techniques, EBDs are a suitable specification model for reactive system software; and, as proven by this thesis, they are also appropriate for use in a design process for strong traceability of behavioural requirements.

EBDs are flexible. They allow the analyst to specify the behaviour of communicating entities using three common communications paradigms (asynchronous, synchronous, and transient broadcast), several event ordering disciplines (based on arrival time, or application defined priority attributes), and several event selection schemes (based on attributes including arrival time, data values, run-time binding of communications partners, and overwrite semantics). EBDs are comprehensive. The assumed management of incoming events is minimized, and as such, is a property of the application-specification as opposed to the underlying communications model; i.e., the specification must explicitly order, accept, screen, and discard events. EBDs support both internally and externally-triggered state transitions with a single notation, and therefore can be used to define individual entity behaviours (flowchart models) as well as the behaviour defined by a system of communicating entities (communicating state machines). EBDs offer an abstraction method to allow for hierarchical behaviour descriptions. The only constraint on a *collapsed behaviour* is that it complete eventually. Finally, but most importantly, EBDs support an explicit separation of application-behaviour and communication semantics in reactive system specifications.

Contribution 2: Verification of EBDs with Unbounded Process Pools

The specification of reactive systems is a complex exercise. Verifiability is therefore an important feature for the specification model, and reachability analysis (among other static analysis techniques) can be adapted to deal with EBD specifications. Although not a specific goal for the thesis, EBDs proved to be appropriate for reachability analysis with extensions for managing unbounded process pools. A verification technique combining reachability analysis and regular expression syntax was defined for EBD models with unbounded process pools, bounded resources, and auxiliary resource counters. The added syntax allowed equivalent processes to be grouped and studied as such, while the addition of auxiliary counter variables for bounded resources switched the focus from requiring a complete configuration to requiring a complete specification. As such, the procedure can determine the minimum number of pool instances required to execute all combined system states. The analyst is also given a means to study the level of concurrency among different pool process states.

Contribution 3: A Technique for Developing and Documenting Design Fragments

The method recognizes that there are several potentially correct alternatives for designing specification segments, in particular, with regard to the design of inter-entity communications. The goal is to present a technique for developing and documenting reusable *programming plans* for the communication requirements of reactive systems. This is achieved using the separation of application-behaviour and communication semantics in EBD specifications, and the entity-to-actor allocation rules, to establish a framework in which to define such alternatives (i.e., the *basic model* actor design). Basic models provide placeholders which are used and manipulated by a *matching* set of independent *design fragments*.

The individuality of design fragments: (1) simplifies the proof of conformance required as they are developed, (2) makes them easier to apply during design, and (3) simplifies their validation when used, by reducing the proof requirements to a list of smaller, and generally independent, *contingencies*. That is, design frag-

ments are added to a library only after most of the conformance issues are already proven. They represent formal relations to specification segments given that the contingencies are provable.

Design fragments capture the expertise of their developer. They are broadly classified by the allocation of entities to actors, and documented in a form which is amenable to the development of a user's handbook. A small initial population is provided to illustrate the technique, and for use in examples of the design method.

Contribution 4: A Method for Designing for Strong Traceability

Here, the goal is to define a method for designing reactive system software with strong traceability and conformance to the specification. This method represents a compromise for formal, automatic transformation processes, and the more typical ad hoc approaches. The method defined in this thesis includes: (1) the allocation of entities in the specification to actors of the design, and (2) the selection of design fragments to cover the communication requirements defined by the specification. A design is constructed in a piece-meal fashion based on specification segments. The selection process is flexible; i.e., it is not constrained to follow any particular ordering of design decisions, yet the associated documentation provides a consistent, and well-structured presentation of the design. The selection process is also creative. The critical, domain related, design decisions are made during the design, without sacrificing strong traceability.

Although the design process assumes that the specification is correct, the design method encourages consideration of the system and design properties. Consequently the method prompts subsequent specification refinement in cases where a suitable fragment can not be found or defined to fit the intended behaviour. Therefore, there is motivation for necessary changes, and help for arguing that these changes are valid and appropriate.

Contribution 5: A Validation Process

Strong traceability is an important objective of this thesis. It is defined by large-sense traceability which illustrates completeness of the design, and small-sense traceability where each design decision is validated for its correct application. Validating

completeness requires consideration of all of the requirements specified by the system EBD model. It becomes a mechanical (potentially automatic) problem given the independence of design fragments, and the structured documentation process. Validating correctness is simplified because: (1) there is no need to address the application-behaviour represented explicitly by ACMs, and (2) much of the conformance proof for communications behaviour is already done in the definition of design fragments. All that is left are the smaller, design fragment contingencies.

Algorithms are defined for proving specification and design contingencies. They indicate that the contingencies can be addressed in any order; and, in general, at any time in the design or validation process. Contingencies can be demonstrated formally, or informally, but in either case support strong traceability through their definition, and documentation. The thesis provides recommendations on the use of formal methods, and reasoning for the contingencies which are common to the defined design fragments.

Contribution 6: A Documentation Process

Since strong traceability is a major goal, documentation of the design process is a consequence; i.e., illustrating completeness and correctness requires documentation. This thesis presents a tabular documentation technique which supports strong traceability by recording: (1) allocation decisions, (2) design fragment decisions, and (3) validation.

7.2 Evaluation

The elements of the design process are illustrated by: (1) several small partial examples (in the case of EBDs), (2) other small, but complete systems specifications (e.g., the Readers/Writers problem, the Producer/Consumer problem, and the 2 of 3 problem), and (3) a larger real-world problem where a single subsystem is defined. (In this example the subsystem behaviour is relatively uncomplicated, but it illustrates the notions of decomposition of large system designs, and using EBDs to define individual behaviour as well as interacting behaviours.) The technique is shown to be effective in these examples which cover the classes of low-level, and rel-

atively complex behaviours (e.g., Readers/Writers), and high-level (possibly more typical) subsystem behaviours. More specific methods of evaluation are listed below.

EBD Specifications — The introduction of design fragments based on EBD specification segments, and the design examples for the systems mentioned above, show that the EBD model is applicable within a design process for strong traceability of behavioural requirements. A limited version of EBD modelling (i.e., without EI directives), has been implemented as a component of the TimeBench designer's workbench which illustrates its adaptability to software tools. EBDs are flexible and useful in their own right. This argument is based on the similarities between EBDs and several accepted methods for specifying reactive systems in terms of their structure and components (however, in many ways EBDs are more expressively powerful).

Verification Procedure — This method was evaluated by example, and also by arguing the similarities and differences between this and the classical approach. The results indicate that the distinct system behaviour is preserved, that the procedure is guaranteed to terminate, and that the resultant system description (tree) is no larger than in the non-general case. Given a complete specification, the procedure can correctly detect deadlock, and indicate, via state descriptions, starvation and critical races, as well as illustrating the levels of concurrency among process pool instances. It was also shown that the procedure can determine the minimum number of pool instances required to execute all combined system states in terms of the auxiliary counter.

Technique for Developing Design Fragments — The development of design fragments is evaluated for the specific basic model defined in this thesis. Examples illustrate the extensibility of the design fragment library, and the potential for presenting design fragments in a handbook form. Example designs illustrate the ease of using design fragments which *modularize* the design of communications in reactive systems.

The imposition of a basic model as a starting concept for design fragments, and

consequently the design process, is considered to be reasonable because it is expected that a single queuing discipline will be used for the bulk of a specification effort. As such, a relatively explicit match can be found for an appropriate basic model and design fragment set. Special cases in the specification will require extra thought, but the design method still presents a structured, stepwise, engineering approach to design. It is also expected that there will be a limited set of useful basic model/design fragment pairs.

Design Method for Strong Traceability — The design method is evaluated for the specific basic model defined in this thesis. Examples which prove that the resulting design is correct, confirm that the approach is valid. The appearances of the designs, are, in the opinion of the author, well-structured and reasonably efficient. The process is shown to encourage specification refinement, and extending the design fragment library with verified, formally defined, design alternatives. Examples also illustrate the potential for reuse of: (1) design fragments, and (2) specifications (i.e., to provide alternative solutions for a given problem, to extend or modify specifications, or reuse at the level of collapsed behaviours).

Validation Process — The thesis argues that the algorithm for proving completeness covers all requirements of the specification (although some checks are redundant). It is argued that this process is completely mechanical, and thus a candidate for automation, when design fragments apply to independent, non-overlapping specification segments. In terms of correctness, the examples illustrate that certain contingencies can be reasoned according to pre-defined approaches. Validating correctness is also a candidate for tools, however, such proofs are limited by the issues faced by these tools (e.g., state explosion).

Documentation — Tables are used in this thesis for several reasons as justified by Parnas et al. in [113, 88, 29, 87, 60]:¹ (1) organization, (2) self-consistency, and hence completeness, (3) versatility, (4) correctness, and (5) modifiability. The

¹Parnas et al. use tables to represent relations (analogously to truth tables for deterministic functions). In this work, the relation is conformance (i.e., one-to-one), and the goal for documentation is to illustrate traceability. The benefits of tables listed here are, however, still applicable.

examples show that the documentation process is clear, well-organized, and relatively concise, regardless of the ordering of design decisions. It motivates future research into using the documented arguments in testing (especially for finding contradictions to test results in the declared assumptions), and knowledge acquisition of design principles used by reactive system designers.

General Evaluation -- The evaluation of the design process can be improved in four inter-related areas: (1) practical application, (2) tool support for developing design fragments, (3) a larger library of design fragments, and (4) tool support for the design method; in particular, assistance in establishing and completing the tables of a design.

Tool support for developing design fragments would be used to establish the lemmas, and consequently, the contingencies of a design alternative. Optimally, the tool would allow for the definition of a specification segment, and the entry of a MachineChart design alternative, from which the contingencies are defined automatically as well as the MCL (MachineChart language) which uses and manipulates the basic model design. This would provide unquestionable assistance in the definition and formalism of design fragments; and hence, the development of larger sets of design fragments, and in practical experience where design fragments might be defined during the course of a design. Although the details of design fragments are fundamentally constrained by the concept of a basic model, their MachineChart components represent the expertise of their developer, and can be used independently of this design method. They represent programming plans in MachineChart format even if they are not formally related to EBD specification segments.

Tool support for designing within the proposed framework, would make it practical for further empirical study through continued practice using: (1) classical academic case studies which would provide a comparison to other accepted methods, and (2) real-world problem definitions which would present the limits of the method and possibly areas for improvement. For example, studies may suggest that extensions for timing are necessary for the EBD specification model, and at the same time, manageable within the defined design process. These studies might also illustrate any deficiencies in the design method, possibly a more suitable basic model

and design fragment set, and issues overlooked in this thesis. Further study could also present new typical contingencies, and methods for reasoning their properties in relation to typical applications.

The concept of designing to conform, with strong traceability to a specification, yet making critical decisions during the design process, has potentially powerful instructional benefits. The concepts behind the design method are strong and intuitive even without the formalisms and the documentation. The design is focussed on maintaining conformance between the specification and design; and as such, emphasizes the need to maintain up-to-date specifications. The approach is piece-meal and structured. By requiring that each design decision be independent of the others, the user is made to consider the dependency relationships of design alternatives.

Furthermore, EBD modelling is a flexible approach to specifying reactive system behaviour; and as such, is useful in its own right, although it may be limited by its inability to specify timing constraints. This evaluation leads to the following list of future research areas.

7.3 Future Research

Empirical Studies — As suggested above, the evaluation of the process is limited by the lack of its use, and further studies are required as suggested above. These studies could: (1) recommend additions or deletions to the EBD modelling technique such as the addition of timing constraints, (2) introduce new, and possibly more appropriate basic models, (3) introduce new, typically used design fragments, (4) introduce new, typical contingencies, (5) define tools which help the validation process, and (6) illustrate deficiencies which have been overlooked by this thesis.

Knowledge Acquisition — As a result of applying the process to further case studies, it is possible that the documentation of the design decisions, and the arguments associated with contingencies, will provide interesting insight into designers of concurrent system. The documented proofs might be helpful in the study of concurrent programming practices and programming plans. As such, there is potential for automated assistance, and possibly automated transformations, based on a given

priority listing of properties such as performance or readability. Others have tried to attain this knowledge with little success [18].

Testing and Debugging - It would be interesting to consider the contingencies and their proofs in a testing and debugging scheme. Contingencies appear to represent testable properties of the design (assuming that the proofs are not performed using formal methods). If a contradiction is found during testing, then the informal proof of this contingency marks a sensible place to start the debugging process. It would also be interesting to examine the relations between contingencies, and their combined outcome when one or more has been implemented erroneously.

Design Optimizations — Studies, not presented in this thesis, showed that the approach is not always able to reproduce proven (e.g., text book) designs. This is known to be the case for semaphore solutions, and other algorithmic solutions (e.g., Dekker's or Lamport's bakery algorithm), to slight variations of the Readers/Writers problem [36, 27, 9]. The problem arises from the fact that these solutions are not based on an entity model: i.e., a system where entities interact in people-like roles with mutually exclusive boundaries on their individual, internal behaviours. These solutions were developed with the implementation domain in mind, and are therefore highly optimal within a supportive environment, but their behaviours are often quite strange in view of an entity model. For example, in the referenced examples, one reader will wait in a queue with the writers, and the rest will wait together in a separate queue. These inconsistencies among like entities are not common in entity-based specifications, however the solutions may be *better* in some circumstances. Studies are required to see if the final designs, or even the design fragments can be systematically matched to these accepted, optimal designs.

Maintenance and Modification — The documentation process should be studied for its ability to maintain old, and even rejected design decisions. Then studies should be carried out to see if this information is useful for re-designs, modifications, and testing.

Tool Support -- This design method is systematic and potentially adaptable to tool support. The tool(s) could be used to: (1) enter EBD specifications,² (2) allocate EBDs to actors,³ (3) browse a design fragment library, (4) automate the documentation process for Actor and Partition Tables, and present templates for Build Tables, (5) automate the binding of segments of the application specification to segments of a design fragment, (6) automate the completeness proof, (7) automate the proof requirements of contingencies, and (8) assist in the development and documentation of design fragments.

Formal methods applied to the design of reactive systems are often considered to be impractical because the specifications are too complex, and the results are not implementable in the given target domain. "Practical" methods are ad hoc, they require intensive testing, and provide little confidence that there is conformance between the specification and the design. This thesis has developed an approach which is practical, and yet systematic. It includes a model for software specification, a design method, and a validation process all of which focus on the goal of achieving strong traceability of behavioural requirements.

²A limited version of this has been implemented as a part of the TimeBench workbench.

³This activity is also already supported by the extensions to TimeBench.

References

- [1] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, MASS: MIT Press, 1985.
- [2] G.R. Andrews et.al., "An Overview of the SR Language and Implementation," *ACM Transactions on Programming Languages and Systems*, vol. 10, no. 1, pp. 51-86, Jan. 1988.
- [3] K.R. Apt, N. Francez, and W.P. de Roever, "A Proof System for Communicating Sequential Processes," *ACM Transactions on Programming Languages and Systems*, vol. 2, pp. 359-385, 1980.
- [4] R. Balzer, "Transformational Implementation: An Example," *IEEE Transactions on Software Engineering*, vol. 7, no. 1, pp. 3-14, Jan. 1981.
- [5] R. Balzer, T. Cheatham, and C. Green, "Software Technology in the 1990's: Using a New Paradigm," *IEEE Computer*, vol. 16, no. 11, pp. 39-45, Nov. 1983.
- [6] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1257-1267, November 1985.
- [7] D. Barstow, "An Experiment in Knowledge-Based Automatic Programming," In *Readings in Artificial Intelligence and Software Engineering*, pp. 133-156, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [8] D. Barstow, "Artificial Intelligence and Software Engineering," In *Ninth International Conference on Software Engineering*, pp. 200-211, March 1987.

- [9] M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, New York, NY:Prentice Hall, 1990.
- [10] E.V. Berard, *Essays on Object-Oriented Software Engineering*, Ft. Collins, Colo.: Prentice Hall, 1992. 1989 and 1990.
- [11] D. Bjorner, "On the Use of Formal Methods in Software Development," In *Ninth International Conference on Software Engineering*, pp. 17-29, Monterey, California, March 1987.
- [12] T. Bolognesi and E. Brinksma, "Introduction to the ISO Specification Language LOTOS," In *Introduction to the ISO Specification Language LOTOS*, pp. 23-73 North-Holland: Elsevier Science Publishers, 1989.
- [13] G. Booch, *Software Engineering with ADA*. Reading, MASS: Benjamin/Cummings, 1987.
- [14] G. Booch, *Object Oriented Design with Applications*. Redwood City, Ca.: Benjamin/Cummings, 1991.
- [15] A. Borgida, S. Greenspan, and J. Mylopoulos, "Knowledge Representations as the Basis for Requirements Specifications," In *Readings in Artificial Intelligence and Software Engineering*, pp. 561-570, Los Altos, Ca.: Morgan Kaufmann Publishers. Inc., 1986.
- [16] J.L. Boyd, *Entity Behaviour Descriptions - A Specification Model for Communicating Entities*, TR-SCE-93-08, Real-Time and Distributed Systems Group, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, On., 1993.
- [17] M. Browne, "Automatic Verification of Finite State Machines Using Temporal Logic," Ph.D. thesis, Pittsburgh, Feb. 1989.
- [18] D. Budgen and M. Marashi, "MDSE Advisor: Knowledge-based Techniques Applied to Software Design Assessment," *Knowledge-Based Systems*, vol. 1, no. 4, pp. 235-239, 1988.

- [19] S. Budkowski and P. Dembinski, "An Introduction to Estelle: A Specification Language for Distributed Systems," *Computer Networks and ISDN Systems*, no. 14, pp. 3-23, 1987.
- [20] R.J.A. Buhr, *System Design With Ada*. Englewood Cliffs, NJ: Prentice-Hall, 1984.
- [21] R.J.A. Buhr, "Practical Visual Techniques for Joint Refinement of Structure and Temporal Behaviour in Reactive System Design," in *Proceedings of Compuro '90*, Tel Aviv, Israel, May 1990.
- [22] R.J.A. Buhr, G.M. Karam, and R. Casselman, "Support for Specifying Temporal Behaviour in Ada Designs," in *Proceedings of 1st Int'l Symp. on Environments and Tools for Ada (SETA1)*, Redondo Beach, CA, May 1990.
- [23] R.J.A. Buhr, *Practical Visual Techniques in System Design: With Applications to Ada*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [24] R.J.A. Buhr, G.M. Karam, C.M. Woodside, R. Casselman, G. Franks, H. Scott, D. Bailey, *TimeBench: a CAD Tool for Real-Time System Design*. Real-Time and Distributed Systems Group, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, On. Sept. 1990.
- [25] R.J.A. Buhr, R.S. Casselman, "Architectures with Pictures," In *OOPSLA '92 Proceedings*, pp. 466-483, Vancouver, B.C., Oct. 1992.
- [26] R.J.A. Buhr, R.S. Cassleman, *Case Studies in Threads and Contracts*, TR-SCE-92-06, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, On. Canada, April 1992.
- [27] D. Bustard, J. Elder, J. Welsh, *Concurrent Program Structures*, New York, NY: Prentice Hall, 1988.
- [28] ACM, *Communications of the ACM; Special Issue: Hypertext*, vol. 31, no. 7, July 1988.

- [29] G. Cantin, "The Use of Function-Tables to Specify Complex Algorithms," CRL Report No. 236, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, On, Sept. 1991.
- [30] A. Che, *User's Guide for Some Special Tools of TimeBench*, Real-Time and Distributed Systems Group, Carleton University, Ottawa, On., Aug. 1992.
- [31] T. Cheatham, G. Holloway, and J. Townley, "Program Refinement by Transformation," In *Fifth International Conference on Software Engineering*, pp. 430-437, San Diego, California, March 1981.
- [32] T. Cheatham, "Reusability Through Program Transformations," *IEEE Transactions on Software Engineering*, vol. 10, no. 5, pp. 589-594, Sept. 1984.
- [33] P.P. Chen, "The Entity-Relationship Model - Toward a Unifying View of Data," *ACM Trans. on Data Base Systems*, vol. 1, no. 1, pp. 9-36, March 1976.
- [34] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244-263, April 1986.
- [35] P. Coad and E. Yourdon, *Object-Oriented Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [36] P.J. Courtois, F. Heymans, and D.L. Parnas, "Concurrent Control with Readers and Writers," *Communications of the ACM*, vol. 14, no. 10, pp. 667-668, Oct. 1971.
- [37] J. Darlington, "An Experimental Program Transformation and Synthesis System," In *Readings in Artificial Intelligence and Software Engineering*, pp. 99-121, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [38] E. Dijkstra, *Invariance and Non-determinacy*, pp. 157-165. Prentice Hall International Series in Computer Science, Englewood Cliffs, NJ: Prentice Hall, 1985.

- [39] T.C. Dobbing, *State Machines for System Design*, Masters Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, On., 1990.
- [40] L. Doldi and P. Gauthier, "VEDA 2: Power to the Protocol Designers," In *Proceedings of the IFIP TC6 5th International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols*, Perros-Guirec, France, Oct. 1992.
- [41] S. Eswara, T. Berriman, P. VAnHoutte, B. Sarikaya, "Towards Execution of TTCN Test Cases," In *Proceedings: Tenth International IFIP WG6.1 Symposium on Protocol Specification, Testing, and Verification*, Ottawa, On. June 1990.
- [42] S.R. Faulk and D.L. Parnas, "On Synchronization in Hard-Real-Time Systems," *Communications of the ACM*, Vol. 31, No. 3, pp. 274-287, March 1988.
- [43] M. Feather, "A System for Assisting Program Transformation," *ACM Trans. on Prog. Languages and Systems*, vol. 4, no. 1, pp. 1-20, Jan. 1982.
- [44] M. Feather, "Constructing Specifications by Combining Parallel Elaborations," *IEEE Transactions on Software Engineering*, vol. 15, no. 2, pp. 198-208, Feb. 1989.
- [45] S. Fickas, "Automating the Transformational Development of Software," *IEEE Transactions on Software Engineering*, vol. 11, no. 11, pp. 1268-1277, Nov. 1985.
- [46] V. Garg, "Analysis of Distributed Systems with Many Identical Processes," In *IEEE Eighth International Conference on Distributed Computing Systems*, (San Jose, Ca.), pp. 358-365, June 1988.
- [47] D. Garlen and N. Delisle, "Formal Specification as Reusable Frameworks," In *VDM '90: VDM and Z - Formal Methods in Software Development, Third International Symposium of VDM Europe*, Lecture Notes in Computer Science #428, Berlin:Springer-Verlag, pp. 150-163, 1990.

- [48] N.H. Gehani, "Message Passing in Concurrent C: Synchronous versus Asynchronous," *Software — Practice and Experience*, vol. 29, no. 6, pp. 571–592, June 1990.
- [49] C. Gilbert, L. Williams, *The ABC's of 1-2-3*, Release 2.3, Second Edition, Sybex, San Francisco, Ca., 1991.
- [50] S. Graf, J.-L. Richier, C. Rodríguez, and J. Voiron, "What are the Limits of Model Checking for the Verification of Real Life Protocols," *Automatic Verification for Finite State Systems*, Lecture Notes in Computer Science #407, Berlin: Springer-Verlag, pp. 275–285, 1989.
- [51] C. Green, D. Luckham, R. Balzer, T. Cheatham, and C. Rich, "Report on a Knowledge-Based Software Assistant," In *Readings in Artificial Intelligence and Software Engineering*, pp. 377–428, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [52] B.T. Hailpern, "Verifying Concurrent Processes Using Temporal Logic," *Lecture Notes in Computer Science #129*, New York, NY: Springer-Verlag, 1982.
- [53] M. Harandi and F.H. Young, "Template Based Specification and Design," In *Third International Workshop on Software Specification and Design*, pp. 94–97 London (UK), Aug. 1985
- [54] D. Harel and A. Pnueli, "On the Development of Reactive Systems," In *Logics and Models of Concurrent Systems*, pp. 477–498, New York: Springer Verlag, 1985.
- [55] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.
- [56] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman, "On the Formal Semantics of Statecharts," In *Proc. 2nd IEEE Symposium on Logic in Computer Science*, pp. 54–64, Ithaca, NY, 1987.

- [57] D. Harel, "On Visual Formalisms," *Communications of the ACM*, vol. 31, no. 5, pp. 514-531, May 1988.
- [58] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, "Statemate: A Working Environment for the Development of Complex, Reactive Systems," in *Proceedings of the 10th International Conference on Software Engineering*, Singapore, April 1988.
- [59] D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*. New York, NY: Dorset House, 1988.
- [60] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Transactions on Software Engineering*, vol. 6, no. 1, pp. 2-13, Jan. 1980.
- [61] C. Hoare, *Communicating Sequential Processes*. Englewood Cliffs, NJ: Prentice-Hall, 1985.
- [62] M. Jaffe, N. Levenson, M. Heimdahl, and B. Melhart, "Software Requirements Analysis for Real-time Process-control Systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 241-258, March, 1991.
- [63] F. Jahanian and A.K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 12, no. 9, pp. 890-904, Sept. 1986.
- [64] F. Jahanian and A.K. Mok, "A Graph-Theoretic Approach for Timing Analysis in Real-Time Logic," In *Real-Time Systems Symposium*, New Orleans, Louisiana, pp. 98-108, Dec. 1986.
- [65] F. Jahanian and D.A. Stuart, "A Method for Verifying Properties of Modechart Specifications," In *Real Time Systems Symposium*, Huntsville, Alabama, pp. 12-21, Dec. 1988.
- [66] F. Jahanian, R. Lee, and A. Mok, "Semantics of Modecharts in Real Time Logic," In *Proceedings of the Twenty-First Annual Hawaii International Conference on Systems Sciences*, pp. 479-489, Kailua-Kona, Hawaii, 1988.

- [67] C.B. Jones, *Systematic Software Development Using VDM (Second Edition)*, London: Prentice-Hall, 1989.
- [68] E. Kant, "On the Efficient Synthesis of Efficient Programs." In *Readings in Artificial Intelligence and Software Engineering*, pp. 157-183, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [69] G. Karam, C. Stanczyk, and G. Bond, "Critical Races in Ada Programs," *IEEE Transactions on Software Engineering*, vol. 15, no. 11, pp. 1471-1480, Nov. 1989.
- [70] A. Klausner and T.E. Konchan, "Rapid Prototyping and Requirements Specification Using PDS," in *ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, pp. 96-105, 1982.
- [71] Lam and Shankar, "Protocol Verification via Projections," *IEEE Transactions on Software Engineering*, vol. 10, no. 7, pp. 325-342, July 1984.
- [72] Y. Ledru, *Towards the Formal Development of Terminating Reactive Systems*, Ph.D. Thesis: Faculté Des Sciences Appliquées Unité D'Informatique, Université Catholique De Louvain, Dec. 1991.
- [73] L. Liu and R. Shyamasundar, "Static Analysis of Real-time Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 373-388, April 1990.
- [74] P. London and M. Feather, "Implementing Specification Freedoms," In *Readings in Artificial Intelligence and Software Engineering*, pp. 285-305, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [75] N.A. Lynch and M.R. Tuttle, *An Introduction to Input/Output Automata*, MIT/LCS/TM-373, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass. Nov. 1988.
- [76] M. Maekawa, A. Oldehoeft, and R. Oldehoeft, *Operating Systems - Advanced Concepts*. Menlo Park, California: Benjamin/Cummings. 1987.

- [77] Z. Manna and A. Pnueli. "Verification of Concurrent Programs: The Temporal Framework." In *The Correctness Problem in Computer Science*, pp. 215-273. New York, NY: Academic Press, 1981.
- [78] C. McNamee and R. Olsson. "Comments on: Critical Races in Ada Programs," *IEEE Transactions on Software Engineering*, vol. 16, no. 12, pp. 1440, Dec. 1990.
- [79] S. Mellor. "Panel Position - Structured Analysis and Object Oriented Analysis." In *OOPSLA ECOOP '90 Proceedings*, p. 137, Ottawa, On., Oct. 1990.
- [80] R. Milner. *A Calculus of Communication Systems*. Lecture Notes in Computer Science #92. New York, NY: Springer-Verlag, 1980.
- [81] N. Minshy and D. Rozenshtein. "Controllable Delegation: An Exercise in Law-Governed Systems." In *OOPSLA '89 Proceedings*, pp. 371-380, New Orleans, Louisiana, Oct. 1989.
- [82] N. Minsky, "The Imposition of Protocols Over Open Distributed Systems," *IEEE Transactions on Software Engineering*, vol. 17, no. 2, pp. 183-195, Feb. 1991.
- [83] J.S. Ostroff. *Temporal Logic For Real-Time Systems*. Toronto, On.: Research Studies Press Ltd., 1989.
- [84] S. Owicki and L. Lamport. "Proving Liveness Properties of Concurrent Programs." *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 455-495, July 1982.
- [85] D.L. Parnas, "On Criteria to be used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 14, no. 1, pp. 221-227, April 1972.
- [86] D.L. Parnas. "Software Aspects of Strategic Defense Systems." *Communications of the ACM*, vol. 28, no. 12, pp. 1326-1335, December 1985.

- [87] D.L. Parnas and P.C. Clements, "Rational Design Process: How and Why to Fake It," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251-257, Feb. 1986.
- [88] D.L. Parnas and J. Madley, "Functional Documentation for Computer Systems Engineering." Tech. Report 90-287, Queen's University, TRIO (Telecommunications Research Institute of Ontario), Sept. 1990.
- [89] H. Partsch and R. Steinbruggen, "Program Transformation Systems," *ACM Computing Surveys*, vol. 15, no. 3, pp. 199-236, Sept. 1983.
- [90] J. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, pp. 223-252, September 1977.
- [91] R. Pressman, *Software Engineering: A Practitioner's Approach*. Second Edition, New York: McGraw-Hill, 1987.
- [92] R.L. Probert, H. Ural, and M.W.A. Hornbeek, "An Integrated Software Environment for Developing and Validating Standardized Conformance Tests," In *Proceedings: 8th International IFIP Symposium on Protocol Specification, Testing, and Verification*, pp. 87-98, Atlantic City, NJ, June 1988.
- [93] Quintus Computer Systems, *Quintus Prolog V2.2 User's Guide*, Quintus Computer Systems, Mountain View, Ca., 1988.
- [94] B. Ramesh and V. Dhar, "Supporting Systems Development by Capturing Deliberations During Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 18, no. 6, pp. 498-510, June 1992.
- [95] H. Reubenstein and R. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 226-240, March, 1991.
- [96] C. Rich and R. Waters, ed., *Artificial Intelligence and Software Engineering*. Los Altos, CA: Morgan Kaufmann Inc., 1986.

- [97] C. Rich and R. Waters, "The Programmer's Apprentice: A Research Overview," *IEEE Computer*, vol. 21, no. 11, pp. 10-25, November 1988.
- [98] R. Ross, *Entity Modelling: Techniques and Application*. Boston, MA: Database Research Group, 1987.
- [99] M. Rossion and E. Gold, "Problem-Solution Mapping in Object-Oriented Design," In *OOPSLA '89 Conference Proceedings*, pp. 7-10, New Orleans, Louisiana:, Oct. 1989.
- [100] C. Roth, *Fundamentals of Logic Design*. New York, NY: West, 1979.
- [101] D. Rozenshtein and N.H. Minsky, "A Law-Governed Object-Oriented System," *Journal of Object-Oriented Programming*, vol. 1, no. 6, pp. 15-29, Mar/Apr 1989.
- [102] R. Saracco and P. Tilanus, "CCITT SDL: Overview of the Language and its Applications," *Computer Networks and ISDN Systems*, vol. 13, pp. 65-74, 1987.
- [103] H.J. Scott and G.M. Karam, "Meta-Code Generation in a Design Environment," SCE-91-25, Dept. of Systems and Computer Eng., Carleton University, Ottawa, Ontario, June 1991.
- [104] K. Shafer and M. Ahuja, "Modeling Asynchronous Distributed Communication," OSU-CISRC-7/91-TR19, Department of Computer and Information Science, Ohio State University, Columbus, Ohio, July 1991.
- [105] S. Shlaer and S. Mellor, *Object-Oriented System Analysis: Modelling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, 1988.
- [106] S. Shlaer and S. Mellor, "An Object-Oriented Approach to Domain Analysis," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 5, pp. 66-77, July 1989.
- [107] D. Smith, G. Kotik, and S. Westfold, "Research on Knowledge-based Software Environments at Kestrel Institute," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 11, pp. 1278-1295, November 1985.

- [108] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," In *Readings in Artificial Intelligence and Software Engineering*, pp. 507-521, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.
- [109] J. Stankovic, "A Serious Problem for Next-generation Systems," *IEEE Computer*, vol. 21, no. 10, pp. 10-19, Oct. 1988.
- [110] M. Stark and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," In *Ada-Europe International Conference*, pp. 213-222, Stockholm, Sweden, 1987.
- [111] C.A. Sunshine, D.H. Thompson, R.W. Erikson, S.L. Gerhart, D. Schwabe, "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models," *IEEE Transactions on Software Engineering*, vol. 8, no. 5, pp. 460-489, Sept. 1982.
- [112] A. Sutcliffe, *Jackson System Development*. New York: Prentice-Hall, 1988.
- [113] A.J. van Schouwen, "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems," CRL Report No. 242, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, On, Feb. 1992.
- [114] J. van den Bos, C. Laffra, "PROCOL: A Parallel Object Language with Protocols," In *OOPSLA '89 Proceedings*, pp. 95-102, Oct. 1989.
- [115] M. Vigder, *Applying Formal Techniques to the Design of Concurrent Systems*, Ph.D. Thesis, Dept. of Systems and Computer Engineering, Carleton University, Ottawa, On., 1992.
- [116] S.T. Vuong and D.D. Cowan, "VALIRA - A Tool for Protocol Validation via Reachability Analysis," In *Proceedings: 7th International IFIP Symposium on Protocol Specification, Testing, and Verification*, pp. 35-41, Zurich, May 1987.
- [117] R. Waters, "The Programmer's Apprentice: A Session with KBEmacs," In *Readings in Artificial Intelligence and Software Engineering*, pp. 351-375, Los Altos, Ca.: Morgan Kaufmann Publishers, Inc., 1986.

- [118] C. West, "An Automated Technique of Communications Protocol Validation," *IEEE Transactions on Communications*, vol. COM-26, no. 8, pp. 1271-1275, August 1978.
- [119] C.H. West, "Protocol Validation - Principles and Applications," In *Tutorial Notes: Tenth International IFIP WG6.1 Symposium on Protocol Specification, Testing, and Verification*, Ottawa, On. June 1990.
- [120] P. Zafiropulo, "Protocol Validation by Duologue-Matrix Analysis," *IEEE Transactions on Communications*, Vol. 26, No. 8, pp. 1187-1275, August 1978.
- [121] P. Zafiropulo, C. West, H. Rudin, D. Cowan, and D. Brand, "Towards Analyzing and Synthesizing Protocols," *IEEE Transactions on Communications*, vol. 28, no. 4, pp. 651-660, April 1980.
- [122] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Transactions on Software Engineering*, Vol. 8, No. 3, pp. 250-269, Sept. 1982.
- [123] P. Zave, "The Operational Versus the Conventional Approach to Software Development," *Communications of the ACM*, vol. 27, no. 2, pp. 104-118, February 1984.
- [124] H. Zimmermann, "OSI Reference Model — The ISO Model of Architecture for Open Systems Interconnection," *IEEE Transactions on Communications*, Vol. Com-28, No. 4, p. 425-432, April 1980.

Appendix A

EI Details

STATE NO.	STATE NAME	MEANING
1	<i>init</i>	This is the initial EI state. It indicates that all buffers, are empty and that there are either: (1) no outstanding meta-event interactions between the EI and EBD, or (2) an <i>in progress</i> list of event sends where one is processed on each entry into the state, and the <i>Complete</i> event is returned to the EBD only after every event in the list has been sent successfully. Initially the <i>end_in_progress</i> boolean is false.
2	<i>wait event</i>	In this state the EI and the EBD are interacting in order to receive a system event. The EI knows that none of the buffered events match any of the events in the satisfiable event set. The EI will timeout, if the time, defined by timeout parameter of the <i>RequestNext(SET.SET_DIRECTIVES)</i> message, has elapsed since this state was entered. This is the only state where broadcast events are not immediately discarded.
3	<i>wait match</i>	In this state, the EI buffers are empty, and the EI is waiting to be told when a synchronous event (sent by its EBD) has been received by the appropriate destination EBD.
4	<i>wait request</i>	In this state the EI buffers are building, but there are no outstanding requests from the EBD to receive an incoming system event. There may be an <i>in progress</i> list of event sends where one is processed on each entry into the state, and the <i>Complete</i> event is returned to the EBD only after every event in the list has been sent successfully.
5	<i>wait match or request</i>	In this state the EI buffers are also non-empty, but the EI is waiting for synchronization on a synchronous system event send.

Table A.1: EI state descriptions

Label	Transition/Description
ClearRequest	$EBD + RequestNext(SET, SET_DIRECTIVES) //$
SendNull	$EBD + SendEvent(Es, Es_DIRECTIVE)[empty_list(Es)]$ $// EBD - Complete$
SendRequest	$EBD + SendEvent(Es, Es_DIRECTIVE)$ $[\neg empty_list(Es)] / send_in_progress := true; \text{ define function } SendSteps(i) \text{ using } Es \text{ and } Es_DIRECTIVE: step := 1 /$
BrInProgress	$[send_in_progress = true \cap SendSteps(step) \text{ processes a broadcast event}] / SendSteps(step) \text{ which assigns values to } SID, \text{ and } SP; \text{ increment } step /$ $(\forall rx(EI(rx) - InteractOn(br(msg), (SID, SP))))$
AsyncInProgress	$[send_in_progress = true \cap SendSteps(step) \text{ processes an asynchronous event}] / SendSteps(step) \text{ which assigns values to } SID, SP, \text{ and } RID; \text{ increment } step /$ $RID - InteractOn(msg, (SID, SP))$
SyncInProgress	$[send_in_progress = true \cap SendSteps(step) \text{ processes a synchronous event}] / SendSteps(step) \text{ which assigns values to } SID, SP, \text{ and } RID; \text{ increment } step /$ $RID - InteractOn(msg, (SID, SP))$
SendComplete	$[send_in_progress = true \cap SendSteps(step) \text{ indicates that all events have been sent}] /$ $send_in_progress := false / EBD - Complete$
Br	$\neg EI + InteractOn(br(m), ATT) //$
Buffer	$\neg EI + InteractOn(sync(m) async(m), ATT) / buffer /$

Table A.2: Legend for EI EBD (table 1 of 2)

Label	Transition/Description
Timeout	$\{timeout(t)\}[emptyq]//$ $EBD - DeliverEBD(timeout)$
TimeoutNotEmptyQ	$\{timeout(t)\}[\neg emptyq]//$ $EBD - DeliverEBD(timeout)$
NoMatch	$\neg EI + InteractOn(E, ATT)[\neg match(E)]/buffer/$
MatchEmptyQ	$\neg EI + InteractOn(E, (SID, SP))[match(E), emptyq]/$ $ID = identifier\ of\ this\ EI / (EBD - DeliverEBD(E),$ $(sync(E) \rightarrow$ $SID - InteractOn(match, (ID, medium))))$
MatchNotEmptyQ	$\neg EI + InteractOn(E, (SID, SP))[match(E), \neg emptyq]/$ $ID = identifier\ of\ this\ EI / (EBD - DeliverEBD(E),$ $(sync(E) \rightarrow$ $SID - InteractOn(match, (ID, medium))))$
MatchReturn	$RID + InteractOn(match, ATT)//$
RequestNoMatch	$EBD + RequestNext(SET, SET_DIRECTIVES)$ $[\neg check_all_for_match(E)]//$
RequestMatchEmptyQ	$EBD + RequestNext(SET, SET_DIRECTIVES)$ $[check_all_for_match(E), emptyq]/$ $dequeue(event(E, SID, SP, T))$ $ID = identifier\ of\ this\ EI / (EBD - DeliverEBD(E),$ $(sync(E) \rightarrow$ $SID - InteractOn(match, (ID, medium))))$
RequestMatchNotEmptyQ	$EBD + RequestNext(SET, SET_DIRECTIVES)$ $[check_all_for_match(E), \neg emptyq]/$ $dequeue(event(E, SID, SP, T))$ $ID = identifier\ of\ this\ EI / (EBD - DeliverEBD(E),$ $(sync(E) \rightarrow$ $SID - InteractOn(match, (ID, medium))))$

Table A.3: Legend for EI EBD (table 2 of 2)

empty_list(arg):
The function is true if <i>arg</i> is an empty list.
define function SendSteps(i):
The SendEvent(<i>Es</i> , <i>Es.DIRECTIVE</i>) message is interpreted to define a function <i>SendSteps(i)</i> which defines the requirements of a single send event for each value of <i>i</i> , and indicates the completion of <i>SendEvent(Es,Es.DIRECTIVE)</i> as its last step. The index variable <i>i</i> is initialized to point to the first event send required by the <i>SendEvent(Es,Es.DIRECTIVE)</i> request. Each <i>se</i> :< <i>E.directives</i> > of <i>Es</i> is used to establish the processing required to define: (1) <i>E</i> where <i>E</i> is one of: <i>async(m)</i> , <i>sync(m)</i> , or <i>br(m)</i> , (2) <i>SID</i> which is the identifier of this EI, (3) <i>SP</i> which is the priority assigned to the individual event send in <i>directives</i> or the default priority= <i>medium</i> , (4) any assignment statements for EI variables defined in a <i>set(EI_var.RID)</i> directive, and (5) the receiver identifier, <i>RID</i> which must be selected from a group of qualifying receivers if the receiver entity is unspecified in <i>Es</i> . This processing then defines one entry in the <i>SendSteps(i)</i> algorithm.
<i>Es.DIRECTIVE</i> (which can be an <i>order(func)</i> directive, and possibly the priorities assigned to each send event in <i>se</i>), are used to order the event sends, and thus defines the <i>ith</i> entry in <i>SendSteps(i)</i> . If <i>Es.DIRECTIVE</i> is undefined, then the order in which the events are listed in <i>Es</i> is used to define the order of the <i>SendSteps(i)</i> algorithm.
SendSteps(step) processes a xxx event:
The condition evaluates what type of event is to be sent next according to the previously established <i>SendSteps(i)</i> algorithm; i.e., asynchronous, synchronous, or transient broadcast.
SendSteps(step):
The processing defined by the current <i>step</i> of the <i>SendSteps(i)</i> algorithm is performed. The parameters of the <i>RID-InteractOn(E.(SID,SP))</i> event are defined.
SendSteps(step) indicates that all events have been sent:
The algorithm defined by <i>SendSteps(i)</i> has been carried out completely.

Table A.4: Operations and Condition Functions of the EI EBD (table 1 of 2)

buffer:
Every incoming asynchronous and synchronous event (i.e., <i>InteractOn</i> (<i>E</i> ,(<i>SID</i> , <i>SP</i>)) where <i>E</i> is either <i>async</i> (<i>m</i>) or (<i>sync</i> (<i>m</i>)) is buffered along with its arrival time (<i>T</i>), as <i>event</i> (<i>E</i> , <i>SID</i> , <i>SP</i> , <i>T</i>). A separate buffer is used for each unique event defined by the EBD.
emptyq:
True if all buffers of the EI are empty.
increment arg:
Increase the value of <i>arg</i> by one.
match(E):
Checks to see if the incoming event, <i>E</i> of <i>InteractOn</i> (<i>E</i> , <i>ATT</i>) matches one of the satisfiable events currently outstanding according to the latest Request-Next(SET.SET_DIRECTIVES) message from the EBD.
check_all_for_match(E):
True if a match can be found among the currently buffered events for one of the elements of the satisfiable event set sent in the last Request-Next(SET.SET_DIRECTIVES) message. <i>E</i> is the event which matches the request. This function begins the search by ordering the unique events according to the <i>order(func)</i> directive defined in the <i>directives</i> of each <i>re</i> :< <i>E</i> , <i>directives</i> >; or, if no order is specified, by the default FIFO ordering. The unique events which are present and satisfy the directives (i.e. according to the <i>when()</i> directives) are noted for the second search defined below. In the second search, the <i>order(func)</i> directive of SET_DIRECTIVES, and possibly the priorities assigned to each unique event (for <i>order(priority(RP))</i>), or the default FAIR ordering, is used to choose from the previously ordered unique events. If an event is selected, where exactly one event is selected if one or more satisfies the request, then any assignments defined by <i>set(EI_var,SID)</i> directives are carried out, and <i>reset</i> directives are used to clear out their corresponding buffers. If no match can be found, then the function returns false, with <i>E</i> undefined.

Table A.5: Operations and Condition Functions of the EI EBD (table 2 of 2)

Appendix B

Readers/Writers Generalized System States

0

$s : [1] : []$
 $w : [1^*] : [new^*]$
 $r : [1^*] : [new^*]$
counter range : [0, 0]
Children : [1, 2] *Parents* : []

2

$s : [1] : [-r + rr^+]$
 $w : [1^*] : [new^*]$
 $r : [1^*, 2^+] : [new^*]$
counter range : [0, 0]
Children : [5, 4, 2] *Parents* : [0, 2, 14]

4

$s : [1] : [-r + rr^*, -w + wr^*]$
 $w : [1^*, 2^*] : [new^*]$
 $r : [1^*, 2^*] : [new^*]$
counter range : [0, 0]
Children : [8, 7, 4] *Parents* : [1, 2, 4, 15, 17]

1

$s : [1] : [-w + wr^+]$
 $w : [1^*, 2^+] : [new^*]$
 $r : [1^*] : [new^*]$
counter range : [0, 0]
Children : [3, 1, 4] *Parents* : [0, 1, 10]

3

$s : [4] : [-w + wr^*]$
 $w : [1^*, 2^+] : [new^*, s + gw]$
 $r : [1^*] : [new^*]$
counter range : [0, 0]
Children : [3, 6, 7] *Parents* : [1, 3]

5

$s : [2] : [-r + rr^*]$
 $w : [1^*] : [new^*]$
 $r : [1^*, 2^+] : [new^*, s + gr^+]$
counter range : [1, 1^+]
Children : [5, 8, 9] *Parents* : [2, 5]

6

$s : [4] : [_w + wr^*]$
 $w : [1^*, 2^*, 3] : [new^*, spon_1]$
 $r : [1^*] : [new^*]$
 $counter_range : [0, 0]$
 $Children : [6, 10, 11] Parents : [3, 6]$

8

$s : [2] : [_r + rr^*, _w + ur^*]$
 $u : [1^*, 2^*] : [new^*]$
 $r : [1^*, 2^*] : [new^*, s + gr^+]$
 $counter_range : [1, 1^+]$
 $Children : [8, 12, 13] Parents : [4, 5, 8]$

10

$s : [4] : [_w + ur^*, _w + wf]$
 $w : [1^*, 2^*] : [new^*]$
 $r : [1^*] : [new^*]$
 $counter_range : [0, 0]$
 $Children : [1, 10, 15] Parents : [6, 10]$

12

$s : [3] : [_r + rr^*, _w + ur^*]$
 $w : [1^*, 2^*] : [new^*]$
 $r : [1^*, 2^*] : [new^*, s + gr^+]$
 $counter_range : [1, 1^+]$
 $Children : [12, 16] Parents : [8, 12]$

14

$s : [2] : [_r + rr^*, _r + rf^*]$
 $w : [1^*] : [new^*]$
 $r : [1^*, 2^*, 3^*] : [new^*, s + gr^*, spon_1^*]$
 $counter_range : [1, 1^+]$
 $Children : [14, 2, 17] Parents : [9, 14]$

7

$s : [4] : [_r + rr^*, _w + wr^*]$
 $w : [1^*, 2^*] : [new^*, s + gw]$
 $r : [1^*, 2^*] : [new^*]$
 $counter_range : [0, 0]$
 $Children : [7, 11] Parents : [3, 4, 7, 18]$

9

$s : [2] : [_r + rr^*]$
 $w : [1^*] : [new^*]$
 $r : [1^*, 2^*, 3^+] : [new^*, s + gr^*, spon_1^+]$
 $counter_range : [1, 1^+]$
 $Children : [9, 13, 14] Parents : [5, 9, 14]$

11

$s : [4] : [_r + rr^*, _w + wr^*]$
 $w : [1^*, 2^*, 3] : [new^*, spon_1]$
 $r : [1^*, 2^*] : [new^*]$
 $counter_range : [0, 0]$
 $Children : [11, 15] Parents : [6, 7, 11]$

13

$s : [2] : [_r + rr^*, _w + ur^*]$
 $w : [1^*, 2^*] : [new^*]$
 $r : [1^*, 2^*, 3^+] : [new^*, s + gr^*, spon_1^+]$
 $counter_range : [1, 1^+]$
 $Children : [13, 16, 17] Parents : [8, 9, 13]$

15

$s : [4] : [_r + rr^*, _w + wr^*, _w + wf]$
 $w : [1^*, 2^*] : [new^*]$
 $r : [1^*, 2^*] : [new^*]$
 $counter_range : [0, 0]$
 $Children : [4, 15] Parents : [10, 11, 15]$

16

 $s : [3] : [-r + rr^*, -w + wr^*]$ $w : [1^*, 2^*] : [new^*]$ $r : [1^*, 2^*, 3^+] : [new^*, s + gr^*, spon_1^+]$ *counter range* : $[1, 1^+]$ *Children* : $[16, 18]$ *Parents* : $[12, 13, 16]$

18

 $s : [3] : [-r + rr^*, -r + rf^*, -w + wr^*]$ $w : [1^*, 2^*] : [new^*]$ $r : [1^*, 2^*, 3^+] : [new^*, s + gr^*, spon_1^*]$ *counter range* : $[1, 1^+]$ *Children* : $[7, 18]$ *Parents* : $[16, 17, 18]$

17

 $s : [2] : [-r + rr^*, -r + rf^*, -w + wr^*]$ $w : [1^*, 2^*] : [new^*]$ $r : [1^*, 2^*, 3^+] : [new^*, s + gr^*, spon_1^*]$ *counter range* : $[1, 1^+]$ *Children* : $[17, 4, 18]$ *Parents* : $[13, 14, 17]$

Appendix C

Design Fragments Framework

This appendix presents an initial population for the design fragments framework as follows. C.1 informally discusses the creation and addition of each design fragment illustrated in C.2. C.1 also presents some example designs and design strategies using these design fragments (however, it does not include any validation of these designs). D.5 presents the design fragments introduced in Chapter 6.

C.1 Populating the Framework

This section presents an initial population for the *Design Fragments Framework*. Several examples will illustrate the use of these design fragments; and, as a result, show that the proof process for contingencies can be quite complex.

Section 4.2.5 introduced the specific details of design fragments. Two types of fragments were defined: (1) IPC fragments, to satisfy inter-actor communications (i.e., communications between EBDs allocated to different actors), and (2) SEQ fragments, to satisfy intra-actor communications (i.e., communications between co-allocated EBDs). These fragments would support the design of the following types of specification segments: (1) individual interactions (e.g., a synchronous interaction), (2) individual primitive operations (e.g., a synchronous send primitive), (3) patterns of interaction (e.g., a typical *client-server* model), (4) collapsed behaviours, and (5) EBDs. This section will generate an initial population of IPC and SEQ design

fragments for the given set of specification formats all of which depend on the concept of an actor satisfying the requirements of a basic model (cref. § 4.2.4).

C.1.1 Inter-Actor Communications — IPC Design Fragments

This section defines the initial population of the framework for inter-actor communications. The design fragments are drawn such that assumed basic model design components (e.g., those in Figure 4.11) are illustrated by dashed lines, and additional design components are shown using solid lines.

C.1.1.1 Individual Interactions

- **Asynchronous Interaction – DF-AsyncInter-IPC-1** This design fragment is defined in Section 4.2.5.

- **Synchronous Interaction – DF-SyncInter-IPC-1** A synchronous interaction is characterized by a transition in the sender entity which matches:

any antecedent/any transition operation/receiver!message

and a corresponding transition (with the same message) in the receiver entity; i.e., matching:

sender?message/any transition operation/any transition consequence

The corresponding, emulation, design fragment supplied for this interaction combination is shown on Page 345. It is labelled DF-SyncInter-IPC-1 to identify it as: Design Fragment for Synchronous Interaction of Communicating Actors Number One. This alternative is application independent and requires no further proofs beyond those given below.

Proof of DF-SyncInter-IPC-1 To prove that this design fragment is always applicable, all of the requirements of a synchronous send and receive must be shown. The sending requirements are satisfied as follows: (1) The engine responds to the *send(E)* action by *pushing* the finger which corresponds to this event. By definition,

a push finger represents a blocking send, therefore the engine is blocked until the push is countered by a pull (which must therefore correspond to the reception of the event). The engine of the receiving gate has no statements within the critical section, therefore it replies to the push immediately upon reception of the event. Thus, the send operation completes when it is received. (2) The event is available for reception on this push finger.

The receiver requirements are also satisfied (generally by the basic model design). (1) An actor gate, by definition, queues the incoming messages of a push finger. (2) Pulling the single choice bar defines a fair selection of all open gates. The waiting place added to open and close the gate is defined by the global boolean parameter which sets the event as a member of the satisfiable event set. (3) Pulling an actor gate with a null critical section immediately releases a blocked sender. (4) Because a pull finger represents a blocking receive, the behaviour of the ACM is held when all open gates are void of visiting messages and a *receive* action is outstanding. QED.

Contingencies associated with DF-SyncInter-IPC-1 This design fragment requires a basic model design for both the sender and receiver actors.

Properties assessed for DF-SyncInter-IPC-1 The sender is delayed awaiting the rendezvous.

• **Transient Broadcast Interaction – DF-TransBInter-IPC-1** A transient broadcast interaction is characterized by a transition in the sender entity which matches:

any antecedent/any transition operation/message<

and a corresponding transition (with the same message) in the receiver entity(s); i.e., matching:

>message/any transition operation/any transition consequence

The corresponding, emulation, design fragment supplied for this interaction combination is shown on Page 350. It is labelled DF-TransBInter-IPC-1 to identify it as: Design Fragment for Transient Broadcast Interaction of Communicating Actors Number One. This alternative is application independent and requires no further

proofs beyond those given below.

Proof of DF-TransBInter-IPC-1 To prove that this design fragment is always applicable, all of the requirements of a transient broadcast send and receive must be shown. The sending requirements are satisfied as follows: (1) The engine responds to the *send(E)* action by initiating a *balking push* finger(s) which corresponds to this event. By definition, a *balking push* finger represents a blocking conditional send, therefore the engine is blocked only if it is received immediately. The engine of the receiving gate(s) has no statements within the critical section, therefore it replies to the push immediately upon reception of the event. Assuming that the system has a finite number of receiving actors, the send operation completes when each *balk* finger has been pushed. (2) The event is available for reception on this push finger.

The receiver requirements are also satisfied. (1) An actor gate, by definition, queues the incoming messages of a push finger, however the *balking push* is conditional to immediate reception; i.e., it is assumed to be discarded by the system if not immediately received. The waiting place added to open and close the gate is defined by the global boolean parameter which sets the event as a member of the satisfiable event set. (2) Because a pull finger represents a blocking receive, the behaviour of the ACM is held when there are no available messages and a *receive* action is outstanding. QED.

Contingencies associated with DF-TransBInter-IPC-1 This design fragment requires a basic model design for both the sender and receiver actors.

Properties assessed for DF-TransBInter-IPC-1 Messages are lost if the receiver is not ready and waiting for the corresponding event arrival.

• **Example:** The framework has been populated with enough information to create a design which emulates any specification for which all collapsed behaviours are *flattened* to give a specification with a single level of hierarchy. For example, a provably correct design for the Readers/Writers specification of Figure 3.18 is given in Figure C.1. The reader and writer pool processes are illustrated by actors which slightly overlay another actor. Since every interaction in the specification is asynchronous, design fragment DF-AsyncInter-IPC-1 is used to build this design interaction by

Figure C.1: Emulation Design for the Readers/Writers Problem

interaction. Every interaction is initiated by a jab finger and received through a gate which is opened and closed via a waiting place whose value is equivalent to, and defined by the satisfiable event set. The engines pull a single *choice* finger to activate all open gates when a receive action is to be performed. All gates have null critical sections, but execute a post visit push on the actor's ACM to indicate the corresponding *received(E)* event. Every requirement of the specification can be strongly traced to the design. Queueing limitations are a function of the number of readers and writers and the speed or priority of the scheduler.

C.1.1.2 Individual Primitive Operations

The initial population of design fragments for the individual primitive operations are relatively easily defined using the interaction design fragments of the previous section. Each design fragment in this section is contingent on a complementary design fragment which completes the interaction. The presentation provided below is very informal.

- **Asynchronous Send Primitive Operation – DF-AsyncTx-IPC-1** The sender requirements are satisfied by a jab finger dedicated to this event, and the engine MCL extension given for the sender in DF-AsyncInter-IPC-1. Given this design fragment, there is an outstanding design contingency for correct inter-process com-

munications; i.e., this finger must be coupled with an asynchronous receiver for the same event.

- **Asynchronous Receive Primitive Operation – DF-AsyncRx-IPC-1** The receiver requirements are satisfied by the guarded gate dedicated to this event as defined for the receiver in DF-AsyncInter-IPC-1. Given this design fragment, the outstanding design contingency for correct inter-process communications is that this gate must be coupled with an asynchronous sender of the same event.

- **Synchronous Send Primitive Operation – DF-SyncTx-IPC-1** The sender requirements are satisfied by a push finger dedicated to this event, and the engine MCL extension given for the sender in DF-SyncInter-IPC-1. Given this design fragment, the outstanding design contingency for correct inter-process communications is that this finger must be coupled with an synchronous receiver for the same event.

- **Synchronous Receive Primitive Operation – DF-SyncRx-IPC-1** The receiver requirements are satisfied by the guarded gate dedicated to this event as defined for the receiver in DF-SyncInter-IPC-1 (i.e., the null critical section for the gate engine and the push to the ACM button with the event *received(msg)*). Given this design fragment, the outstanding design contingency for correct inter-process communications is that this gate must be coupled with an synchronous sender of the same event.

- **Transient Broadcast Send Primitive Operation – DF-TransBTx-IPC-1** The sender requirements are satisfied by a balking push finger dedicated to this event, and the engine MCL extension given for the sender in DF-TransBInter-IPC-1. Given this design fragment, the outstanding design contingency for correct inter-process communications is that this finger must be coupled with an transient receiver for the same event.

- **Transient Broadcast Receive Primitive Operation – DF-TransBRx-IPC-1** The receiver requirements are satisfied by the guarded gate dedicated to this event as defined for the receiver in DF-TransBInter-IPC-1. Given this design fragment, the outstanding design contingency for correct inter-process communications is that this gate must be coupled with an transient broadcast sender of the same event.

These additions to the framework have not altered the possible resultant designs of an EBD specification: i.e., there is only one alternative for sender and receiver designs, given a particular interaction. Unfortunately, not all implementation languages support all of these design elements, thus alternative design fragments are often needed. For example, Ada does not directly support the semantics of a jab finger, therefore to accommodate Ada, alternative solutions are required. Below are some example alternatives.

● **Rendezvous Send Primitive Operation – DF-AsyncTx-IPC-2** The sender requirements are satisfied a design fragment equivalent to DF-SyncTx-IPC-1 if: (1) a synchronous receiver for the same event is matched with the synchronous send, and (2) there is a guarantee that the send eventually completes (which was not the case for a synchronous sender). Synchronous receives are guaranteed to complete once initiated (see definition in Table 4.2), but there is no requirement to synchronize with a sender if the corresponding receive is not initiated. Therefore, to conform to the requirements of an asynchronous send, which requires that the send complete eventually, the synchronous receiver in this design fragment, must be shown to initiate a receive for each initiated send. This added contingency is application-specific; i.e., it is necessary to examine the system behaviour to prove that the push finger is eventually pulled. For this design fragment there is an issue of performance: i.e., the delay associated with the newly required synchronization. For example, if the reception is tied to interactions with an external entity, the delay can become non-deterministic, and this should be a consideration when using this design fragment. This fragment represents the sender side of the “rendezvous” solution discussed in Section 4.2.5.

Other common solutions for asynchronous senders and receivers (using only synchronous communications for implementation) use *courier* and *buffer* actors respectively to reduce the time that a sender is blocked waiting to complete a send operation (cref. § 4.2.2). A courier can only initiate calls thus the original sender can wait on other incoming events while trying to send the previously determined, outgoing, asynchronous event. The courier acts out the simple loop: *get_message; put_message*. It is typically used by a *server*- or *dispatcher-type* entity as discussed in Section 4.2.5

as DF-AsyncInter-IPC-3. A buffer can only receive messages, (i.e., it pulls on gates *get_message* and *put_message*, and *get_message* is closed when the buffer is empty). It cannot deny a *put_message* call, therefore a sender is always guaranteed to be freed relatively quickly. Both of these descriptions assume unbounded buffers, therefore, in reality both designs have the property that the associated buffer can overflow. This is a relevant issue to highlight for these design fragments.

● **Courier Send Primitive Operation – DF-AsyncInter-IPC-3** In this design fragment a courier is used to transport a message from the sender to the outgoing finger of the courier. A detailed presentation is given in Section 4.2.5.

● **Buffer Receive Primitive Operation – DF-AsyncRx-IPC-3** In this design fragment the receiver actor is making a call to a buffer actor to receive a message instead of pulling the corresponding *choice* finger (see Page 344). Therefore all of the events, of any satisfiable event set which includes this event, must be selected by the buffer according to the associated EI directives; i.e., the *receive_place* for this satisfiable event set has been moved to the buffer along with the *receive_placeholders* for each event in the satisfiable event set. In this example, only one event is communicated using the buffer actor.

The asynchronous send is guaranteed to complete because (assuming an unbounded buffer) the buffer actor is never blocks the sender because the *msg* gate (i.e., the *receive_placeholder* for *msg* which is now part of the buffer actor) is always open and eventually pulled. Again, this design fragment is typically used when only synchronous interactions can be implemented. Because the buffer is bounded, there are performance issues when using this fragment; i.e., delays or losses can result from a full buffer.

The buffer inside the buffer actor of DF-AsyncRx-IPC-3 must be large enough that it does not overflow; i.e., the receiver must push the *get_msg* entry of the buffer often enough to avoid overflow.

Finally, for the case where synchronous interactions are not provided by the implementation language, DF-SyncInter-IPC-2 illustrates a design fragment for conforming to a synchronous event interaction in the specification using only asynchronous implementation components.

• **Dual Call Synchronous Interaction – DF-SyncInter-IPC-2** In this case, a synchronous send in the specification is designed using an asynchronous implementation as shown on Page 346. Here, the behaviours are equivalent if either: (1) the receiver is always waiting when the event is sent; i.e., the synchronization is guaranteed without blocking, or (2) the sender does not perform any functional requirements until after the receiver has received the event. The second case occurs, for example, when the original sender awaits an acknowledgment from the receiver immediately after the send, and this is the premise for DF-SyncInter-IPC-2. This design fragment uses two asynchronous messages to emulate the send-wait-acknowledgment and receive-wait-send semantics of a synchronous interaction: (1) the original event interaction, and (2) a *wait acknowledgment* interaction.

C.1.1.3 Patterns of Interaction

Specific segments of a specification, or sequences of interactions, have distinct properties from which alternative designs are possible. The framework must include potential solutions for recognizable interaction scenarios. For example, the courier (DF-AsyncInter-IPC-3) and buffer (DF-AsyncRx-IPC-3) solutions are applicable to server scenarios. The courier is used by a server (dispatcher) to forward messages or results without blocking the server, while a buffer is used by a server (administrator) to collect requests so as to avoid blocking the original senders; e.g., as DF-Dispatcher-IPC-1 and DF-Administrator-IPC-1 respectively. Another example interaction pattern is the client/server scenario and design fragment discussed in Section 4.2.5.1.

• **Simple Client/Server Interaction – DF-ClientServer-IPC-2** In this design fragment the receiver (server) has been changed from an actor to a box. The interaction is therefore a procedure call. This has similar semantics to the rendezvous alternative described above in terms of the blocking and sending of events. This design alternative has other contingencies which are a result of the using a box in place of an actor. Where an actor guarantees mutual exclusion of event receives, the box does not. An example, where this contingency is guaranteed, is a receiver actor that has only a single source of events (e.g. entity A interacts only with entity B). A

single event source cannot generate more than one event at a time when restricted to procedure call interactions. QED.

C.1.2 Intra-actor Communications — SEQ Design Fragments

This section defines the initial population of the framework for intra-actor communications.

C.1.2.1 Individual Interactions

- **Actor-Level Scheduled Interactions – DF-Inter-SEQ-1** The first potential solution for a many-to-one allocation builds a design by abstracting a single correct schedule (order) for the requirements of the co-allocated behaviours. This design fragment is defined in Section 4.2.5.2.

- **Segment-level Scheduled Interactions DF-Inter-SEQ-2** This design fragment is the segment-level alternative of DF-Inter-SEQ-1; i.e., it allows for the sequential design of a collapsible portion of co-allocated behaviours (see Page 356). In this case, `send_` and `receive_placeholders` are again defined by the schedule, and since the service being designed must, in itself, complete eventually, proof for the individual inter-actor communications become components of the overall proof of a collapsible operation.

This design fragment is placed at the `action_` or `interaction_placeholder` of the initial behaviour in the specification segment. At the end of the sequence, each ACM involved in the schedule must be advanced to correctly reflect the execution of the behaviour pattern. The default implementation at all `action_placeholders` in the basic model and associated with the behaviour pattern, are removed: they should never execute. An example is given later in this section.

C.1.2.2 Individual Primitive Operations

A potential solution which considers each primitive operation individually is one where the EIs are emulated using an approach analogous to an event driven simu-

start(OP) - - This represents the action_placeholder for the *start(OP)* action. The default implementation has the engine call the operation, *OP*. When this completes, the *completed(op)* event is sent to the originating ACM, and the action returned by the ACM is placed at the end of the event queue.

Table C.1: Processing for *start(op)* action in DF-Prim-SEQ-N

lation. In this design, the co-allocated ACMs are managed by a single *simulation* engine. To start the behaviour of an actor, the engine sends an *init* event to each of the ACMs, and queues the returned action in a FIFO *event queue*. The actions stored in the queue are processed as follows as defined in Tables C.1, C.2, C.3.

• **Actor-level Simulation Engine – DF-Prim-SEQ-1** This design fragment is illustrated on Page 357. The engine of the single actor interacts with the co-allocated ACMs according to the algorithm detailed above and shown on Page 357. The event queue is initialized with the actions returned in response to *init* events which are sent to each ACM (cref. § 4.2.4). All actions defined by the combined behaviour are managed accordingly with each application-specific action being considered separately in an initial design, thus allowing segment-level design fragments to be positioned appropriately. Once an actor's design is complete, the actions which remain (i.e., those which have not been removed by the addition of a segment-level design fragment), must be expected by the actor-level design (e.g., the basic model).

Proof of DF-Prim-SEQ-1 In this design, EIs are implemented by a single engine component. The processing of ACM actions is constrained to a FIFO ordering equivalent to the order in which the actions are sent given a static initialization. Therefore one particular, but correct path of the reachability tree is executed for the default queueing and selection policy of an EI. Design contingencies exist when a satisfiable set is split between inter-actor and intra-actor events. In this case a non-FIFO schedule must be defined to manage the receipt of the event; i.e., to define the *receive_placeholder*. The design must be proven to satisfy the selection discipline of the EI; i.e., to be fair. Also, when inter-actor events are sent or received, the interactions must be proven to complete eventually (unless the combined behaviour of the co-allocated EBDs is blocked because of the interaction). These are

receive an intra-actor event — This is the *action_placeholder* for a *receive* action when an ACM's currently defined satisfiable event set contains only intra-actor events. By default, the engine looks for a match between the satisfiable event set of the originating ACM and the *send(E)* actions of the event queue (where *E* is a single message in this design of the event queue). If a match exists then the *interaction_placeholder* is defined. By default, the *send(E)* action is removed from the queue, and the receiver is sent a *received(E)* event. When the receiver ACM returns, that action is queued. The ACM for the sender of *E* is sent a *next_action* event, and the returned action is also queued. When all actions are queued, the engine examines the next action at the front of the event queue.

If no match is found, then the action is returned to the end of the queue. It is processed when the matching *send(E)* reaches the head of the queue, or when this action is once again at the head of the queue: i.e., it is processed eventually.

receive an inter-actor event — This is an *action_placeholder* for a *receive* action. Here, the ACM's satisfiable event set contains only inter-actor events. By default, the engine defines the *receive_placeholder* for the given satisfiable event set, and initiates the inter-actor receive. The engine must, therefore pull a gate or choice bar (where the guards define the satisfiable set) for this action such that the requirements of a basic model are satisfied. It is possible that one or more of the behaviours encapsulated by this actor are independent of this interaction, and in this case, it is necessary to prove that the receive will complete eventually.

receive on a satisfiable set containing both inter and intra-actor events — This is an *action_placeholder* for the corresponding *receive* action. For the default implementation, this case requires a separate design decision which defines a fair schedule for accepting the events of the satisfiable set. For example, the designer could prove that the set of inter-actor events and the set of intra-actor events define disjoint satisfiable sets; i.e., either a inter-actor event is satisfiable, or an intra-actor event is satisfiable, but never both at the same time. In this case it could be fair to check for a match with the intra-actor event set, and if this fails, attempt (with a conditional receive design) to satisfy the action with an inter-actor event. If both fail, then the action is returned to the end of the queue.

As another example, it might be possible to prove that the co-allocated behaviour will, at some point and infinitely often, require an inter-actor interaction from this satisfiable set. In this case, the schedule would accept intra-actor events until the behaviour is dependent on the external interaction. At that point, the inter-actor receive takes place.

The design chosen is in all cases contingent on fairness; i.e., "fair" is the selection policy defined for the EI when dealing with a satisfiable event set. The design must also correctly define all of the corresponding placeholders.

Table C.2: Processing for *receive* action in DF-Prim-SEQ-N

send(E) on intra-actor event — This defines the *action_placeholder* of the *send(E)* action. The default implementations are as follows.

Asynchronous Event Send — If a corresponding *receive* action is in the event queue, then the *interaction_placeholder* is defined. By default, the action is removed from the queue, and the ACM sending the message is pushed with a *next_action* event. After the action returned by the sender ACM is queued, the receiver ACM is called with a *received(E)* event. When this ACM returns an action, the action is queued in the event queue, and the engine continues by processing the action at the front of the event queue.

Otherwise, the *send(E)* action is placed at the end of the event queue, the sender ACM is called with a *next_action* event, and, after the returned action is queued, the engine continues with the action currently at the head of the event queue.

Synchronous Event Send — If a corresponding *receive* action is in the event queue, then this *interaction_placeholder* is defined. The default implementation for a match found is the same as for asynchronous messages. Otherwise, the *send(E)* action is placed at the end of the event queue, and the engine continues by processing the action at the head of the event queue; i.e., the sender is not advanced by a *next_action* event.

Transient Broadcast Event Send — If a corresponding *receive* action is in the event queue, then this *interaction_placeholder* is defined. Again, the default approach is the same as for asynchronous messages. Otherwise, the *send(E)* action is discarded, and the engine continues by processing the action at the head of the event queue.

send(E) on inter-actor event — This is an *action_placeholder* for the corresponding *send(E)* action. By default, the engine defines the *send_placeholder* for this event and initiates the *send(E)* action. As with the *receive* action, it is necessary to prove that the send completes eventually if any of the other co-allocated behaviours could progress without completing this particular interaction.

Table C.3: Processing for *send(E)* action in DF-Prim-SEQ-N

specification contingencies. QED.

• **Segment-level Simulation Engine – DF-Prim-SEQ-2** This design fragment is illustrated on Page 358. A specification segment is designed using the simulation event queue approach. Because the segment must complete eventually, the event queue is empty on completion of the segment, thus the simulation defined above

continues until the queue is empty. The queue is initialized with the actions that are outstanding at the beginning of the segment: e.g. if the segment begins with an interaction, then it is placed at the appropriate *interaction_placeholder*: here, it is known that the representative actions are outstanding. If the segment begins with an action, then it is placed at the corresponding *action_placeholder* with this action as the contents of the event queue. The default implementations become redundant for all *action_placeholders* covered by the specification segment, and are removed.

Proof of DF-Prim-SEQ-2 The proof is as given for DF-Prim-SEQ-1.

• **Example:** The classic producer/consumer problem with the EBD specification shown in Figure C.2 will be considered. The rhombus shaped EBD states are initial, *start*, and termination, *stop*, states [23]. In the termination state, the EBD has exhausted all behavioural requirements. The design will allocate the producer and buffer entities to one actor, and the consumer entity to its own actor. An emulation design will be used as a basis for the producer/buffer combination (i.e., DF-Prim-SEQ-1), and for the interactions with the consumer design (i.e., using DF-AsyncInter-IPC-1).

To complete the system design, the producer/buffer actor must correctly manage the interactions with the consumer. Therefore, the buffer's receive operation must accept, in a manner deemed as *fair*, both producer requests for empty buffer slots and consumer requests for full buffer slots¹. It can be proven that the producer and buffer are both blocked whenever the buffer is full, therefore the requirement to receive consumer requests can initiate a *pull RqstFull* whenever both the producer and the buffer entities are unable to otherwise continue. The consumer will complete its *service* which includes the event *Return* before the producer/buffer combination could continue.

This design will constrain the behaviour to the following pattern (where *p* represents the producer accessing the buffer slots, and *c* represents the consumer):

$(p;p;p;\dots;p);(c;p;c;p;c;p;\dots);(c;c;\dots;c)$; i.e.,

¹The events sent by the producer (*RqstEmpty* and *PutBuff*) are disjoint (never both outstanding), as are the consumer events (*RqstFull* and *Return*).

Figure C.2: Producer/Consumer EBD Specification

the producer initially fills the buffer, and then the consumer and producer take turns at accessing the slots until the producer enters its final state where it no longer has any behavioural requirements. This design is correct under the assumption that the producer is either continuously filling the buffer, or it has entered its final, termination state. However, if the producer takes a coffee break, then the consumer is forced to take a break at relatively the same time. Details of this design are given in Section 5.1.8.

C.1.2.3 Patterns of Interaction

As in the case of design fragments for inter-actor communications, the first design fragment in this category is for client/server scenarios. In this case, when the server

EBD (was entity A in the discussions given earlier) matches the required pattern for all incoming events, a design fragment which uses direct calls can be applied in the design. This design fragment is typically used to cover the requirements of a single service within a combined behaviour of co-allocated EBDs. It replaces the default implementation for each action defined by the segment, and is placed at the *interaction_placeholder* of the initial *request* interaction.

• **Segment-level Client/Server Sequence – DF-ClientServer-SEQ-1** For this design fragment, the engine makes a direct call to the server operation in place of the default implementations for each action in the sequence (see Page 359). The ACMs are advanced according to the specification segment, however, the returned information is redundant because this is essentially a schedule (DF-Inter-SEQ-1) of this segment. Therefore, when placed within a design where the information of the ACMs is always redundant, this advancement of the ACMs is not necessary. The design fragment is placed at the *interaction_placeholder* for the initial interaction (the *request* interaction).

Proof of DF-ClientServer-SEQ-1 The proof here is equivalent to that for DF-ClientServer-IPC-2, however the contingency for proving mutually exclusiveness of the server is removed. Because all of the EBDs are allocated to the same actor, mutual exclusion of every operation is guaranteed by the nature of a sequential execution. All operations are allowed to complete before another is started. The information of the server EBD is redundant since the semantics of a button are equivalent to: (*receive/perform operation/return to original sender*). QED.

• **Example:** If the Readers/Writers problem is given an all-to-one allocation, the design of Figures C.3 and C.4 is provably correct. The design was achieved as follows. Initially, the designer chooses to apply DF-ClientServer-SEQ-1 for the read and write services; i.e., matching $s+gr;read;s-rf$ as the server specification in the read case, and matching $s+gw;write;s-wf$ as the server specification in the write case. DF-Inter-SEQ-2 is then considered as a potential solution for sequencing the $s-rr;wait\ grant;s+gr$ and $.r+rr\{set(r=SID)\};[C'=C+1];r-gr$ interacting segments. Finally, the designer considers DF-Prim-SEQ-1 as the overall actor design frag-

Figure C.3: Emulation of the Readers/Writers Problem Allocated to One Actor - Part 1

ment. The design process is detailed below.

Step 1. Using DF-ClientServer-SEQ-1 — The designer wishes to use DF-ClientServer-SEQ-1 for the reader and writer *server* sequences: $s+gr;read;s-rf$ and $s+gw;write;s-wf$ respectively. For the writers, there is a direct match for both the client and server behaviours: i.e., the (client) behaviour is the $w-gw;writer;_w+wf$ of the scheduler, where *writer* is a wait state, and the server behaviour is as shown above. For the readers, the match is not explicit in the (client) scheduler specification. In this case, the scheduler transitions to state 2, *readers*, when the grant event, *gr*, is issued. This is not a wait state with a single exit to a different state as required by DF-ClientServer-SEQ-1, therefore to use this design fragment, the designer must prove that it is appropriate to effectively convert the scheduler behaviour such that it will conform to the required pattern.

The events $_w+wr$, $_r+rr$, and $_r+rf[C>1]$ must be removed from the satisfiable event set of state 2 of the scheduler, and yet the constrained behaviour must still conform to a correct system behaviour. All of these events must be deferrable to scheduler state 1, *init*, or be undefined in the constrained specification. Both

```

engine is begin
- DF-Prim-SEQ-1
event = init; push scheduler_ACM; push reader_ACM; push writer_ACM;
- place returned actions into event queue
loop
- remove head of event queue
case action is
  when send(s-rr) => if matching receive for List where  $.r+rr \in \text{List}$  then
    - dequeue the matching receive action
    - DF-Inter-SEQ-2
    skip; - push empty operation
    - advance the ACMs
    event = received; push scheduler_ACM; - returns send(r-gr)
    event = next_action; push reader_ACM; - returns receive for (s+gr)
    - part of DF-Inter-SEQ-2 or DF-ClientServer-SEQ-1
    event = next_action; push scheduler_ACM; - return receive for ( $.r+rf$ )
    event = received; push reader_ACM; - returns start(Read)
    - strictly DF-ClientServer-SEQ-1
    push Read;
    - advance the ACMs
    event = completed(read); push reader_ACM; - returns send(s-rf)
    event = next_action; push reader_ACM; - returns send(s-rr)
    event = received; push scheduler_ACM; - returns receive for ( $.r+rr|.w+wr$ )
  else - queue send(s-rr) at end of the event queue
  when send(s-wr) => if matching receive for List where  $.w+wr \in \text{List}$  then
    - dequeue the matching receive action
    - advance the ACMs
    event = received; push scheduler_ACM; - returns send(w-gw)
    event = next_action; push writer_ACM; - returns receive for (s+gw)
  else - queue send(s-wr) at end of the event queue
  when receive for ( $.r+rr|.w+wr$ ) => if matching send(s-rr) then
    - dequeue the matching send action
    - same as above for this match
  elseif matching send(s-wr) then - same as above for this match
  else - queue receive for ( $.r+rr|.w+wr$ ) at end of event queue
  when send(w-gw) => if matching receive for List where  $s+gw \in \text{List}$  then
    - dequeue the matching receive action
    - DF-ClientServer-SEQ-1
    event = received; push writer_ACM; - returns start(write)
    event = next_action; push scheduler_ACM; - returns receive for ( $.w+wf$ )
    push Write;
    - advance the ACMs
    event = completed(write); push writer_ACM; - returns send(s-wf)
    event = next_action; push writer_ACM; - returns send(s-wr)
    event = received; push scheduler_ACM; - returns receive for ( $.r+rr|.w+wr$ )
  else - queue send(w-gr) at end of the event queue
  when receive for (s+gw) => if matching send(w-gw) then
    - dequeue the matching send action
    - same as above for this match
  else - queue receive for (s+gw) at end of event queue
end case; - queue returned actions
end loop; end.

```

Figure C.4: Emulation of the Readers/Writers Allocated to One Actor – Part 2

Refined Scheduler EBD Specification:

Scheduler

1: Init

2: Readers

4: Writer

legend:

s: scheduler

w: writer

r: reader

rr: read request

rf: read finished

gr: grant read

wr: write request

wf: write finished

gw: grant write

Figure C.5: Refined Readers/Writers Specification

$_w+wr$ and $_r+rr$ can be deferred to state 1 and still be given fair consideration by the specification as it is defined for state 1. Given that $_r+rr$ events are no longer accepted at state 2, the auxiliary counter variable, C , is constrained to the value 1, and therefore $_r+rf[C > 1]$ is never satisfiable. Therefore, the design fragment can be applied if it is fair to constrain the value of C to one. This implies that readers are managed sequentially instead of being allowed to access the resource in parallel.

Given an all to one allocation, only one co-allocated behaviour can be active at one time anyway, therefore this is not an unreasonable restriction of the behaviour specification. (Any other design would require a design fragment which time-sliced the read operations.) To implement this design fragment, the scheduler specification is refined as shown in Figure C.5; and, as such, the contingencies of the client/server scenario are satisfied.

Given this refinement, the design process must be re-started. The client/server scenario for writers is still valid, and thus DF-ClientServer-SEQ-1 is applied to both reads and writes. The *read* and *write* operations are therefore designed as buttons which are activated whenever the corresponding behaviour sequences are

to be carried out (see Figure C.3). The ACMs must be advanced accordingly, and the corresponding action_ and interaction_placeholders are defined for the actions/interactions of the sequence within this design fragment implementation. Figure C.4 illustrates the advancement of the ACMs, however, space did not permit the identification of all placeholders. The placeholders can be determined using the design fragment description on Page 359. All default implementations for the actions involved in these segments are now redundant and removed. QED.

Step 2. Using DF-Inter-SEQ-1 — The reader sequence: *s-rr;wait grant;s+gr*, is always matched with the scheduler sequence: *r+rr;empty operation;r-gr*. Therefore, whenever an *s-rr* and *r+rr* interaction occurs, it is known that the scheduler's empty transition operation follows, and that the *s+gr* and *r-gr* interaction follows again. The design fragment places the empty operation at the interaction_placeholder for the *rr* interaction, and removes the default implementations for the *start(empty op)* action_placeholder, the *send(r-gr)* action_placeholder, the *receive* action_placeholder for the following satisfiable event set: *(s+gr)*, and the *gr* interaction_placeholder. It also advances the reader and scheduler ACMs through the sequence, and defines new action_ and interaction_ placeholders for the actions/interactions involved (see Figure C.4).

Therefore, the client/server design fragment for the read operation defined above, is placed at the *gr* interaction_placeholder defined by this design fragment².

Step 3. Using DF-Prim-SEQ-1 — The overall actor design is covered by the DF-Prim-SEQ-1 design fragment. Here, an event simulation design is used to drive the combined behaviour. To start, this design fragment considers every application-specific, ACM defined, action. Therefore, there is a case statement for each application-defined action which then, by default, deals with that action according to the algorithm defined by the design fragment. This design fragment is applicable to all combined behaviours: it represents an emulation of the specifica-

²When segment-level SEQ design fragments overlap, one replaces the overlapped segments defined by the other. In this example, the overlapped segment is the *gr* interaction. In both designs the implementation advances the ACMs involved in the interaction, therefore, applying one before the other has no affect on the final design.

tion.

The client/server design fragment for the writer and scheduler interactions, defined above, replaces the actions covered by the fragment. The implementation defined by the fragment: (1) is placed at the interaction_placeholders for the *gw* interaction (there are two in this example, one after matching a *send(w-gw)* action in the corresponding case statement of DF-Prim-SEQ-1, and one after matching the *receive* action with satisfiable event set: $(s+gw)$ in this corresponding case statement), and (2) removes the implementations (the cases) for the following actions: *start(write)*, *send(s-wf)*, and *receive* for the $(\neg w+wf)$ event.

The client/server design for the reader/scheduler interactions would have a similar result if the DF-Inter-SEQ-2 fragment, applied to the read request and grant sequence, was not being used. However, this fragment is applied. It is placed at the interaction_placeholders for the *rr* interaction, and removes the default implementations for the following actions (cases): *send(r-gr)*, and *receive* for the $(s+gr)$ event. New action_ and interaction_placeholders are defined by the application of DF-Inter-SEQ-2. The client/server design fragment for the reading operation is placed at the new *gr* interaction_placeholder replacing that implementation with, in this case, exactly the same MCL. This fragment also removes the default implementations for the *start(read)* action, the *send(s-rf)* action, the *receive* action for the $(\neg r+rf)$ event, and the *rf* interaction. This leaves only the *send(s-rr)*, *send(s-wr)*, *receive* for $(\neg r+rr)\neg w+wr$, *send(w-gw)*, and *receive* for $(s+gw)$ case statement implementations for the overall DF-Prim-SEQ-1 design, as shown in Figure C.4.

With this design, the reader and writer ACMs can be made redundant if the event queue is initialized with a randomly generated set of *send(s-rr)* and *send(s-wr)* events, and a DF-Inter-SEQ-1 design fragment is used for the writer/scheduler request and grant interactions as shown in Figure C.6. In this case, the overall design fragment is not DF-Prim-SEQ-1, but another (not yet defined: e.g., DF-Prim-SEQ-3) fragment which has this property for randomly initiating the simulation event queue.

engine is

begin

- **DF-Prim-SEQ-3**

- randomly generate and queue a sequence of read and write requests

event = init;

push scheduler.ACM;

- place returned actions into event queue

loop

- remove head of event queue

case action is

when send(s-rr) => matching receive is always true

- dequeue matching receive action

- **DF-Inter-SEQ-2**

event = received; push scheduler.ACM; - returns send(r-gr)

- part of **DF-Inter-SEQ-2** or **DF-ClientServer-SEQ-1**

event = next.action; push scheduler.ACM; - return receive for (r+rf)

- strictly **DF-ClientServer-SEQ-1**

push Read;

event = received; push scheduler.ACM; - returns receive for (r+rr|w+wr)

else - never occurs

when send(s-wr) => matching receive is always true

- dequeue matching receive action

- **DF-Inter-SEQ-2**

event = received; push scheduler.ACM; - returns send(w-gw)

- part of **DF-Inter-SEQ-2** or **DF-ClientServer-SEQ-1**

event = next.action; push scheduler.ACM; - return receive for (w+wf)

- strictly **DF-ClientServer-SEQ-1**

push Write;

event = received; push scheduler.ACM; - returns receive for (r+rr|w+wr)

else - never occurs

when receive for (r+rr|w+wr) => - stop the simulation;

there are no more send(E) actions

end case;

- queue returned actions

end loop

end.

Figure C.6: Simulation of the Readers/Writers Allocated to One Actor

C.1.3 Collapsed Behaviours

Collapsed behaviours represent the *local* behaviour characteristics of an entity.³ If an operation includes a sequential, collapsed behaviour, then this is a *procedural operation*, requiring the same design considerations as an operation state which performs a single functional requirement. However, if the collapsed behaviour assumes that its incoming events are queued, or that the operation might have to await the arrival of an event, then the operation can, for example, be: (1) translated into a separate actor, or (2) flattened to the same level as the global behaviour. For a "flat" (fully expanded) EBD, the interactions are designed using the design fragments provided in earlier sections. In many cases this behaviour has been collapsed because there is an implicit modularization of this operation from the main, overall functionality of the entity.

• **Collapsed Operation – DF-CollapsedOP-IPC-1** This design fragment is illustrated on Page 360. The *start(OP)* action of the ACM is used to call the *start* gate of the operation actor. This is a rendezvous scenario, where the reply is not returned until the operation is complete. When the rendezvous completes, the actor engine for the entity invokes its ACM with the *complete(OP)* event. The collapsed behaviour is placed into an ACM for the operation actor with an added *idle* state from which the operation is started, and at which the operation will always complete. That is, the ACM of the collapsed behaviour transitions from its *idle* state to the default entry state of the behaviour when a *start* message is received, and it transitions back to the *idle* state when the operation is complete.

Proof of DF-CollapsedOP-IPC-1 The entity actor starts the operation when the *start(OP)* action is returned by the ACM, and is blocked waiting on this rendezvous until the operation is complete. The semantics of the operation are emulated by this design fragment. The set of incoming events must not intersect with the set of incoming events for the actor-level entity behaviour. This specification constraint ensures a unique connection point for entities sending events to this entity. QED.

³Local behaviours are typically externally coupled behaviours; i.e., those which are pre-defined, and generally accept or emit external events [16].

C.2 Framework

C.2.1 Asynchronous Communications

C.2.1.1 Interactions

DF-AsyncInter-IPC-1

Performance Issues:

1. "msg" is buffered at "rx". Delays or losses are possible if buffer limitations are stressed.
2. Otherwise, "msg" is guaranteed to complete.

DF-AsyncInter-IPC-2

Specification Segments

Spec. Contingencies:

1. "tx+msg" must be received eventually by "rx".

Design Fragment

Design Contingencies:

1. Requires the basic model defined in 4.2.4.

tx @send_placeholder for tx-msg:
push msg;

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg);
push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-AsyncInter-IPC-3

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. Requires the basic model defined in 4.2.4.

tx @send_placeholder for rx-msg:
insert(msg,msg_buffer); get_msg := true;

tx@receive_place for wait state:
include get_msg in the choice frame..

add get_msg gate:
begin critical section
msg := dequeue_head(msg_buffer);
end critical section;
if is_empty_queue(msg_buffer) then
get_msg := false;
end get_msg;

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical section, and post visit statements:
event = received(msg);
push ACM;

courier task:
engine is
begin
loop
push get_msg;
push msg;
end loop;
end courier.

Performance Issues:

1. "rx" must receive "msg" and release "courier" often enough to avoid buffer overflow at "msg_buffer"
2. size of "msg_buffer" is dependant on the rate of reception of "msg" by "rx".
3. "rx-msg" can be delayed by any of the "N" paths leading from "wait state".

DF-AsyncInter-IPC-4

Specification Segments

Spec. Contingencies:

None

Design Fragment

Design Contingencies:

1. "actual_param" and "formal_param" must be type compatible.

tx @send_placeholder for rx-msg:
push msg(actual_param); -- jab

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg(formal_param));
push ACM;

Performance Issues:

1. "msg" is buffered at "rx". Delays or losses are possible if buffer limitations are stressed.
2. Otherwise, "msg" is guaranteed to complete.

DF-AsyncInter-IPC-5

Specification Segments

Spec. Contingencies:

1. "tx+msg" must be received eventually by "rx".

Design Fragment

Design Contingencies:

1. "actual_param" and "formal_param" must be type compatible.

tx @send_placeholder for tx-msg:
push msg(actual_param);

rx @receive_placeholder for tx+msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg(formal_param));
push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

C.2.1.2 Primitive Operations

DF-AsyncTx-IPC-1

Spec. Contingencies:

None

Specification Segments

Design Fragment

Design Contingencies:

1. "tx+msg" must be designed as an asynchronous receive of "rx".

Match Rules:

1. "rx-msg" is an asynchronous send of "tx".

2. "tx+msg" must be an asynchronous receive of "rx".

tx @send_placeholder for rx-msg:
push msg; -- jab

Performance Issues:

1. "rx-msg" is guaranteed to complete.

DF-AsyncRx-IPC-1
Spec. Contingencies:

None

Specification Segments

Design Fragment

Design Contingencies:

1. "rx-msg" must be designed as an asynchronous send in "tx".

Match Rules:

1. "tx+msg" is an asynchronous receive of "rx".
2. "rx-msg" must be an asynchronous send of "tx".

rx @receive_placeholder for tx+msg:
 rx gate "msg" has a null critical
 section, and post visit statements:
 event = received(msg);
 push ACM;

Performance Issues:

1. "msg" is buffered by "rx". Delays or losses are possible if buffer limitations are stressed.

DF-AsyncTx-IPC-2

Specification Segments

Spec. Contingencies:

1. "tx+msg" must be received eventually by "rx".

Design Fragment

Design Contingencies:

1. "tx+msg" must be designed as a synchronous receive of "rx".

Match Rules:

1. "rx-msg" is a synchronous send of "tx".
2. "tx+msg" must be a synchronous receive of "rx". tx @send_placeholder for tx-msg:
push msg;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-AsyncRx-IPC-2

Specification Segments

Spec. Contingencies:

None

Design Fragment

Design Contingencies:

1. "rx-msg" must be designed as a synchronous send of "tx".

Match Rules:

1. "tx+msg" is a synchronous receive of "rx".
2. "rx-msg" must be a synchronous send of "tx".

rx @receive_placeholder for tx+msg:
 rx gate "msg" has a null critical
 section, and post visit statements:
 event = received(msg);
 push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-AsyncTx-IPC-3

Specification Segments

Spec. Contingencies:

1. "tx+msg" must be received eventually by "rx".

Design Fragment

Design Contingencies:

1. "tx+msg" must be designed as a synchronous receive of "rx".
2. "actual_param" must be type compatible with the formal parameter defined by "rx".

Match Rules:

1. "rx-msg" is a synchronous send of "tx".
2. "tx+msg" must be a synchronous receive of "rx".

tx @send_placeholder for tx-msg:
push msg;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-AsyncRx-IPC-3

Specification Segments

Spec. Contingencies:

1. Whenever "msg" can be received, there must be no other satisfiable events; i.e., all satisfiable event sets of "rx" that contain "msg", have no other elements.

Design Fragment

Design Contingencies:

1. "rx-msg" must be designed as an asynchronous send in "tx".
2. "actual_param" sent by "tx" and "formal_param" must be type compatible.

Match Rules:

1. "tx+msg" is an asynchronous receive of "rx".
2. "rx-msg" must be an asynchronous send of "tx".

task buffer is begin

get_msg := false; -- close gate

loop

pull choice;

end loop; end buffer.

gate msg is begin

critical section is begin

-- buffer incoming data(formal_param)

end critical section;

get_msg := true;

end gate msg;

gate get_msg is begin

critical section is begin

-- remove head of buffer and assign it to formal_param

end critical section

if is_empty_buffer then get_msg := false; --close gate

end gate get_msg.

rx @receive_place for tx+msg:

push get_msg;

event = received(msg);

push ACM;

Performance Issues:

1. "msg" is buffered by "buffer". Delays or losses are possible if buffer limitations are stressed.
2. "formal_param" is copied into "buffer" for every "tx-msg".
3. "formal_param" is copied into "rx" for every "get_msg".
4. size of "msg_buffer" is dependent on the rate of reception of "msg" by "rx".
5. Otherwise, "msg" is guaranteed to complete.

C.2.2 Synchronous Communications

C.2.2.1 Interactions

DF-SyncInter-IPC-1

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

None.

tx @send_placeholder for tx!msg:
push msg;

rx @receive_placeholder for tx?msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg);
push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-SyncInter-IPC-2

Specification Segments

Spec. Contingencies:

None

Design Fragment

Design Contingencies:

None.

tx @send_placeholder for rx!msg:
 push msg; -- jab
 pull_reply;

rx @receive_placeholder for tx?msg:
 rx gate "msg" has a null critical
 section, and post visit statements:
 push reply;
 event = received(msg);
 push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-SyncInter-IPC-3

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. "actual_param" and "formal_param" must be type compatible.

tx @send_placeholder for tx!msg:
push msg(actual_param);

rx @receive_placeholder for tx?msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg(formal_param));
push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

C.2.2.2 Primitive Operations

DF-SyncTx-IPC-1

Spec. Contingencies:

None.

Specification Segments

Design Fragment

Design Contingencies:

1. "tx?msg" must be designed as a synchronous receive of "rx".

Match Rules:

1. "rx!msg" is a synchronous send of "tx".
2. "tx?msg" must be a synchronous receive of "rx".

tx @send_placeholder for tx!msg:
push msg;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

DF-SyncTx-IPC-1

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. "rx!msg" must be designed as a synchronous send of "tx".

Match Rules:

1. "tx?msg" is a synchronous receive of "rx".
2. "rx!msg" must be a synchronous send of "tx".

rx @receive_placeholder for tx?msg:
 rx gate "msg" has a null critical section, and post visit statements:
 event = received(msg);
 push ACM;

Performance Issues:

1. "tx" blocks until "rx" acknowledges that "msg" has been received.
2. "tx" is delayed by "rx".

C.2.3 Transient Broadcast Communications

C.2.3.1 Interactions

DF-TransBInter-IPC-1

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

None.

tx @send_placeholder for msg<:
push msg; -- balk

rx @receive_placeholder for >msg:
rx gate "msg" has a null critical
section, and post visit statements:
event = received(msg);
push ACM;

Performance Issues:

1. "msg" is lost if "rx" is not ready and waiting when "tx" issues "msg".

C.2.3.2 Primitive Operations

DF-TransBTx-IPC-1

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. ">msg" must be designed as a transient broadcast receive.

Match Rules:

1. "msg<" is a transient broadcast send.
2. ">msg" must be a transient broadcast receive.

tx @send_placeholder for msg<:
push msg; -- balk

Performance Issues:

1. "msg" is lost if "tx" is not ready and waiting when "tx" issues "msg".

DF-TransBRx-IPC-1

Specification Segments

Spec. Contingencies:

None.

Design Fragment

Design Contingencies:

1. “msg<” must be designed as a transient broadcast send.

Match Rules:

1. “>msg” is a transient broadcast receive.
2. “msg<” must be a transient broadcast send.

rx @receive_placeholder for >msg:
rx gate “msg” has a null critical section, and post visit statements:
event = received(msg);
push ACM;

Performance Issues:

1. “msg” is lost if “rx” is not ready and waiting when “tx” issues “msg”.

C.2.4 Patterns of Interaction

DF-ClientServer-IPC-1

Specification Segments

Spec. Contingencies:

1. "wait_ack" must be a null operation with satisfiable set: $\{ \text{server+ack} \}$
2. "operation" must complete eventually.

Design Fragment

Design Contingencies:

None.

client @send_placeholder for server-request:
 push request_ack;
 event= next_action; push ACM;
 event = received(ack);
 push ACM;

server @receive_placeholder for client+request:
 server gate request_ack is
 begin critical section
 event = received(request);
 push ACM;
 push operation; -- button not shown
 event= completed(operation);
 push ACM; -- absorb client-ack action
 end;
 event = next_action; push ACM;
 end.

DF-ClientServer-IPC-2

Specification Segments

Spec. Contingencies:

1. "wait_ack" must be a null operation with satisfiable set: {server+ack}
2. "operation" must complete eventually.
3. "operation" must be re-entrant and mutually exclusive.

Design Fragment

Design Contingencies:

None.

```

client @send_placeholder for server-request:
    push request_ack;
    event= next_action; push ACM;
    event = received(ack);
    push ACM;

```

```

button server.request_ack is
begin

```

```

    push operation;
end.

```

```

-- assumes ACM is redundant and naot used
in package server.

```

C.2.5 Intra-actor Communications

DF-Inter-SEQ-1

Spec. Contingencies:

1. For every incoming message, "rxmsg":
Unless the entire agenda of this actor depends on "rxmsg" to initiate its processing (or processing loop), then "rxmsg" must be sent eventually when this actor is blocked on the corresponding receive_placeholder.
2. For every outgoing synchronous event: "txmsg":
"txmsg" must be received eventually by "rx".

Design Contingencies:

1. "actor" must satisfy the requirements of a basic model; specifying all send_, receive_, action_, and interaction_placeholders.
2. "behaviour engine" must define a fair static schedule for the combined behaviour of "ebd1", "ebd2", ...

Performance Issues:

1. MCL to advance ACMs through their behaviours can be removed if, when the actor's design is complete, all actions returned by the ACMs are redundant information; i.e., already pre-determined by the design.
2. "behaviour_engine" defines priorities and preference of events and entities in "actor".

Specification Segments

Any co-allocated behaviour where the behaviour of all co-allocated EBDs are considered; i.e., this is an actor-level SEQ design fragment.

Design Fragment

"behaviour engine" is a static schedule for the combined behaviours of all EBDs allocated to "actor"

MCL at each interaction_placeholder

(= receive and send action_placeholders respectively):

```
event=received(rxmsg); push receiver_ACM;
event= next_action; push sender_ACM;
```

MCL at each start(OP) action_placeholder:

```
push OP;
event= completed(OP); push ACM;
```

DF-Inter-SEQ-2

Spec. Contingencies:

1. For every incoming message: "rxmsg" must be sent eventually by its sender.
2. For every outgoing synchronous event: "txmsg" must be received eventually by "rx".
3. The behaviour segment must either: 1) represent a sequential segment of the co-allocated behaviour, or 2) define one correct path through the combined behaviour in terms of the respective behaviour.
4. The behaviour of the specification segment must complete eventually.

Design Contingencies:

1. "actor" must satisfy the requirements of a basic model in terms of the specification segment; specifying all send_, receive_, action_, and interaction_placeholders.
2. "segment behaviour" must define a fair static schedule for the combined behaviour of the specification segment.

Performance Issues:

1. MCL to advance ACMs through their behaviours can be removed if, when the actor's design is complete, all actions returned by the ACMs are redundant information; i.e., already pre-determined by the design.
2. The button is not required if the implementation is placed inline of "actor's" engine.
3. "specification_behaviour" defines priorities and preference of events and entities in "actor".

Specification Segments

Any behaviour segment of one or more interacting, co-allocated EBDs.

"specification behaviour" is a fair static schedule for the specification segment. It is placed at the action_ or interaction_ placeholder of the first action/interaction in the combined behaviour.

MCL at each newly defined interaction_ placeholder (= receive and send action_placeholders respectively):
 event=received(rxmsg); push receiver_ACM;
 event=next_action; push sender_ACM;

MCL at each newly defined start(OP) action_placeholder:
 push OP;
 event=completed(OP); push ACM;

The default implementation of "actor", for each action_placeholder covered by the specification segment, is removed.

DF-Prim-SEQ-1

Spec. Contingencies:

1. For every incoming message:
Unless the entire agenda of this actor depends on "rxmsg" to initiate its processing (or processing loop), then "rxmsg" must be sent eventually when this actor is blocked on the corresponding receive_placeholder.
2. For every outgoing synchronous event:
"txmsg" must be received eventually by "rx".

Design Contingencies:

- 1 "actor;" must satisfy the requirements of a basic model: specifying all send_, receive_, action_, and interaction_placeholders.
2. For every receive action with both inter- and intra-actor communications: the design must choose to interact on the inter- or intra-actor communications in an order such that all are considered fairly throughout the processing of the combined behaviour.

Performance Issues:

- 1 This design fragment requires a lot of overhead processing to manage the event queue.

Specification Segments

Any co-allocated behaviour where the behaviour of all co-allocated EBDs are considered: i.e., this is an actor-level SEQ design fragment.

```

"behaviour engine" is
begin
  event = init;
  for each ACM: push ACM; queue action
    in "event queue";
loop
  action := head of event queue;
  case action is
    -- action placeholder for send(E)
    when send(E) => if matching receive with
      E in the satisfiable event set then
      -- interaction_placeholder for msg of E
      -- advance the ACMs
      event = next_action; push sender_ACM;
      event = received(E); push receiver_ACM;
      else add send(E) to end of event queue.
    when send(E2) => send_placeholder for E2.
    -- continue with each application-specific action
    -- of the co-allocated EBDs according to the
    -- algorithm given in C.1.1
  end case;
  queue the returned actions;
end loop;
end.
  
```

DF-Prim-SEQ-2

Spec. Contingencies:

1. For every incoming message:
Unless the entire agenda of this actor depends on `rxmsg` to initiate its processing (or processing loop), then `rxmsg` must be sent eventually when this actor is blocked on the corresponding `receive_placeholder`.
2. For every outgoing synchronous event:
`txmsg` must be received eventually by `rx`.

Design Contingencies:

1. `segment behaviour` must satisfy the requirements of a basic model: specifying all `send_`, `receive_`, `action_`, and `interaction_placeholder`.
2. For every receive action with both inter- and intra-actor communications: the design must choose to interact on the inter- or intra-actor communications in an order such that all are considered fairly throughout the processing of the combined behaviour

Performance Issues:

1. This design fragment requires a lot of overhead processing to manage the event queue.

Design Fragment con't:

`segment behaviour` is placed at the `action_` or `interaction_placeholder` of the first action/interaction of the segment.

The default implementation of `actor` for each `action_placeholder` covered by the specification segment, is removed.

Specification Segments

Any specification segment with interacting, co-allocated EBDs

DF-ClientServer-SEQ-1

Specification Segments

Spec. Contingencies:

1. "wait_ack" must be a null operation with satisfiable set: {server+ack}
2. "operation" must complete eventually

Design Fragment

Design Contingencies:

None

Perfor. Issue Issues:

1. The MCL to advance ACMs through their behaviours can be removed if, when the actor's design is complete, all actions returned by the ACMs are redundant information, i.e., already pre-determined by the design

service is
begin

```

-- action_ and interaction_placeholders for the "request"
-- message
event = next_action; push client_ACM;
event = received(client+request); push server_ACM;
-- action_placeholder for the start(operation) action
push operation;
event = completed(operation); push server_ACM;
-- action_ and interaction_placeholders for the "ack"
-- message
event = next_action; push server_ACM;
event = received(server+ack); push client_ACM;
end;

```

"service" is placed at the interaction placeholder for the "request" interaction.

The default implementation of "actor" for each action_placeholder covered by the specification segment, is removed

C.2.6 Collapsed Behaviours

DF-CollapsedOp-1

Specification Segments

Spec. Contingencies:

1. "collapsed operation" must not perform any receive event actions.

Design Contingencies:

None.

```
tx @action_placeholder for
start(collapsed_operation)::
  push start;
  event = completed(collapsed_operation);
  push ACM;
```

```
actor collapsed_operation is
begin
  loop
    pull start;
  end loop; end.
```

```
gate start is
begin critical section
  event = received(start); push ACM;
  while action != completed loop
    -- perform operation behaviour
  end loop
end gate.
```


C.3 Framework Extensions

DF-Priority-Overwrite-Buffer-1

Spec. Contingencies:

None.

Specification Segments

Design Fragment

Design Contingencies:

1. "actual_param1" of "put_txs" and "formal_param1" must be type compatible.
2. "actual_param2" of "get_txs" and "formal_param2" must be type compatible.

Performance Issues:

1. "actual_param" sent by "put_txs" is copied into "buffer" for every "put_msg".
2. "formal_param" is copied from "buffer" for every "get_msg".
3. Because of overwrite, data is lost.

```

task buffer is begin
  get_msg := false; -- close gate
  loop
    pull choice;
  end loop; end buffer.
gate put_msg is begin
  critical section is begin
    --priority overwrite buffer incoming data(formal_param1)
  end critical section;
  get_msg := true;
end gate msg;
gate get_msg is begin
  critical section is begin
    -- remove head of buffer and assign it to formal_param2
  end critical section
  if is_empty_buffer then get_msg := false; --close gate
end gate get_msg.
  
```

DF-Overwrite-Buffer-1

Spec. Contingencies:

None.

Specification Segments

Design Fragment

Design Contingencies:

1. "actual_param1" of "put_txs" and "formal_param1" must be type compatible.
2. "actual_param2" of "get_txs" and "formal_param2" must be type compatible.

Performance Issues:

1. "actual_param" sent by "put_txs" is copied into "buffer" for every "put_msg".
2. "formal_param" is copied from "buffer" for every "get_msg".
3. Because of overwrite, data is lost.

```

task buffer is begin
  get_msg := false; -- close gate
loop
  pull choice;
end loop; end buffer.
gate put_msg is begin
critical section is begin
  --overwrite buffer incoming data(formal_param1)
end critical section;
  get_msg := true;
end gate msg;
gate get_msg is begin
critical section is begin
  -- remove head of buffer and assign it to formal_param2
end critical section
  if is_empty_buffer then get_msg := false; --close gate
end gate get_msg.
  
```

DF-Periodic-Loop-1

Specification Segments

Spec. Contingencies:

1. "Wait" is a null state.
2. There are no incoming events hidden in the "Evaluate" operation.
3. "rx-msg" is sent at a rate of "timeout period" by "tx".

Design Fragment

Design Contingencies:

1. "formal_param" and "actual_param" of "tx" must be type compatible.

Performance Issues:

1. It must be sufficient to evaluate "msg" at a period of "timeout period".
2. It must be sufficient to send "other events" only after the source!ntuating_event(formal_param) interaction is complete; i.e., at a rate "timeout period".

DF-Periodic-Loop-2

Specification Segments

Spec. Contingencies:

1. There are no incoming events hidden in the "Evaluate" operation.

Design Contingencies:

1. "source" must be designed as a synchronous receiver of "initiating_event(formal_param)".
3. "formal_param" and "actual_param" of "source" must be type compatible.

Match Rules:

1. "source!initiating_event" is an synchronous send of "rx".
2. "rx?initiating_event" must be a synchronous receive of "source".

Performance Issues:

1. It must be sufficient to evaluate "initiating_event" at a period of "timeout period".
2. It must be sufficient to send "other events" only after the source!initiating_event(formal_param) interaction is complete; i.e., at a rate "timeout period".

DF-CASE-SEQ+Couriers

Spec. Contingencies:

1. Each "wait state" must be a null state.
2. for every external synchronous response: "actor?response" must be received eventually by "final_rx".

Specification Segments

actor

... combine N such entities.

Design Fragment

```

actor is begin
-- for all couniered responses:
courier_guard := false -- close gate;
loop
  pull choice;
end loop;
end.

actor @receive_placeholder for
each couniered response:
gate couniered response is
begin critical section
-- copy head of couniered response
  buffer into msg;
end critical section;
if is_empty_courier_buffer then
  courier_guard := false; -- close gate
end;
end;

```

```

actor @receive_placeholder for each
externally triggered stimulus:
gate stimulus is begin
  push deterministic service;
  -- manage_response;
end;

```

```

courier task is
begin
  loop
    push couniered_response;
    push response; -- at receiver
  end loop;
end courier.

```

```

manage_response:
-- if final_rx is external
-- and response is not
-- couniered
  push response;
-- if final_rx is external
-- and response is
-- couniered
-- buffer couniered
-- response
  courier_guard := true;
-- if final_rx is internal
  push deterministic
  service activated
  by this
  interaction
-- manage new final
-- response

```

Design Contingencies:

1. For every couniered asynchronous response: "actor+couniered_response" must be designed as a synchronous receive in "final_rx". Match Rules: seeDF-AsyncTx-IPC-2.
2. formal and actual parameters must be type compatible.

Performance Issues:

1. for every couniered asynchronous response: "final_x" must receive "response" and release "courier" often enough to avoid buffer overflow at "couniered_response_buffer".
2. size of "couniered_response_buffer" is dependant on the rate of reception of "response" by "final_rx".
3. "couniered_response" can be delayed by any of the "N" paths leading from the combined "wait state".

Appendix D

Final EBDs and Tables for Example Design of Chapter 6

D.1 Initial Partition Tables

PARTITION TABLE - Top-Level Partition PARTITION TABLE - Top-Level Partition

Partition: ?SPA?
Parent: \$spa\$

Partition: ?TA?
Parent: \$ta\$

Message	Build Table Reference	Message	Build Table Reference
?update_SPA(type)?		?update_TA(type)?	
?change_SPA(params)?		?change_TA(params)?	
?refresh_SPA(type)?		?refresh_TA(type)?	
?start?		?start?	

PARTITION TABLE - Top-Level Partition PARTITION TABLE - Top-Level Partition

Partition: ?KEY?
Parent: \$ii\$

Partition: ?SOFT?
Parent: \$iii\$

Message	Build Table Reference	Message	Build Table Reference
?keypressed(value)?		?soft_enter?	
?change_SPA(params)?		?soft_change(button)?	
?refresh_SPA(type)?		?change_SPA(params)?	
?change_TA(params)?		?refresh_SPA(type)?	
?refresh_TA(type)?		?change_TA(params)?	
?soft_enter?		?refresh_TA(type)?	
?anc_move(direction)?		?ok?	
?ok?		?err?	
?err?		?reset?	
?display(command)?		?display(command)?	
?start?		?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?SB?**Parent:** \$ii\$

Message	Build Table Reference
?display(command)?	
?soft_change(button)?	
?reset?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?ANC?**Parent:** \$ii\$

Message	Build Table Reference
?display(command)?	
?anc_move(direction)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?ERR?**Parent:** \$ii\$

Message	Build Table Reference
?ok?	
?err?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?CA?**Parent:** \$schInfo\$

Message	Build Table Reference
?chc_pos(x,y)?	
?display(command)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?CHC?**Parent:** \$oih\$

Message	Build Table Reference
?trk_pos(x,y)?	
?display(command)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?TRK?**Parent:** \$oih\$

Message	Build Table Reference
?trk_int(x,y)?	
?hc_pos(x,y)?	
?trk_pos(x,y)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?JC?**Parent:** \$oih\$

Message	Build Table Reference
?jstc_pos(x,y)?	
?display(command)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?JSTK?**Parent:** \$oih\$

Message	Build Table Reference
jstc_int(x,y)?	
?jstc_pos(x,y)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** 'KBD?**Parent:** \$oih\$

Message	Build Table Reference
?kbd_int(char)?	
?keypress_d(value)?	
?start?	

PARTITION TABLE - Top-Level Partition**Partition:** ?DSPLY?**Parent:** \$display\$

Message	Build Table Reference
?display(command)?	
?display_command(code)?	
?start?	

D.2 Final EBDs (if changed)

TA?get_update_ta(why,what)
 [queue not empty and dequeue head(why,what) =true]//

TA.Buffer EBD

Final EVT EBD

Notes:

1. change and update are mutually exclusive and cover all possible values.

Final TA EBD

Notes: Operator Request =

[spa change]//(SPA_Buffer-update_spa(change,params),ERR-ok)

OR

[ta change]//(TA_Buffer-update_ta(change,params),ERR-ok)

OR

[spectral data change]//(SPA_Buffer-update_spa(refresh,type),
TA_Buffer-update_ta(refresh,type),ERR-ok)

OR

[other chnage]//(TA_Buffer-update_ta(refresh,type), ERR-ok)

OR

[local echo]//(DSPLY-display(command), ERR-ok)

OR

[Local error]//(DSPLY-display(command), ERR-err)

OR

[joystick button]//(SOFT-soft_enter, ERR-ok)

OR

[anc movement]//(ANC-ANC_move(direction, ERR-ok)

Final KEY EBD

Notes: Not_Menu_Reset =

[not menu reset AND spa change]//(SPA_Buffer-update_spa(change,params),ERR-ok)
OR

[not menu reset AND ta change]//(TA_Buffer-update_ta(change,params),ERR-ok)
OR

[not menu reset AND spectral data change]//(SPA_Buffer-update_spa(refresh,type),
TA_Buffer-update_ta(refresh,type),ERR-ok)

OR

[not menu reset AND other change]//(TA_Buffer-update_ta(refresh,type), ERR-ok)
OR

[not menu reset AND local echo]//(DSPLY-display(command), ERR-ok)

OR

[not menu reset AND Local error]//(DSPLY-display(command), ERR-err)

Final SOFT EBD

SPA?get_update_spa(why,what)

[queue not empty and dequeue head(why,what)=true]//

SPA_Buffer EBD

5 of/de 5

PM 13 . x4 PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

Notes:

1. Get_Update = //SPA_buffer!get_update_SPA(why,whi;
2. track update, spectral update, and change request are mutually exclusive, and cover all possibilities.

Final SPA EBD

CA?get_chc_pos(x,y)
 [queue not empty and dequeue head(x,y)=true]//

CHC_Pos_Buffer EBD

Final TRK EBD

Final CA EBD

D.3 Final Actor Tables (if changed)

ACTOR TABLE

Actor: \$spa\$ (Spectral Display Area)

EBDs: *spa*

ACMs: \$spa\$

Basic Model: Design Fragment of Figure 4.11 is used.

Top-Level Partitions	Comments
?SPA?	Operations require lengthy processing.
Layered Partitions	Comments
SPA:?START SPA?	set default screen
SPA:?GENERATE SPA?	includes functionality of ?UPDATE SPECTRAL DATA? and ?UPDATE TRACKING MARKERS? below.
SPA:?UPDATE SPECTRAL DATA?	
SPA:?UPDATE TRACKING MARKERS?	
SPA:?DETERMINE UPDATE TYPE?	
Comments	
The operations provide data (e.g. ranges and tracking marker positions and details) used by the CA entity.	

ACTOR TABLE**Actor:** \$ta\$ (Textual Area Display)**EBDs:** ta **ACMs:** \$ta\$**Basic Model:** Design Fragment of Figure 4.11 is used.

Top-Level Partitions	Comments
?TA?	Operations require lengthy processing.
Layered Partitions	Comments
TA: ?START TA? TA: ?GENERATE TA? TA: ?UPDATE TEXT? TA: ?DETERMINE UPDATE TYPE?	set default screen includes functionality of ?UPDATE TEXT? below.
Comments	
The operations provide data (e.g. values of editable parameters, and locations accessible by the alpha-numeric cursor) used by the KBD and KEY, and ANC entities respectively.	

ACTOR TABLE**Actor:** \$chc_info\$ (CHC Information)**EBDs:** ca **ACMs:** \$ca\$**Basic Model:** Build Table Reference 15.

Top-Level Partitions	Comments
?CA?	
Layered Partitions	Comments
CA: ?START CA? CA: ?DETERMINE OVERLAP?	This functions uses data defined by the SPA entity, and includes the event <i>DSPLY-display(command)</i> .
Comments	

ACTOR TABLE**Actor:** \$ta.buffer\$**EBDs:** TA_BUFFER**ACMs:** \$ta.buffer\$**Basic Model:** Build Table Reference 2.

Top-Level Partitions	Comments
?TA_BUFFER?	
Layered Partitions	Comments
?QUEUE EMPTY? ?QUEUE NOT EMPTY? ?PRIORITY OVERWRITE QUEUE(WHY,WHAT)? ?DEQUEUE HEAD(WHY,WHAT)?	
Comments	

ACTOR TABLE**Actor:** \$spa.buffer\$**EBDs:** SPA_BUFFER**ACMs:** \$spa.buffer\$**Basic Model:** Build Table Reference 7.

Top-Level Partitions	Comments
?SPA_BUFFER?	
Layered Partitions	Comments
?QUEUE EMPTY? ?QUEUE NOT EMPTY? ?PRIORITY OVERWRITE QUEUE(WHY,WHAT)? ?DEQUEUE HEAD(WHY,WHAT)?	
Comments	

ACTOR TABLE

Actor: \$sch_pos.buffer\$

EBDs: *CHC_POS_BUFFER* **ACMs:** \$sch_pos.buffer\$

Basic Model: Build Table Reference 12.

Top-Level Partitions	Comments
?CHC.POS.BUFFER?	
Layered Partitions	Comments
?QUEUE EMPTY?	
?QUEUE NOT EMPTY?	
?OVERWRITE QUEUE(X,Y)?	
?DEQUEUE HEAD(X,Y)?	
Comments	

D.4 Final Partition Tables

PARTITION TABLE - Top-Level Partition

Partition: ?EVT?

Parent: \$sch\$

Message	Build Table Reference
?update_sta(type)?	Build Table Reference 1
?update_la(refresh,type)?	Build Table Reference 3
?update_spa(refresh,type)?	Build Table Reference 8
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition

Partition: ?STA?

Parent: \$sch\$

Message	Build Table Reference
?update_sta(type)?	Build Table Reference 1
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition: ?SPA?****Parent: \$spa\$**

Message	Build Table Reference
?get_update_spa(why,what)?	Build Table Reference 11
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition: ?TA?****Parent: \$ta\$**

Message	Build Table Reference
?get_update_ta(why,what)?	Build Table Reference 6
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition: ?KEY?****Parent: \$ii\$**

Message	Build Table Reference
?keypressed(value)?	Build Table Reference 23
?update_spa(change,params)?	Build Table Reference 9
?update_spa(refresh,type)?	Build Table Reference 9
?update_ta(change,params)?	Build Table Reference 4
?update_ta(refresh,type)?	Build Table Reference 4
?soft_enter?	Build Table Reference 16
?anc_move(direction)?	Build Table Reference 16
?ok?	Build Table Reference 16
?err?	Build Table Reference 16
?display(command)?	Build Table Reference 17
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition: ?SOFT?****Parent: \$ii\$**

Message	Build Table Reference
?soft_enter?	Build Table Reference 16
?soft_change(button)?	Build Table Reference 16
?update_spa(change,params)?	Build Table Reference 10
?update_spa(refresh,type)?	Build Table Reference 10
?update_ta(change,params)?	Build Table Reference 5
?update_ta(refresh,type)?	Build Table Reference 5
?ok?	Build Table Reference 16
?err?	Build Table Reference 16
?reset?	Build Table Reference 16
?display(command)?	Build Table Reference 17
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition: ?SB?****Parent: \$ii\$**

Message	Build Table Reference
?display(command)?	Build Table Reference 17
?soft_change(button)?	Build Table Reference 16
?reset?	Build Table Reference 16
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?ANC?**Parent:** \$ii\$

Message	Build Table Reference
?display(command)?	Build Table Reference 17
?anc_move(direction)?	Build Table Reference 16
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?ERR?**Parent:** \$ii\$

Message	Build Table Reference
?ok?	Build Table Reference 16
?err?	Build Table Reference 16
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?CA?**Parent:** \$chc.info\$

Message	Build Table Reference
?get_chc_pos(x,y)?	Build Table Reference 14
?display(command)?	Build Table Reference 18 (in ?DETERMINE OVERLAP?)
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?CHC?**Parent:** \$oih\$

Message	Build Table Reference
?trk_pos(x,y)?	Build Table Reference 22 (history[20])
?display(command)?	Build Table Reference 19
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?TRK?**Parent:** \$oih\$

Message	Build Table Reference
?trk_int(x,y)?	Build Table Reference 23
?chc_pos(x,y)?	Build Table Reference 13
?trk_pos(x,y)?	Build Table Reference 22 (history[20])
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?JC?**Parent:** \$oih\$

Message	Build Table Reference
?jstl_pos(x,y)?	Build Table Reference 22 (history[20])
?display(command)?	Build Table Reference 19
?start?	Build Table Reference 21

PARTITION TABLE Top-Level Partition**Partition:** ?JSTK?**Parent:** \$oih\$

Message	Build Table Reference
?jstk_int(x,y)?	Build Table Reference 23
?jstk_pos(x,y)?	Build Table Reference 22 (history[20])
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?KBD?**Parent:** \$oih\$

Message	Build Table Reference
?kbd_int(char)?	Build Table Reference 23
?keypressed(value)?	Build Table Reference 22
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?DSPLY?**Parent:** \$display\$

Message	Build Table Reference
?display(command)?	Build Table Reference 17,18,19
?display_command(code)?	Build Table Reference 24
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?TA.BUFFER?**Parent:** \$ta_buffer\$

Message	Build Table Reference
?update.ta(why,what)?	Build Table Reference 3,4,5
?get.update.ta(why,what)?	Build Table Reference 6
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?SPA.BUFFER?**Parent:** \$spa_buffer\$

Message	Build Table Reference
?update.spa(why,what)?	Build Table Reference 8,9,10
?get.update.i.spa(why,what)?	Build Table Reference 11
?start?	Build Table Reference 21

PARTITION TABLE - Top-Level Partition**Partition:** ?CHC.POS.BUFFER?**Parent:** \$chc_pos_buffer\$

Message	Build Table Reference
?chc_pos(x,y)?	Build Table Reference 13
?get.chc_pos(x,y)?	Build Table Reference 14
?start?	Build Table Reference 21

D.5 Build Tables

BUILD TABLE REFERENCE: 1 of Display Subsystem

Build Table for: DF-Inter-SEQ-1 (Static Schedule)

Generic Specification Segment	
any co-allocated behaviour	
Application Bindings	
actor	<i>SEH</i>
list co-allocated EBDs	<i>EVT, STA</i>
behaviour engine	<i>?sch_engine?</i>
list events received from external EBDs	<i>>start</i>
list events sent to external EBDs	<i>TA.Buffer-update_ta(type),</i> <i>SPA.Buffer-update_spa(type)</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. Unless the entire agenda of this actor depends on <i>>start</i> to initiate its processing (or processing loop), then <i>start</i> must be sent eventually when this actor is blocked on the corresponding receive placeholder.	confirmed
Design Contingencies	
Contingency	Proof
1. <i>?sch_engine?</i> must define a fair static schedule for the combined behaviour of <i>EVT</i> and <i>STA</i> .	confirmed (<i>?sch_engine?</i>)
2. <i>>start</i> requires a fair receive_placeholder	confirmed
3. <i>TA.Buffer-update_ta(type)</i> requires a fair send_placeholder	confirmed
4. <i>SPA.Buffer-update_spa(type)</i> requires a fair send_placeholder	confirmed
Performance Issues	
Issue	Comment
1. Redundant MCL can be removed	done.
2. <i>?sch_engine?</i> defines priorities and preferences of events and entities in S	OK

?sch_engine?

The two behaviours (STA and EVT) are combined as follows:

?START STA? - action_placeholder for *start(Sta : STA)* operation

?START EVT? - action_placeholder for *start(Sta : EVT)* operation

receive_placeholder for *>start*

?GENERATE STA? - action_placeholder

loop

 ?GET SYSTEM EVENTS? - action_placeholder

logic and send_placeholders for TA_Buffer!update.ta(refresh,type)
 and SPA_Buffer!update.spa(refresh,type) (were ta-ta.update(type) and
 spa-spa.update(type)).

 - action_ and interaction_placeholder(s) for *sta_update(type)*

 ?UPDATE STA? - action_placeholder

end loop

BUILD TABLE REFERENCE: 2 of Display Subsystem**Build Table for: DF-Overwrite-Priority-Buffer-1**

Generic Specification Segment	
{QUEUE EMPTY;A;B} A= put_txs+put_msg(formal_param1); PRIORITY OVERWRITE QUEUE(formal_param1) B= { {A;B}* {get_txs?get_msg(formal_param2){QUEUE NOT EMPTY $\cap$ DEQUEUE HEAD(formal_param2)};{([QUEUE EMPTY]; A;B) ([QUEUE NOT EMPTY]; B)}}}	
Application Bindings	
actor	<i>\$TA_Buffer\$</i>
EBD	<i>TA_Buffer</i>
put_txs	<i>EVT, KEY, SOFT</i>
get_txs	<i>TA</i>
put_msg	<i>update.ta</i>
formal_param1	<i>(why,what)</i>
actual_param1	<i>[(refresh,type) (change,params)]</i>
priority param	<i>(change,params)</i>
get_msg	<i>get_update.ta</i>
formal_param2	<i>(why,what)</i>
actual_param2	<i>[(refresh,type) (change,params)]</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>(why,what)</i> and <i>[(refresh,type) (change,params)]</i> must be type compatible	confirmed
2. <i>(why,what)</i> and <i>[(refresh,type) (change,params)]</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>[(refresh,type) (change,params)]</i> is copied into <i>TA_Buffer</i> for every <i>update.ta</i> .	ok
2. <i>[(refresh,type) (change,params)]</i> is copied from <i>TA_Buffer</i> for every <i>get_update.ta</i> .	ok
3. Because of overwrite, data is lost.	ok

BUILD TABLE REFERENCE: 3 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	EVT
rx	TA_Buffer
msg	update.ta
actual_param	(refresh.type)
formal_param	(why,what)
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>EVT+update.ta(what,why)</i> must be received eventually by <i>TA_Buffer</i>	<i>update.ta</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>(refresh.type)</i> and <i>(why,what)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>EVT</i> blocks until <i>TA_Buffer</i> acknowledges that <i>update.ta</i> has been received.	ok
2. <i>EVT</i> is delayed by <i>TA_Buffer</i> .	ok

BUILD TABLE REFERENCE: 4 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	KEY
rx	TA.Buffer
msg	update.ta
actual_param	{(refresh,type) (change,params)}
formal_param	(why,what)
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. KEY+update.ta(what,why) must be received eventually by TA.Buffer	update.ta is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. [(refresh,type) (change,params)] and (why,what) must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. KEY blocks until TA.Buffer acknowledges that update.ta has been received.	ok
2. KEY is delayed by TA.Buffer.	ok

BUILD TABLE REFERENCE: 5 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx+msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>SOFT</i>
rx	<i>TA_Buffer</i>
msg	<i>update_ta</i>
actual_param	<i>[(refresh,type) (change,params)]</i>
formal_param	<i>(why,what)</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>SOFT+update_ta(what,why)</i> must be received eventually by <i>TA_Buffer</i>	<i>update_ta</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>[(refresh,type) (change,params)]</i> and <i>(why,what)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>SOFT</i> blocks until <i>TA_Buffer</i> acknowledges that <i>update_ta</i> has been received.	ok
2. <i>SOFT</i> is delayed by <i>TA_Buffer</i> .	ok

BUILD TABLE REFERENCE: 6 of Display Subsystem**Build Table for: DF-SyncInter-IPC-3**

Generic Specification Segment	
tx: rx!msg(actual_param)	
rx: tx?msg(formal_param)	
Application Bindings	
tx	TA
rx	TA.Buffer
msg	get_update_ta
actual_param	(why, what)
formal_param	(why, what)
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. (what, why) and (why, what) must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. TA blocks until TA.Buffer acknowledges that get_update_ta has been received.	ok
2. TA is delayed by TA.Buffer.	ok

BUILD TABLE REFERENCE: 7 of Display Subsystem**Build Table for: DF-Overwrite-Priority-Buffer-1**

Generic Specification Segment	
{QUEUE EMPTY;A;B} A= put_txs+put_msg(formal_param1); PRIORITY OVERWRITE QUEUE(formal_param1) B= { {A;B}* {get_txs?get_msg(formal_param2){QUEUE NOT EMPTY $\cap$ DEQUEUE HEAD(formal_param2)};{([QUEUE EMPTY]; A;B) ([QUEUE NOT EMPTY]; B)}}}	
Application Bindings	
actor	<i>\$SPA_Buffer\$</i>
EBD	<i>SPA_Buffer</i>
put_txs	<i>EVT, KEY, SOFT</i>
get_txs	<i>SPA</i>
put_msg	<i>update_spa</i>
formal_param1	<i>(why,what)</i>
actual_param1	<i>[(refresh,type) (change,params)]</i>
priority param	<i>(change,params)</i>
get_msg	<i>get_update_spa</i>
formal_param2	<i>(why,what)</i>
actual_param2	<i>[(refresh,type) (change,params)]</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>(why,what)</i> and <i>[(refresh,type) (change,params)]</i> must be type compatible	confirmed
2. <i>(why,what)</i> and <i>[(refresh,type) (change,params)]</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>[(refresh,type) (change,params)]</i> is copied into <i>SPA_Buffer</i> for every <i>update_spa</i> .	ok
2. <i>[(refresh,type) (change,params)]</i> is copied from <i>SPA_Buffer</i> for every <i>get_update_spa</i> .	ok
3. Because of overwrite, data is lost.	ok

BUILD TABLE REFERENCE: 8 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	EVT
rx	SPA_Buffer
msg	update_spa
actual_param	(refresh.type)
formal_param	(why.what)
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>EVT+update_spa(what,why)</i> must be received eventually by <i>SPA_Buffer</i>	<i>update_spa</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>(refresh,type)</i> and <i>(why,what)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>EVT</i> blocks until <i>SPA_Buffer</i> acknowledges that <i>update_spa</i> has been received.	ok
2. <i>EVT</i> is delayed by <i>SPA_Buffer</i> .	ok

BUILD TABLE REFERENCE: 9 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>KEY</i>
rx	<i>SPA_Buffer</i>
msg	<i>update_spa</i>
actual_param	<i>{(refresh,type) (change.params)}</i>
formal_param	<i>(why,what)</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>KEY+update_spa(what,why)</i> must be received eventually by <i>SPA_Buffer</i>	<i>update_spa</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>{(refresh,type) (change.params)}</i> and <i>(why,what)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>KEY</i> blocks until <i>SPA_Buffer</i> acknowledges that <i>update_spa</i> has been received.	ok
2. <i>KEY</i> is delayed by <i>SPA_Buffer</i> .	ok

BUILD TABLE REFERENCE: 10 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>SOFT</i>
rx	<i>SPA_Buffer</i>
msg	<i>update_spa</i>
actual_param	<i>[(refresh,type) (change,params)]</i>
formal_param	<i>(why,what)</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>SOFT+update_spa(what,why)</i> must be received eventually by <i>SPA_Buffer</i>	<i>update_spa</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>[(refresh,type) (change,params)]</i> and <i>(why,what)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>SOFT</i> blocks until <i>SPA_Buffer</i> acknowledges that <i>update_spa</i> has been received.	ok
2. <i>SOFT</i> is delayed by <i>SPA_Buffer</i> .	ok

BUILD TABLE REFERENCE: 11 of Display Subsystem**Build Table for: DF-SyncInter-IPC-3**

Generic Specification Segment	
tx: rx!msg(actual_param)	
rx: tx?msg(formal_param)	
Application Bindings	
tx	SPA
rx	SPA_Buffer
msg	get_update_spa
actual_param	(why, what)
formal_param	(why, what)
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. (what, why) and (why, what) must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. SPA blocks until SPA_Buffer acknowledges that get_update_spa has been received.	ok
2. SPA is delayed by SPA_Buffer.	ok

BUILD TABLE REFERENCE: 12 of Display Subsystem
Build Table for: DF-Overwrite-Buffer-1

Generic Specification Segment	
{QUEUE EMPTY:A:B} A= put_txs?put_msg(formal_param1); OVERWRITE QUEUE(formal_param1) B= { {A:B}* {get_txs?get_msg(formal_param2)[QUEUE NOT EMPTY () DEQUEUE HEAD(formal_param2)];{([QUEUE EMPTY]: A:B) ([QUEUE NOT EMPTY]: B)}}}	
Application Bindings	
actor	<i>\$CHC_Pos_Buffer\$</i>
EBD	<i>CHC_Pos_Buffer</i>
put_txs	<i>TRK = (11)</i>
get_txs	<i>CA = (chc_info)</i>
put_msg	<i>chc_pos</i>
formal_param1	<i>(x,y)</i>
actual_param1	<i>(x,y)</i>
get_msg	<i>get_chc_pos</i>
formal_param2	<i>(x,y)</i>
actual_param2	<i>(x,y)</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>(x,y)</i> and <i>(x,y)</i> must be type compatible	confirmed
2. <i>(x,y)</i> and <i>(x,y)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>(x,y)</i> is copied into <i>CHC_Pos_Buffer</i> for every <i>chc_pos</i> .	ok
2. <i>(x,y)</i> is copied from <i>CHC_Pos_Buffer</i> for every <i>get_chc_pos</i> .	ok
3. Because of overwrite, data is lost.	ok

BUILD TABLE REFERENCE: 13 of Display Subsystem**Build Table for: DF-SyncInter-IPC-3**

Generic Specification Segment	
tx: rx!msg(actual_param)	
rx: tx?msg(formal_param)	
Application Bindings	
tx	<i>TRK (=II)</i>
rx	<i>CHC_Pos_Buffer</i>
msg	<i>chc_pos</i>
actual_param	<i>(x,y)</i>
formal_param	<i>(x,y)</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>(x,y)</i> and <i>(x,y)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>TRK (=II)</i> blocks until <i>CHC_Pos_Buffer</i> acknowledges that <i>chc_pos</i> has been received.	ok
2. <i>TRK (=II)</i> is delayed by <i>CHC_Pos_Buffer</i> .	ok

BUILD TABLE REFERENCE: 14 of Display Subsystem**Build Table for: DF-SyncInter-IPC-3**

Generic Specification Segment	
tx: rx!msg(actual_param)	
rx: tx?msg(formal_param)	
Application Bindings	
tx	CA (= chc_info)
rx	CHC_Pos_Buffer
msg	get_chc_pos
actual_param	(x,y)
formal_param	(x,y)
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. (x,y) and (x,y) must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. CA (= chc_info) blocks until CHC_Pos_Buffer acknowledges that get_chc_pos has been received.	ok
2. CA (= chc_info) is delayed by CHC_Pos_Buffer.	ok

BUILD TABLE REFERENCE: 15 of Display Subsystem**Build Table for: DF-Periodic-Loop-2**

Generic Specification Segment	
>init; { source!initiating_event(formal.param). {EVALUATE; other events (all outgoing); wait timeout period; }*	
Application Bindings	
actor	<i>\$chc.info\$</i>
EBD	<i>CA</i>
source	<i>CHC.Pos.Buffer</i>
initiating event	<i>get_chc_pos</i>
formal param	<i>(x,y)</i>
actual param	<i>(x,y)</i>
timeout period	<i>to be evaluated</i>
other events (all outgoing)	<i>DSPLY-display(command)</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>(x,y)</i> and <i>(x,y)</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. It must be sufficient to evaluate the message <i>get_chc_pos(x,y)</i> at a period <i>to be evaluated</i> .	to be evaluated
2. It must be sufficient to send <i>DSPLY-display(command)</i> only after the <i>ca!get_chc_pos(x,y)</i> interaction is complete; i.e., at a period <i>to be evaluated</i> .	to be evaluated.

BUILD TABLE REFERENCE: 16 of Display Subsystem (page 1 of 2)**Build Table for: DF-Inter-SEQ-1** (Static Schedule)

Generic Specification Segment	
any co-allocated behaviour	
Application Bindings	
actor list co-allocated EBDs behaviour engine list events received from external EBDs list events sent to external EBDs MachineChart Design Fragment	<i>//</i> <i>KEY, SOFT, SB, ANC, ERR</i> <i>?u_engine?</i> <i>>start, KBD+keypressed(value)</i> <i>DSPLY-display(command),</i> <i>SPA.Buffer-</i> <i>update_spa(change, params),</i> <i>TA.Buffer-</i> <i>update_ta(change, params),</i> <i>SPA.Buffer-</i> <i>update_spa(refresh, type),</i> <i>TA.Buffer-</i> <i>update_ta(refresh, type)</i>
Specification Contingencies	
Contingency	Proof
1. <i>>start</i> must not block the progression of the combined behaviour.	confirmed
2. <i>KBD+keypressed(value)</i> must not block the progression of the combined behaviour.	confirmed

BUILD TABLE REFERENCE: 16 of Display Subsystem (page 2 of 2)**Build Table for: DF-Inter-SEQ-1 (Static Schedule)**

Design Contingencies	
Contingency	Proof
1. <i>?u_engine?</i> must define a fair static schedule for the combined behaviour of <i>KEY</i> , <i>SOFT</i> , <i>SB</i> , <i>ANC</i> , and <i>ERR</i> .	confirmed (<i>?ii_engine?</i>)
2. <i>>start</i> requires a fair receive_placeholder	confirmed
3. <i>DSPLY-display(command)</i> requires a fair send_placeholder	confirmed
4. <i>SPA_Buffer-update_spa(change.params)</i> requires a fair send_placeholder	confirmed
5. <i>SPA_Buffer-update_spa(refresh.type)</i> requires a fair send_placeholder	confirmed
6. <i>TA_Buffer-update_spa(change.params)</i> requires a fair // send_placeholder	confirmed
7. <i>TA_Buffer-update_spa(refresh.type)</i> requires a fair // send_placeholder	confirmed
Performance Issues	
Issue	Comment
1. Redundant MCL can be removed	done.
2. <i>?u_engine?</i> defines priorities and preferences of events and entities in <i>II</i>	OK

?u_engine?

The five behaviours (KEY, SOFT, SB, ANC, and ERR) are combined as follows.

service: the initialization operations includes *receive_placeholder* for the *start* message, and *action_placeholders* for the operation states.

loop

receive_placeholder for a keypressed(value) message from KBD

?EVALUATE KEYSTROKE? - *action_placeholder*

logic for correct management of input; e.g.,

case *keystroke* of

when *soft button enter* => *using joystick position shared with OIH*

?EVALUATE BUTTON EQUIVALENT? - *interaction_placeholder* for
KEY and SOFT: *soft_enter*

if *soft button menu change is required* then

?RESET SOFT BUTTON DISPLAY? - *interaction_placeholder* for
SOFT and SB: *soft_change* and *reset*

send_placeholder for DSPLY-display(command)

else if *display change* then

send_placeholder for TA_Buffer!update_ta(change,params) or
SPA_Buffer!update_spa(change,params)

else if *function* then

(using data shared with ta and/or spa actors as needed)

update system data and issue system events

send_placeholder for - *interaction_placeholder*

for KEY or SOFT and ERR: *ok*

send_placeholder for TA_Buffer!update_ta(refresh,type) and
SPA_Buffer!update_spa(refresh,type,)

OR *send_placeholder DSPLY-display(command)*

interaction_placeholder for KEY or SOFT and ERR: err

when *ANC arrow* => **?DETERMINE AND SAVE ANC POSITION?**

- *interaction_placeholder for ANC and KEY: anc_move*

send_placeholder for DSPLY-display(command)

when *hard button or keyboard input* => **?EVALUATE BUTTON EQUIVALENT?**

- *action_placeholder*

similar processing to soft button case adding functionality of edits

end case

end loop

BUILD TABLE REFERENCE: 17 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	KEY, SOFT ANC, or SB (= II)
rx	DSPLY
msg	display
actual_param	command
formal_param	command
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. II+display(command) must be received eventually by DSDPLY	?Display-Queue?
Design Contingencies	
Contingency	Proof
1. command and command must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. KEY, ANC, SB, or ERR (= II)) blocks until DSDPLY acknowledges that display has been received.	ok
2. KEY, ANC, SB, or ERR (= II)) is delayed by DSDPLY.	ok
?Display-Queue?	
II+display(command) must be received eventually by DSDPLY. -> DSDPLY receives display in all states after initialization, -> therefore it is always part of the satisfiable set, and thus received, -> eventually.	
<div>Proof prove</div>	

BUILD TABLE REFERENCE: 18 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5 (Rendezvous with Message)**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	CA
rx	DSPLY
msg	display
actual_param	command
formal_param	command
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. CA+display(command) must be received eventually by DSPLY	?Display_Queue?
Design Contingencies	
Contingency	Proof
1. command and command must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. CA blocks until DSPLY acknowledges that display has been received.	ok
2. CA is delayed by DSPLY.	ok

BUILD TABLE REFERENCE: 19 of Display Subsystem**Build Table for: DF-AsyncInter-IPC-5**

Generic Specification Segment	
tx: rx-msg(actual_param)	
rx: tx+msg(formal_param)	
Application Bindings	
tx	<i>CHC, JC (= OIH)</i>
rx	<i>DSPLY</i>
msg	<i>display</i>
actual_param	<i>command</i>
formal_param	<i>command</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>OIH+display(command)</i> must be received eventually by <i>DSPLY</i>	?Display.Queue?
Design Contingencies	
Contingency	Proof
1. <i>command</i> and <i>command</i> must be type compatible	confirmed
Performance Issues	
Issue	Comment
1. <i>CHC, JC (= OIH)</i> blocks until <i>DSPLY</i> acknowledges that <i>display</i> has been received.	ok
2. <i>CHC, JC (= OIH)</i> is delayed by <i>DSPLY</i> .	ok

BUILD TABLE REFERENCE: 20 of Display Subsystem (page 1 of 2)**Build Table for: DF-Inter-SEQ-1 (Static Schedule)**

Generic Specification Segment	
any co-allocated behaviour	
Application Bindings	
actor	<i>OIH</i>
list co-allocated EBDs	<i>CHC, TRK, JC, JSTK, KBD</i>
behaviour engine	<i>?oth_engine?</i>
list events received from external EBDs	<i>>start, external+kbd_int(char), external+trk_int(x,y), external+jstk_int(x,y)</i>
list event sent to external EBDs	<i>CHC_Pos_Buffer!chc_pos(x,y), DSPLY-display(command), KEY-keypressed(value)</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>>start</i> must not block the progression of the combined behaviour.	confirmed
2. <i>kbd_int(char)</i> must not block the progression of the combined behaviour.	confirmed by OR synchronization of three initiating external events (<i>kbd_int</i> , <i>trk_int</i> , and <i>jstk_int</i>).
3. <i>trk_int(x,y)</i> must not block the progression of the combined behaviour.	confirmed by OR synchronization of three initiating external events (<i>kbd_int</i> , <i>trk_int</i> , and <i>jstk_int</i>).
4. <i>jstk_int(x,y)</i> must not block the progression of the combined behaviour.	confirmed by OR synchronization of three initiating external events (<i>kbd_int</i> , <i>trk_int</i> , and <i>jstk_int</i>).

BUILD TABLE REFERENCE: 20 of Display Subsystem (page 2 of 2)**Build Table for: DF-Inter-SEQ-1 (Static Schedule)**

Design Contingencies	
Contingency	Proof
1. <i>?oish_engine?</i> must define a fair static schedule for the combined behaviour of <i>CHC</i> , <i>TRK</i> , <i>JC</i> , <i>JSTK</i> , and <i>KBD</i> .	confirmed (<i>?oish_engine?</i>)
Specification Contingencies	
Contingency	Proof
4. <i>!!chc_pos(x,y)</i> must be received eventually by <i>CHC.Pos.Buffer</i>	<i>chc_pos</i> is always an element of the satisfiable event set, therefore it is received infinitely often.
Design Contingencies	
Contingency	Proof
1. <i>?oish_engine?</i> must define a fair static schedule for the combined behaviour of <i>CHC</i> , <i>TRK</i> , <i>JC</i> , <i>JSTK</i> , and <i>KBD</i> .	confirmed (<i>?oish_engine?</i>)
2. <i>>start</i> requires a fair receive_placeholder	confirmed
3. <i>>external+kbd_int(char)</i> requires a fair receive_placeholder	confirmed
4. <i>>external+trk_int(x,y)</i> requires a fair receive_placeholder	confirmed
5. <i>>external+jstk_int(x,y)</i> requires a fair receive_placeholder	confirmed
6. <i>CHC.Pos.Buffer!chc_pos(x,y)</i> requires a fair send_placeholder	confirmed
7. <i>DSPLY-display(command)</i> requires a fair send_placeholder	confirmed
Performance Issues	
Issue	Comment
1. Redundant MCL can be removed	done.
2. <i>?oish_engine?</i> defines priorities and preferences of events and entities in <i>!!</i>	OK

?oth_engine?

The five behaviours (CHC, TRK, JC, JSTK, and KBD) are combined as follows.

serialize the initialization operations - includes *receive_placeholder* for the *start* message, and *action_placeholders* for the operation states.

loop

receive_placeholder for OR synchronization of external messages:

kbd_int(char), trk_int(x,y), or jstk_int(x,y)

logic for correct management of input; e.g.,

case interrupt type of

when keyboard input => *send_placeholder* for *KEY-keypressed(value)*

when trackball interrupt => *send_placeholder* for *DSPLY-display(command)*
from CHC EBD

send_placeholder for *CHC_Pos_Buffer!chc_pos(x,y)*

when joystick interrupt => ?SAVE JSTK POS? for SOFT entity - *action_placeholder*
send_placeholder for *DSPLY-display(command)* from JC EBD

end case

end loop

BUILD TABLE REFERENCE: 21 of Display Subsystem**Build Table for: DF-TransBInter-IPC-2 (Rendezvous)**

Generic Specification Segment	
tx: msg<	
rx: >msg	
Application Bindings	
tx	system
rxs	every entity in display subsystem
msg	start
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>start</i> must be received eventually by <i>every entity in display subsystem</i> .	confirmed.
Design Contingencies	
Contingency	Proof
1. <i>system</i> must be able to individually invoke <i>every entity in display subsystem</i> .	confirmed (done in particular order)

BUILD TABLE REFERENCE: 22 of Display Subsystem (page 1 of 3)**Build Table for: DF-CASE-SEQ+Couriers**

Generic Specification Segment	
collection of servers of the form:	
servers: (wait; init.tx+stimulus; deterministic service; final.rx-response)*	
Application Bindings	
<p>actor</p> <p>list co-allocated EBDs</p> <p>There are several possible combinations for managing the stimulus and response. Only applicable combinations are shown. Servers with receive_placeholders for stimulus, and either (1) send_placeholder for response or (2) response which triggers other co-allocated service(s): server:(wait; init.tx+stimulus; deterministic service; final.rx-response)</p> <p>Services stimulated internally with send_placeholder for response: server:(wait; init.tx+stimulus; deterministic service; final.rx-response)</p> <p>Services with receive_placeholder for stimulus and courier for response: server:(wait; init.tx+stimulus; deterministic service; final.rx-response)</p> <p>Server: Courier Name for <i>keypressed(value)</i></p> <p>MachineChart Design Fragment</p>	<p><i>OIH</i></p> <p><i>CHC, TRK, JC, JSTK, KBD</i></p> <p><i>TRK:(Wait Trackball Interrupt; external+trk.int(x,y); Null; (CHC.Pos.Buffer!chc_pos(x,y),chc_trk_pos(x,y)))</i></p> <p><i>JSTK:(Wait Joystick Interrupt; external+jstk.int(x,y); Null; JC-jstk_pos(x,y))</i></p> <p><i>CHC:(Wait Move CHC; TRK+trk_pos(x,y); A:assemble Display Command; DSPLY-display(command))</i></p> <p><i>JC:(Wait JSTK Move; JSTK+jstk_pos(x,y); Equate Button Position; DSPLY-display(command))</i></p> <p><i>KBD:(Wait Keyboard Interrupt; external+kbd.int(char); Null; KEY-keypressed(value))</i></p> <p><i>KBD: key_courier</i></p>
Specification Contingencies	
Contingency	Proof
1. <i>OIH?chc_pos(x,y)</i> must be received eventually by <i>CHC.Pos.Buffer</i>	<i>chc_pos</i> is always an element of the satisfiable event set, therefore it is received infinitely often.

BUILD TABLE REFERENCE: 22 of Display Subsystem (page 2 of 3)**Build Table for: DF-CASE-SEQ+Couriers**

Design Contingencies	
Contingency	Proof
1. <i>OIH+keypressed(value)</i> must be designed as a synchronous receive in <i>KEY</i>	match
2. <i>value</i> must be type compatible with the formal parameter defined by <i>key_courier</i>	
3. <i>TRK-trk_int(x,y)</i> must be designed as a synchronous send of <i>external</i>	match
4. <i>(x,y)</i> must be type compatible with the actual parameter defined by <i>external</i>	
5. <i>JSTK-jstk_int(x,y)</i> must be designed as a synchronous send of <i>external</i>	match
6. <i>(x,y)</i> must be type compatible with the actual parameter defined by <i>external</i>	
7. <i>KBD-kbd_int(x,y)</i> must be designed as a synchronous send of <i>external</i>	match
8. <i>char</i> must be type compatible with the actual parameter defined by <i>external</i>	

BUILD TABLE REFERENCE: 22 of Display Subsystem (page 3 of 3)**Build Table for: DF-CASE-SEQ+Couriers**

Performance Issues	
Issue	Comment
1. <i>KEY</i> must receive <i>keypressed(value)</i> and release <i>key_courier</i> often enough to avoid buffer overflow at <i>keypressed_buffer</i> .	<i>KEY</i> is reading the <i>value</i> once per cycle of its loop processing as defined in <i>?u_engine?</i>
2. size of <i>keypressed_buffer</i> is dependent on the rate of reception of <i>keypressed(value)</i> by <i>KEY</i>	ok
3. <i>KEY-keypressed(value)</i> can be delayed by any of the <i>N</i> paths leading from the combined wait state. Each service is delayed by the other services, therefore each should complete in a reasonable time; i.e., the deterministic service and response interactions must complete in reasonable time. <i>Assemble Display Command</i> <i>Equate Button Position</i> <i>DSPLY-display(command)</i> <i>CHC_Pos_Buffer!chc_pos(x,y)</i>	relatively simple relatively simple DSPLY is allocated to higher priority actor. This interaction is designed by DF-AsyncInter-5, but is assumed to complete in reasonable time CHC_Pos_Buffer is an overwrite buffer which accepts values to be buffered infinitely often, and with a relatively small worst case delay.

BUILD TABLE REFERENCE: 23 of Display Subsystem**Build Table for: DF-AsyncRx-IPC-2**

Generic Specification Segment	
rx: tx+msg(formal param)	
Application Bindings	
rx	<i>KEY</i>
tx	<i>KBD</i>
msg	<i>keypressed</i>
formal param	<i>value</i>
MachineChart Design Fragment	
Design Contingencies	
Contingency	Proof
1. <i>KEY-keypressed</i> must be designed as a synchronous send of <i>KBD</i>	match
2. <i>value</i> must be type compatible with actual parameter sent by <i>KBD</i> .	ok
Performance Issues	
Issue	Comment
1. <i>KBD</i> blocks until <i>KEY</i> acknowledges that <i>keypressed</i> has been received.	a courier actor is used between these entities (see Build Table 23).
2. <i>KBD</i> is delayed by <i>KEY</i>	Not true in this design because of the added courier actor <i>key_courier</i> .

BUILD TABLE REFERENCE: 24 of Display Subsystem**Build Table for: DF-AsyncTx-IPC-3** (Rendezvous Send with Message)

Generic Specification Segment	
tx: rx-msg(actual_param)	
Application Bindings	
tx	<i>DSPLY</i>
rx	<i>external</i>
msg	<i>display_command</i>
actual_param	<i>code</i>
MachineChart Design Fragment	
Specification Contingencies	
Contingency	Proof
1. <i>DSPLY+display_command(code)</i> must be received eventually by <i>external</i>	
Design Contingencies	
Contingency	Proof
1. <i>DSPLY+display_command(code)</i> must be designed as a synchronous receive of <i>external</i>	match
2. <i>code</i> must be type compatible with the formal parameter defined by <i>external</i>	
Performance Issues	
Issue	Comment
1. <i>DSPLY</i> blocks until <i>external</i> acknowledges that <i>display_command</i> has been received.	
2. <i>DSPLY</i> is delayed by <i>external</i> .	

END

1 2 - 1 2 - 9 3

FIN