

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

**ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600**

UMI[®]

**COMBINING SEMI-FORMAL AND FORMAL NOTATIONS
IN SOFTWARE SPECIFICATION: AN APPROACH TO
MODELLING TIME-CONSTRAINED SYSTEMS**

by

Sergiu-Mihai Dascalu

Submitted
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

at

DALHOUSIE UNIVERSITY

Halifax, Nova Scotia

September, 2001

© by Sergiu-Mihai Dascalu, 2001



**National Library
of Canada**

**Acquisitions and
Bibliographic Services**

**395 Wellington Street
Ottawa ON K1A 0N4
Canada**

**Bibliothèque nationale
du Canada**

**Acquisitions et
services bibliographiques**

**395, rue Wellington
Ottawa ON K1A 0N4
Canada**

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

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

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

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

0-612-63474-4

Canada

Dalhousie University
Faculty of Computer Science

The undersigned hereby certify that they have examined, and recommend to the Faculty of Graduate Studies for acceptance, the thesis entitled "Combining Semi-Formal and Formal Notations in Software Specification: An Approach to Modelling Time-Constrained Systems" by Sergiu-Mihai Dascalu in partial fulfillment of the requirements for the degree of the Doctor of Philosophy.

Dated : November 1, 2001

Supervisor:



Dr. PETER HITCHCOCK

External Examiner:



Dr. GREGORY BUTLER

Examiners:



Dr. PETER BODORIK



Dr. TREVOR SMEDLEY



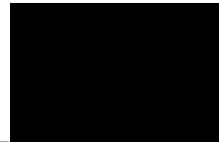
Dr. WILLIAM PHILLIPS

Dalhousie University
Faculty of Computer Science

DATE: September 14, 2001

AUTHOR: Sergiu-Mihai Dascalu
TITLE: Combining Semi-Formal and Formal Notations in Software
Specification: An Approach to Modelling Time-Constrained Systems
MAJOR SUBJECT: Computer Science
DEGREE: Doctor of Philosophy
CONVOCATION: October, 2001

Permission is herewith granted to Dalhousie University to circulate and to have copied for non-commercial purposes, at its discretion, the above thesis upon the request of individuals or institutions.



Signature of Author

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

The author attests the permission has been obtained for the use of any copyrighted material appearing in this thesis (other than brief excerpts requiring only proper acknowledgement in scholarly writing), and that all such use is clearly acknowledged.

**To my daughter Diana, my wife Alexandra,
and my parents Niculina and Vasile**

Table of Contents

List of Tables.....	x
List of Figures	xi
List of Abbreviations	xv
Acknowledgements	xvi
Abstract.....	xviii
1 Introduction.....	1
1.1 Three Paradigms and a View of the Field	1
1.1.1 The First Paradigm or Objects as Conquerors	1
1.1.2 The Resilient Field of Real-Time Applications	2
1.1.3 The Second Paradigm or Formalisation as a Controlling Factor	3
1.1.4 The Third Paradigm or The Power of Pictures	5
1.2 Motivations.....	6
1.2.1 Effectiveness and Simplicity.....	6
1.2.2 Capability of Tackling Complex Tasks.....	6
1.2.3 Early Detection of Errors.....	7
1.2.4 Powerful Combination of Paradigms.....	7
1.2.5 Understandability and Practicality.....	7
1.2.6 Ease of Communication	7
1.2.7 Expressiveness and Modernity	8
1.2.8 Rigor and Precision	8
1.2.9 Refinement.....	8
1.3 Challenges.....	8
1.3.1 Efficient Combination of Techniques and Notations	9
1.3.2 Approaching Time-Constrained Systems from an Object-Oriented Perspective	10
1.3.3 Developing Mechanisms for Formalisation of Graphical Representations.....	10
1.3.4 Rigorous Treatment of Temporal Constraints	11
1.3.5 Provision for User Acceptance	11
1.3.6 Tool Support.....	12
1.3.7 Capability of Extension	12
1.4 Notes on Terminology	13
1.5 The Proposed Approach.....	14
1.6 Overview of the Thesis.....	17
1.7 Chapter Summary.....	18

2	Background: Context and Concepts	19
2.1	Introduction	19
2.2	Research Space and Topic Location	20
2.3	On Specifying Real-Time Systems	22
2.3.1	Characteristics of Real-Time Systems	22
2.3.2	Focus On Time	29
2.4	Brief Immersion in Object-Orientation	30
2.4.1	On Objects and Their Modelling Power	30
2.4.2	Object-Orientation in the Real-Time Domain	33
2.5	On The Importance of Graphical Notations	34
2.6	Formal Notations in Software Development	36
2.6.1	Alexander's Definition of a Formal System	36
2.6.2	Classifications and Examples of Formal Methods	38
2.6.3	Advantages and Disadvantages of Formal Methods	40
2.6.4	Formal Techniques within the Software Development Process	43
2.6.5	A New Trend: Lighter Use of Formal Methods	43
2.7	Chapter Summary	44
3	Background: Notations	45
3.1	Introduction	45
3.2	Z and Flavours of Z	45
3.2.1	The Z Notation	46
3.2.1.1	Sets, Types, and Predicates	47
3.2.1.2	Relations, Functions, and Sequences	50
3.2.1.3	Schemas and Schema Calculus	55
3.2.2	Z Variants and Tools	58
3.2.3	A Glance at Z++	61
3.3	On UML and Its Capability of Dealing with Time	62
3.3.1	A Bird's Eye View on UML	63
3.3.2	UML Support for Modelling Real-Time Systems	76
3.3.3	The UML Promise	82
3.4	Chapter Summary	87
4	Related Work	88
4.1	Introduction	88
4.2	Integration of Semi-formal and Formal Notations in Software Specification	89
4.3	Semi-formal/Formal Integrations of Notations Not Involving Z	92
4.4	Semi-formal/Formal Integrations of Notations Involving Variants of Z	95
4.5	Closely Related Approaches	96
4.5.1	Jia's Augmented Object-Oriented Modeling Language	96
4.5.2	Noe and Hartrum's Extension of Rational Rose 98	99
4.5.3	Blending Octopus and Z	101
4.5.4	Headway System's RoZeLink	103

4.5.5	Object Z and UML	104
4.6	Modalities of Specifying Temporal Constraints in Z	106
4.6.1	Time Refinement in Z	107
4.6.2	The Quartz Alternative	108
4.6.3	Andy Evans' Approach	109
4.6.4	RTOZ	111
4.6.5	TCOZ	112
4.6.6	Other Approaches	114
4.6.7	The Z++ Alternative	116
4.7	Chapter Summary	116
5	Formal Specification of Temporal Constraints	118
5.1	Introduction	118
5.2	Dasarathy's Classification of Temporal Constraints	119
5.3	On the Rigorous Specification of Temporal Constraints	123
5.4	Real-Time Logic (RTL)	124
5.4.1	The Event-Action Model	124
5.4.2	RTL Concepts and Notations	125
5.5	Using RTL in Z++	127
5.5.1	Lano's Key Extensions to RTL	128
5.5.2	Events	128
5.5.3	Terms	129
5.5.4	Formulae	130
5.5.5	Abbreviations	130
5.5.6	Axioms	131
5.6	Chapter Summary	131
6	Translations Between UML and Z++: Formalisation and Deformalisation ...	132
6.1	Introduction	132
6.2	Preliminary Remarks	133
6.3	Formalisation of UML Class Diagrams in Z++	136
6.3.1	Rules for Developing Well-Formed Class Diagrams	136
6.3.1.1	Rules for Class Diagrams	138
6.3.1.2	Rules for Classes	139
6.3.1.3	Rules for Relationships	144
6.3.2	Translation Principles for Class Diagrams	146
6.3.2.1	Translation of Types	147
6.3.2.2	Translation of Attributes	149
6.3.2.3	Translation of Operations	150
6.3.2.4	Translation of Classes	151
6.3.2.5	Translation of Relationships	152
6.3.3	Algorithm for Formalising Class Diagrams (AFCD)	154
6.3.3.1	AFCD Input	155

6.3.3.2	AFCD Output.....	157
6.3.3.3	AFCD Pseudocode	158
6.4	Formalisation of UML State Diagrams in Z++	171
6.4.1	Constraints on the Contents of State Diagrams.....	171
6.4.2	Translation Principles for State Diagrams.....	174
6.4.2.1	General Principles and Terminology.....	174
6.4.2.2	Translation of States	176
6.4.2.3	Translation of Transitions.....	179
6.4.3	Algorithm for Formalism State Diagrams (AFSD).....	183
6.4.3.1	AFSD Input.....	183
6.4.3.2	AFSD Output.....	185
6.4.3.3	AFSD Pseudocode	185
6.4.4	Example of Formalising a State Diagram	189
6.5	Deformalisation: From Z++ Specifications to UML Representations	193
6.5.1	Principles of Deformalisation	193
6.5.1.1	Assigning Types.....	193
6.5.1.2	Generating Attributes for UML Classes	194
6.5.1.3	Generating Operations for UML Classes	195
6.5.1.4	Generating UML Classes	196
6.5.1.5	Generating Relationships.....	197
6.5.1.6	Generating State Diagrams	198
6.5.2	Outline of the Algorithm for Deformalisation (ADF).....	199
6.6	Notes on the Application of Formalisation and Deformalisation Algorithms	202
6.7	Chapter Summary.....	204
7	A Procedural Frame.....	205
7.1	Introduction.....	205
7.2	Modelling Focus	206
7.3	Artefacts.....	207
7.4	Activities	210
7.4.1	Conventions in the Diagrammatic Representation of the Procedural Frame.....	210
7.4.2	Simplifications in the Diagrammatic Representation of the Procedural Frame	212
7.4.3	Stages and Steps.....	212
7.4.4	The Regular Sequence of Modelling Activities.....	215
7.4.5	Alternative Flows of Modelling Activities	217
7.5	Chapter Summary.....	219
8	An Application: The Case of the Elevator System.....	221
8.1	Introduction.....	221
8.2	On the Elevator Case Study	222
8.3	The Problem	223
8.3.1	General Requirements for the Elevator System.....	224
8.3.2	Temporal Constraints for the Elevator System	225

8.3.3	Coverage of Dasarathy Constraints by the Elevator's Timing Requirements.....	227
8.4	The Modelling Solution.....	228
8.4.1	Definition of Use Cases.....	228
8.4.2	Elaboration of Scenarios.....	229
8.4.3	Construction of the Class Diagram.....	233
8.4.4	Specification of Sequence Diagrams.....	234
8.4.5	Elaboration of Class Compounds.....	236
8.4.6	Formalisation through the AFCD and the AFSD.....	240
8.4.7	Enhancement of the Formal Specification.....	249
8.5	Chapter Summary.....	252
9	Towards an Integrated Environment: A Prototype for Harmony.....	253
9.1	Introduction.....	253
9.2	General Principles.....	254
9.3	Overall Organisation.....	257
9.4	The Project Pane.....	259
9.5	The UML Space.....	262
9.6	The Z++ Space.....	263
9.7	Other Features.....	266
9.8	Chapter Summary.....	268
10	Conclusions.....	269
10.1	Introduction.....	269
10.2	Summary Comparison with Closely Related Approaches.....	269
10.3	Main Contributions.....	271
10.4	Other Contributions.....	272
10.5	More On the Limitations of the Proposed Approach.....	273
10.6	A Look Forward.....	274
	Bibliography.....	276
	Appendix A: Summary Overview of Z++.....	295
A.1	BNF Syntax of the Z++ Class Declaration.....	295
A.2	Invocation of Operations.....	297
A.3	Notes on Semantics.....	298
A.4	Extending and Restructuring the Specification.....	300
A.5	Translation to Standard Z.....	301
	Appendix B: Java Implementation of the AFCD.....	302
B.1	Contents of the Program Listing.....	302
B.2	Program Listing.....	303
	Appendix C: Harmony's User Interface.....	359

List of Tables

Table 2.I	Classification of Research Approaches Based on Domains of Exploration	20
Table 3.I	A Summary of Sequences	54
Table 3.II	UML Things (Model Elements)	65
Table 3.III	UML Relationships	66
Table 3.IV	UML Extension Mechanisms	67
Table 3.V	Types of Events in UML	76
Table 3.VI	UML Markings and Expressions for Time and Location	81
Table 4.II	Examples of Semi-Formal/Formal Integrations Not Involving Z	94
Table 5.I	Examples of Semi-formal/Formal Integrations Involving Z	97
Table 7.I	Abbreviations for Modelling Artefacts	209
Table 8.I	Correspondence between ELS Timing Requirements and Dasarathy Constraints	227
Table 10.I	Summary Comparison with Closely Related Approaches	271
Table B.I	Contents of the FCD Program	303
Table C.I	Menus of Menu Bar	360
Table C.II	Shortcut Buttons and Their Equivalent Menu Options	361
Table C.III	File Menu Items	362
Table C.IV	Other Icons for Artefacts	363
Table C.V	Edit Menu Items	364
Table C.VI	View Menu Items	365
Table C.VII	UML Menu Items	366
Table C.VIII	Z++ Menu Items	367
Table C.IX	Tools Menu Items	368
Table C.X	Windows Menu Items	369
Table C.XI	Help Menu Items	370
Table C.XII	UML Toolbox Items	371
Table C.XIII	Items of the Z++ Symbol Box	372

List of Figures

Fig. 2.1	Domains of Research Space and Topic Location	21
Fig. 2.2	Hard, Firm, and Soft Real-Time Systems	23
Fig. 3.1	Partial Z Description of a Robot Arm	57
Fig. 3.2	General Form of Z++ Class Declaration	61
Fig. 3.3	Snapshot of Rational Software Corporation's Rational Rose	64
Fig. 3.4	The 4+1 Architectural Views and UML Diagrams That Express Them	69
Fig. 3.5	Overview of the Automatic Camshaft Testing System (ACTS)	70
Fig. 3.6	Example of Use Case Diagram: Excerpt from ACTS Specification	72
Fig. 3.7	Example of Class Diagram: Excerpt from ACTS Specification	73
Fig. 3.8	Example of Object Diagram: Excerpt from ACTS Specification	74
Fig. 3.9	Example of Sequence Diagram: Excerpt from ACTS Specification	75
Fig. 3.10	Example of Signal: Excerpt from ACTS Specification	77
Fig. 3.11	UML Symbols for Asynchronous and Synchronous Communication	79
Fig. 3.12	Example of Statechart Diagram: A 2-Speed DC Motor for ACTS Axis X	80
Fig. 4.1	First Zoom-In on The Research Space	93
Fig. 4.2	Second Zoom-In on The Research Space	95
Fig. 4.3	Jia's AML-based Approach	98
Fig. 4.4	Noe and Hartrum's Approach	100
Fig. 4.5	The Octopus and Z Integration Approach	102
Fig. 4.6	The RoZeLink Tool	103
Fig. 4.7	The UML/Object-Z Combination	105
Fig. 6.1	The Top-Level FCD Procedure	159
Fig. 6.2	The CheckCDSyntax Procedure	159
Fig. 6.3	The CheckRelationships Procedure	161
Fig. 6.4	Alternative CheckRelationships Procedure	161
Fig. 6.5	The CheckAcrossCD Procedure	162

Fig. 6.6	The CheckClasses Procedure	162
Fig. 6.7	The CDTranslate Procedure	164
Fig. 6.8	The TranslateClasses Procedure	164
Fig. 6.9	The TranslateClass Procedure	164
Fig. 6.10	The TranslateAttributes Procedure	165
Fig. 6.11	The TranslateAttribute Procedure	165
Fig. 6.12	The Translate Operations Procedure	166
Fig. 6.13	The TranslateOperation Procedure	166
Fig. 6.14	The ProcessOpParams Procedure	167
Fig. 6.15	The ProcessOpReturn Procedure	167
Fig. 6.16	The PlaceZPPAttributes Procedure	168
Fig. 6.17	The PlaceZPPOperations Procedure	168
Fig. 6.18	The TranslateRelationships Procedure	169
Fig. 6.19	The TranslateAggregation Procedure	169
Fig. 6.20	The TranslateAssociation Procedure	170
Fig. 6.21	General Form of A State Transition	172
Fig. 6.22	The SDTranslate Procedure	186
Fig. 6.23	The TranslateStates Procedure	187
Fig. 6.24	The TranslateState Procedure	187
Fig. 6.25	The TranslateTransitions Procedure	188
Fig. 6.26	The ProcessCallTrans Procedure	188
Fig. 6.27	The GenerateTransitOperation Procedure	189
Fig. 6.28	The WriteHistoryPredicates Procedure	189
Fig. 6.29	DCMotor State Diagram from the ACTS	190
Fig. 6.30	Z++ Class DCMotor Generated by the AFSD	191
Fig. 6.31	The AFD Procedure	200
Fig. 6.32	The TranslateZPPClass Procedure	201
Fig. 6.33	The GenerateUMLClass Procedure	201
Fig. 7.1	The Procedural Frame	211
Fig. 7.2	Regular Sequence of Modelling Activities	216
Fig. 7.3	An Example of Irregular Flow of Modelling Activities	218

Fig. 8.1	ELS Use Case Diagram	228
Fig. 8.2	ELS Scenario: Outside Request A	230
Fig. 8.3	ELS Scenario: Outside Request B	231
Fig. 8.4	ELS Scenario: Outside Request C	232
Fig. 8.5	ELS Scenario: Inside Request A	232
Fig. 8.6	ELS Class Diagram: Initial Structure	233
Fig. 8.7	ELS Sequence Diagram: Outside Request A	235
Fig. 8.8	ELS Class Button	237
Fig. 8.9	ELS State Diagram for the Button Class	237
Fig. 8.10	ELS Class Elevator	237
Fig. 8.11	ELS State Diagram for the Elevator Class	238
Fig. 8.12	ELS Class Diagram with Attributes and Operations Attached to Classes	239
Fig. 8.13	ELS Z++ Specification Generated by the AFCD	241
Fig. 8.14	ELS Z++ Class Button Updated by the AFSD	246
Fig. 8.15	ELS Z++ Class Elevator Updated by the AFSD	247
Fig. 9.1	Harmony's Look	258
Fig. 9.2	The View Menu	258
Fig. 9.3	The Project Pane	260
Fig. 9.4	The New Model Element Selector	260
Fig. 9.5	Harmony with New Project Just Created.....	261
Fig. 9.6	The UML Menu	262
Fig. 9.7	A UML Toolbox	263
Fig. 9.8	Harmony's Z++ Menu	264
Fig. 9.9	The Z++ Symbol Box	265
Fig. 9.10	Harmony Specific Buttons	266
Fig. 9.11	The Legend Pane	267
Fig. C.1	The Harmony Window	359
Fig. C.2	Harmony's Menu Bar	360
Fig. C.3	Harmony's Main Toolbar	361
Fig. C.4	Harmony's File Menu	362

Fig. C.5	Harmony's New Model Element Selector	363
Fig. C.6	Harmony's Edit Menu	364
Fig. C.7	Harmony's View Menu	365
Fig. C.8	Harmony's UML Menu	366
Fig. C.9	Harmony's Z++ Menu.....	367
Fig. C.10	Harmony's Tools Menu	368
Fig. C.11	Harmony's Window Menu	368
Fig. C.12	Harmony's Help Menu	369
Fig. C.13	Harmony's UML Toolboxes	370
Fig. C.14	Harmony's Z++ Symbol Box	371
Fig. C.15	Harmony's Legend Pane: COMP and CD Symbols	373
Fig. C.16	Examples of Harmony Messages	374
Fig. C.17	Harmony's About Message	374

List of Abbreviations

ACTS	Automatic Camshaft Testing System
ADF	Algorithm for Deformalisation
AFCDD	Algorithm for Formalising Class Diagrams
AFSD	Algorithm for Formalising State Diagrams
CASE	Computer-Aided Software Engineering
CD	Class Diagram
CLS	Class Specification
CLSTD	Class State Diagram
COMP	Class Compound
CSP	Communicating Sequential Processes
DFD	Data Flow Diagrams
ERD	Entity Relationship Diagrams
IDE	Integrated Development Environment
ISE	Integrated Specification Environment
GUI	Graphical User Interface
ELS	The Elevator System
OMG	Object Management Group
OMT	Object Modelling Technique
OO	Object-Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOSE	Object-Oriented Software Engineering
OOZ	Object-Oriented Variant of Z
ROOM	Real-Time Object Oriented Modelling
RT	Real-Time
RTL	Real-Time Logic
RTS	Real-Time Systems
SC	Scenario
SCG	Scenario Group
SQD	Sequence Diagram
SQDG	Sequence Diagram Group
TCS	Time-Constrained Systems
TL	Temporal Logic
UC	Use Case
UCD	Use Case Diagram
UML	Unified Modelling Language
ZPPC	Z++ Class
ZSPEC	Z++ Specification

Acknowledgements

First of foremost I would like to express my deepest gratitude to my thesis supervisor, Dr. Peter Hitchcock, who guided my research with infinite wisdom and patience, as well as with unabated confidence in my ability of completing the work presented here. I am much indebted to Dr. Hitchcock for introducing me to the rigorous, admirable world of formal notations, and for spending countless hours with me guiding the shaping of the approach described in this thesis.

I would also like to express my heartfelt gratitude to the members of my examining committee, Dr. Peter Bodorik, Dr. Trevor Smedley, and Dr. William Phillips, for their guidance and support throughout the course of my studies. In particular, I would like to thank Dr. Bodorik for supervising my earlier coursework and research, to express my gratitude to Dr. Smedley for opening for me the realm of visual notations, and to thank Dr. Phillips for his advice and encouragement.

I would also like to extend my thanks to the external examiner, Dr. Gregory Butler, from Concordia University, Montreal, for his thorough review and constructive critique of the thesis. Dr. Butler has pointed out with great clarity both the merits and the limitations of our thesis, his observations about the latter being extremely helpful for us to better evaluate our work and to define its future directions.

I would also like to express my gratitude to the former School of Computer Science and the former Technical University of Nova Scotia for the financial support I received in the form of scholarships and awards during my doctoral studies. I would like to thank both the former School of Computer Science at TUNS and the current Faculty of Computer Science at Dalhousie University, in particular to their directors, Dr. Jonathan Barzilai and, respectively, Dr. Jacob Slonim, for trusting me with teaching appointments as lecturer for the courses Software Development with Ada, Software Engineering, Database Management Systems, and Computer Organization for Electrical Engineers. Teaching these courses offered me the privilege of being constantly stimulated by the inquiring minds of my students and helped me consolidate the foundation of the present work.

Among the students who steadily motivated and encouraged me in my work Paul Evans, Greg Power, Chad Seward, Patrick Lee, Silvano Da Ros, Carmen Wong, Fernand Boudreau, Stephen Nickerson, Atreya Basu, Samer Mansour, and Jas Singh are the ones to whom I am especially grateful. Many thanks are due as well to my graduate colleagues Dawn Jutla, Eve Rosenthal, Marcel Karam, Krys Gawetski, and Arun Sood who provided me, in various phases of my studies, with a friendly and stimulating environment.

I am deeply indebted to Dr. Traian Ionescu, my former team leader and head of department at the Polytechnic University of Bucharest, Romania, who mentored my earlier professional steps with wisdom, goodwill, and grace. To him and his wife, Mrs. Mihaela Ionescu, I owe special thanks for their guidance and friendship throughout many years.

I was very fortunate to work with and be guided by several other senior professors at the Polytechnic University of Bucharest. I would like to take this opportunity and thank Dr. Radu Dobrescu, Dr. Sergiu Iliescu, and Dr. Aurelian Stanescu for giving me, years ago, through their personal examples, an excellent definition of the word *academia*.

To my longtime friend, Cristian Mierlea, I would like to say thank you for many things but especially for teaching me, many years ago, that in essence the computers are an extraordinary combination of mathematics and literature. And for inducing me, in those high school years, to value them dearly. For their constant support and encouragement throughout the years my heartfelt thanks go as well to my friend Mike Cailean and to his wife, Rodica Cailean. I would also like to express my thanks to a special friend, close to my heart and mind, Jorge Luis Borges, the great Argentine writer, presently in Heaven, in all probability narrating one of his fabulous stories. The beauty and finesse of his writings lifted my spirit in countless nights.

I have many special thanks for my parents, my wife, and my daughter. Words can hardly express my gratitude for them, but I trust they know how deeply I have appreciated their support throughout my studies. My daughter Diana grew up with a student father, quietly accepting to have me away from her in so many occasions. Finally, I would like to particularly thank my wife for her unfailing understanding, extended sacrifice, and unconditional love. I would like her to know that without her support this thesis would never have happened.

Abstract

This thesis is about the integration of semi-formal, graphical representations with formal notations within a modelling approach aimed at the construction of time-constrained systems (TCS). We believe that the two types of notation, graphical (semi-formal) and, respectively, formal, can efficiently complement each other and provide the basis for a software specification approach that can be both rigorous and practical. Although many authors have envisaged the advantages of combining informality with formality in software construction, there are very few reports that address the problem within the context of object-orientation and project its solution over the canvas of TCS modelling.

The pillars of our approach are the following: the combination of formal and semi-formal notations for specification purposes, the integration into an object-oriented approach of modelling capabilities that target properties of TCS, the elaboration of detailed algorithms for UML to Z++ translations, and the proposal of a procedural frame for effective and reliable development of TCS. Principles and an outline of an algorithm for the reverse translation, from Z++ to UML, are also included in the approach.

While the graphical notation employed is a subset of the UML, the formal notations used are Lano's Z++, an object-oriented variant of Z, and Jahanian and Mok's Real Time Logic. Both structural and dynamic aspects of the system are considered and a new modelling element denoted class compound is proposed.

From a methodological point of view, after several UML-based modelling steps are completed the formalisation process can take place, the result being a formal specification derived from the graphical representations obtained in the earlier steps. The integrated, semi-formal and formal model of the system can be subsequently enhanced while the designed translation mechanisms allow changes in the graphical representations to be reflected into the formal specifications as well as modifications of the formal specifications to be fed back into the diagrammatic descriptions of the system.

A case study, an Elevator System, is included in the thesis to illustrate the application of the proposed approach and the GUI-centred design of Harmony, an integrated specification environment intended to support the approach, is also presented.

Although we believe the proposed approach offers a viable solution for modelling software systems, it has nevertheless a number of limitations that need be pointed out. Firstly, the translation of UML constructs is restricted to a subset of the notation, and the treatment of state diagrams is confined to sequential, non-composite executions (composite states and aspects related to concurrency are not covered). Secondly, although timing constraints can be attached to structural UML constructs in the regular way, we have not tackled their mechanised translation to Z++, and there is a limited incorporation of such constraints in the state diagrams considered. Thirdly, the formal language employed, Z++, is currently lacking in supporting tools, which could be an impediment to the use of the proposed approach in industrial applications. Fourthly, for the formalisation algorithms a set of rules for well-formedness and a set of principles for translation are given without using meta-models for UML and Z++/RTL, yet the use of these meta-models would have probably allowed a more concise and precise description of the algorithms.

However, our belief is that, through future work, the above limitations can be overcome and our proposal can thus become a stronger contender in the landscape of object-oriented approaches for modelling TCS.

1 INTRODUCTION

“ ‘Where shall I begin, please your Majesty?’ he asked.
‘Begin at the beginning,’ the King said, gravely,
‘and go on till you come to the end: then stop.’ ”

[Lewis Carroll, *Alice's Adventures in Wonderland*, 1865]

1.1 Three Paradigms and a View of the Field

1.1.1 The First Paradigm or Objects as Conquerors

Few paradigms have had such a significant impact on the field of software development as the object-oriented approach. One can argue that actually there is nothing really new under the sun of technology, and that the object-oriented paradigm simply built upon the results of many honest structured methods exercised intensively on various domains of application over a significant number of years. The time of object-orientation just had to come, one may say, and this is probably true considering the constant progress within the computers' world, but we still cannot stop admiring its fundamental naturalness and the benefits it made possible.

The object-oriented paradigm has shifted the developers' focus from the solution domain (computer implementation) to the problem domain (the real-world that we relentlessly try to model and control) and brought with it a much greater modelling power –resulting primarily from the natural correspondence between objects and real-world entities. The object-oriented approach has also come with solutions for improved control of complexity –mainly

through abstraction, information hiding, and localisation, and provided effective answers for code reusability and extensibility via encapsulation and inheritance. The object concept proves to be remarkably powerful while essentially simple –the key characteristics of any true successful solution. And of any true conqueror.

1.1.2 The Resilient Field of Real-Time Applications

While a large variety of general-purpose object-oriented development methods have been proposed, among the most notable Shlaer and Mellor [Shlaer88, Shlaer91], Coad and Yourdon [Coad90, Coad91], OMT (Object Modeling Technique) [Rumbaugh91], Booch [Booch94], OOSE (Object-Oriented Software Engineering) [Jacobson94], Fusion [Coleman94], and more recently the Unified Software Development Process [Jacobson99], there has been comparatively a smaller production of object-oriented methods dedicated to real-time systems. This type of applications seemed to be more resilient to potential conquerors, including the objects. The explanation resides mostly in the efficiency concerns developers of real-time systems may have. As Bran Selic points out, even though the object paradigm is suitable for real-time applications (due to its equal emphasis on both structure and behaviour, which appropriately answers the needs of real-time systems development methodologies) it nevertheless extended over the real-time domain more slowly than over other areas of software development [Selic98]. The cause, the author indicates, lies in the rather scarce attention paid to important aspects of real-time execution, such as concurrency and efficient allocation of memory. Indeed, the constraints on execution speed and memory space are much stricter for real-time systems which, among other things, must meet deadlines and operate in typically unfriendly environments. Consequently, the traditional solution for ensuring both high execution speed and low memory utilization was to write lower level code using assembly language or languages such as C, Ada or Occam. These languages in turn provided relatively little support for the implementation of object-oriented designs. On the other hand, where support was provided (e.g., C++, Smalltalk) the overhead for manipulating objects at run time seemed to be costly, precluding the implementation of real-time systems in all but the more relaxed (softer) cases.

Nevertheless, some newer object-oriented approaches for real-time development such as ROOM [Selic94], Octopus [Awad96], and Comet [Gomaa00] have been successfully developed over the last years. This is certainly related to the constant improvements in hardware –faster, more powerful, and more compact processors being able to alleviate a number of issues related to the development of real-time systems in the “object-oriented way” and extend the application range of the OO paradigm in areas never tackled before. Commonly, the object-oriented analysis and design techniques that focus on real-time systems extend the traditional capabilities of general-purpose object-oriented methodologies with support for modelling aspects such as concurrency, distribution, timing constraints, synchronisation, communication, interrupts, and exceptions. At the implementation level, newer languages such as Ada95 [Barnes96] and Java [Gosling96] offer good support for writing real-time applications in an object-oriented manner (Java’s capability for real-time programming is amply illustrated in [Bollella00]). These realities provide solid grounds for us to predict, for the near future, an increased interest in applying the object-oriented technology to the field of real-time applications. In other words, the field’s resilience has been eroded to the point of the complete acceptance of the conqueror objects.

1.1.3 The Second Paradigm or Formalisation as a Controlling Factor

Software developers need to be resourceful, imaginative, alert, and quick to react to new challenges. This is due to the dynamics of their profession, in which daily novelties represent the only constant characteristic of the work environment. The need for fast and efficient solutions for new problems exercises tremendously the creativity of developers. But in the rush for delivering the expected solutions errors happen and bugs sneak in the software produced. Sometimes, the entire architecture of a program turns out to be erroneous. The craft of software developers needs reality checks, more so if the application domain is safety-critical or security-critical. Formalisms are needed as controlling factors of a developer’s work; creativity must be channeled properly, and some moderation in art is necessary. It is well known that the best masterpieces brightly combine inspiration with rigor. In software development, formal methods are precisely employed to bring in the latter.

As shown by Gerhart et al., “formal methods are mathematical synthesis and analysis techniques used to develop computer-controlled systems” [Gerhart94, pp.5]. While it is observed that the technological transfer of formal development approaches from the academia to the industry is rather slow, an increased interest in the application of formal methods to software construction has been signaled over the last years [Fraser94, Clarke96, Hall98, Abernethy00]. Typically, what prompts the usage of formal techniques are safety concerns, regulatory standards, or the need to demonstrate that the implementation of a system corresponds to the system’s requirements. However, we believe that the most important reason for applying formal methods in industrial applications lies in the improved understanding of the system under construction and, generally speaking, in increased intellectual control over the software being developed.

Numerous formalisms or formal development frameworks have been proposed, among the most notable being Temporal Logic (TL) [Rescher71, Pnueli77], the Vienna Development Methodology (VDM) [Bjørner78, Jones90], Communicating Sequential Processes (CSP) [Hoare78, Hoare85], Calculus of Communicating Systems (CCS) [Milner80], Larch [Guttag85, Guttag93], Statecharts [Harel87, Harel96], and the Language of Temporal Logic Specification (LOTOS) [ISO89], but we will focus our attention on the formalism that emerged as one of the most popular over the last decade: the specification language Z, originated from the Oxford University Computing Laboratory, U.K., and currently used by many organisations all over the world. Very good classifications of formal approaches can be found in [Fraser94], [Gaudel94], and [Liu97], while authoritative references on Z are [Spivey92] and [Wordsworth92].

While successfully employed for formally describing and analysing numerous data-intensive, non real-time applications, the specification language Z has been only occasionally utilised for the development of time-constrained systems. Although mathematically sound, mature, expressive, and elegant, Z has been traditionally deemed of limited applicability in describing systems essentially characterised by strict demands on their meeting of prescribed deadlines, systems that most often are also concurrent in nature and complex, and possibly even safety

critical. This limitation is due mainly to Z's intrinsic lack of support for capturing temporal properties of systems and to its reduced capability for simulation, which makes difficult the construction of executable prototypes that could allow developers to interactively refine and validate the specifications. In addition, due precisely to its generality and expressiveness, Z does not typically allow for automated translation of specification into implementation code. However, newer studies have been focused on finding modalities of using Z for specifying real-time systems [Fidge97, Periyasamy97, Mahony98] and it has also been shown that by employing additional conventions and structuring mechanisms it is possible to animate a large subset of Z descriptions [Utting95, Jia98b]. Both these studies and the well-known, solid mathematical foundation offered by Z for formally capturing various properties of systems have encouraged us to investigate the possibility of using Z (more precisely, an object-oriented extension of Z) in the development of real-time systems (which, for reasons explained later in this chapter, we will refer to as *time-constrained systems*). In short, to approach successfully the field of real-time systems, we believe that objects alone are not sufficient: mathematical rigor is needed, and should be provided as early as possible in the software development process.

1.1.4 The Third Paradigm or The Power of Pictures

Descriptions of computer applications, at least in what regards the software components, used to be mostly if not entirely textual. There were hardly any other forms of representation but text and perhaps formulae and tables (both of them in essence some other forms of organised text). Driven by the technological engine that has produced increasingly faster processors and constantly larger-capacity devices, the world of software itself has changed in the last decade or so. The words of David Harel, in a 1988 seminal article, proved to be prophetic: "We are entirely convinced the future is visual. We believe that in the next few years many more of our daily technical and scientific chores will be carried out visually, and graphical facilities will be far better and cheaper than today's" [Harel88, pp. 528]. While after more than a decade we can extend this prediction to incorporate multimedia facilities, today we feel fortunate to witness the truthfulness of Harel's prediction and admire the

accuracy of his vision. The graphical symbols (for practical purposes we exclude from them the classical letters of the alphabet), the icons, the visual metaphors, the animation, are now common parts of our daily interaction with the computers. Actually, it is hard to imagine today any significant software development environment based exclusively on text. Even the more conservative Unix systems have included graphical interfaces into their environments. As Harel predicted, the present *is* and indeed the future *will* continue to be visual. We have complied with this reality by incorporating a graphical notation in our modelling approach and by providing a graphical user interface to the tool that supports this approach.

1.2 Motivations

The motivations for our research approach can be summarised as follows:

1.2.1 Effectiveness and Simplicity

First and foremost, we have the fundamental belief that any new, practical approach should necessarily be both effective and simple –or, to be more precise and use one of Einstein’s well known quotes, “as simple as possible, but no simpler” (this quote is cited, among others, by Stroustrup in his landmark book on C++ [Stroustrup97, pp. 723]). Obviously, any academic research should have a motivation that ultimately relates to practical needs. Overcomplicated software development approaches have difficulties gaining widespread acceptance in industrial environments, and as such they take the risk of remaining mere exercises in abstraction. The point here is not to underestimate the need for complex, sound, thoroughly refined theoretical foundations for new software development techniques, but rather to emphasise the necessity of hiding such foundations under apparently unsophisticated facades. In other words, we are driven in our approach by the desire to “engineer the illusion of simplicity” [Booch94, pp. 6].

1.2.2 Capability of Tackling Complex Tasks

We see the real-time systems as a complex, challenging field of investigation that is open to new research and offer the promise of rewarding methodological improvements. Benefits of

effective application development in this area are potentially enormous [Kopetz97, McUumber99, Douglass99].

1.2.3 Early Detection of Errors

Cost-benefit considerations also provide for us the compelling reason to focus on the early stages of the software development process, where detecting and correcting an error is usually between tens and hundreds of times less expensive than later, during implementation and maintenance [Boehm84, Schach99].

1.2.4 Powerful Combination of Paradigms

We consider that the accurate combination of several major paradigms that emerged vigorously within the software development world can provide the basis for a technologically sound, useful, and efficient methodological solution.

1.2.5 Understandability and Practicality

Effectiveness requires excellent communication and minimal departure from the problem domain in terms of description of functionality. As such, we see use cases and scenarios as the most appropriate means of interactivity, as key elements for bridging the gap between the users' understanding of the system under development and the developers' view of the same thing (the system). In software specification, capturing the behaviour of a system is probably more important than describing the system's structure, because the latter can generally be subjected to some approximations and refined in later stages.

1.2.6 Ease of Communication

Speed of communication and shared understanding depend on the way the information is organised and on the quality of the information's conveyor. Visual representations and graphical symbols are very powerful means of transmitting information. One cannot rely exclusively on unadorned text for capturing the intricacies of real-time systems. We are

compelled by today's technology, in which visual descriptions play a very important role in conveying information, to incorporate in our approach forms of graphical representation.

1.2.7 Expressiveness and Modernity

Because we deal with the specification of software systems we are compelled, for reasons outlined in Subsection 1.1.1, to proceed in an object-oriented manner. We use object-orientation as the wrapper paradigm of our approach that also incorporates formality and focuses on real-time issues. The widespread success of this paradigm accounts for our choice, there is no real competition for objects at this point in time.

1.2.8 Rigor and Precision

Formality or, in other words, mathematical rigor is a condition for dependability and assurance when dealing with real-time systems. Not only are we convinced that the key parts of the more complex software specifications should be treated formally, but we make out of formalisation an important component of our approach.

1.2.9 Refinement

Finally, to supply our approach with the necessary characteristic of "naturalness" (synonymous to "developer-friendly") we have included the classical technique of refinement in the modelling approach proposed (the term is used here in the sense of iterative revision of the model for gradual improvement, not in the sense of successive detailing of the model up to executable code). It should be point out that refinement is not used simply as a universal remedy, but as an important constituent of our approach, as shown in Chapter 7 of the thesis.

1.3 Challenges

Based on the above-mentioned considerations, our essential goal, stated briefly, is to propose a new, theoretically sound, yet user-friendly and pragmatic methodological approach for

specifying time-constrained systems. The approach aims at incorporating both object-oriented principles and formal techniques for describing the software under construction. We have identified a number of major challenges for our endeavor, as outlined below.

1.3.1 Efficient Combination of Techniques and Notations

There is an apparent dichotomy between graphical (specifically, semi-formal or informal) and formal techniques for software specifications, but there is also a growing number of approaches that attempt to integrate them and reap the benefits of both, as shown in Chapter 4. (To be precise, graphical notations can be formal, as discussed more in Section 2.5 of this thesis, but unless specified otherwise we refer in our dissertation to the larger category of semi-formal and informal graphical notations—see also the notes on terminology in Section 1.5). Typically, specification approaches based on semi-formal or informal graphical representations are designed to provide a user-friendly apparatus for software development, and focus primarily on suitable methodological steps and on the inclusion of an easy to manipulate set of modelling symbols. The concern for rapid development plays an important role in the definition of such approaches. Conversely, formal techniques are employed rather as sophisticated tools for demonstrating properties of the systems, and are generally used only in situations that require special attention, such as safety analysis or security enforcement. Formal methods can provide greater intellectual control even though, as pointed out by Gerhart et al., no single method is general enough to completely cover an application domain, and it is rather unclear how to combine formal methods with other methods [Gerhart94]. However, as indicated by Perry Alexander, the two types of models, formal and informal, are not competitive, but complementary [Alexander95]. On the one hand, graphical models are natural and easy to understand and on the other hand formal models ensure precise specification and proof capability. The integration of the two models would combine, as well said by Alexander, “the best of both worlds,” thus offering a solution for reliable, efficient software development. The challenge remains, obviously, to seamlessly integrate them in an efficient, unified approach, balanced between formal and informal, flexible enough to be used for a large class of applications, and able to adapt to various degrees of rigorousness demands.

1.3.2 Approaching Time-Constrained Systems from an Object-Oriented Perspective

The application of the object-oriented paradigm has been extended relatively recently to the area of real-time systems (more details are presented in Chapters 2, 3, and 4 of the thesis). However, there are numerous aspects of such systems that need particular attention when dealt with from an object-oriented point of view. As pointed out by numerous authors, the specification of real-time systems using the object paradigm remains an area of ongoing research [Yang96, Evans99, McUmbler99]. The modelling challenges in the case of time-constrained systems embrace both structural aspects (such as identification and structuring of classes, establishing relationships, and deciding on object responsibilities) and behavioural aspects, including message passing, synchronisation, communication, parallel execution and, of course, capturing of time properties in the form of precise temporal constraints imposed on the run-time execution of the system.

1.3.3 Developing Mechanisms for Formalisation of Graphical Representations

The translation of models described using a semi-formal graphical representation into their formal, mathematically sound counterparts has been the object of previous research work, as presented in more detail in Chapter 4 of the thesis. However, rules for formalisation have been designed primarily in the context of structured methods, such as SSADM (Structured Systems Analysis and Design Method) [Pollack92] or the RRT (Rigorous Review Technique) [Aujla94], while relatively few attempts have targeted the object-oriented models, and even fewer have been dedicated to the specification of real-time systems. With the emergence of the modelling standard UML (Unified Modelling Language) [Booch98] some recent approaches have focused on translating UML notations into formal equivalents or on employing UML in conjunction with formal notations, as discussed in more detail in Chapters 3 and 4. Yet, there is still a need for continued work in this new direction, especially if we take into consideration real-time aspects of the systems.

1.3.4 Rigorous Treatment of Temporal Constraints

In a frequently cited paper, Dasarathy stresses the importance of specialised constructs in requirements languages for capturing timing constraints [Dasarathy85]. He points out that temporal restrictions typically considered are performance constraints (placed on the system's response) although the same importance should be given to behavioural constraints, which impose limits on the rate of stimuli on a system. Dealing with time in a rigorous fashion is in itself a complex problem. Accurately capturing timing properties of systems has been for some time the subject of considerable research work [Hoare78, Dasarathy85, Ostroff89, Shaw92, Mathai96, and many others]. However, including temporal aspects in object-oriented models is an even greater issue, a subject that over the last few years has increasingly attracted the attention of researchers (examples of research in this direction include [Vishnuvajjala96], [Selic99a], [Alagar00], and [Kim00b]). We strongly agree with Leung and Chan that "being such an important notion, time deserves a proper treatment" [Leung96, pp. 246]. Consequently, we attempt to include in our notation a set of modelling constructs capable to provide the necessary support for expressing our expectations of punctuality and collaboration regarding the components of the system being developed.

1.3.5 Provisions for User Acceptance

As previously mentioned, one of our goals is that of understandability and practicality. We advocate the application of formal techniques in software development, particularly in software specification, but we are aware that the acceptance of such techniques by the software development community can be achieved only by proposing a well-defined, relatively small, yet expressive set of notations, incorporated into a straightforward and easy-to-follow modelling technique. Therefore, the challenge is to reach the equilibrium between the true expressiveness of the approach and its apparent complexity, which must not be perceived as too complicated to its intended users. Of course, it will not be possible to completely hide the mathematical foundation of the approach behind graphical symbols but, as pointed out by Gerhart et al., the main challenge for applying formal methods consists not of teaching the developers the mathematics involved, but of training the users how to model

the systems properly [Gerhart94]. Hence, we need work on the notation, but must not forget the method.

1.3.6 Tool Support

A recognised issue with the formal techniques in general is the lack of tool support [Gerhart94, Dill96]. Software tools are necessary for enhanced interaction with the user, including navigation and visualisation, for type checking, and for reasoning about the consistency of specifications across larger projects. Also, improved mechanisms of version control, as well as facilities for maintaining conformance between formal specifications and their corresponding design rationales are needed [Johnson96]. Consequently, our intention is to supply the theoretical results of our work with suitable tool support, in the form of an environment for object-oriented, visual and formal modelling of systems. Even though some desirable capabilities of this environment, such as formal proof and animation, would not be included in our tool at this stage (such features would require separate, complex research investigations) our intention is to include sufficient functionality in the tool to illustrate the practicality of our approach.

1.3.7 Capability of Extension

Even though potentially very rewarding, dealing with formal aspects at the specification level must be seen only as a starting point towards the application of the proposed dual approach, formal and semi-formal, to the entire software development process. We would like to see beyond the present dissertation and leave the door open for potential extensions beyond the modelling phase, for instance for prototyping and simulation, refinement to executable code, specification based testing, and formally-conducted maintenance. In practical terms, the challenge is that both the notation and the deliverables of our specification approach should be ready for use in subsequent software development phases as well as in association with alternative software construction techniques and tools.

1.4 Notes on Terminology

Before outlining the approach proposed in this thesis several notes on terminology are necessary.

First of all, we rely on Fraser et al. to distinguish between formal, semi-formal, and informal specification techniques [Fraser94]. Specifically, *informal techniques*, represented by natural language and unstructured pictures, “do not have complete sets of rules to constrain the models that can be created,” *semi-formal techniques* have well-defined syntax and their “typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities,” while *formal techniques*, such as specification languages based on predicate logic, have precise syntax and semantics and “there is an underlying model against which a description expressed in a mathematical notation can be verified.” [Fraser94, pp. 79].

Secondly, as many other authors, for instance [Spivey92] and [France97], we use the term *notation* as a substitute for *language*, although rigorously speaking notation refers only to the set of symbols belonging to the language. This commonly used promotion of the term helps avoiding tedious repetitions and simplifies the discourse of the thesis.

Thirdly, we use the word *specification* in the sense defined by Alan Davis, that of a document containing a description (in our case, of the software under construction). According to this definition, one can use terms such as requirements specification, design specification, or test specification [Davis93, pp. 372].

Fourthly, the word *modelling*, which also appears in the title of our thesis, is used to denote the activity of creating a model; that is, of developing a representation of the real thing (which, again, in our case is the software system being built). We see the two concepts, specification and modelling, closely connected and the difference between them of rather

fine nuance. Specification, in our view, is a description that may propose a model, while a model, in its analytical form, is recorded in a specification (for the sake of completeness, generally speaking a specification may not contain a model and a model may not have a specification). In our approach the distinction between specification and modelling is especially difficult to highlight; using well-established terminology, we employ the *modelling notation* UML and a variant of the *specification language* Z to create object-oriented *models* of the system, described in documents (*specifications*) that encompass both analysis and design aspects.

Finally, we use the term *time-constrained systems* (TCS) as an alternative to real-time or reactive systems in order to emphasise the temporal restrictions imposed on such systems and to shift the focus from specialised, less approachable products confined to rather restricted domains (military, nuclear energy generation, or medical devices), to more accessible products such as operating systems, transaction processing systems, cellular phones, and microwave oven controllers. In our view, one can consider the term time-constrained systems a substitute for both hard and soft real-time systems –a substitute that stresses the importance of timing properties that characterise these systems. Nevertheless, in order to avoid repetitions and employ recognised terminology when necessary, the terms time-constrained systems and real-time systems are used interchangeably in this thesis.

1.5 The Proposed Approach

This thesis is about the integration of semi-formal, graphical representations with formal notations within a modelling approach aimed at the construction of time-constrained systems. We believe that the two types of notation, graphical (semi-formal) and, respectively, formal, can efficiently complement each other and provide the basis for a software specification approach that can be both rigorous and practical. The former notations, relying on graphical symbols and diagrams, bring the “power of pictures,” which manifests through better representation of abstractions and higher expressiveness. The latter notations, precise, based on mathematics, increase the developer’s assurance and intellectual control and make possible automated synthesis and verification. Although many authors have envisaged the

advantages of combining informality with formality in software construction, there are very few reports that address the problem within the context of object-orientation and project its solution over the canvas of TCS modelling.

The pillars of our approach are the following: the combination of formal and semi-formal notations for specification purposes, the integration into an object-oriented approach of modelling capabilities that target properties of TCS, the elaboration of detailed translation algorithms from diagrammatic representations to formal specifications, and the proposal of a procedural frame for effective and reliable development of TCS. Principles and an outline for the reverse translation, from formal specifications to graphical representations, an auxiliary process intended to support the understanding of the system's model by developers and users not trained in formal methods, are also included in the approach.

While the graphical notation employed is a subset of the UML, the formal notations used are Lano and Haughton's Z++ object-oriented variant of Z [Lano91, Lano94a, Lano95] and Jahanian and Mok's Real Time Logic [Jahanian86, Jahanian94]. Both structural and dynamic aspects of the system are considered and a new modelling element, denoted class compound and consisting of a simple yet practical aggregation of the UML class and state diagram constructs, is proposed in order to facilitate the specification process.

From a methodological point of view, after several UML-based modelling steps are completed the formalisation process can take place, the result being a formal specification derived from the graphical representations obtained in the earlier steps. The integrated, semi-formal and formal model of the system can be subsequently enhanced while the designed translation mechanisms allow changes in the graphical representations to be reflected into the formal specifications as well as modifications of the formal specifications to be fed back into the diagrammatic descriptions of the system.

A case study, an Elevator System, is included in the thesis to illustrate the application of the proposed approach and the GUI-centred design of Harmony, an integrated specification environment intended to support the approach, is also presented.

Although we believe the proposed approach offers a viable solution for modelling software systems, it has nevertheless a number of limitations that need be pointed out. Firstly, the translation of UML constructs is restricted to a subset of the notation, and the treatment of state diagrams is confined to sequential, non-composite executions (composite states and aspects related to concurrency are not covered), which reduces the applicability of the translation algorithms to modelling TCS. Secondly, although temporal constraints can be attached to structural UML constructs in the regular way (using UML time marks, time expressions, and timing constraints –see Table 3.VI for details), we have not tackled their mechanised translation to Z++, and there is a limited incorporation of such constraints in the state diagrams employed. More precisely, the timing information pertaining to state diagrams considered in the formalisation process is only in the form of bounds [lower, upper] included in the label of transitions and in the form of transition triggers of the kind *passage of time events* (all other sorts of temporal constraints need be added manually by the Z++ specifier). Thirdly, the formal language employed, Z++, is currently lacking in supporting tools, which can be an impediment to the use of the proposed approach in industrial applications (our Harmony tool is not yet implemented, and we have not intended to deal with tool-supported formal analysis and formal refinement in the present thesis). In fact, we are aware of tools for Z++ only via [Lano94d], in which it is mentioned that such tools have been written in Quintus Prolog and ProWindows, but we have not investigated the possible connection of our approach to these tools. Fourthly, for the formalisation algorithms a set of rules for well-formedness and a set of principles for translation are given without using meta-models for UML and Z++/RTL, yet the use of these meta-models would have probably allowed a more concise and precise description of the algorithms. Also, there are a number of issues related to the application of the formalisation and deformalisation algorithms, indicated in Section 6.6, that deserve further investigation and require additional work.

However, our belief is that, through future work, the above limitations can be overcome and our proposal can thus become a stronger contender in the landscape of object-oriented approaches for modelling TCS.

1.6 Overview of the Thesis

The present thesis, in its remaining chapters, is organised as follows. *Chapter 2, Background: Context and Concepts*, defines the space of our research, localises in this space the topic of our dissertation, and presents the most significant aspects of the “domains” that belong to the space of our investigation. The distinguishing characteristics of real-time systems are examined, essential object-oriented principles and concepts are surveyed, observations on the value of graphical notations are presented, and the utilisation of formal methods in software development is discussed. In *Chapter 3, Background: Notations*, the focus is shifted from general concepts to the two particular specification languages employed in our integrated approach. A description of the specification language Z is given, together with a short presentation of some of Z’s variants. In particular, Z++, the object-oriented variant of Z used in the proposed approach is briefly introduced. Also, an overview of UML, including its capability for modelling real-time systems, as well as a look on UML’s perspectives are included. A survey of reported research that is similar to ours is taken in *Chapter 4, Related Work*. In this chapter, the major ways of integrating informality with formality in the specification phase are identified, related approaches focused on real-time systems are examined, and existing ways of dealing with time in Z-based approaches are discussed. Details on the formal resources employed for dealing with time in a rigorous manner are presented in *Chapter 5, Formal Specification of Temporal Constraints*. This chapter includes a section on the major types of timing constraints that are considered when modelling time-constrained systems and gives details on the specific RTL constructs employed for capturing time-related properties of the systems. Details on the translation processes from UML class diagrams to Z specifications, including the automated formalisation of classes, relationships, and state diagrams are given in *Chapter 6, Translations between UML and Z++: Formalisation and Deformalisation*. Guidelines for completing the reverse translation, from Z++ to UML, are also suggested in this chapter. *Chapter 7, A Procedural Frame*, brings the translation mechanisms proposed in the previous chapter under the methodological umbrella of a complete modelling approach. The proposed dual (semi-formal and formal) modelling process is detailed through a series of steps organised in stages,

in each step a set of artefacts being produced, making up the combined diagrammatic and formal model of the system. An illustration of applying the proposed dual, integrated approach to modelling time-constrained systems is provided in *Chapter 8, An Application: The Case of the Elevator System*. Since any new methodological approach for software development is best served by an accompanying tool, *Chapter 9, Towards an Integrated Environment: A Prototype for Harmony*, presents the GUI-centred design of the software specification environment that we have envisaged as supporting tool for the proposed modelling approach. Finally, *Chapter 10, Conclusions*, analyses the merits and limitations of our approach, presents a summary of our contributions, and opens a window to the future by pointing to a series of connected research directions that we believe deserve further investigation.

1.7 Chapter Summary

In this chapter we have taken a view on the big picture, that of today's computer-related technologies, and introduced the larger scene of our research. We have explained the motivations of our endeavor, pointed out the major challenges related to our work, and outlined the proposed dual, integrated formal/semi-formal software specification approach. This approach, aimed at the development of time-constrained systems, has the main goal of harmoniously integrating graphical (semi-formal) and mathematical notations in a theoretically sound, yet friendly, flexible, and easy-to-use software specification methodology. An overview of the chapters that follow has been presented as well. In this initial chapter a brief analysis of three major paradigms that pervade today's software development world was also included. We believe that the foundation for sound, effective improvements of software development methodologies resides in the right combination of the three paradigms, object-orientation, formal specification, and visual representation. At the convergence of these powerful paradigms we place the topic of our thesis.

2 BACKGROUND: CONTEXT AND CONCEPTS

“You must pin down the butterfly of time.”

[Michael Jackson, *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*, Addison-Wesley, 1995, pp. 78]

2.1 Introduction

In this chapter the research space and the coordinates of the thesis' topic are defined using a classification based on 'domains and 'sub-domains' of exploration and the major aspects of the larger framework in which we have undertaken our research are overviewed. By analysing the distinctive features of real-time (or, in our vocabulary, time-constrained) systems, object-oriented modelling, and formality in software development the larger contour of our work is drawn. The main characteristics of real-time systems are analysed with the dual intent of establishing the context of the present research and of identifying specific challenges of capturing temporal properties of systems. The impact of the object-oriented paradigm on the software development process is also discussed and the value of graphical notations is emphasised. Some of the most significant aspects of employing formal notations in various phases of the software life-cycle are examined, and arguments pro and contra this employment are reviewed. As part of the examination of formality and formalisms, the newer category of light formal methods, which circumscribes our approach, is briefly discussed. Thus, Chapter 2 sets the scene for a closer look (in Chapter 3, "Background: Notations") at the two specification languages used in our approach, one formal (Z) and the other graphical, semi-formal, and object-oriented (UML). The same scene is then used in Chapter 4, "Related Work," to identify existing research approaches that are situated in the vicinity of our topic's location.

2.2 Research Space and Topic Location

The research space that encompasses the topic of the present thesis can be described by considering three *domains of exploration* (Table 2.I and Fig. 2.1). For simplification, in the case of the first domain, which characterises the formality of a modelling approach, only its ‘formal’ *sub-domain* is considered. The second domain describes the methodological paradigm used for software development, which can be either object-oriented or non object-oriented. The third domain classifies approaches as having or not having RT modelling capabilities. Within each of the three domains of exploration a number of sub-domains (*areas*) of interest can be further delimited according to various criteria. We have been interested in specifying the Z, Z++, and UML “dimensions” of a given approach, hence the classification in Table 2.I, which distinguishes areas denoted A, B, and C in the first domain, 1, 2, and 3 in the second, and • and + in the third.

Table 2.I Classification of Research Approaches Based on Domains of Exploration

Formality Domain	Methodology Domain	Real-Time Domain
Area A [non-Z]: Formalism involved, but not Z-centred	Area 1 [non-OO]: Not an object-oriented methodology	Area • [non-RT]: No RT modelling capabilities
Area B [Z but non-OOZ]: Formalism involved, Z-centred, but not OOZ	Area 2 [OO, non-UML]: An OO methodology, but UML not involved	Area + [RT]: RT modelling capabilities provided
Area C [OOZ]: Formalism involved, and an OO version of Z used	Area 3 [UML]: An OO methodology that uses UML	N/A

The 18 possible combinations of areas from the three domains provide a classification scheme in which a given approach has a class between A1• and C3+ (with the exception of classes C1• and C1+, which do not make sense because an OOZ notation can be used only in conjunction with an OO modelling strategy). This sharp delimitation of domains involves a certain simplification, since things are almost never purely “black or white” (formal or categorically non-formal, for instance), but it nevertheless serves well our localisation purpose. Based on the classification presented in Table 2.1, a graphical representation of domains can be drawn, as presented in Fig. 2.1.

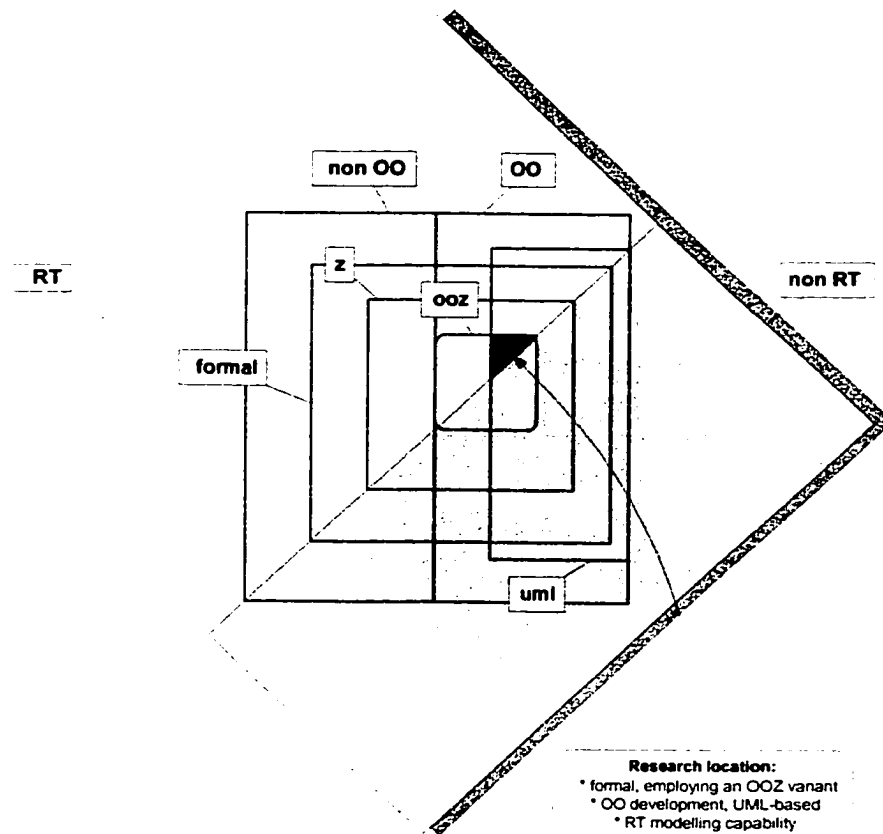


Fig. 2.1 Domains of Research Space and Topic Location

The figure indicates that our class C3+ approach is placed at the intersection of the three major domains described above, and also enjoys the special characteristics provided by its

“OOZ-centred” and “UML-based” dimensions. Before moving to investigate further the three domains defining the research space and discuss the details of the Z and UML notations used to demarcate areas in these domains, several comments are necessary. Firstly, to keep the figure simple, the areas in Fig. 2.1 have not been textually labelled as indicated in Table 2.I, but the identification of specific classes, e.g., B2●, should be straightforward. Secondly, the graphical representation of the abstract topology presented in Fig. 2.1 is not intended to reflect the proportionality of existing approaches (for instance, there is no intention on our part to claim a 50-50 distribution between OO and non-OO approaches, as the figure might suggest). Thirdly, the classification presented in Table 2.I and its depiction shown in Fig. 2.1 will be used again in Chapter 4, where a survey of related approaches is presented.

2.3 On Specifying Real-Time Systems

2.3.1 Characteristics of Real-Time Systems

In today’s fast evolving world of computing, real-time systems are taking an increasingly important role and are extending their reign over a growing number of application domains. Real-time systems prove to be useful in many areas of human activity: numerous commercial, industrial, medical, and military products that must pay careful attention to the precious resource which is the time are used on daily basis. As pointed out by Stankovic, “a real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced” ([Stankovic96b], pp. 751). In a similar way, Everett and Honiden indicate that “a real-time system must respond to externally generated stimuli within a finite, specifiable time delay” [Everett95, pp.13]. Severe consequences may result if timing as well as logical correctness properties are not satisfied. Based on the severity of consequences, the real-time systems can be classified as *hard real-time systems*, where the failure of meeting the deadline can result in an important loss (including loss of human life, injury, and/or major equipment damage), or *soft real-time*

systems, in which the deadline can be occasionally missed, but the utility of the result decreases after the deadline [Burns97, Kopetz97]. Some authors take into consideration an intermediate category, *firm real-time systems*, which in essence can be described as having a shorter soft deadline and a longer hard deadline [Douglass98] (Fig. 2.2 presents a summary characterisation of the three types of real-time systems based on a generic utility-time function.) Hard real-time systems encompass aircraft controllers, process control systems, factory robots, traffic lights controllers, and medical devices such as heart pacemakers, while examples of soft real-time systems include automatic banking machines, ticket reservation systems, general-purpose communication systems, and embedded commercial products such as television sets and videocassette recorders. An example of firm real-time system is that of a patient ventilator system, in which an occasional late breath in the range of few seconds is tolerated, while a several minute delay is catastrophic [Douglass98].

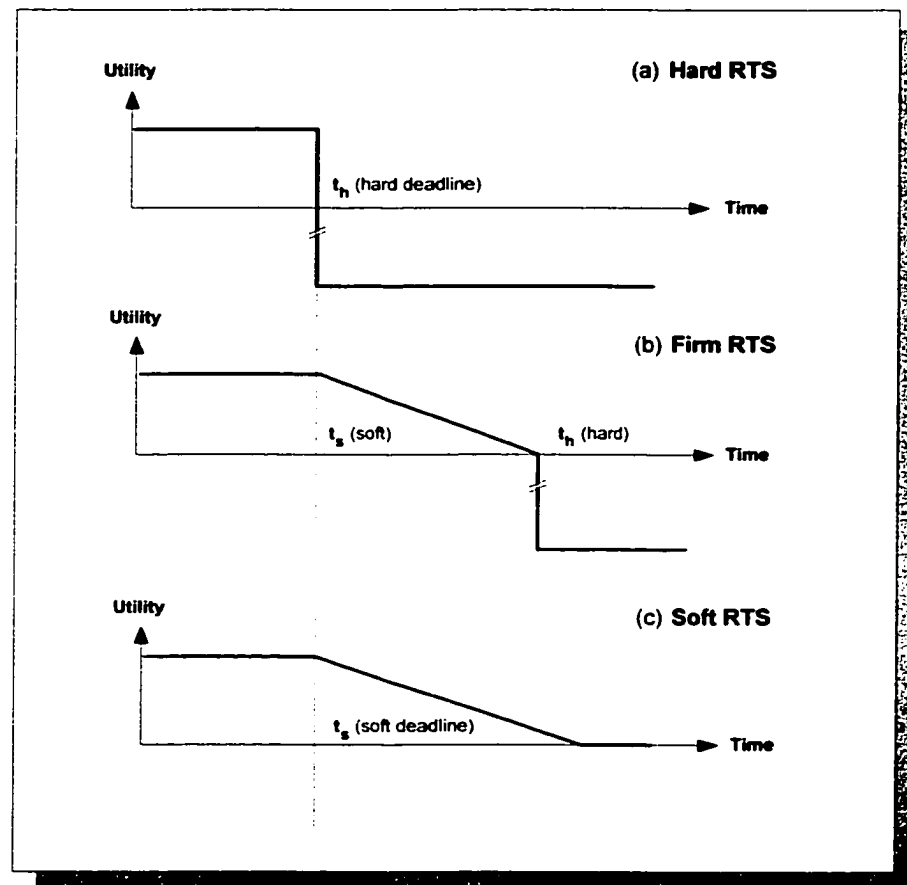


Fig. 2.2 Hard, Firm, and Soft Real-Time Systems

As pointed out by Stankovic, most of the activities of real-time systems have to occur in a timely fashion, but some non time-critical activities also coexist. The former activities are referred to as *real-time tasks* (or *time-critical tasks*) while the later can be simply called *tasks* [Stankovic88]. Timing constraints on tasks can be *periodic*, if activated every T units of time, *aperiodic* if activated at unpredictable times, or *sporadic*, if they aperiodic behaviour is further restrained by a minimum interval of time between activations [Stankovic96a]. The complexity of designing RTS also arises from additional types of constraints and requirements such as resource constraints, concurrence constraints, precedence relationships, placement constraints, communication requirements, criticalness. A real-time system differs from a traditional system (non real-time) in at least the following aspects: deadlines are attached to some or all of the system's tasks, faults in the system –including timing faults– may lead to catastrophic consequences, the system should have the ability to deal with exceptions, the system must be fast, predictable, reliable, and adaptive [Stankovic88]. Lin and Burke show that RTS are very difficult to debug and modify, and –since there are always demands for new functions and configurations– they must be easy to change and reconfigure [Lin92]. Other authors also point out that the design of real-time software is resource-constrained, the software itself is intricate and contains highly complex time critical parts, and the real-time software should be able to detect the occurrence of failures [Natarajan92, Everett95]. Everett and Honiden show that “development of most software focuses on how to handle a normal situation, but real-time, critical-application development also focuses on how to handle the abnormal situation” (Everett95, pp.15). And, unfortunately, as Gibbs points out, “errors in real-time systems ... are devilishly difficult to spot because, like that suspicious sound in your car engine, they only occur only when conditions are just so” [Gibbs94, pp.88]. In short, as noted by Douglass, RTS “must operate under more-severe constraints than ‘normal’ software systems yet perform reliably for long periods of time” [Douglass99, pp. 57].

In what follows, we give a more detailed account, albeit not exhaustive, of characteristics pertaining to RTS and analyse their implications on the design of a dedicated specification approach. Of course, it is rather difficult to find an example that exhibits all the properties

listed below, and it will be a massive task, if not impossible, to develop a specification method capable of rigorously handling all these properties. In fact, in Subsection 2.3.2 we focus on a reduced number of capabilities we have aimed to include in our modelling approach, but at this point it is useful to have a closer look at the impressive complexity of the RT domain. The starting point of our selection has been the list of requirements for specification languages presented by Narayan and Gajski in [Narayan93] and further analysed by Narayan in [Narayan96]. Their list of requirements is concerned, however, with the more restricted category of embedded systems so we have resorted to additional references in order to describe the larger class of real-time systems.

- *Timeliness.* The essential characteristic of RTS is that deadlines are imposed to some or all the tasks of the system. Timeliness is part of the definition of a real-time system [Douglass99], such system being required to work under predefined temporal constraints and correctly react to stimuli from its environment “on time” [Selic94]. In specification terms, the modelling notation should incorporate a time metric, as well as facilities for expressing both relative and absolute timing constraints;
- *Reliability.* One of the most imperative requirements placed on RTS, particularly on hard RTS, is that of reliability. Due to the gravity of the potential damages that can result as a consequence of a real-time system failing to function correctly, additional measures must be taken into consideration. As pointed out by Nancy Leveson, the vast majority of software faults have roots in incorrect specification [Leveson86], therefore the specification languages and techniques employed in the development of RTS must provide adequate support for incorporating reliability measures and for assessing the system’s safeness. Essentially, means to deliver specifications that are complete, consistent, comprehensible, and unambiguous are necessary [Burns97]. The use of formal techniques is required, at least for the security and safety-critical parts of the system;
- *Intensive dynamics.* Due to the typically intensive dynamics of RTS, modelling the states of such systems is an essential requirement for a dedicated specification language. Diagrammatic notations with solid mathematic foundation have been proposed (among

the most notable Petri Nets [Petri62, Reisig85] and Statecharts [Harel87]) and proved to be extremely valuable for specifying the dynamic behaviour of RTS. In particular, finite-state machines have been used successfully in various phases of the software development process [Avnur90, Ding93, Harel96];

- *Input/Output.* Obviously, due to the continuous interaction with the environment in which a real-time system typically operates, an adequate set of symbols and operators for describing input/output operations should be included in the specification language;
- *Exceptions.* Real-time systems must react promptly to stimuli from their environment or to internal events that necessitate immediate attention. Some events, external or internal, are more important than others, and taking appropriate measures in response to critical situations is a strict requirement for such systems. Exception and interrupt handling are inherent in the implementation of RTS and it is desirable to have them described at the specification level;
- *Concurrency.* Even though concurrency is not part of the definition of RTS, many such systems exhibit concurrent behaviour. Moreover, as pointed out in [Shaw92], due to the very nature of RTS it is not sufficient to model only the system, it is also necessary to capture the environment in which it operates. And, this environment is inherently concurrent, with multiple sources of stimuli that influence the behaviour of the system. Consequently, a specification language for RTS should provide appropriate support for expressing concurrency;
- *Distribution.* As in the case of concurrency, distribution is not necessarily a characteristic of RTS, but it is nevertheless impossible to ignore it, at least in the case of large-scale systems. Capturing the distributed nature of a complex real-time system is becoming a necessary feature in these days when the Internet and the World-Wide Web have obtained the status of common nouns. However, the task of expressing both concurrency and distribution is very demanding [Douglass99];
- *Communication and synchronisation.* One cannot possibly imagine a useful system in which various software components do not communicate and synchronise, more so in a real-time system that is required to be both concurrent and distributed. Clear description

of communication and synchronisation is necessary and the specification notation must provide constructs and mechanisms to support it;

- *Resource allocation.* Because many RTS are also distributed, it is desirable that facilities for describing allocation of resources should be included in a specification language that aims at modelling such systems;
- *Size.* RTS are not only special in their dealing with time but in many cases they are also large and complex, involving numerous processes and threads, as well as a significant number of input/output variables. Size alone is obviously an element that affects the development of a system, but in the case of RTS a complicating factor is that largeness is inherently associated with continuous change, so provisions for extensibility should be built in the design of such systems [Burns97]. Both structured and object-oriented methods provide means of dealing with increasingly more demanding requirements on size; modules, classes, components, and patterns are typical solutions for dealing with large-size software products. Operators for expressing composition and decomposition, as well as mechanisms for modelling hierarchical structures are necessary;
- *Non time-constrained activities.* Although it would appear that non time-critical activities should not be deemed an issue, it has been shown that incorporating such activities in the development of RTS may prove to be a complicating factor. The most common problem raised by non time-constrained activities is that a worst-case execution time for them (e.g., the answer from a human user) cannot be easily evaluated [Audsley96];
- *Computations.* Typically, RTS must continuously interact with their environment and provide appropriate response under conditions imposed by the environment. The computation of the system's response can be complex, for instance in the case of process control systems, which involve solving systems of possibly complicated differential equations. Consequently, the implementation of RTS requires the ability of manipulating real, fixed or floating-point numbers [Burns97]. This translates into a requirement for the specification language, which must be able to handle both quantitative and qualitative intricacies of RTS;
- *Data modelling.* RTS are the most complex type of systems –as put by Alderson et al., they “have proved troublesome to produce, with all the difficulties of the other kinds of

software-based systems together with a number of specific additional problems” [Alderson98, pp. 442]. Among the traditional difficulties, data modelling is a challenging issue if not for all, but for an increasing number of time-constrained systems. In fact, a branch of time-constrained systems is that of real-time database systems (RTDBS), in which both timely response and the ability to manipulate data that has temporal validity are required [Lin94]. In this respect, solutions to unambiguously specify aspects such as relationships between consistency constraints and timing constraints, the validity of external data consistency, abstractions for data, and also data transformations are needed [Sahraoui97];

- *Reuse*. As pointed out by Mrva, the real-time systems, particularly the embedded systems, appear to be poor candidates for reuse [Mrva97]. This can be explained by the fact that most of the RTS are specialised, typically required to resolve needs of a rather particular nature. Reuse seems hard to achieve with RTS for the simple reason that rarely two applications exhibit more than limited similarity. However, as indicated by Mrva, reuse is not only desirable but also possible within the realm of such systems and the major factors on which the reusability value of a real-time system depends on are the frequency and the utility of reuse, which are related to comprehensibility, habitability (measures how “at home” a potential user feels with the reusable components), and independence of components with respect to their environment [Mrva97]. The object-oriented paradigm offers an avenue of investigation for the designers of RTS, together with the newer pattern-based techniques;
- *Animation/execution*. The need for animation is advocated by many authors who stress the importance of rapid-prototyping and early client feedback in the development of RTS. Animation of specifications is generally desired, because it can provide a rapid feedback to the designer and facilitate a better understanding of the system’s requirements. Although the more ambitious goal of an animation system is to generate a full-scale prototype or even a complete implementation of the system being developed, animation can be used interactively for immediate exploration purposes: the consequences of a specification can be evaluated dynamically, during the composition of

the system's specifications, thus allowing the refining and optimisation of specifications [Utting95].

2.3.2 Focus On Time

Obviously, the real-time domain is very complex and very demanding. The approach we take is to tackle some of its complexity and deal with several of the aspects mentioned above. Since the defining property of a real-time system is timeliness, we decided to focus on expressing temporal properties of the systems at the specification level. Because of this, and for reasons outlined in Subsection 1.4, we use *time-constrained systems* (TCS) as the preferred term in denoting the systems our approach is focused on, although when needed (primarily, for referencing purposes) the traditional RTS denomination is also used in this dissertation.

Our “selection of emphasis” has also been based on the observation that while timeliness is a characteristic of both hard and soft real-time systems if we speak about TCS (as opposed to RTS) more stringent (“harder”) requirements placed on these systems, such as reliability and safety, are gently pushed towards the background. The intention, of course, has not been to ignore such demanding requirements, but to come up with a “more popular,” more pragmatic specification approach that would appeal to both software developers and users and would not scare them away by suggesting an emphasis on the more difficult (and less “popular”) subclass of complex safety-critical applications. And, while our method can address the modelling of hard RTS (e.g., traffic lights controllers), it is only fair to say that “really hard” RTS –if we may introduce this distinction– such as aircraft autopilot controllers or nuclear process control systems would need supplementary treatment, provided by some additional techniques and tools. On the other hand, while we acknowledge our approach's focus to the “softer side” of RTS (in fact, not necessarily soft, since it could be either “lighter hard”, “firm,” or indeed “soft”!) we note that the term TCS has an additional advantage: it covers both *reactive* (or *event-driven*) systems and *time-based* (or *time-driven*) systems

because timeliness is part of both of them. (If it were to speak simply about time-based systems it would have meant that we address only systems whose behaviour is driven by the passage of time or the arrival of time epochs [Douglass99], and this would have been somewhat too restrictive).

Our emphasis on timing properties is illustrated by the fact that the starting point in the design of our approach has been provided by the archetypical classes of temporal constraints identified in [Dasarathy85] and that Real-Time Logic (RTL) [Jahanian86, Jahanian94], which offers very good support for expressing both absolute and relative timing properties, has been included in the proposed integrated specification method (more details are provided in Chapters 5 and 6). In terms of the characteristics of RTS discussed in the preceding subsection, the approach presented in this thesis can be summarily described as follows: it places primary emphasis on timeliness, provides a good modelling coverage of intensive dynamics, input/output, exceptions, non time-constrained activities, computations, data-modelling and reuse, offers a fair support for dealing with concurrency, communication, synchronisation, and size, and does not address distribution, resource allocation, and animation/execution. In what regards reliability, the formal basis is here, with Z++ and RTL its pillars, but the particular specification approach we propose here need be complemented by analysis techniques that have been left outside the scope of the present dissertation.

2.4 Brief Immersion in Object-Orientation

2.4.1 On Objects and Their Modelling Power

Over the years, the structured paradigm proved to be less effective than initially thought. By mid-eighties, the practitioners in the field became aware that it did not live up to earlier expectations, particularly in two major respects: it did not cope well with the increasing size of modern software products and did not support adequately the maintenance of such products [Schach99]. As indicated by Schach, the essential limitation of the structured paradigm is that its approaches for software development are either action-oriented or data-

oriented, but not both. In response to this situation, a new alternative, soon to be known as *object-oriented*, emerged with remarkable power. Although an important breakthrough in software development, the apparition of the new approach was not spontaneous, but the cumulated result of the work of many scientists and developers [Booch94]. The origins of some concepts that helped shape the new approach can be traced back to as early as the 1960s, most notably to Dahl and Nygaard (the class construct in Simula67), and to Alan Kay, Adele Goldberg and their team at the Xerox Palo Alto Research Center, California (messages and inheritance in Smalltalk) [Page-Jones99]. Other major contributors, according to the same author, include Larry Constantine (coupling and cohesion), Dijkstra (layers of abstraction), Barbara Liskov (abstract data types), David Parnas (information hiding), Jean Ichbiah (packages and genericity in Ada83), Bjarne Stroustrup (C++), Bertrand Meyer (Eiffel), Grady Booch, Ivar Jacobson, and James Rumbaugh (OOA, OOD, and UML). To these, we have to add Peter Chen, whose ERD (Entity-Relationship Diagrams) contribution [Chen76] is a recognised source of inspiration for object-oriented approaches. And, interestingly, if we follow Kouichi Kishida's observations and look carefully we can find precursors to OO even in ancient times (Confucius) as well as in the 19th century (the German philosopher Max Weber) [Kishida96]! In fact, this should not be so surprising, since in his survey of the foundations of the object model, Booch also makes references to ancient philosophy, Greek in his case, as well as to Descartes [Booch94, pp. 36-37].

The newer approach, the object-oriented paradigm, is founded on the concept of *object*, which can be defined concisely as “a unified software component that incorporates both the data and the actions that operate on that data” [Schach99, pp. 17] or as “a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand” [Rumbaugh91, pp. 21]. More completely, an object is “an entity that: has state; is characterized by the actions that it suffers and that it requires of other objects; is an instance of some (possibly anonymous) class; is denoted by a name; has restricted visibility of and by other objects; may be viewed either by its specification or by its implementation” [Booch86, pp. 215]. The internal structure of an object is described by *attributes*, and *messages* can be sent to an object to invoke one of its *methods* (or *operations*) –that is, to invoke actions that

generally operate on the internal structure of the object. Typically, we ignore many details of objects, and are concerned mostly with ways of manipulating them through operations. In software, the notion of *object* covers tangible things (such as book, floor, door, or thermometer), persons (e.g., student, teacher, employee), roles (e.g., dispatcher, supervisor, controller), events (e.g., take-off, interrupt, shutdown) and an infinite variety of other things (e.g., proposals, meetings, poetic ideas, eulogies, referrals, rebuttals, etc.). In OO terminology a *class* is a template for objects that have similar features, more precisely the objects belonging to the same class have the same structure and the same behaviour (e.g., `raymondsAlarmClock` is an object of the class `AlarmClock`). A *class* can be seen as an abstract data type that supports inheritance.

Three major principles are promoted by the OO paradigm:

- *Encapsulation*, the defining principle of object-orientation, which signifies putting together in a single unit of both data and operations pertaining to some entity that can be qualified as an object (or, to be more precise, as a class, the “blueprint for creating objects” [Mughal00, pp. 2]). Encapsulation supports abstraction and information hiding, key ingredients for developing high-quality software products;
- *Inheritance*, the mechanism of creating a new class from existing ones and the provider of the strongest foundation for reuse;
- *Polymorphism*, essentially an instrument for abstraction and an enhancer of flexibility, with its meaning taken from the Greek equivalent of “having multiple forms,” and used in the OO world with the significance “same name for different behaviours.”

The major breakthrough brought by the OO approach comes from the fact that the conceptual and physical independence of components reduces the level of complexity of software. Thus, both development and maintenance are simplified [Schach99]. Among the most important benefits of the OO approach we would nominate:

- Greater modelling power, since objects correspond more naturally to real-world entities and as such the problem domain is better described;

- Increased code reusability and extensibility, due to encapsulation and inheritance, which offer strong support for code reuse and product extension;
- Improved control of complexity, mainly through abstraction, information hiding, and localisation –the management of complexity is helped since the emphasis is on interfaces and interactions among independent, collaborating entities (objects).

These benefits, together with a series of other advantages of the OO approach, such as production of software more resilient to change, greater level of confidence in the correctness of software through separation of its state space [Booch94], greater stability of designs over time, more flexible and adaptable development, and easier transition between the development phases [Johnson00], have lead to a proliferation of OO techniques and tools for software construction. Of course, there are less beneficial aspects of OO development that the software professionals are aware of, most significantly longer initial development time, decreased run-time performance, and unavailability of adequate OO DBMS, but overall the newer approach has gained the confidence of the software development world [Johnson00]. And, while there are some isolated opinions that the OO paradigm is only a “hype,” possibly less effective than the structured one [Niemann99], and some scientists have even proclaimed its impending demise [Davis98], we share Bertrand Meyer’s position that “OO solutions are our best bet” and, in fact, “it’s the only game in town” [Meyer99, pp. 144], the newly emerged component-based development actually assuming and making use of the OO technology.

2.4.2 Object-Orientation in the Real-Time Domain

For reasons mentioned in Subsection 1.1.2, the “conqueror objects” have only relatively recently expanded over the real-time domain. However, as the OO technology has matured, the focus of numerous scientists has shifted towards tackling the complexity of real-time applications via the OO avenue. Currently, there is a significant amount of work in this direction, and a number of important methods and methodologies have been proposed, among the most notable ROOM [Selic94, Selic96], TRIO [Bucci94, Ciapessoni99],

Octopus [Awad96], and Comet [Gomaa00]. The considerable attention currently paid by researchers and developers to the application of the OO techniques to the development of RT software is both indicative of the economical importance of RTS and illustrative for the general recognition of the OO paradigm's modelling prowess. And, there is probably no better illustration for the current concerted effort in this direction than the development of powerful dedicated commercial tools such as I-Logix Inc.'s Rhapsody [Rhapsody01] and Rational Software Corporation's Rational Rose Real-Time [RationalRoseRT01]. In addition, the hottest general OO programming language of the moment, Java, has recently enhanced its support for RT applications through the definition of the preliminary version of the Real-Time Specification for Java (RTSJ), expected by E. Douglas Jensen "to become the first real-time programming language to be both commercially and technologically successful" [Bollella00, pp. xxi]. With strong research directions and important programs such as OMG's Real-Time Analysis and Design Initiative [Selic99a], major advances in the development of industrial-use IDEs, and considerable RT support from an OO language such as Java that is used by a large number of programmers, the trend is obvious. We can safely assume that it will continue strongly in the foreseeable future.

2.5 On The Importance of Graphical Notations

It has been mentioned in Chapter 1 that today is almost impossible to create a viable software development tool without an adequate GUI interface. The provision for an easy-to-use, friendly and functionally complete graphical interface is not simply a trend of the moment but a stringent requirement for any development tool intended for practical use. Although it might seem like a futile argumentation, it is nevertheless useful to stress the importance of visual interfaces in such tools. And perhaps there is no better way to emphasise this idea than by paraphrasing David Taylor who, in a recent article, recalls the following prediction he made more than 15 years ago about the OO paradigm: "by the year 2000 no one would talk about objects any more because the technology would be so thoroughly absorbed into the mainstream that no one would think to mention it" [Taylor99, pp. 50]. While we share Meyer's position and question the accuracy of Taylor's affirmation in the

OO context (Meyer considers that several more years are still needed before Taylor's affirmation can be fully supported [Meyer99]), we believe that this is truly the case for graphical interfaces in the context of software tools. Thus, we can state that they are here for quite a while and practically taken for granted, so "nobody would think to mention them." In fact, animation and multimedia capabilities are an important part of our interaction with the computers and they are expected to have an increasingly larger presence in modern professional tools, so perhaps discussing GUI advantages runs the risk of obsolescence.

But it is not only the graphical user interface we are referring to; the use in our approach of UML, defined as a "visual modelling language" [Quatrani98], corresponds to another reality, that of the need for visual notations in analysis and design. In Chapter 1 the motivations for a combination graphical notation (semi-formal in our case) with a formal language for software specification have been presented and while we do not intend to discuss here visual languages and environments in general, we refer nevertheless to [Green96] for a complete list of cognitive dimensions that can be used to evaluate the benefits of visual notations, including closeness of mapping, abstraction gradient, role-expressiveness, consistency, progressive evaluation, and visibility. Also, for a thorough rebuttal of some common objections to the use of visual representations in the computing process we refer to [Cox93]. However, the task is simpler in our case, since OO methodologies have been traditionally supported by graphical notations, and it is quite hard today to imagine such a methodology without an accompanying set of graphical symbols for classes, relationships, collaboration diagrams, etc.

In our opinion, the use of visual representations, as opposed to simply employing text, is strongly justified by enhanced support for abstraction, better representation of information in terms of structures (components and their relationships), increased expressiveness (richness of information content), simpler syntax, capability for direct manipulation, and increased naturalness (which facilitates communication).

Many scientists have acknowledged the advantages of visual notations in software development by adding a "visual dimension" to their specification approaches, for instance

Buhr's diagrams for the design of Ada applications [Buhr90], Dillon et al.'s Graphical Interval Logic (GIL) aimed at representing the temporal evolution of concurrent systems' properties [Dillon94], Roman et al.'s custom built Pavane visualisations for capturing formally expressed specifications and designs [Roman96], and Taentzer's visual rules for declarative specification of behaviour in OO modelling techniques [Taentzer99]. In the "Z area" an interesting approach is the one taken by Kim and Carrington who, based on Kent's Constraint diagrams [Kent97] and Kent and Gil's Contract Box notation [Kent98] propose 3D visualisations of Z expressions to facilitate the understanding of specifications [Kim99b].

It is also important to note that visual notations are not necessary semi-formal (or informal) because when accompanied by precise semantics they fit in the class of formal notations (this is the case, for example, of Petri Nets and Statecharts, two powerful techniques used for modelling specific aspects of RTS). But even in the case of more general semi-formal notations such as DFD (Data Flow Diagrams), ERD, or UML the expressive power provided by their graphical representation is of considerable help during the development process.

And, to conclude the case for graphical notations, perhaps apparently a minor aspect, but nevertheless solidly backed by its acceptance in practice is Together Soft Corporation's inclusion of colours in the modelling process [TogetherSoft00b]. Colours and other elements of visualisation are, in our opinion, great enhancers of productivity in developing software products.

2.6 Formal Notations in Software Development

2.6.1 Alexander's Definition of a Formal System

A clear and concise definition of a formal system can be found in [Alexander95]. The author uses the following terminology (key terms are highlighted by us using italics):

- A *formal system* consists of a *formal language* and a *deductive apparatus*;
- A *formal language* has two essential components: an *alphabet of symbols* and a *set of grammar rules*;
- The grammar rules are used to construct *well-formed formulas*;
- A *deductive apparatus* is a set of *axioms* (basic truths) plus a set of *inference rules* (e.g., substitution, simplification, expansion rules);
- The inference rules produce a well-formed formula from other well-formed formulas; the deductive apparatus also provides means to establish whether a well-formed formula is a direct consequence of another;
- To apply a formal system to a problem, the formal system must be given *semantics*, which in essence provide a mapping between objects in the problem domain and well-formed formulas in the formal language;
- With the semantic mapping established, the formal system can be used to create a *formal model* of “known characteristics” of the problem domain.

As pointed out by Alexander, software systems requirements describe the desired behaviour of a system within its operational environment. In essence, the execution of a software artefact can be described formally by a precondition $I(x)$ and a post-condition $O(x,z)$, where x is the input of the execution and z is its output. In short, when $I(x)$ is true, the execution of the software artefact generates z , which satisfies $O(x,z)$. The key issue in software development is to find some *program* $P(x)$ that produces z under the conditions stipulated by the pre-condition I and the post-condition O . This process of determining an appropriate $P(x)$ is complex, and requires successive refinements, each producing a more concrete model of the system (the starting point being a high-level model of the requirements). Each refinement involves two fundamental processes, *synthesis* and *analysis*. Alexander points out that in general both synthesis, the creation of a new model of the system, and analysis, the verification of the model with respect to the original model, can be reliable only if formal models are employed. Semi-formal models are unable to predict or verify most of the system’s characteristics.

According to the same author, “a *software specification* is a model of a developing software system” and “*formal specification* is representing software specification using a formal model” [Alexander95, pp. 30]. The “foundations picture” drawn by Alexander can be extended with a couple of definitions proposed earlier by Jeannette Wing:

- “A *formal specification language* provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification” [Wing90, pp. 10];
- Formal specification languages supply the mathematical basis for *formal methods*, which are “mathematically based techniques for describing system properties” [Wing90, pp. 8].

2.6.2 Classifications and Examples of Formal Methods

The usual way of classifying formal methods is based on the traditional model-oriented versus property-oriented criterion. The distinction between these two categories of methods stems from the way the behaviour of the system is defined, directly or indirectly. A *model-oriented* method directly describes the behaviour of a system in terms of sequences of states (each state being characterised by a set of instance variables) and operations that can cause state transitions. The *property-oriented* methods can be further classified as *axiomatic* or *algebraic*, depending on their underlying mathematical foundation (first order predicate logic or many sorted algebras). In both cases, property-oriented methods define the behaviour of the system indirectly, via a set of properties usually expressed as axioms that the system must satisfy [Wing90]. In their comprehensive survey of formalisms Liu and Zedan propose a more refined taxonomy by identifying five classes of formal methods, specifically *model-based*, *logic-based* (logics are employed to express the desired properties of the systems, including temporal and probabilistic behaviours), *algebraic*, *process algebra-based* (differ from algebraic by supporting explicit representation of concurrency), and *net-based* (graphical notations with precise formal semantics) [Liu97]. On the same topic, Gaudel points out that a finer distinction between formal methods can be made by using additional criteria, specifically [Gaudel94]:

- Their level of formality –the methods can be classified according to three key terms, namely ‘formalised,’ ‘conceptual,’ and ‘deductive’. Formal methods are obviously formalised but the degree of the notation’s formalisation and the potential of performing various types of checks are different from method to method. Similarly, different techniques emphasise in various degrees their capability of modelling conceptual aspects of systems and exhibit deduction systems of various degrees of complexity;
- The life-cycle stages where the techniques are applied. The classification encompasses activities such as domain specification, requirements engineering, design by refinement, proof of correctness, software re-engineering, and reuse;
- The specific aspects of computing they address. Algebraic methods are focused on describing abstract data types in an implementation-independent manner, model-oriented techniques aim at explicitly dealing with the dynamics of state-based systems, while other approaches address aspects specific to reactive and distributed systems, such as communication and concurrency;
- The mathematical foundation on which they are based, in terms of conceptual framework and deduction system. The conceptual foundations include process algebras, automata, set theory, and partial functions, while the deduction systems can be based on first-order predicate logic, higher-order logic, temporal logic, etc.;
- The methodological apparatus accompanying the method. Typically, this may consist of data tool kits in the case of model-oriented techniques, or may be provided as a kernel for property specification in the case of algebraic or axiomatic methods.

Some of the most representative formal methods are, in alphabetical order, Abrial’s B-Method [Abrial96], Hoare’s CSP [Hoare78, Hoare85], Milner’s CCS [Milner80], ITL (Interval Temporal Logic) [Moszkowski86], Larch [Gutttag85, Gutttag93], LOTOS [ISO89], Petri Nets [Petri62, Reisig85], RTL (Real-Time Logic) [Jahanian86], RTTL (Real-Time Temporal Logic) [Ostroff89], Statecharts [Harel87], Temporal Logic [Rescher71, Pnueli77, Manna81], VDM [Jones90], and Z [Spivey92]. A large variety of environments and tools have been developed to accompany the existing formal methods and a significant number of

extensions and variations have been proposed. Several notable variants and tools pertaining to the “Z sub-domain” are discussed in Subsection 3.2.2 of this thesis.

2.6.3 Advantages and Disadvantages of Formal Methods

There has been a fair amount of debate over the applicability of formal methods in practice and especially over their potential of becoming working instruments for the large community of software developers. The attitudes vary from strong skepticism [Lawrence96, Glass96] to resolute conviction [Hall90, Meyer97, Kapur00], with many views within the range delimited by the above positions. We note however that the tendency is to recognise the benefits of formality in software development, but to caution also about its perceived disadvantages.

In what follows we present a summary of both benefits and disadvantages of applying formal techniques but not before mentioning that precisely the intricacies of these techniques prompted us to decide on the fundamental theme of our thesis, that of integrating formality with semi-formality in software specification.

The main reasons for employing formal methods are related to achieving the following goals:

- Better understanding of the system through formal specification and increased intellectual control [Gerhart94, Sommerville95, Clarke96, Hall96]. Daniel Jackson, in particular, remarkably refutes Brian Lawrence’s opinion that, due to the difficulties associated to their application, formal methods may not be actually needed. Jackson considers that documents written in a natural language cannot be adequate repositories of an analyst’s insights and that the greatest benefits of formalising requirements reside in clarifying ideas, revealing unexpected issues, and providing relevant feedback for the discussion with the client (Jackson’s counterpoint to Lawrence’s opinion [Lawrence96], in [Jackson96a]). Also, Jeannette Wing remarkably notes that “the greatest benefit in

applying a formal method often comes from the process of formalizing rather than from the end result” [Wing90, pp. 13];

- Higher degree of confidence through rigorous verification and property proving, particularly needed for the development of safety or security-critical systems [Sommerville95, Liu97, Schach99, Kapur00];
- Increased customer satisfaction and higher quality of products, including earlier detection and minimisation of errors, as well as enhanced functionality and performance [Gerhart94, Larsen96];
- Improved communication via supplemental notations [Gerhart94, Jackson96a];
- Power of abstraction or, as expressively stated by D. Jackson, “simplicity by omission” [Jackson96a, pp. 21];
- Competitive advantage resulting from applying the best practice [Gerhart94, Kelley-Sobel00];
- Compliance with standards or certification requirements [Hinchey96, Kapur00];
- Possibility of automatic transformation from specification to implementation [Sommerville95];
- Potential for reuse by enhanced identification of commonality [Bowen95a, Jackson96a, Meyer97];
- Educational benefits, including better understanding of research-and-design issues [Gerhart94] and improvement of complex problem solving skills [Kelley-Sobel00].

On the negative side, the following are considered the main disadvantages of formal methods:

- Difficult to use in practice due to their underlying mathematics, perceived by many developers as being hard to master [Gaudel94, Sommerville95, Lawrence96]. Representatively, Stephen Schach lists as weaknesses of formal specification methods “hard for team to learn, hard to use, almost impossible for most clients to understand” [Schach99, pp. 364];

- Lack of supporting tools [Gerhart94, Morgan94, Dill96, Holloway96];
- Not general enough, and not yet sufficiently employed in combination with other methods, formal or informal [Gerhart94, Clarke96, Lawrence96];
- Insufficient formal education and training of developers [Jones96, Hinchey96, Clarke96, Zimmerman00] and lack of educational support, including suitable textbooks [Kelley-Sobel00];
- Slow technology transfer from research to industry [Gaudel94, Glass96, Clarke96];
- Unwillingness of customers to invest effort in acquiring the necessary skills for dealing with formal representations of the systems [Sommerville95];
- Insufficient management support [Sommerville95];
- Inadequate notation, difficult to understand and use [Parnas96];
- Lack of application on significant, complex real-world problems [Holloway96, Dill96, Zimmerman00] and lack of truly impactful, convincing results [Parnas96].

Based on the analysis of a number of negative opinions about formal methods Hall [Hall90] and Bowen and Hinchey [Bowen95b] point out that many of the perceived disadvantages are actually “myths” and aptly dispel these myths with counterexamples and solid justification. Among the typical “myths” (or misconceptions) about formal methods the most common are: they increase development costs, can be applied only to safety critical systems, delay the delivery of the product, require a high level of mathematical skill, and are not actually necessary.

Overall, we share the view of those who advocate the application of formal methods, and believe that the difficulties of learning them are well paid off by the benefits they can bring. On the other hand, we agree with Anthony Hall that they are not a universal panacea [Hall90] and believe that integrating them into a software development approach that combines formality with informality can increase their chances of success in practice. Also, we need not forget that like most other things in life, formal methods should not be overused, otherwise they may turn out to be actual obstacles in the completion path of a

software product. Or, in Bowen and Hinchey's words, "thou shalt formalize, but not overformalize" [Bowen95a, pp. 57].

2.6.4 Formal Techniques within the Software Development Process

As indicated in Subsection 2.6.1, a formal system can essentially perform two kinds of activity, analysis and synthesis. On practical terms, formal techniques can be applied during all stages of formal development. Specific activities include rigorous specification of requirements, specification verification and validation, program refinement from specifications, specification-based testing, re-engineering, and reuse [Wing90, Gaudel94]. As pointed out by many authors, the greatest benefits can be obtained by applying formal techniques in the initial stages of development, when the early detection of errors saves a considerable amount of time and money [Leveson86, Morgan94, Larsen96, Schach99].

2.6.5 A New Trend: Lighter Use of Formal Methods

Recognising the need for a larger acceptance of formal methods, a new direction of investigation has emerged within the last few years, focused on a more pragmatic application of formalisms in software development. In order to increase the use of formal methods in industrial applications, including large-scale projects and applications outside the safety-critical area, cost-effective ways of improving the quality of software have been proposed. In this direction, Jones suggests the use of *formal methods light*, an approach focused on sketching the abstract model of the system, seen as crucial for understanding the architecture of the system, with minimum emphasis on notational details [Jones96]. In the same line of research, Jackson and Wing consider that *lightweight formal methods*, characterised by partiality in language, modelling, analysis, and composition, can bring greater benefits at reduced cost by allowing economically feasible automatic analysis of selected parts of the system [Jackson96b]. The authors' opinion is that the generality of an expressive language such as Z is an impediment for tool-supported analysis while simpler, less expressive, but more "focused" formal methods, can have greater effect in practical applications. An

exponent of the new direction, the lightweight modelling notation Alloy, based on a subset of Z and incorporating a limited number of extra features necessary for object modelling, has been recently developed at the Massachusetts Institute of Technology, together with a supporting tool entitled Alloy Constraint Analyzer [Jackson00a, Jackson00b]. Under the same umbrella of lightweight formal methods, Easterbrook et al. report very promising results of applying, in three NASA projects, “partial analysis on partial specifications, without a commitment to developing and baselining complete, consistent formal specification” [Easterbrook98, pp. 5]. Also, based on two other NASA case studies, Feather concludes that lightweight formal methods are useful for rapid analysis of specifications, yield results in a cost-effective and timely manner, and can be successfully used as complements to other forms of quality assurance [Feather98]. In a similar direction, Cau et al. propose the use of *lean formal methods*, envisaged as methods adequately accompanied by suites of affordable and practicable tools capable of supporting rapid prototyping, testing, and verification [Cau98], and Rushby suggests “invisible” *formal methods*, unobtrusively integrated in familiar software engineering tools [Rushby00]. The approach presented in this thesis also proposes a lighter application of formal methods.

2.7 Chapter Summary

In this chapter the larger space of our research has been surveyed and the topic of the dissertation has been localised on precise coordinates by using a “zoom-in” technique of exploration. Since the location of the thesis’ topic lays at the intersection of three major domains of software development and investigation, namely real-time systems, formality, and object-orientation, an overview of these domains has been presented and specific challenges, advantages, and disadvantages have been pointed out. This overview has provided the groundwork for next focusing the “investigation lense” on the two specification notations used in our approach (in Chapter 3) and, respectively, on the existing research studies that share similarities with our work (in Chapter 4).

3 BACKGROUND: NOTATIONS

“A good notation has a subtlety and suggestiveness which at times make it almost seem like a live teacher.”

[Bertrand Russell, in Introduction to L. Wittgenstein's
Tractatus Logico-Philosophicus, 1922]

3.1 Introduction

This chapter shifts the focus from the distinguishing aspects of the domains that make up the thesis' research space to particular details of the two specification languages, Z and UML, that provide the notational basis for the modelling approach proposed in this thesis. Presentations of the main features of both Z and UML are included, and the salient components of the two notations are illustrated by short examples. A look at the larger family of Z-based languages and a brief introduction of Z++ (the OO variant of Z employed in our approach) are also included and UML's support for modelling RTS is examined. In order to illustrate UML an Automatic Camshaft Testing System (ACTS) inspired from our previous work on developing software for an automobile manufacturing company is employed as a “recurrent theme” from which short examples are extracted. Observations regarding current directions of exploiting the modelling power of UML are also presented.

3.2 Z and Flavours of Z

The formal specification language Z has been developed by the Programming Research Group at the Oxford University Computing Laboratory from the influential work of Jean-

Raymond Abrial [Abrial80] and has been used by various industrial organisations all over the world. Numerous variants of Z have been proposed over the years, including object-oriented alternatives. A model-oriented formal language based on set theory and first order predicate logic, Z is undoubtedly one of the most successful formal specification notations in terms of its acceptance by the software development community. The specialised literature contains numerous accounts of software and hardware products that have employed Z as a formal instrument for specification, for instance IBM's Customer Information Control System (CICS) [Nix88], Inmos transputers [Barrett89], Tektronix oscilloscopes [Delisle90], Bellcore's PLAN system for planning and administrating feeder loop networks [Morgan94], and Lloyd's Register's COBOL parser [Neil98]. More on the industrial use of Z can be found in [Gerhart94] and [Bowen95b] and a comprehensive set of pointers to Z resources is available at "The World Wide Web Virtual Library: The Z Notation" [Zed01]. Authoritative books on the syntax and semantics of Z are [Spivey92] and [Wordsworth92], while very good texts on the application of Z in practice are [Barden94] and [Jacky97]. Object-oriented versions of Z are comprehensively surveyed in [Stepney92a, Stepney92b] and amply illustrated in [Lano94a], while the most complete reference for Z++ is [Lano95].

This section continues with a summary overview of the main features of Z, including its types, predicates, relations, functions, schemas, and schema calculus. These features provide the foundation on which all Z variants have been built, including the object-oriented alternative Z++ employed in our dissertation. Variants of Z are then briefly surveyed in Subsection 3.2.2, followed by a succinct introduction of Z++ in Subsection 3.2.3. The presentation of Z++ is kept to a minimum here since the notation is further detailed in Chapter 6, in conjunction with the proposed formalisation process of UML constructs. Z++ is also briefly presented in Appendix A.

3.2.1 The Z Notation

In essence, a Z specification consists of a number of *schemas*, which describe both the static and the dynamic aspects of the system. The static properties of the system are captured in the

collection of possible states of the system and in the invariant relationships that must be satisfied as the system transitions from state to state. The dynamic properties of the system are modelled by the system's operations, the relationship between their input and outputs, and the changes in the system state. Schemas provide the necessary support for modularisation and refinement, allowing the developer to specify pieces of the software product separately and then relate and combine them using the rules of Z schema composition. Refinement is supported, abstract specifications being transformed into equivalent concrete schemas that contain additional details. This powerful mechanism for composition and refinement accounts for the notation's successful application to larger projects. Schemas are expressed formally and the effect of each operation is described abstractly using first order predicate logic expressions, but the entire Z specification can (and normally should) include textual annotations, natural language descriptions that clarify the meaning of the rather arid mathematical statements.

The remaining part of this Subsection summarises the main features of Z, as described in [Spivey92] and [Wordsworth92], and illustrates them with several short examples. The notation style follows the one of [Barden94].

3.2.1.1 Sets, Types, and Predicates

Z's set theory is a typed set theory, which means that every value in the specification is assigned to a type. From simple, basic types, it is possible to define more complex types via three ways of type composition: set types, Cartesian product types, and schema types. A *type* can be introduced in Z by a *given set* (or *basic type*), declared by writing its name in the *given set brackets*. For instance:

[LIGHT] (3.1)

introduces the given set with the name LIGHT, declared to be a type covering values of a specific kind, used as "atomic" entities in a given application (in this case, we need to

describe the various lights that can be turned on or off in an apartment). As already mentioned, each variable declared in Z must have a type, for instance:

```
l balconyLight: LIGHT
```

 (3.2)

A set can be defined by *set enumeration*, that is by listing its members (of the same type) in order, separated by commas, and enclosed in brackets. For instance, possible kinds of room of interest in a particular application can be written:

```
{livingroom, bedroom, bathroom, kitchen, den, hall, extra}
```

 (3.3)

A name can be given to a set introduced via set enumeration by using *syntactic equivalence*, specified by the `==` symbol, for instance:

```
ROOMKIND == {livingroom, bedroom, bathroom, kitchen,
             den, hall, extra}
```

 (3.4)

A set with just one member is called a *singleton set*, while a set with no members is an *empty set* or a *null set*. The equality of two sets means that both sets have exactly the same members, so for two sets A and B with members of different types, both the $A = B$ and $A \neq B$ notations are not well-formed because comparison is not possible between such sets. Typical set operations such as union (\cup), intersection (\cap), difference (\setminus), and Cartesian product (\times), as well as relationships such as subset (\subseteq), strict subset (\subset), and membership (\in) can be applied. The cardinality of a finite set A is denoted $\#A$, while the set of all subsets of set A is denoted $\mathbb{P}A$ (powerset of A). The set of integer numbers is denoted \mathbb{Z} and its type is $\mathbb{P}\mathbb{Z}$. Similarly, natural numbers are in the set \mathbb{N} , which has the type $\mathbb{P}\mathbb{N}$. These two types are included by default in any specification and need not be introduced formally. Types with smaller number of values can be introduced using data definition, specified by the *data definition operator* (`:=`), for instance:

STATUS ::= on | off (3.5)

Another way of defining a set is by *set comprehension*, in the form:

{D | P • E} (3.6)

where D is a declaration, P a constraint imposed on values, and E an expression denoting the terms. The expression (3.6) denotes the set of values of term E for everything declared in D that satisfies the constraining predicate P. For instance, if TEMPERATURE == ℤ then:

{t: TEMPERATURE | t ≥ 0 ∧ t < 20 • t} (3.7)

gives the set of all positive temperatures that are less than 20-degree Celsius.

More complex structures can be defined using *schema types*, for instance, assuming LENGTH and WIDTH are already introduced (e.g., LENGTH == ℕ and WIDTH == ℕ) then:

Room	
kind:	ROOMKIND
dimension:	LENGTH x WIDTH
temperature:	TEMPERATURE
lights:	PLIGHT

(3.8)

The components of composite type variables can be accessed using *the dot notation*. In the above case if libraryRoom is a Room variable then the specification can make use of libraryRoom.kind, libraryRoom.dimension, etc.

In Z, *predicates* provide formal ways of expressing the meaning conveyed by declarative sentences of the natural language. It is possible to build more complex predicates from simpler ones by using the connectives of the first-order logic: negation (¬), conjunction (∧), disjunction (∨), equivalence (⇔), and implication (⇒). In addition, the universal quantifier (∀) and the existential quantifier (∃) are available, as well as the unique existential quantifier

(\exists_1), which indicates the fact that there is a single item satisfying a certain property. An example of a predicate, describing a situation that requires turning a heater on, is the following (assume that a `Heater` type, a `roomHeater` variable of `Heater` type, and the `comfortLevel` constant have been defined, and note the single quote decoration that indicates the “after-state” value of `roomHeater.status`):

$$\begin{aligned} &(\text{roomHeater.status} = \text{off}) \wedge \\ &(\text{libraryRoom.temperature} < \text{comfortLevel}) \\ &\Rightarrow \text{roomHeater.status}' = \text{on} \end{aligned} \tag{3.9}$$

3.2.1.2 Relations, Functions, and Sequences

A *relation* is a set of ordered pairs. In a relation R , the first member of the pair belongs to a set X , while the second member of the pair to a set Y (X and Y need not be different). The notation for the relation R is:

$$| R : X \longrightarrow Y \tag{3.10}$$

where X is the *from-set* and Y is the *to-set* of relation R . To indicate the fact the $x \in X$ and $y \in Y$ are in relation R the following notation is used:

$$| x \longmapsto y \in R \text{ or, equivalently, } x R y \tag{3.11}$$

The generic definition:

$$| X \longrightarrow Y == \mathbb{P}(X \times Y) \tag{3.12}$$

describes all possible relations that can be defined from X to Y . For a given relation R defined as in (3.10), the *domain of R* , denoted $\text{dom } R$, is the set of first members of all pairs in

the relation, while the *range of R*, denoted $\text{ran } R$, is the set of second members of all pairs in the relation. They are defined, respectively, by the following expressions:

$$| \{x:X \mid \exists y \in Y \bullet x \mapsto y \in R\} \quad (3.13)$$

$$| \{y:Y \mid \exists x \in X \bullet x \mapsto y \in R\} \quad (3.14)$$

Two relations R and S , defined as $R : X \mapsto Y$ and, respectively, $S : Y \mapsto Z$ can be subjected to *relation composition*, denoted $R \circ S$ and defined formally as:

$$| \{x:X ; z:Z \mid (\exists y \in Y \bullet (x \mapsto y \in R \wedge y \mapsto z \in S)) \bullet x \mapsto z\} \quad (3.15)$$

A number of operators are useful when working with relations. Assuming that M is a set of members from the domain type X and N is a set of members from the range type Y , then the *domain restriction operator* \triangleleft , which confines relation R to those pairs whose first members are in the set of interest M is defined as:

$$| M \triangleleft R == \{x:X; y:Y \mid x \in M \wedge x \mapsto y \in R \bullet x \mapsto y\} \quad (3.16)$$

The *range restriction operator* \triangleright is defined symmetrically:

$$| R \triangleright N == \{x:X; y:Y \mid y \in N \wedge x \mapsto y \in R \bullet x \mapsto y\} \quad (3.17)$$

It is also possible to use the *domain subtraction operator* \triangleleft , which restricts the relation to those pairs whose first members are not in the set M . This operator is defined as:

$$| M \triangleleft R == \{x:X ; y:Y \mid x \notin M \wedge x \mapsto y \in R \bullet x \mapsto y\} \quad (3.18)$$

Finally, the *range subtraction operator* \triangleright restricts the relation to those pairs whose second members are not in the set N . The range subtraction operator is defined as:

$$| \quad R \triangleright N == \{x: X ; y: Y \mid y \notin N \wedge x \mapsto y \in R \bullet x \mapsto y\} \quad (3.19)$$

A *function* is a particular case of relation, in which each member of the from-set can be in relation with at most one member of the to-set. This is expressed formally by the following generic definition, which defines all possible functions from X to Y :

$$| \quad X \mapsto Y == \{f: X \mapsto Y \mid (\forall x: X; y_1, y_2: Y \bullet (x \mapsto y_1) \in f \wedge (x \mapsto y_2) \in f \Rightarrow y_1 = y_2)\} \quad (3.20)$$

In Z there are predefined symbols for partial functions, total functions, partial injections, total injections, partial surjections, total surjections, and bijections. *Partial functions* correspond to the definition (3.20), since there is no constraint on the from-set –specifically the domain of f may not be the entire from set X . The domain of a *total function* is the entire from-set of the function. The formal generic definition for total functions is:

$$| \quad X \longrightarrow Y == \{f : X \mapsto Y \mid \text{dom } f = X\} \quad (3.21)$$

An *injection* has the property that the second members of its pairs are unique. Injections can be *partial injections*, described by:

$$| \quad X \mapsto\!\!\!\rightarrow Y == \{f : X \mapsto Y \mid (\forall x_1, x_2 : \text{dom } f \bullet f(x_1) = f(x_2) \Rightarrow x_1 = x_2)\} \quad (3.22)$$

or can be *total injections*, corresponding to:

$$| \quad X \longmapsto Y == (X \longrightarrow Y) \cap (X \mapsto\!\!\!\rightarrow Y) \quad (3.23)$$

A *surjection* has the property that its range is the whole of its to-set. *Partial surjections* are described by:

$$| X \twoheadrightarrow Y == \{f : X \rightarrow Y \mid \text{ran } f = Y\} \quad (3.24)$$

while *total surjections* are defined as:

$$| X \twoheadrightarrow Y == (X \rightarrow Y) \cap (X \twoheadrightarrow Y) \quad (3.25)$$

Finally, a *bijection* is a function both injective and surjective:

$$| X \xrightarrow{\sim} Y == (X \twoheadrightarrow Y) \cap (X \rightarrow Y) \quad (3.26)$$

Since functions are relations, all operators that apply to relations apply as well to functions. In addition, an operator that guarantees that the result is a function and can be used for updating information is the *function overriding* operator \oplus . For instance, with the `[LIGHT]` given set (3.1) and the `STATUS` type (3.5), a partial function that keeps track of the lighting situation in a given environment can be defined as:

$$\text{lightsynopsis} : \text{LIGHT} \twoheadrightarrow \text{STATUS} \quad (3.27)$$

and turning on the particular light `balconyLight` can be described as:

$$\text{lightsynopsis} = \text{lightsynopsis} \oplus (\text{balconyLight} \mapsto \text{on}) \quad (3.28)$$

Sequences are particularly useful in software specification. A sequence of values of type X is an ordered collection of values and can be defined as a partial function from the natural numbers \mathbb{N} to X with the property that its domain is $1 \dots n$ (where n is the length of the sequence). The generic definition that describes all possible sequences of values of type X is:

$$| \text{seq } X == \{f : \mathbb{N} \rightarrow X \mid (\exists n : \mathbb{N} \bullet \text{dom } f = 1 \dots n)\} \quad (3.29)$$

As shown in Table 3.I, where the definition (3.4) is used for X , a number of operators permit the extraction of useful information from a sequence s , namely $\text{head}(s)$, $\text{tail}(s)$, $\text{last}(s)$, and $\text{front}(s)$. Also, a sequence s can be reversed by applying the reverse operator $\text{rev}(s)$ and two sequences s and t can be concatenated using the *concatenation operator* \wedge .

Table 3.I A Summary of Sequences

Item	Description	Example [assume $X = \text{ROOMKIND}$ as defined in (3.4)]
finite sequences	$\text{seq}X == \{f : \mathbb{N} \rightarrow X \mid \text{dom } f = 1 \dots n\}$	$s = \langle \text{hall, kitchen, den, bedroom, bedroom} \rangle$ $\text{emptyseq} = \langle \rangle$
non-empty finite sequences	$\text{seq}_1 X = \{f : \text{seq}X \mid \#f > 0\}$	$t = \langle \text{livingroom, bedroom, extra, hall, extra} \rangle$
injective sequences	$\text{iseq}X == \text{seq}X \cap (\mathbb{N} \rightarrow X)$	$u = \langle \text{hall, den, livingroom} \rangle$
head	$\text{head } s = s(1)$	$\text{head } s = \text{hall}$
tail	$\text{tail } s = \{k : 1 \dots \#s - 1 \bullet k \mapsto s(k+1)\}$	$\text{tail } s = \langle \text{kitchen, den, bedroom, bedroom} \rangle$
last	$\text{last } s = s \#s$	$\text{last } s = \text{bedroom}$
front	$\text{front } s = \{\#s\} \triangleleft s$	$\text{front } s = \langle \text{hall, kitchen, den, bedroom} \rangle$
reverse	$\text{rev } s = \{k : 1 \dots \#s \bullet (\#s - k + 1) \mapsto s(k)\}$	$\text{rev } s = \langle \text{bedroom, bedroom, den, kitchen, hall} \rangle$
concatenation	$s \wedge t = s \cup \{k : 1 \dots \#t \bullet (k + \#s) \mapsto t(k)\}$	$s \wedge t = \langle \text{hall, kitchen, den, bedroom, bedroom, livingroom, bedroom, extra, hall, extra} \rangle$

Z includes definitions for non-empty sequences ($\text{seq}_1 X$) as well as for injective sequences ($\text{iseq } X$). The notion of bag can also be used in Z, with the expression:

$$\mid \text{ bag } X == X \leftrightarrow N_1 \quad (3.30)$$

indicating that items $x_k \in X$ have a number of apparitions $n_k \geq 1$ in the bag.

3.2.1.2 Schemas and Schema Calculus

The fundamental building block for Z specification is the *schema*, which can be defined as a mathematical description of a part of the system under construction. Z schemas are used for describing both structural and behavioural properties of the system. They can express both the state space and the operations of the systems, and can be interconnected using the mechanisms of Z schema calculus. Thus, they provide support for modularity and composition, allowing the development of large-scale specifications. Schemas also provide the basis for refinement through schema transformation.

A Z schema has a name (used for reference throughout the specification) and consists of two parts: a *declaration part*, and a *predicate part* (some examples of schemas are given in Fig. 3.1). In the declaration part schema components are introduced and their types are specified. These components act as local variables that, together with the global variables defined in the system, can be used by the first-order expressions included in the predicate part. These expressions define conditions that must be satisfied by the variables introduced in the declaration part. The association between the names introduced in the declaration part and the types of values that those names denote is referred to as the *signature of the schema*. The *property of the schema* is given by the constraints included in the predicate part, together with the predicates implicit in the declaration part. The property of the schema is also called *the schema invariant* and together with the signature of the schema makes up, when the schema defines the abstract state of an abstract data type, the *data space*, which describes all possible data states of the abstract data type. Schemas also describe operations and operations

on the abstract state are called *abstract operations*. When schemas define such operations the following conventions are used:

- undashed names denote the values of the components in the *starting state*, before the operation;
- dashed names denote the values of the same components in the *ending state*, after the operation;
- names postfixed by a question mark denote the input values to the operation;
- names postfixed by an exclamation mark denote the output values from the operation.

Predicates that refer only to the input values and to the starting state define the *precondition* of the schema. The precondition must hold in order for the operation to behave as defined. The remaining predicates included in the predicate part are concerned with input, output and the ending state. They are referred to as the *postcondition* of the schema and describe the conditions that must be obtained after the operation has behaved as specified. Z provides a number of conventions for schema presentation aimed at reducing the amount of specification shown in a document. These conventions include *schema decoration* (for systematical inclusion of dashed names associated to a schema, together with the schema's invariant), *schema inclusion* (which has the result of bringing into the declaration part of the enclosing schema all the declarations of the enclosed schema), the *delta convention*, which makes use of the symbol Δ and indicates change in the schema's variables, and the *xi convention*, which uses the symbol Ξ and provides a shorthand notation for situations in which the schema's variables are not changed by an operation. Schemas can be combined using the following schema calculus mechanisms: *disjunction*, *conjunction*, *negation*, *implication*, *quantification*, *pipng*, and *composition*. Both the signatures and the properties of two or more schemas can be combined according to the significance of the logical connectives indicated above. A short example of a Z specification illustrating some of the above conventions is shown in Fig. 3.1 (the example is an adaptation of a small part of the case study presented in [Evans97]).

Description of a robot arm. The robot loads bottles from a conveyor and unloads them at a filling machine. The bottles are loaded and unloaded one at a time. Given type for identifying bottles:

```
[BOTTLEID]
```

The two opposite positions of the robot arm and a type useful in describing the loading process (the robot arm is either unloaded or loaded with an identifiable bottle):

```
ArmPosition ::= at_coveyor | at_filling
bottleLoaded ::= loaded «BOTTLEID» | unloaded
```

Composite type describing the robot arm:

```
Arm
  position: ArmPosition
  status: bottleLoaded
```

Robot arm initialisation:

```
InitArm
  Arm
  position = at_conveyor
  status   = unloaded
```

Robot arm operations. The delta convention for Load and Unload operations indicates that the Arm state is changed:

```
Load
  Δ Arm
  bottle? : BOTTLEID
  position = at_conveyor
  status   = unloaded
  status'  = loaded (bottle?)
  position' = position
```

```
Unload
  Δ Arm
  bottle! : BOTTLEID
  position = at_filling
  status   = loaded (bottle!)
  status'  = unloaded
  position' = position
```

Fig 3.1 Partial Z Description of a Robot Arm

In conclusion of this summary presentation of Z we note again that there is much more about this language than presented here and for all the necessary details we refer the reader once more to the sources cited at the beginning of this section.

3.2.2 Z Variants and Tools

Currently, there is an expanding community of Z users and researchers in many parts of the world, with major concentrations in the Great Britain, and significant presence in other countries, most notably Australia, Canada, France, Germany, and the U.S.A. The Internet makes available numerous Z resources, many of them accessible through the Z community's web-site [Zed01], which provides pointers to a variety of materials, including papers, reports, books, and tools. As pointed out from this web-site, as well as from Stepney et al.'s surveys [Stepney92a, Stepney92b] and Lano and Haughton's collection of case studies [Lano94a], various groups have worked on developing extensions to Z, some of the most notable being:

- *Object-Z*, which provides a construct for classes that encapsulates both state and operation schemas and includes a non-conformant inheritance mechanism (operations in derived classes can be redefined by strengthening the operation's precondition or by re-writing it) [Duke94]. Provisions for specifying the allowable sequences of operations are included, and a temporal logic notation is employed. The formal semantics of classes in Object-Z are based on event histories, and operators for specifying parallel operations are available. This object-oriented variant of Z uses a graphical notation for classes that extends Z's basic construct, the schema, and provides a useful visual aid for the developers;
- *Z++*, which follows an approach similar to the one taken by the authors of Object-Z for specifying systems in an object-oriented fashion [Lano91, Lano95]. Z++ has class definition and an inheritance mechanisms similar to that of Object-Z but, as indicated in [Stepney92b], while Object-Z is fairly abstract, the design of Z++ has been more influenced by object-oriented programming language constructs. Z++, which also

includes temporal logic support, is further covered in Subsection 3.2.3, in Chapter 6, and in Appendix A of this thesis;

- *ZEST (Z Extended with Structuring)*, developed by British Telecommunications as an object-oriented dialect of Z intended primarily for modelling network structures and open distributed systems [Zadeh96]. ZEST extends the conventional Z language by including a class construct that consists of five distinct clauses (inheritance, interface, axiomatic clause, unnamed schema, and named schemas) and has a syntax similar to that of Object-Z. However, its semantics are significantly different and it has no special provisions for capturing temporal properties of systems;
- *OOZE (Object-Oriented Z Environment)*, which has an algebraic formal semantics based on OBJ3, employs Z's notation and specification style, and consists of a full-fledged environment that includes a database for indexes, dependency relations, and module extensions, as well as support for animating the specifications [Alencar94]. OOZE permits the nesting of schema boxes and incorporates readability enhancements such as separate exception schemas and separate pre- and post- conditions;
- *Sum*, an extension of the conventional Z oriented towards refinement and translation of formal specifications to Ada [Utting95]. Sum is part of the Cogito formal development environment, which includes among its components a type-checker, a configuration management tools, and the Ergo theorem prover (the inspired header of their web-site is "Cogito, Ergo Sum" [Cogito97]). Sum specifications are translated into constructs of the functional programming language Haskell. This particular language was chosen because its type system is similar to that of Z, its strong typing allows for a fair amount of checking the specifications during editing, and its lazy evaluation increases the chance of termination in the case of some specifications;
- *S*, designed as a "gentler Z" (hence, its name, suggesting a Z without asperities), a machine readable notation developed at the University of British Columbia in the context of Hughes Aircraft of Canada's complex project CAATS (Canadian Automated Air Traffic System) [Joyce94]. The purpose of the language is to satisfy the requirements of applying formal specifications in large scale industrial projects by overcoming the perceived limitations of Z, specifically difficulty in explaining it to non-expert users and

impossibility of creating Z specifications using a standard text editor. The advantages of S, which has a Z-like syntax and HOL semantics, stem from its relatively simple semantics, entire printable set of characters, and the availability of a proven verification tool, the HOL system;

- *Alloy*, the “new kid in town,” is defined by its originator as “a little language for describing structural properties” [Jackson00b, pp. 1], both sufficiently simple to allow a completely automatic semantic analysis and sufficiently powerful to express complex constraints. The new modelling language, which has its semantic basis taken from Z and a structuring mechanism similar to that of existing object-oriented notations such as UML, is aimed at supporting lightweight formal development of object-oriented systems. Alloy is less powerful than Z but compensates Z’s unsuitability for object-oriented modelling with a number of features such as more flexible state declarations and distinct types of schemas. It also has a graphical notation with a textual counterpart that allows building a model entirely textually and, in comparison to UML, is more abstract (since it is based on sets and not classes) and, of course, has precise semantics.

Several well-known Z tools are the already mentioned COGITO, which proposes a methodology and toolset for the formal development of software [Bloesch94]; Logica’s Formaliser environment, consisting of a ZEST Specific Formaliser, a Z Specific Formaliser, and a Generic Formaliser that supports the development of grammars for new languages [Logica01]; Z/EVES, a complex analysis system from ORA, Canada, that allows the examination of Z specifications through syntax and type checking, precondition calculation, schema expansion, domain checking, and theorem proving [Meisels97, ZEVES00]; Wizard, a type-checker for Object-Z [Uttig95, Wizard01]; Jia’s ZTC and ZANS tools, a type checker and, respectively, an animator for Z specifications [Jia98a, Jia98b, ZANS98, ZTC98]; ZETA, an open environment written in Java that provides an integration framework for editing, analysing, and animating Z specifications [ZETA00], and the more recent Alloy Constraint Analyzer, developed by Daniel Jackson and his colleagues at the Massachusetts Institute of Technology [Alloy00]. Additional information on several other Z tools, specifically ProofPower Z, Zola, CADiZ, and HOL-Z, can be found in an earlier paper by

Steggles and Hulance [Steggles94]. Since a number of newer Z-centred specification approaches are currently being developed it is expected that novel Z tools, supporting these approaches, will emerge in the near future. Some of these recent approaches are described in Chapter 4, where work related to ours is surveyed.

3.2.3 A Glance at Z++

One of the OO extensions of Z, Z++ distinguishes itself through its support for capturing timing properties of systems. Since the language constitutes an integral part of the foundation on which our modelling approach is built, the purpose of this subsection is to provide only a quick look at Z++, its features being described in more detail in Appendix A, which contains a summary overview of Z++, and in Chapter 6, where rules for formalising UML constructs are presented. Here, only the general form of Z++'s most important construct, the class declaration, is given, based on [Lano94e] and [Lano95] (Fig. 3.2).

```

ZPP_Class ::= CLASS Identifier [TypeParams]
[EXTENDS Ancestors]
[TYPES TypeDefs]
[FUNCTIONS AxiomaticDefs]
[OWNS Locals]
[RETURNS OpTypes]
[OPERATIONS OpTypes]
[INVARIANT Predicate]
[ACTIONS Actions]
[HISTORY History]
END CLASS

```

Fig. 3.2 General Form of Z++ Class Declaration

As indicated by Lano and Haughton, in the Z++ class declaration *TypeParams* represents a list of generic type parameters, *EXTENDS* specifies the superclasses of the class, *TYPES* introduces type identifiers used in the declaration of local variables, *FUNCTIONS* gives a list of axiomatic definitions of constants, *OWNS* specifies the local variables (attributes), *RETURNS* lists the operations that do not change the state of the object, *OPERATIONS* declares the operations that may change the attributes of the object, *INVARIANT* specifies a predicate that describes the internal state of the object and is guaranteed to be true between executions of the object's operations, *ACTIONS* defines the class operations that can be performed on objects of the class (the operations are specified using regular Z schema definitions), and *HISTORY* specifies the admissible sequences of execution for the objects of the class in the form of temporal-logic formulae that make use of operators such as \square (*henceforth*), \bigcirc (*next*), \diamond (*eventually*), *before* and *until*. In essence, it is here, in the *HISTORY* clause, where Z++'s capability of dealing with temporal aspects of the systems resides. As described in Chapter 6, this clause can also contain RTL (Real-Time Logic) formulae (Jahanian and Mok's RTL is introduced in Chapter 5).

3.3 On UML and Its Capability of Dealing with Time

The Unified Modeling Language (UML) has emerged from the combination of several widely-used, practice-validated object-oriented notations developed over the last decade by a number of prominent authors, primarily from the Booch notation [Booch94], Rumbaugh et al.'s OMT [Rumbaugh91], and Jacobson et al.'s OOSE [Jacobson94]. The fact that Booch and Rumbaugh joined forces at Rational Software Corporation in October 1994, followed by Jacobson in October 1995, created the premises for a standard notation and a common methodology for the object-oriented development community. In order to accommodate the variety of existing object-oriented analysis and design approaches and gain a large industry support, UML has borrowed concepts and notations from several other methods, including Coad and Yourdon [Coad90, Coad91], Shlaer and Mellor [Shlaer88, Shlaer91], Fusion [Coleman94], and Statecharts [Harel87]. In November 1997 version 1.1 of UML has been adopted as object modelling standard notation by the Object Management Group (OMG)

and as of March 2001 minor revisions have been included in versions up to 1.4, the work on the last one being currently in progress. A major revision, version 2.0, is tentatively scheduled for standardisation in 2001 [Kobryn99]. The complete version 1.3 of the language's specification, published in March 2000, is available from the OMG web-site [UML00]. In our opinion, the incorporation of concepts from numerous sources, although justified by the goal of ending "the methods war" by providing a comprehensive, widely accepted modelling language for object-oriented development, requires some "fine-tuning" in practical terms – the generality of the notation and its higher level of diffuseness [Green96] typically necessitating decisions on what to be used, what customisations to be made (what "stereotypes" to be employed), and what to be left out from UML in a particular methodology and/or application. As detailed later in the thesis, for practical reasons we focus in our approach on only a subset of the UML notation and thus minimise the notational alternatives that UML would provide if considered in totality. Here, in Subsection 3.3.1 an overview of UML's general capabilities is presented, while in Subsection 3.3.2 the features of the language that support the modelling of RTS are examined. Although the most authoritative reference for UML is [UML00] for a shorter and less formal description of UML we have relied on [Booch98] as primary reference for the two Subsections that follow, with additional sources consulted [Quatrani98], [Si-Alhir98], [Douglass98] and [Douglass99]. The examination of UML is completed in Subsection 3.3.3 with a look at some of the current research and industrial developments involving UML.

3.3.1 A Bird's Eye View on UML

As indicated by its authors, UML is a "graphical language for visualizing, specifying, constructing, and documenting the artifacts of software-intensive systems" [Booch98, pp. xv]. The notation is primarily designed for supporting the analysis and design phases of the software development process, but it is useful also for the deployment and maintenance of software. UML is supported by the industrial-strength software development environment Rational Rose (latest version 2001), commercially available from Rational Software Corporation [RationalRose01], and has been integrated in a number of other CASE tools,

some of which are mentioned in Section 3.3.3 of the thesis. Recently, a real-time version has also been made available [RationalRoseRT01]. The environment is powerful and supports not only the modelling of large software systems, but also processes such as reengineering and automated code generation. Fig. 3.3 contains an image of the Rational Rose environment (version 2001), showing its main components in the case of a logical view specification: the menu bar, the toolbar, the browser, the palette for class modelling, and the pane with drawing windows (the example shown is of an Elevator system, based loosely on [Dong97b], where an OMT description is given).

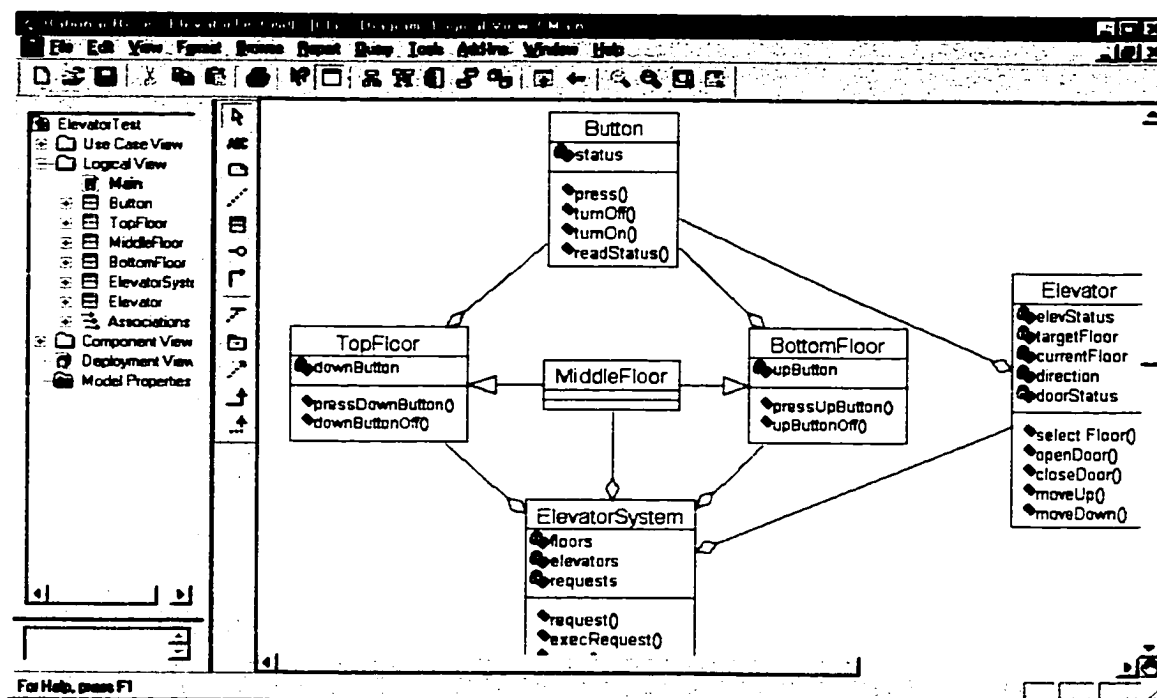
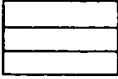



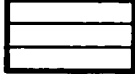



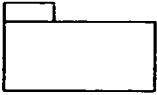
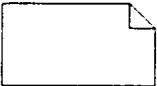


Fig. 3.3 Snapshot of Rational Software Corporation's Rational Rose

Booch et al. indicate that there are three types of *building blocks* in the language, namely *things* (first-class abstractions in UML's conceptual model, alternatively referred to by us as *model elements*), *relationships* that interconnect abstractions, and *diagrams* that present collections of things together with the relationships that exist among these things [Booch98]. A summary of UML basic things (there are also variations for them), further classified as *structural*, *behavioural*, *grouping*, and *annotational* is presented in Table 3.II, while the

Table 3.11 UML Things (Model Elements)

UML Thing (Model Element)	Kind	Description	Symbol
class	structural	represents a set of objects with the same attributes, operations, relationships and semantics	
interface	structural	defines a collection of operations (a set of services) that represent the externally visible behaviour of an entity	
collaboration	structural & behavioural	describes an interaction and includes several cooperating elements with their specific roles	
use case	structural & behavioural	represents a set of sequences of actions that yield an observable result to some actor(s)	
active class	structural	a class whose instances owns threads or processes and thus can initiate control activities	
component	structural	a physical and replaceable part of a system that packages implementation	
node	structural	a run-time computational resource generally having memory and often processing capability as well	
interaction	behavioural	defines a behaviour and consists of a set of messages exchanged among collaborating objects within a particular context	(message, the basic element) 
state machine	behavioural	specifies the sequences of states an object or an interaction can go through during its lifetime	details in Subsection 3.3.2
package	grouping	general purpose mechanism for organising model elements in groups	
note	annotational	explanatory item (comments, constraints, etc.)	

fundamental relationships of the language, namely *dependency*, *association*, *aggregation* (with its particular form *composition*), *generalisation*, and *realisation* are succinctly described in Table 3.III (diagrams, the third kind of building blocks, are discussed in more detail later in this Subsection). The authors indicate that besides building blocks the conceptual model of UML consists also of a number of *rules* for putting together these blocks (such as rules for names, scope, visibility, and integrity) and *common mechanisms* that are consistently applied within the language (specifications, adornments, common divisions, and extensibility mechanisms).

Table 3.III UML Relationships

UML Relationship	Description	Symbol
dependency	a "using" relationship from a client <i>C</i> to a supplier <i>S</i> ("C uses S" or "C depends on S"); changes in the specification of <i>S</i> may affect the using class <i>C</i>	$S \leftarrow \text{-----} C$
association	structural relationship between two model elements that declares a connection between their instances (on the association symbol the <i>name</i> of the relationship, the <i>roles</i> of the two model elements and the <i>multiplicity</i> of their instances can be specified)	$E_1 \xrightarrow[\text{role1}]{\text{rel_name}}_{\text{role2}}^{m_2} E_2$
aggregation / composition	aggregation is a particular case of association that specifies a "has-a" relationship between the whole <i>W</i> and its part <i>P</i> ; composition is an aggregation with strong ownership and the lifetime of <i>P</i> subsumed to the lifetime of <i>W</i>	$W \text{---} \diamond \text{---} P$ $W \text{---} \blacklozenge \text{---} P$
generalisation	relationship between a more general model element (parent <i>P</i>) and a more specific kind of that element (child <i>C</i>); "is-a" relationship	$P \leftarrow \text{---} C$
realisation	a relationship in which the contract stipulated in the model element <i>X</i> is carried out in the model element <i>R</i>	$X \leftarrow \text{-----} R$

Since extensibility is one of the most important characteristics of the language the mechanisms that ensure UML remains open-ended, namely *stereotypes*, *tagged values*, and *constraints* are presented in Table 3.IV.

Table 3.IV UML Extension Mechanisms

UML Extension Mechanism	Description	Symbol
stereotype	allows the creation of new building blocks from the existing ones; extends the language by allowing the addition of new, problem-specific model elements	<<stereotype_name>>
tagged value	attaches new information to an existing model element; extends the language by allowing the addition of new properties	{property_description}
constraint	extends the semantics of UML building blocks by adding new rules or modifying the existing ones	{constraint_description}

In essence, the UML allows the modelling of a system through a number of diagrams that capture either the static or the dynamic aspects of the system and can be organised in a number of views, each view being “a projection into the organization and structure of the system, focused on the particular aspect of that system” [Booch98, pp.31]. Static aspects are captured in *use case diagrams*, which contain actors (an actor being someone or something that externally interacts with the system), use cases, and their relationships; *class diagrams*, which show classes, interfaces, collaborations, and their relationships; *object diagrams*, which provide snapshots of instances of classes and their relationships; *component diagrams*, illustrating the organisation of and the dependencies among software implementation components (a component typically maps to one or more classes, interfaces and collaborations); and *deployment diagrams*, which show the configuration of nodes and the run-time allocation of components to nodes. Behavioural aspects are described in *sequence*

diagrams, which depict a succession of messages exchanged among objects and emphasise the time ordering of messages; *collaboration diagrams*, which are similar to sequence diagrams, but stress the organisation and the roles of objects that send and receive messages; *statechart diagrams*, which essentially contain states and transitions that describe the event-driven life cycle of objects; and *activity diagrams*, whose role is to indicate how the control flows from activity to activity within a system. The sequence diagrams and collaboration diagrams, which convey the same information and can be easily transformed one into the other, are referred to commonly as *interaction diagrams*.

Although with the exception of the use case view the terminology related to views varies from author to author (for instance, [Booch98] speaks of use case, design, process, deployment, and implementation views, [Si-Alhir98] uses the terms use case, structural, behavioural, environment, and implementation views, and [Quatrani98], whose presentation is based on the Rational Rose tool, discusses the use case, logical, process, component, and deployment views) and there are also different nuances involved in the meanings these authors associate to views, it is generally acknowledged that a “4+1” architectural view model allows a comprehensive description of the system. Useful to note, the view concept is not part of UML specification, but the language supports this generally accepted “4+1” view of architecture that facilitates the organisation of knowledge and allows the modelling of the system from various interconnected perspectives. Interestingly, the Rational Rose environment, including its latest edition [RationalRose01], has predefined sections for only four views (use case, logical, component, and deployment) and relies on component diagrams to render the process view employed by Booch et al. and by Quatrani.

In Fig. 3.4 we have set for a “4+1” view solution that associates meanings and diagrams to views in Si-Alhir’s way, which emphasises a more traditional demarcation between structure and behaviour [Si-Alhir98]. However, in order to use as much as possible the more prevalent Rational terminology, the names of views have been borrowed from all the references mentioned above.

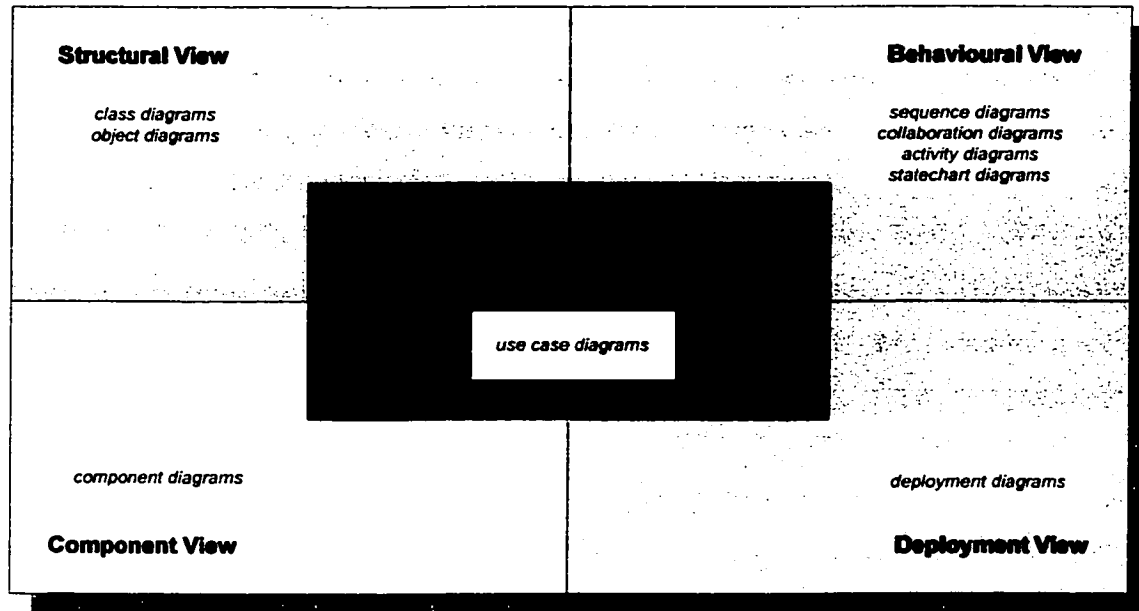


Fig. 3.4 The 4+1 Architectural Views and UML Diagrams That Express Them

In this “4+1 views” architectural model of the system the *use case view* describes the behaviour of the system as seen by its users. It employs use case diagrams to capture the functionality provided by the system to its external interactors and constitutes the “central perspective” that binds together the different angles under which the system can be scrutinised. The *structural view* relies on class and object diagrams to describe the system’s structural elements and their interconnections. The *behavioural view* is concerned with the dynamic aspects of the system and uses all four types of behavioural diagrams to capture them. The *component view* (or implementation view) makes use of component diagrams to capture both the behavioural and static aspects of a system’s realisation and shows all the components and files that are needed to assemble the physical system. Finally, the *deployment view* presents in its associated deployment diagram the nodes that form the hardware topology on which the system executes. Of course, for managing the complexity of a problem each view can be considered separately but the complete “picture” of the system is obtained by interconnecting them. In fact, as pointed out by Booch et al., the views interact inherently, for instance the nodes of the deployment view contain components (defined in the component view) that realise classes (specified in the structural view) and behaviours

(described in the behavioural view), all derived from use cases (captured in the use case view). In our dual-notation specification approach focused on time-constrained systems we are primarily interested in the use-case, structural, and behavioural views, and leave aside details pertaining to the component and deployment view (details are given in Chapter 7).

To keep the description shorter and in agreement with the selection of the UML subset used in our approach only five out of the nine possible kinds of UML diagrams are illustrated below. A use-case diagram (Fig 3.6), a class diagram (Fig 3.7), an object diagram (Fig. 3.8), and a sequence diagram (Fig. 3.9) pertaining to a common “theme”, an Automatic Camshaft Testing System (ACTS, Fig. 3.5) inspired from our previous work [Dascalu89, Ionescu93], are presented in this Subsection, while a statechart diagram is included in Subsection 3.3.2.

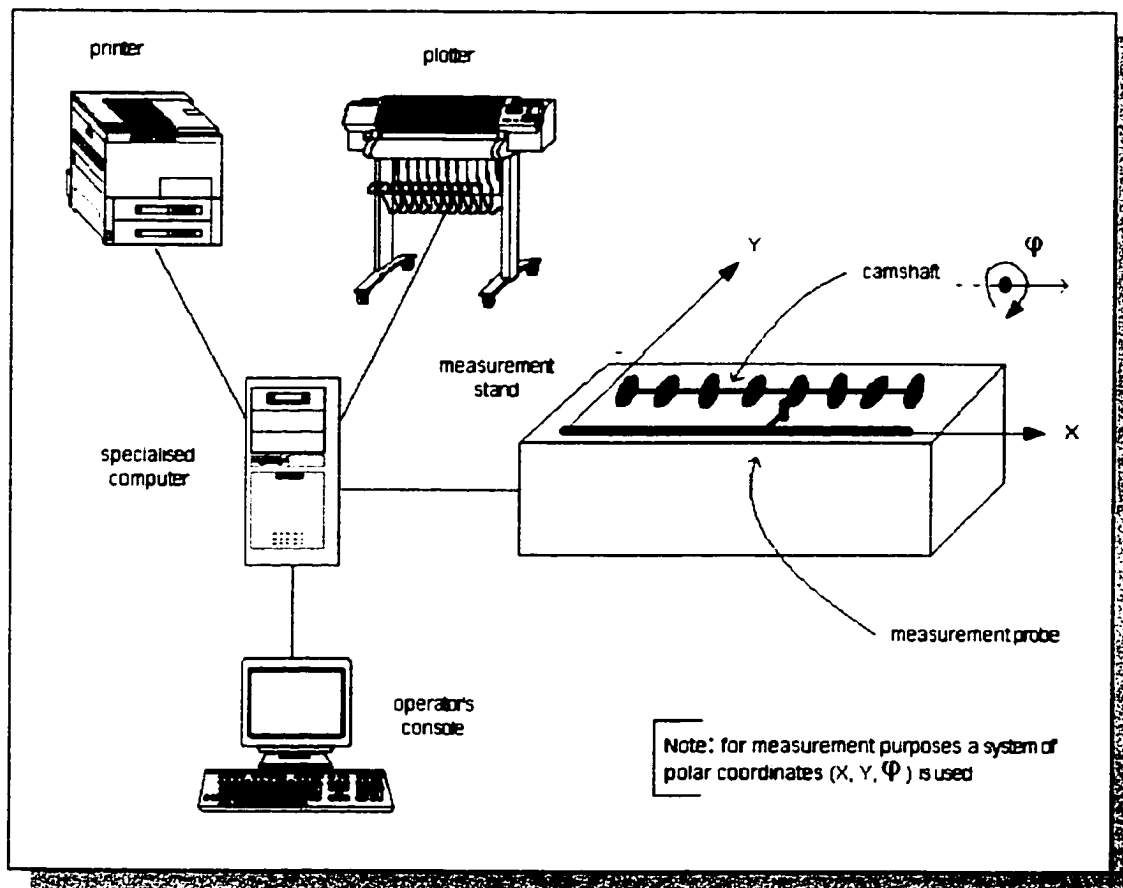


Fig. 3.5 Overview of the Automatic Camshaft Testing System (ACTS)

These diagrams are only intended to illustrate several of the basic UML concepts and not to capture the entire complexity of the automatic camshaft testing problem, so they should be viewed only as small excerpts from a larger specification.

However, in order to provide the necessary context, a short description of the system presented in Fig. 3.5 is necessary. Functionally, what is important to know is that in the ACTS actual profile data (Y values) for each of the N cams (typically $N = 8$ or $N = 12$) of an automobile engine camshaft are automatically collected and then a number of associated diagrams (“height,” speed, and acceleration) can be drawn and a variety of comparisons can be performed against theoretical values. To achieve this, for each cam the measurement probe is first moved along the X axis to a position that corresponds to the middle of the cam’s lateral surface, pushed subsequently against the cam (Y movement) and then the camshaft is rotated a little more than 360° (movement on φ axis) while profile values Y are collected from the cam. The ACTS operator has a number of options, including selective testing of cams (e.g., only cams 1, 4, 5, and 8 are inspected), variable “angular step” for measurement (e.g., 0.5° or 1°), and various formats for test certificates (out of tolerance values only or full diagrams, printed or plotted profiles, etc.).

In the four related UML diagrams the view on ACTS is “sequential” in the sense that the three axis controllers (X , Y , φ) work only one at time and the sensors are polled by the controllers. As shown later in Subsection 3.3.2 a concurrent approach can also be considered with sensors sending signals to the axis controllers and the camshaft being rotated towards a predefined position while the probe is “cruising” on X and Y directions (in principle, simultaneous movements on X and Y axes are not excluded either). In the UML diagrams pertaining to the ACTS specification italics have been used to highlight key elements, many of them introduced in Tables 3.II to Table 3.IV. Besides the references cited at the beginning of this section for more examples of UML diagrams we suggest [TogetherSoft00a], which contains one of the most concise and clear UML tutorials we found during our survey of the notation.

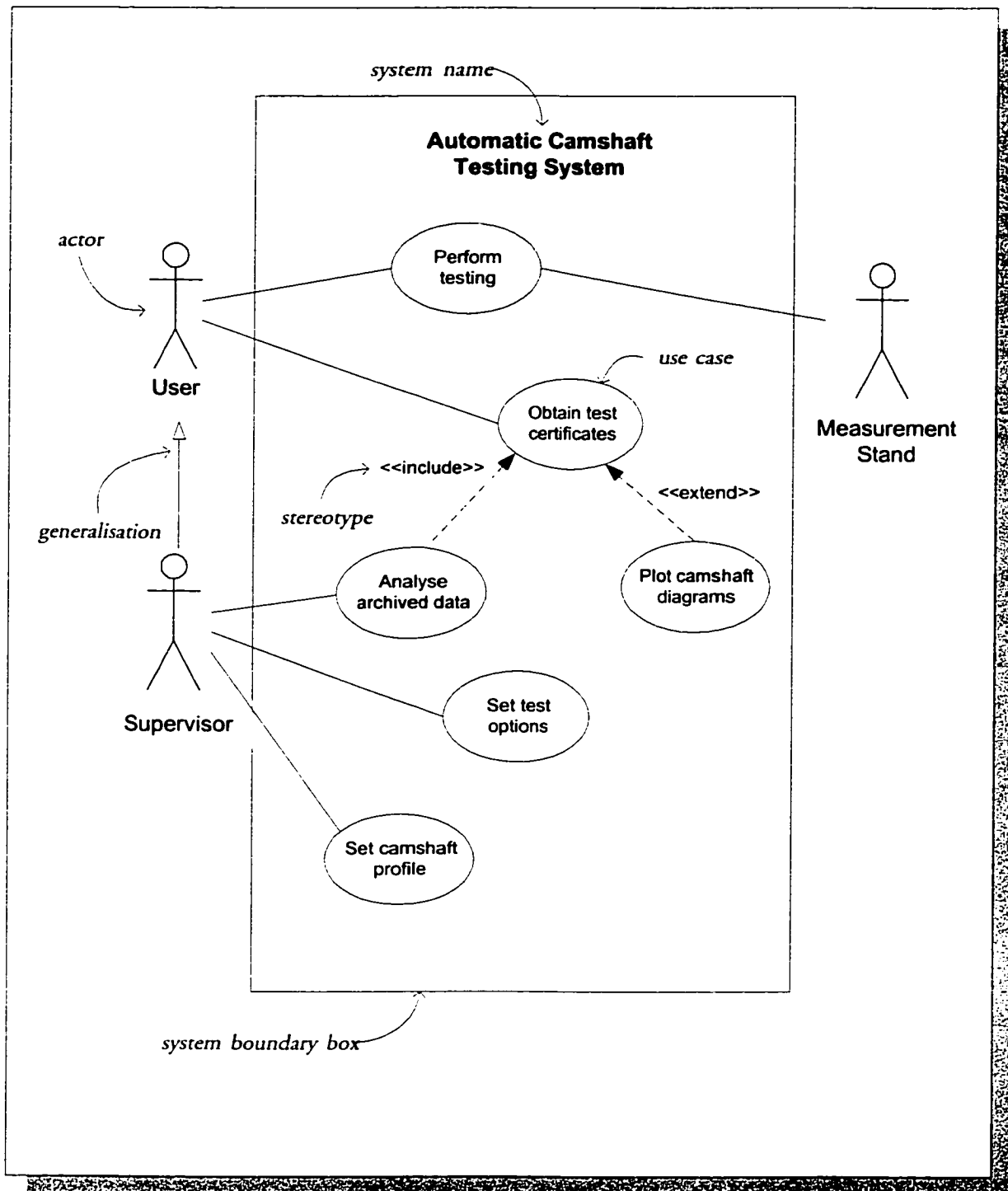


Fig. 3.6 Example of Use Case Diagram: Excerpt from the ACTS Specification

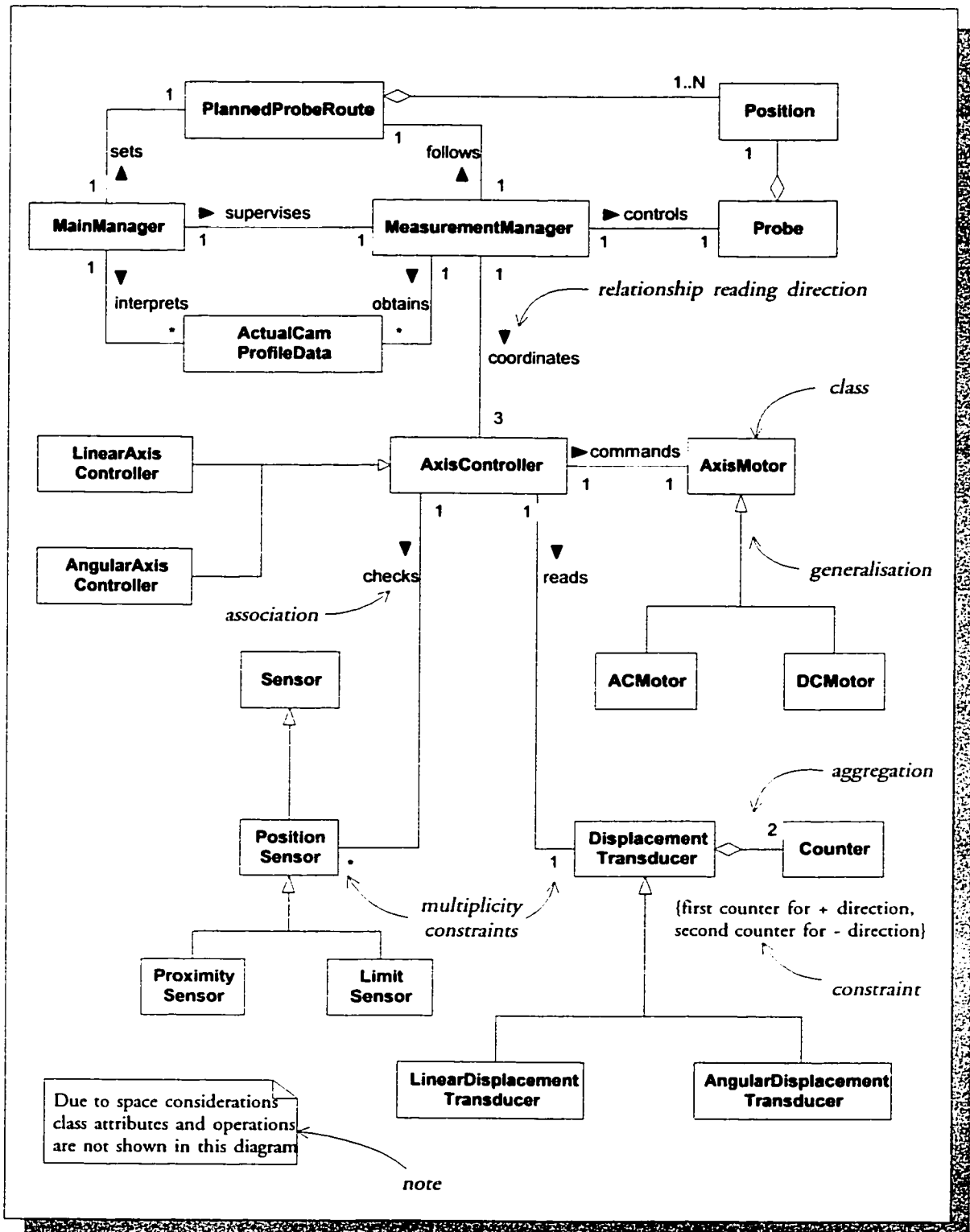


Fig. 3.7 Example of Class Diagram: Excerpt from the ACTS Specification

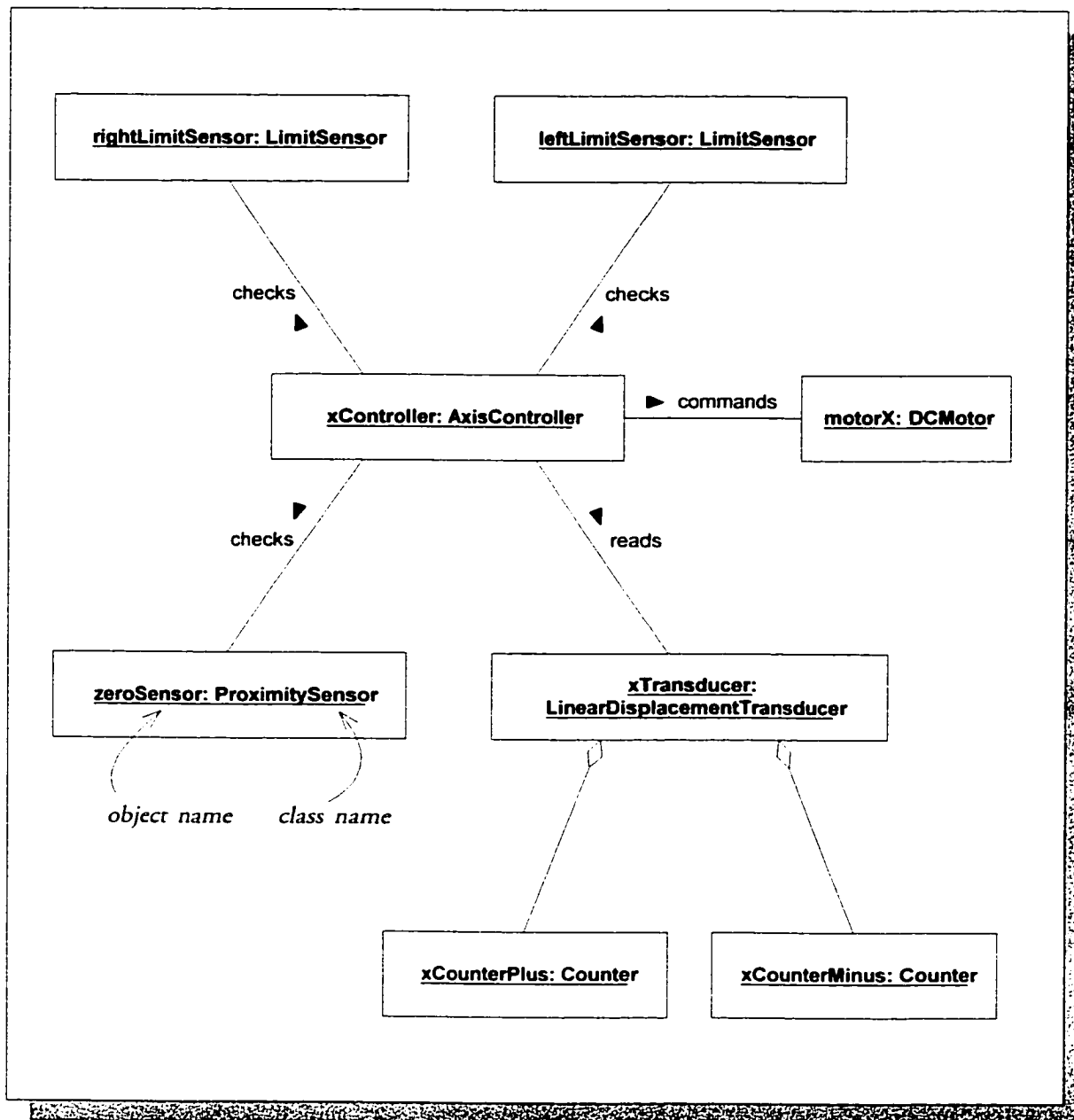


Fig. 3.8 Example of Object Diagram: Excerpt from the ACTS Specification

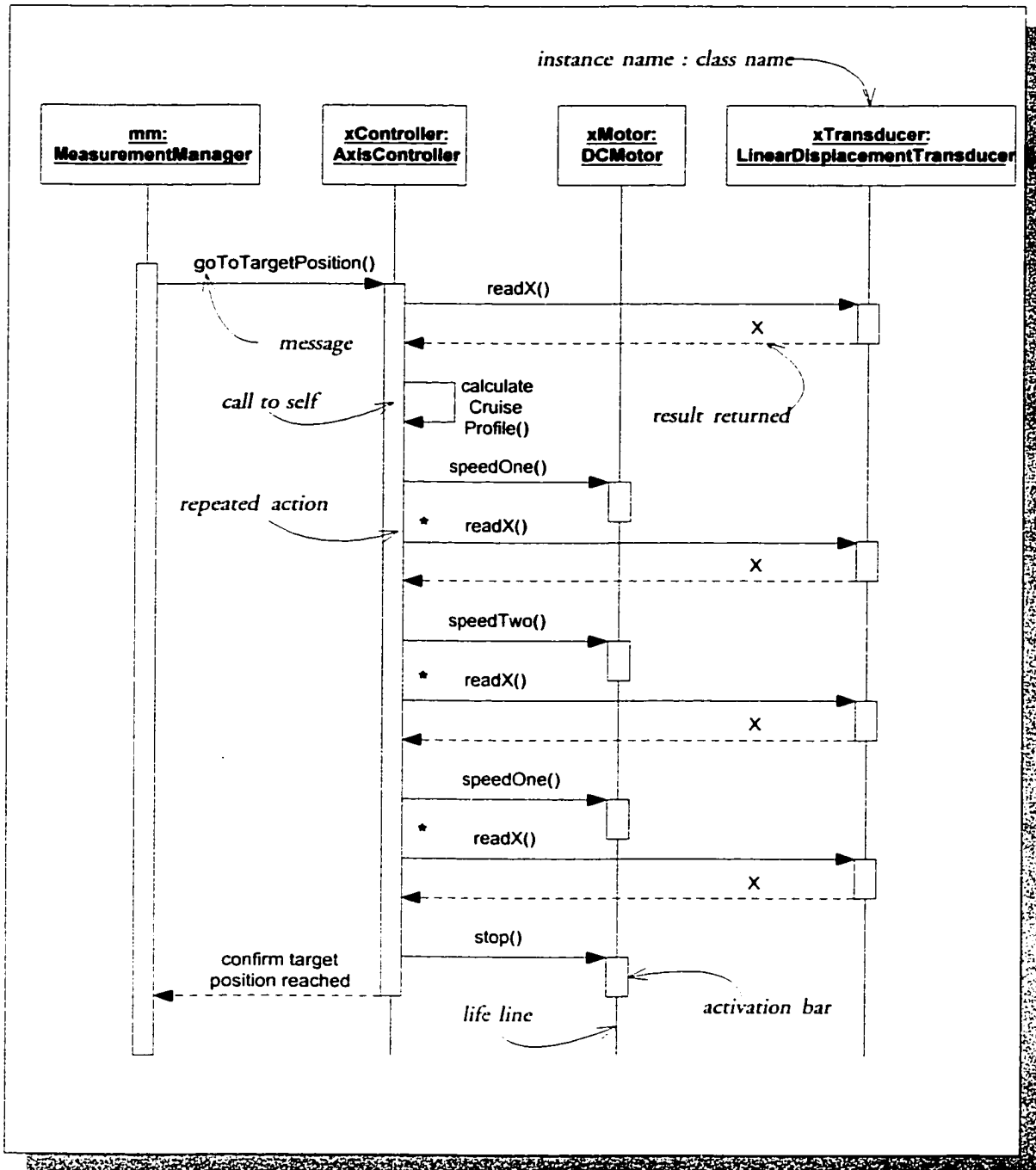


Fig. 3.9 Example of Sequence Diagram: Excerpt from the ACTS Specification

3.3.2 UML Support for Modelling Real-Time Systems

As described in [Booch98], UML provides support for modelling real-time systems via a number of special constructs and mechanisms.

Firstly, there are events and signals. According to Booch, Rumbaugh, and Jacobson, *events* are “things that happen” [Booch98, pp. 37], each event specifying some occurrence that has identifiable location in time and space. A *signal* is a particular type of event that represents an asynchronous stimulus communicated between instances of classes. Other types of events are *call event*, *passage of time event*, and *change event* (Table. 3.V). Signals are similar to classes; they can have instances, attributes, and operations, and can be organised in hierarchies. Represented as classes stereotyped with the <<signal>> mark, they are in relation

Type	Description	Symbol
signal event	an occurrence of interest packed as an object and dispatched asynchronously from an object to another	<<signal>>
call event	method invocation from an object to another; synchronous notification from the caller object to the object whose operation is invoked	no special symbol, graphical notations for regular operation invocation apply
passage of time event	event that specifies that a given duration has elapsed	after (duration)
change event	event that indicates the satisfaction of some condition (typically based on the changing of some attribute's value);	when (condition)
	in particular, can be used as a time event, marking the arrival of an absolute moment of time	when (time_value)

Table 3.V Types of Events in UML

“send” with the class operation that dispatches them. As a notational convention, receivers of signals may include in their class symbol, below attributes and operations, an additional compartment showing the list of accepted signals. An important kind of signal is *exception*, predictably stereotyped with the mark <<exception>>. All types of event can be either *internal* or *external* to a system, *synchronous* or *asynchronous* (depending on whether or not their sender waits for the receiver’s response), and can be *multicast* (send to a specified set of receivers) or *broadcast* (dispatched to all objects in the system that might be listening). Events other than signals are typically involved in state diagrams only as transition triggers but they can also be modelled as classes. Considering again the ACTS problem used in Subsection 3.3.1 but opting this time for a concurrent solution (several threads of control involved), an example of signal communicated between classes is given in Fig. 3.10.

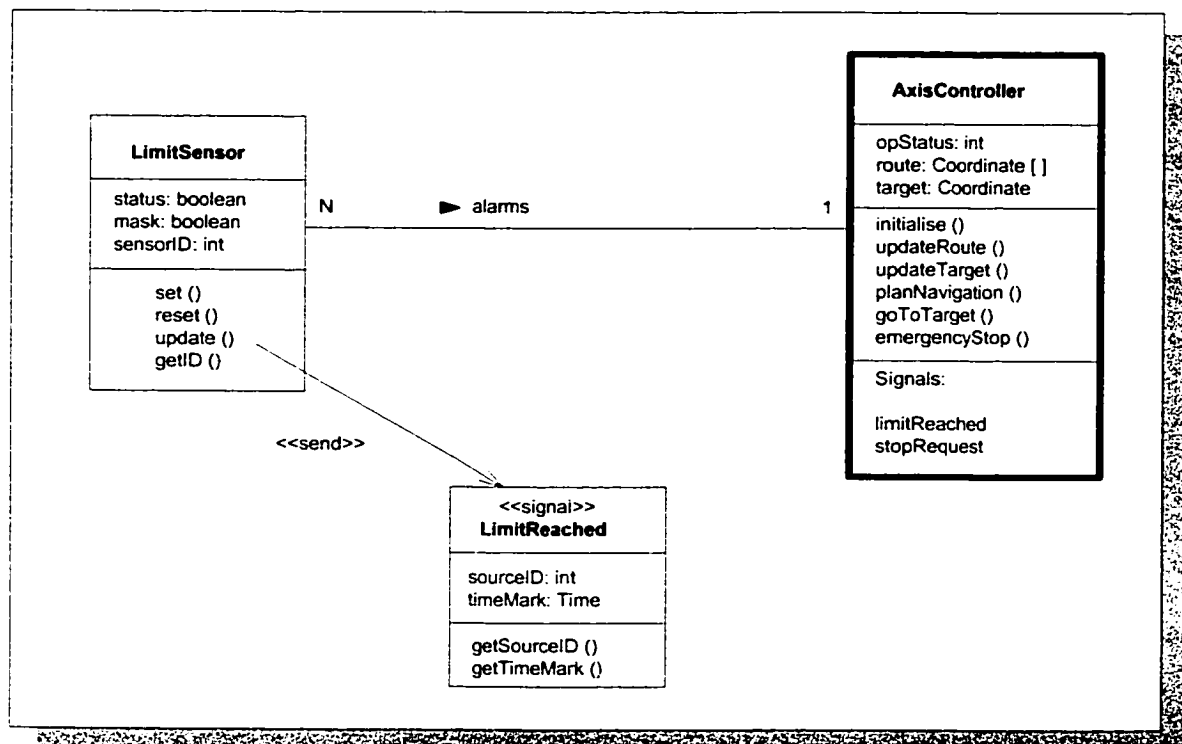


Fig. 3.10 Example of Signal: Excerpt from the ACTS Specification

In this figure, the thick-line class “AxisController” is an active class, as explained below, while “LimitSensor” is a regular class whose objects are within the flow of control rooted in some active class (not included in the diagram).

Secondly, there is UML support for describing processes and threads. While a *process* represents a heavyweight flow of control that is known to the operating system, has its own address space, and competes with peers on the same node, a *thread* is a lightweight flow of control that runs in the address space of an enclosing process and competes with peers within this process. In a concurrent system an *active class* is used to show an independent flow of control, each *active object* representing a thread or process that can initiate control activity. An active class can be stereotyped either as <<process>> or <<thread>>, is graphically represented with a thick line, and typically contains the extra compartment “signals” in its representation. Communication between thread objects can be achieved using either signals (asynchronous) or call events (synchronous) while process objects usually communicate via message passing (asynchronously) or remote procedure calls (synchronously). As shown in Fig. 3.11, UML includes graphical symbols for representing both asynchronous communication, which has mailbox semantics, and synchronous communication, characterised by rendezvous semantics. Two special cases of rendezvous are also included in Fig. 3.11: *timeout rendezvous*, meaning that the sender will wait for the receiver to respond to the message up to some preset period of time before aborting the transmission and continuing with its processing, and *balking rendezvous*, describing the situation in which the sender aborts the communication and continues its processing if the receiver is not immediately ready to accept the message [Booch94]. In UML it is also possible to indicate a critical region by attaching the {concurrent} constraint to operations.

Thirdly, although not restricted to the specification of RTS, finite-state machines and statechart diagrams are of great help for modelling such systems, especially if the behaviour of these systems is event-driven. A *finite-state machine* serves for representing the lifecycle of objects and contains a set of states and all possible transitions among these states. A *state* can be viewed as a situation in the life of an object during which it performs some activity, satisfies a specific condition, or simply waits for an event. A *stable state* is a state in which the object may exist for an identifiable period of time. In its most complete description a state has a *name*, executes some *entry/exit actions*, contains *internal transitions* (which are

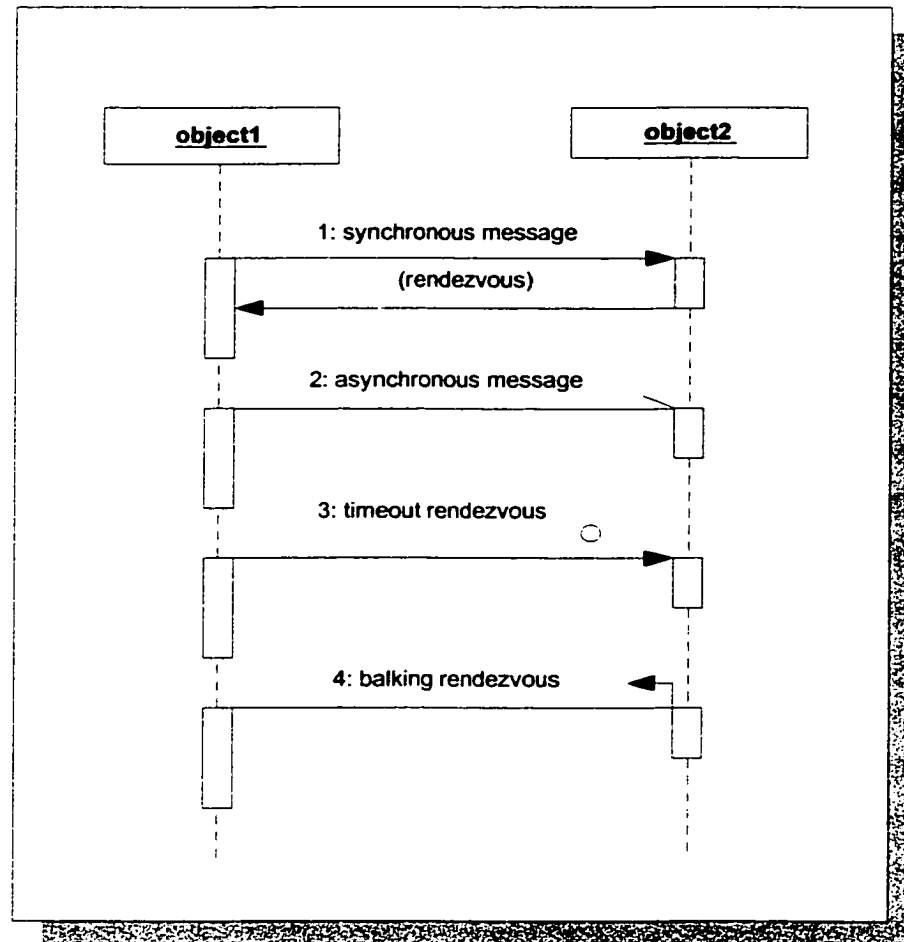


Fig. 3.11 UML Symbols for Synchronous and Asynchronous Communication

handled without causing a state change), has a number of *substates* (either sequential or concurrent), executes some *activities*, and declares *deferred events* (events queued and handled by the object in a different state). A *transition*, in its most complete form, has five parts: a *source state*, the state from which the transition originates, an *event trigger*, the event that makes the transition eligible to fire, a *guard condition*, a Boolean expression that must be true for the transition to take place, an *action*, which is an atomic computation and may act directly on the object described by the state machine and indirectly on other objects, and a *target state*, the state that becomes active as the result of firing the transition. Modelling reactive objects means specifying stable states, events, actions that occur on state changes, conditions for these actions to take place, as well as initial and final states. It is useful to note

that in a Mealy machine actions are attached to transitions, while in a Moore machine actions are attached to states, both approaches being handled properly by the state and transition concepts described above. A *statechart diagram* (see also Section 3.3.1) shows a state machine, emphasises the flow of control from state to state, and can be used in the context of the whole system, of a subsystem, or of a class. In Figure 3.12 an example of finite state machine is presented, illustrating some of the most commonly used state-transition

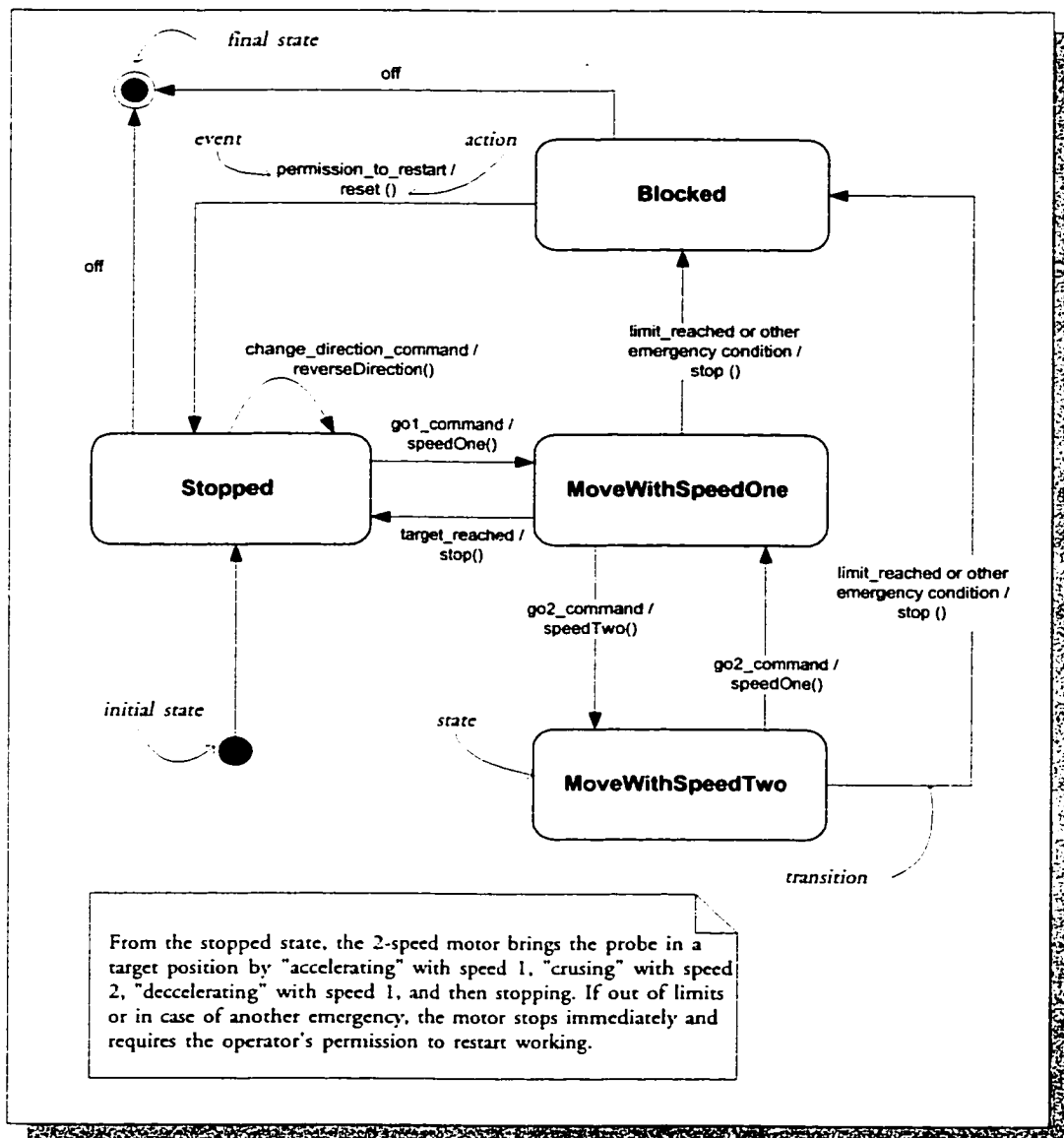


Fig. 3.12 Example of Statechart Diagram: A 2-Speed DC Motor for ACTS' Axis X

elements. The example is again related to the ACTS problem and describes the operation of a 2-speed DC motor that moves the measurement probe on axis X under the supervision of the X axis controller.

Fourthly, modelling RTS can make use of some UML markings and expressions that are dedicated to capturing time and space properties of systems. These are *timing marks*, *time expressions*, *time constraints*, and *location tagged values* (Table 3.VI). Additionally, *semantics tagged values* can be attached to operations and a time expression may be used to specify the operation's *time complexity*: minimum, maximum, and/or average execution time.

Table 3.VI UML Markings and Expressions for Time and Location

Type	Description	Example
timing mark	indicates the time of an occurrence and is denoted by a lowercase letter attached to an event (message)	a: updateRoute() b: goToTarget()
time expression	an expression that evaluates to a time value; it may include the predefined functions for messages <i>startTime</i> , <i>stopTime</i> , and <i>executionTime</i>	b.startTime < a.stopTime + 0.5
time constraint	a statement about relative or absolute value(s) of time; as all other UML constraints it is represented by a string in brackets	{a.executionTime < 2 ms} {every 12 hours}
location tagged value	specification of a component's placement in a node; typically written below the name of the component, it is primarily used in deployment diagrams	{location = RemoteController}

Finally, the modelling of any particular class of systems, including RTS, is supported by UML's *extension mechanisms* described in Subsection 3.3.1. In particular, stereotypes are of

great value for satisfying the modelling requirements of particular types of application and for supporting specialised methodologies. A prominent example in this respect is Selic's definition of new stereotypes, including <<capsule>>, <<protocol>>, and <<port>> [Selic99b]), as means of expressing the concepts put forth by the ROOM approach.

However, UML alone is not sufficient for rigorous development of RTS and supplementary formalisation is necessary [Evans99, Howerton99, McUmbert99, Alagar00]. Solutions in this direction have already been proposed, as indicated in the next Subsection, where several of UML's avenues of development are overviewed.

3.3.3 The UML Promise

As a new and promising language for specifying object-oriented systems, UML has been recently approached from various perspectives by practitioners and researchers. Without claiming that our survey of the latest developments involving UML has exhaustively covered the related literature, we have identified several major directions of exploiting the benefits brought by the notation. These directions, some of which intersect in many of the surveyed reports, can be described as follows:

- *Application to the development of industrial systems*, from medium-sized to complex. The range of reported applications extends from the construction of an MPEG-4 decoder [Barrios99] to the development of a next-generation AWACS (Airborne Warning and Control System) [Bell99], and encompasses projects such as a wine bottling production line [Becker00], a car radio assembly line [Fernandes00], a gas turbine engine simulation system [Xie99], and a GSM (Global System for Mobile Communication) [Jigorea00];
- *Extensions covering special application domains*. Equipped with the new device, UML, teams from both industry and academia have recently started to employ its modelling capabilities in areas traditionally more difficult to tackle, such as heterogeneous systems and distributed systems. In many cases, extensions to the notation have been proposed,

from simple additions consisting of a few of stereotypes to more intricate modifications involving both syntax and semantics. For instance, in the [Barrios99] proposal aimed at heterogeneous system design the additions consist of a number of tags attached to classes to indicate the target implementation language and of a number of stereotypes for hardware entities, in the [Fernandes00] tackling of embedded systems the extensions come in the form of tagged values called *references* associated to use cases for better correspondence with object diagrams, in the [Conallen99a] paper comprehensive stereotype-based extensions to UML that allow the modelling of Web applications are presented, while in the [Price99] treatment of spatiotemporal data modelling a set of supplementary definitions to UML is proposed, resulting in what the authors have termed STUML (Spatio-Temporal UML);

- *Support for new development methodologies.* In order to accommodate particular aspects of the modelling process within specific application domains or to promote novel software construction approaches not only extensions to UML but also new development methods and methodologies have been devised. Examples of such UML-based methods and methodologies, some of them accompanied by specific extensions of the notation, include D'Souza and Wills' comprehensive Catalysis approach for object and component-based development [D'Souza98], Conallen and Bebick's proposal for modelling Web applications [Conallen99b], Muller's utilisation of UML for database design [Muller98], Cheesman and Daniels' process for server-side component-based development [Cheesman00], Fernandes et al.'s alternative for modelling embedded systems [Fernandes00], Lu et al.'s UDRE (User-driven Domain-specific Requirements Engineering), focused on the user's involvement in the development process and on the relevance of requirements throughout it [Lu99], and Oldevik et al.'s methodology for developing distributed systems, a methodology that employs RM-ODP (the ISO standard Reference Model for Open Distributed Processing) as a conceptual and architectural framework and UML as a flexible modelling notation "that can be used for virtually anything" [Oldevik98, pp.13];
- *Combinations with other notations.* There are also situations where simply extending UML is not sufficient for capturing all the aspects of a particular class of application (or

for capturing these aspects in the desired way), yet employing the modelling power of UML still brings significant advantages. In such cases, UML can act as companion to some other language or languages, and play various roles. Examples of partners for UML include IDL (Interface Description Language) in the object-oriented development of distributed systems [Watkins98], Java in the construction of simulation systems [Kortright97], E-DFDs (extended data flow diagrams) in modelling distributed real-time systems [Becker00], and a variety of rigorous languages such as cTLA (a variant of Lamport's Temporal Logic of Actions) [Graw00], GSBL[∞] (an algebraic specification language based on inheritance [Favre99]), and VHDL [McUmbler99] in specification approaches with enhanced formalism. Since combining notations is also the avenue of research we follow in the present thesis, more about combinations of notations, in particular about those involving UML and variants of Z, is presented in Chapter 4, Related Work, of this thesis.

- *Strengthening of UML's underpinning formalism* and, generally speaking, *formalisation of UML*. Representatively, the need for concerted efforts in this direction has led to the formation of a dedicated group, pUML ("precise UML") whose primary objective is "to clarify and make precise the semantics of UML" [pUML01a] and whose membership include well known scientists such as Andy Evans, Robert France, David Harel, Kevin Lano, Stuart Kent, and Bernard Rumpe. The two working groups of pUML focus on building a rigorous meta-model semantics for UML and, respectively, on defining precise semantics for OCL (Object Constraint Language, the standard constraint language of UML [Warmer98]). Approaches taken by pUML members include the proposal of a meta-model semantics for structural constraints in UML [Kent99], the incorporation of rigorous reasoning techniques within UML's component abstractions and representations [Evans98], the development of an axiomatic semantics model for a large part of the notation [Lano98], and the formalisation of key UML constructs [Shroff97] (for a more complete image of the group's research we refer the reader to pUML's list of publications available at [pUML01b]). Sustained work in the same direction of formalising UML has also been ongoing for some time in other centres of research (e.g.,

- Alagar and Muthiayen's proposal for Real-Time UML [Alagar00] and Alemán and Alvarez's work on foundations of developing UML model verification tools [Alemán00]);
- *Eclectic uses*, for instance as supporting notation in an approach aimed at helping students understand the value of precise specifications [Stoecklin98] or in the generation of the OOHyTime meta-model, whose purpose is to facilitate both the understanding and the utilisation of HyTime (Hypermedia/Time-Based Structured Language), a standard for interchanging hypermedia documents [Scott99];
 - *Tool support*. The growing popularity of the modelling language has also been fuelled by the development of a variety of CASE tools that incorporate support for the notation. Besides Rational Software Corporation, with its vanguard tools Rational Rose [RationalRose01] and Rational Rose Real-Time [RationalRoseRT01], a number of other major commercial vendors have already provided the software development community with CASE tools that incorporate support for the UML notation. Among these, major vendors are TogetherSoft Corporation, developers of Together Control Center, Together Enterprise, and Together Solo [TogetherSoft00b], I-Logix, Inc., with its Rhapsody environments for modelling real-time embedded systems [Rhapsody01], Popkin Software, creators of System Architect 2001 [SystemArchitect01], Computer Associates International, Inc., providers of the application lifecycle management tool Paradigm Plus [ParadigmPlus01], and Microsoft Corporation, who have recently acquired Visio Corporation, the developers of Visio, an intelligent diagrammatic editor [Visio00]. However, it is worth noting that providing comprehensive tool support for UML is not easy to achieve. For instance, according to [Bell99] the coverage of the notation need be improved in the case of the two (unnamed) design tools that were used in the AWACS project and, as the author indicates, provided incomplete support for UML.

A number of observations can be drawn from our survey of the recent UML literature. First of all, that the wealth of projects and applications domains making use of the new modelling language as well as the variety of directions from which the language has been approached provide a solid justification for the term *general notation* associated with UML.

Also, even before the apparition of a dedicated tool such as Rational Rose Real-Time we could note as a fact the significant number of approaches aiming at dealing with complex systems such as real-time embedded systems and distributed systems. This reflects a clear need for notational support for modelling such systems, and it appears that in the near future UML will have a major role to play in the distributed and real-time areas.

The application of UML to specific domains has led to a number of extensions or adaptations of the notation and in not a few cases a new, custom-made methodology has also been proposed. In fact, extensions to the notation appear to represent the norm rather than the exception in the employment of UML. However, this should not come as a surprise, since UML was built from the beginning with provisions for extension, but the spawning of notations can lead to an “inflated” UML, hard to manipulate efficiently. Also, the number of new methodologies proposed is rather large, and unless the proposal of a new methodology is fully justified, the present methodological gusto may actually backfire, and undermine the very idea of unification behind UML. In truth, however, we should note that many of the new methodologies reported in the literature are necessary to fill the gap between UML’s generality and the development idiosyncrasies of specific application domains. Also, the impending “stabilisation” of the Unified Modelling Process will most probably reduce significantly the number of new methodologies that bring only marginal modifications to existing practices.

Finally, we should note the quasi-unanimous acceptance of UML and the fact it is generally perceived as a very useful tool, with luminous future. For instance, Xie et al. are particularly satisfied with the support provided by UML for iterative development and the value of frequent shifts between views, especially between the use-case view and the logical view, which are deemed to accelerate the understanding of the requirements and the developing of new ideas [Xie99]; Barrios and Lopez’s study highlights UML’s versatility in supporting, at higher levels of abstraction, the design of complex systems that necessitate mixed implementation solutions [Barrios99]; the [Becker00] paper shows that it provides adequate input to a CASHE tool such as SIM2SYS (developed by the authors to support their

methodology); Lu et al. consider that UML's semantics help the clarification of requirements while its diagrammatic notations enhance the "understandability, traceability, verifiability, and modifiability of the requirements" [Lu99, pp. 133]; and Watkins et al. deem it a "rich methodology," well equipped to "express the requirements of large systems" [Watkins98, pp.149].

Of course, not all UML is shining brightly in the limelight, and beyond shortcomings mentioned at the beginning of Section 3.3 and inherent limitations that have triggered a significant number of UML extensions and combinations with other notations there are also some other deficiencies in the language, for example logical flaws in the definition of certain UML concepts, as pointed out by [Simons99] who strongly disputes the soundness of a number of features pertaining to use cases. But the vast preponderance of useful features present in the language, as well as UML's versatility and widespread acceptance have provided us with good reasons to decide to employ it in our approach. Of course, because the magic concoction is still boiling over the fire, we probably have to cool it a bit first and then see if its flavour is the one promised by its tempting aroma. Or, to put it in a different way, we believe that only time and practice will tell us what things deserve to stay.

3.4 Chapter Summary

In this chapter the two specification languages, Z and UML, that pertain to the thesis' research space and further characterise the topic of this dissertation have been presented. Examples of application as well as surveys of both Z and UML landscapes, including descriptions of extensions, research directions, and ways of utilisation, have been included. UML's support for RT software development has been described and Z++, the object-oriented variant of Z used in the thesis has been succinctly introduced, with a view of further detailing it in Chapter 6, where it provides fundamental support for the formalisation approach proposed in this thesis. Remarks on the current expectations raised by UML have also been presented.

4 RELATED WORK

"But search the land of living men
Where wilt thou find their like agen?"

[Walter Scott, Introduction to
Canto First, *Marmion*, 1808]

4.1 Introduction

The purpose of this chapter is to narrow the research space, focus on the topic location and discuss current specification approaches that are related to ours. Based on the examination of these approaches, the contour of our work can be drawn with greater accuracy, leaving to the remaining of the thesis the task of completing the detailed picture of our approach. Some general observations regarding the integration of notations in software specification are presented first, followed by a brief review of a number of semi-formal/formal combinations of notations involving formalisms other than Z. Then, the examination of integrations of notations is narrowed down to research projects that involve Z or variants of Z. In particular, five approaches that share significant characteristics with the modelling solution presented in this thesis are discussed in more details and both commonalities and differences are highlighted. Because the approach proposed in this dissertation places special emphasis on capturing temporal properties of systems, a review of existing modalities of dealing with time in the context of Z-based specifications is also included.

4.2 Integration of Semi-formal and Formal Notations in Software Specification

Integrating formal with semi-formal or informal notations in software development is not a new idea, some forms of combinations being present in a fair number of approaches. After all, formal languages like Z include provisions for textual, plain language annotations, intended to alleviate the difficulty of following complex mathematical expressions and to relate abstract descriptions with real-world entities. However, as pointed out by several authors, one of the main reasons that, in addition to lack of tools support, have prevented the wider application of formal methods is that not sufficient attention has been paid to the integration of formal techniques with traditional, semi-formal methods [Gerhart94, Clarke96, Lawrence96].

Many authors consider the integration of formal techniques with conventional, informal (or semi-formal) approaches as highly beneficial in software development. For instance, [Aujla94] points out that formal techniques are portable and extendable and can be used in various ways and in various phases of the development. They can be applied as complementary techniques or as alternatives to conventional approaches. Their application leads to the detection of a significant number of errors in specifications. On the other hand, Aujla et al. show that formal techniques themselves benefit from being included in the larger frame of an integrated methodology; they are provided with both context and method, which they may lack if considered in isolation. Alexander sees the combination of formality and informality as a way to obtain “the best of both worlds” [Alexander95] and Bruel et al. point out that “the main objectives of integrated formal/informal approaches is to make formal methods easier to apply and to make informal methods more rigorous” [Bruel98b, pp. 52].

Integration, which in general covers combination of notations, models, and even methods [Bruel98b], has nevertheless its own issues, most notably the fact that interpretations underlying the translation rules from informal to formal are seldom explicitly stated, the

focus of formalisation is in general on basic constructs, and not structures, and little attention is paid to relating the results of analysing the generated formal models to the corresponding components of the informal counterparts (Bernhard Rumpe, in the [Bruel98b] panel).

However, in general, using complementary, concerted techniques for modelling software systems brings a series of benefits, the most important being the increased modelling power provided by the combination and the higher level of confidence they bring in regarding the correctness of the software product being developed. Evidently, these advantages did not pass unnoticed by the researchers and practitioners of the software engineering field, and various combination strategies have been proposed. Some of these strategies are briefly reviewed in the rest of this chapter but, before that, it is useful to point out that, in broad terms, the relationship between the formal and the semi-formal (or informal) components of a specification can be one of the following (notice that we refer in particular to semi-formal or informal graphical notations):

- If the graphical (semi-formal or informal) part is built initially and then a translation process is applied to obtain its formal counterpart, we can speak about *derivation* of the formal model from the informal model or simply of *formalisation* (e.g., [Lee95], where diagrammatic and text elements of Bailin's object-oriented requirements specification method OOS [Bailin89] are translated into Z counterparts, or [Laleau00], where the translation is from UML to B). Certainly, it is also possible to obtain a visualisation of the formal part, in which case the derivation is from formal to visual (e.g., [Salek94], where the REVIEW system is used in the larger frame of the METAVIEW meta-system –which facilitates the development of CASE environments– to generate natural language descriptions from Environment Definition Language (EDL)/Environment Constraint Language (ECL) specifications, or [Kim99b], where graphical representations for Z constructs are proposed). The later form of derivation can also be called *deformalisation*;
- If in addition to diagrammatic representations some related formal specifications are produced independently (e.g., [Jia97], where Z specifications supplement UML models),

the approach can be characterised as *complementary formalisation*. Typically, this approach also involves derivation from informal to formal, a subset of the diagrammatic description of the system being translated into formal specifications (this is the case for the cited [Jia97] approach, which is discussed in more detail in Subsection 4.5.1);

- if changes in any of the specification's parts are continuously propagated in the other, we can speak of a *tight integration of notations* (e.g., [RoZeLink99], where UML models are connected to corresponding ZEST descriptions).

In the above classification the terms semi-formal part and formal part of a specification are used but we should point out that, due to the costs involved, formalising the entire specification of a software product is generally impractical, if not impossible, and the typical approach is to apply formal techniques only to the critical sections of the software being developed [Gerhart94]. As such, the correspondence between the diagrammatic (semi-formal or informal) and textual (formal) parts of a specification is typically limited to a subset of the specification's components.

Depending on the number of notations involved, a combination of notations can take the form of either a *dual-notation integration* (e.g., [Björkander00], where UML is combined with SDL) or of a *multiple-notation integration* (e.g., the multi-paradigm specification technique devised by Zave and Jackson [Zave96], the pure formal method integration (PFMI) strategy suggested by Paige to allow the combined usage of formal methods such as Z, refinement calculus, predicative programming, and Larch [Paige98], or the framework solution proposed by Day and Joyce for integrating multiple notations [Day00]). Generally speaking, the integration does not necessarily involve a *formal/semi-formal* (or *informal*) combination; it can be of the *formal/formal* type (e.g., [Sowmya98], where the dynamic aspects of RTS are modelled using both Statecharts and FNLOG, a logic-based language built on first-order predicate calculus and TL) or *semi-formal/semi-formal* (e.g., [Scogings01], where an integration UML/Lean Cuisine+ is proposed for supporting the early stages of interactive system design). And, as mentioned in the classification proposed above, it has also been considered useful to deformalise the formal models [Salek94, Kim99b].

For the purpose of comparing various integration approaches we also introduce the notion of *monolithic environment*, which means that a single CASE tool is used for developing both semi-formal and formal models, and various subsequent formal processing (such as analysis and refinement) can be invoked from this tool. The alternative, the *non-monolithic environment*, refers to a combined use of CASE tools, with separate invocations from the operating system.

We believe that the integration of notations does provide a viable solution for modelling complex systems because various aspects of the systems need various ways of description, which can beneficially complement each other (for a classification and examination of forms of method complementarity, primarily in terms of notations and processes, we suggest [Paige99]). In particular, in the case of formal/semi-formal integrations, it is always possible to “fine tune” the formality level and adjust the balance between the less rigorous diagrammatic representations and the formal specifications to best answer the needs of a given application. An important point we must not forget about integration is, as well stated by Clarke et al., that the end result should be a compound, and not a mixture (in other words, a solution in which the components are tightly united, and not one in which the components simply intermingle) [Clarke96].

Within the research space introduced in Section 2.1, we look next at a number of integration approaches that propose a formal/semi-formal combination of notations.

4.3 Semi-formal/Formal Integrations of Notations Not Involving Z

By resorting again to the classification proposed in Table 2.1 and to the research space presented in Fig. 2.1 we can detail now the areas that neighbour our thesis’ C3+ topic location and have a look at research approaches that “reside” in these areas. Figure 4.1 contains an enlarged depiction of a major portion of Fig. 2.1, which covers all 16 combinations of integration as defined in Section 2.2 (recall that the classes –or *areas*– C1•

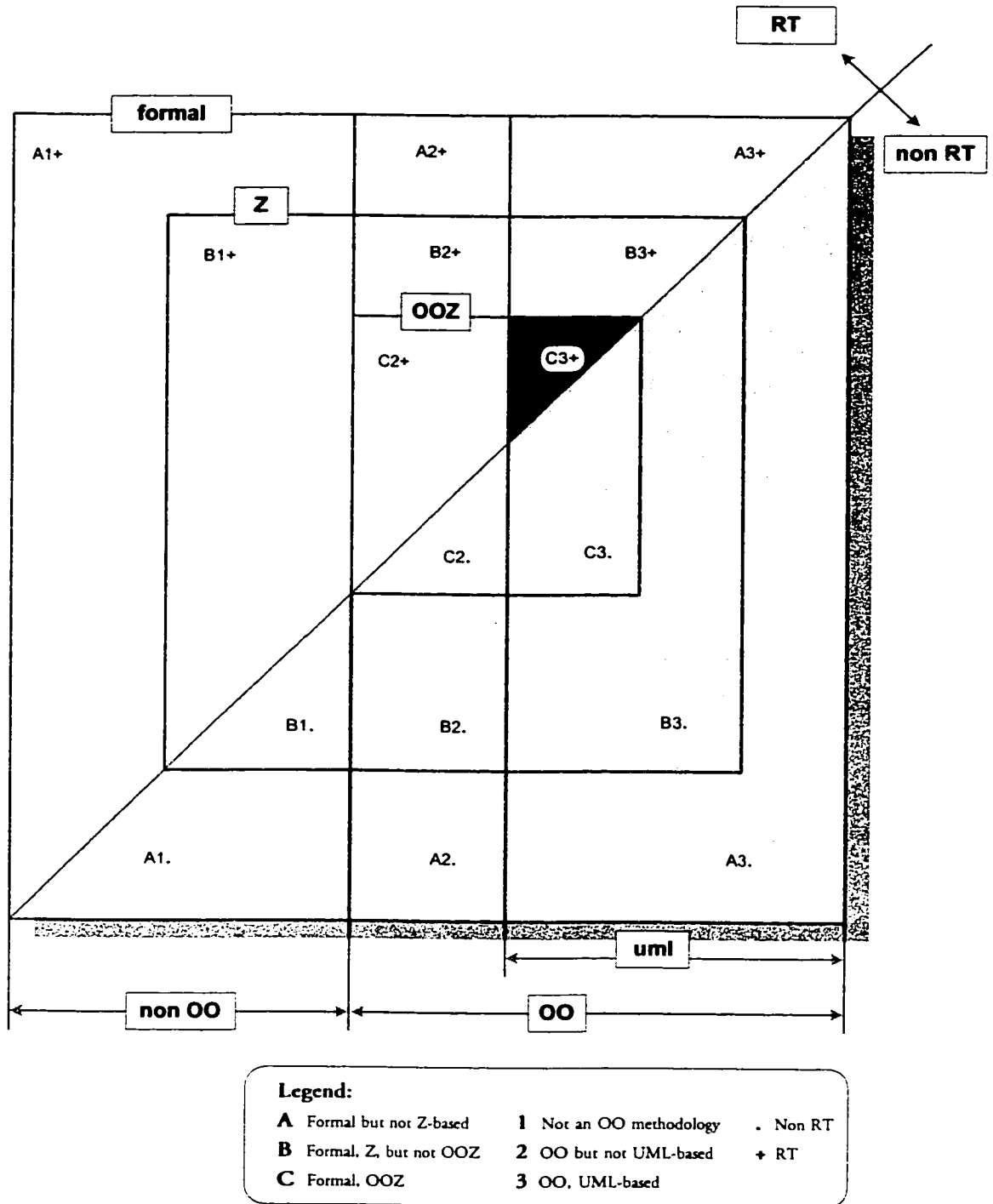


Fig 4.1 First Zoom-In on the Research Space

and C1+ are discarded). In this section we focus only very briefly on some examples of projects that fit in the A1• to A3+ areas, as an introduction to the next section, where Z-based approaches are discussed. This introduction is intended to be simply illustrative and by no means comprehensive (areas A1 to A3 are quite large, because they include “everything but not Z” of all possible semi-formal/formal combinations of notations that cover our research space). Examples of A-type approaches, with succinct descriptions, are presented in Table 4.I. It can be inferred from this table that the topic of semi-formal/formal integration has been pursued constantly by researchers, and no remote area (“remote” in the sense defined by our classification) has been left uncovered.

Table 4.I Examples of Semi-formal/Formal Integrations Not Involving Z

Area	Area Characteristics	Example Approach	Summary Description of the Example
A1•	Formal non-Z, Non OO, Non RT	[D’Almeida92]	Translation from Modified Entity-Relationship diagrams (MER) and textual Keyboard-based Formatted Descriptions (KDF) to VDM
A1+	Formal non-Z, Non OO, RT	[Sahraoui97]	DFD-based methods integrated with TL constructs of the Zaman language [Sahraoui92]
A2•	Formal non-Z, OO non UML, Non RT	[Cheng94]	The VISUALSPECS environment supports the formalisation of OMT models in algebraic languages such as Larch
A2+	Formal non-Z, OO non UML, RT	[Chen98]	Integration of HRT-HOOD (Hard Real Time- Hierarchical OO Design) models [Burns95] with TAM (Temporal Agent Model) specifications [Scholefield92]
A3•	Formal non-Z, UML, Non RT	[Laleau00]	B specifications generated from UML diagrams
A3+	Formal non-Z, UML, RT	[Bordbar00]	Petri Nets serve for representing and analysing dynamic models in a UML-based approach aimed at modelling discrete-event dynamic systems

4.4 Semi-formal/Formal Integrations of Notations Involving Variants of Z

After the introduction to semi-formal/formal integrations of notations based on examples that do not involve Z, it is now time to look at the closer neighborhood of the thesis topic, represented by areas B2• to C3+, which make up the “Z sub-domain”. The best way to do this is to enlarge again the original representation of Fig. 2.1 and discard the peripheral areas A1• to A3+, thus resulting the depiction shown in Fig. 4.2. Examples that serve the

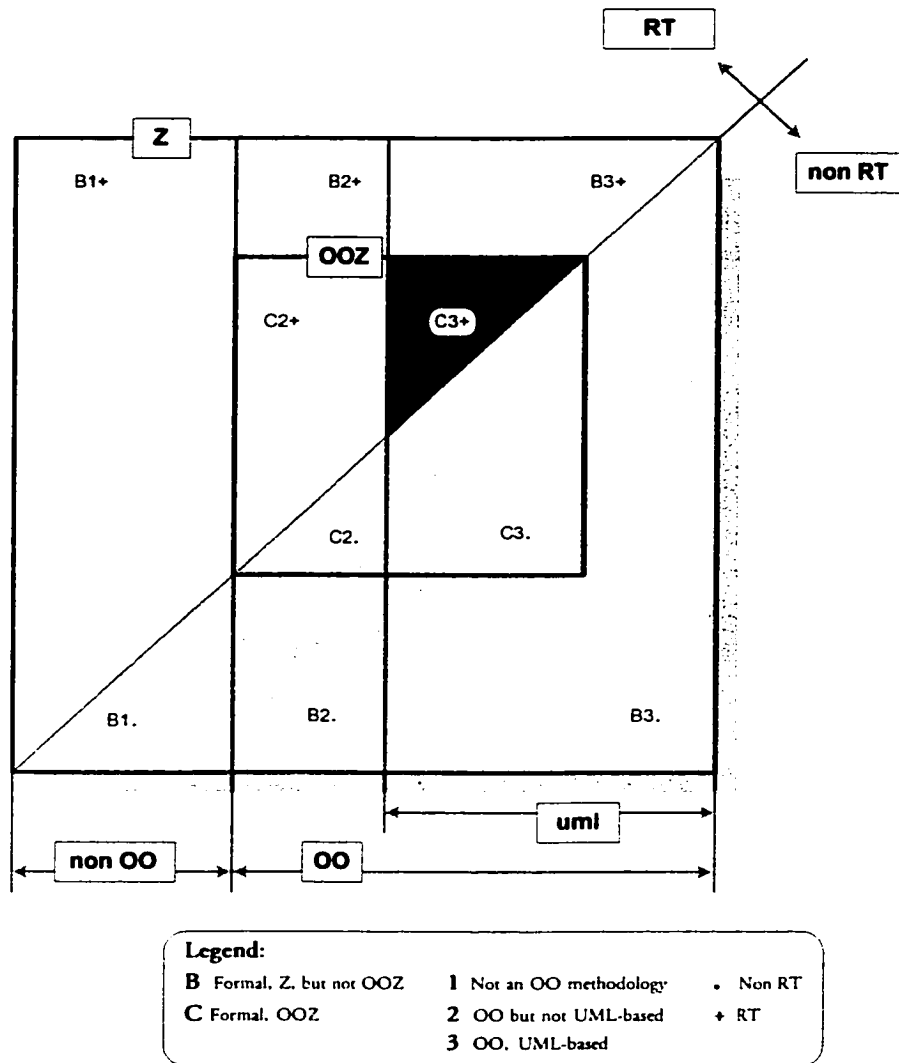


Fig. 4.2 Second Zoom-In on The Research Space

illustration of integration classes proposed in Chapter 2 are given again in tabular form (Table 2.II). Regarding the completion of this table, it can be noted that even though all ten areas of the “Z sub-domain” have been covered, examples for some classes have been more difficult to find than for other. In particular, the example for B1+ is the only one we find after a rather long search (typically, in the earlier approaches, when Z was integrated with notations of structured approaches, the focus was not on RT applications). Also, for the C2+ category we had to resort again to [Lano95], the only other candidate we found being [Dong97b], but there the addition of an OMT description to the Object-Z specification of a multiple-elevator controller is rather accidental, and not suggested as an integration approach per-se. The closer categories B3 to C3 are also not very populated, and in fact the few approaches that fit in these areas of the thesis’ topic’s “near vicinity” constitute the more restricted group of “closely related approaches,” discussed next at the last and most detailed level of investigation of the thesis’ research space.

4.5 Closely Related Approaches

While, as previously shown, there are numerous approaches that integrate in various degrees graphical, semi-formal representations with formal notations, very few are aimed at explicitly dealing with TCS using a Z-based formalism incorporated in the larger frame of the OO paradigm. We have identified five specific approaches that in our view are the closest to the direction of work that we have pursued. However, of the five approaches, only two include provisions for explicitly dealing with temporal properties of the systems, as we also have attempted.

4.5.1 Jia’s Augmented Object-Oriented Modeling Language

Xiaoping Jia, the author of the well-known Z type checker ZTC [Jia98a], takes a pragmatic approach in combining the strengths of semi-formal graphical OO notations with those of formal specifications [Jia97, Jia98c]. The author indicates that only a partial automation of

Table 4.11 Examples of Semi-formal/Formal Integrations Involving Z

Area	Area Characteristics	Example Approach	Summary Description of the Example
B1•	Z but not OOZ, Non OO, Non RT	[Aujla94]	ERD and DFD formalised using Z within the Rigorous Review Technique (RRT)
B1+	Z but not OOZ, Non OO, RT	[Coombes92]	Formalisation in Z of casual timing diagrams (diagrams inspired from those used by electrical engineers to illustrate temporal properties of digital devices)
B2•	Z but not OOZ, OO but not UML, Non RT	[Lee95]	Constructs of Bailin's OOS method transformed into equivalent Z specifications (also mentioned in Section 4.2)
B2+	Z but not OOZ, OO but not UML, RT	[Bruel96]	Fusion models translated into Z specifications (precursor of [France97] shown in the B3+ area)
B3•	Z but not OOZ, UML, Non RT	[Jia97] [Noe00]	Formalisation in Z of UML constructs (details in Subsection 4.5.1 and, respectively, 4.5.2)
B3+	Z but not OOZ, UML, RT	[France97]	Structural and behavioural Octopus analysis models expressed in UML are formalised using Z (details in Subsection 4.5.3)
C2•	OOZ, OO but not UML, Non RT	[Nguyen96]	Proposal of a 4-submodel specification based on the integration of OMT and Object-Z* (a slightly modified version of Object-Z); RT properties not explicitly targeted
C2+	OOZ, OO but not UML, RT	[Lano95]	Formalisation of OMT constructs in Z++ (more details in Chapter 6)
C3•	OOZ, UML, Non RT	[RoZeLink99]	Two-way link between UML constructs supported by Rational Rose 98 and ZEST specifications (details in Subsection 4.5.4)
C3+	OOZ, UML, RT	[Kim00b]	UML and Object-Z combine forces for describing a lights control system (details in Subsection 4.5.5)

code generation can be achieved from semi-formal OO models, in the form of a skeletal implementation. Thus, in his approach formal notations are used for partial description of the system, as a complement of the traditional OOAD models (as indicated in the definition of complementary formalisation proposed in Section 4.2). The approach is driven by practical reasons and its aim is to minimise changes and extensions of widely-used semi-formal and formal notations while providing an intuitive and easy to use, yet powerful software development method. Specifically, Jia proposes a language denoted AML (Augmented Object-Oriented Modeling Language) that essentially combines notations from UML and Z. For pragmatic reasons, minimal additions to the Z notations have been included (mostly for handling the specification of classes), making up the slightly extended Z notation referred to as Zext. A supporting tool called Venus was developed to provide the very useful capabilities of model analysis, animation of a large subset of Z specifications, refinement of the design based on a fixed, yet comprehensive library of data structures and algorithms, and extensive C++ code generation.

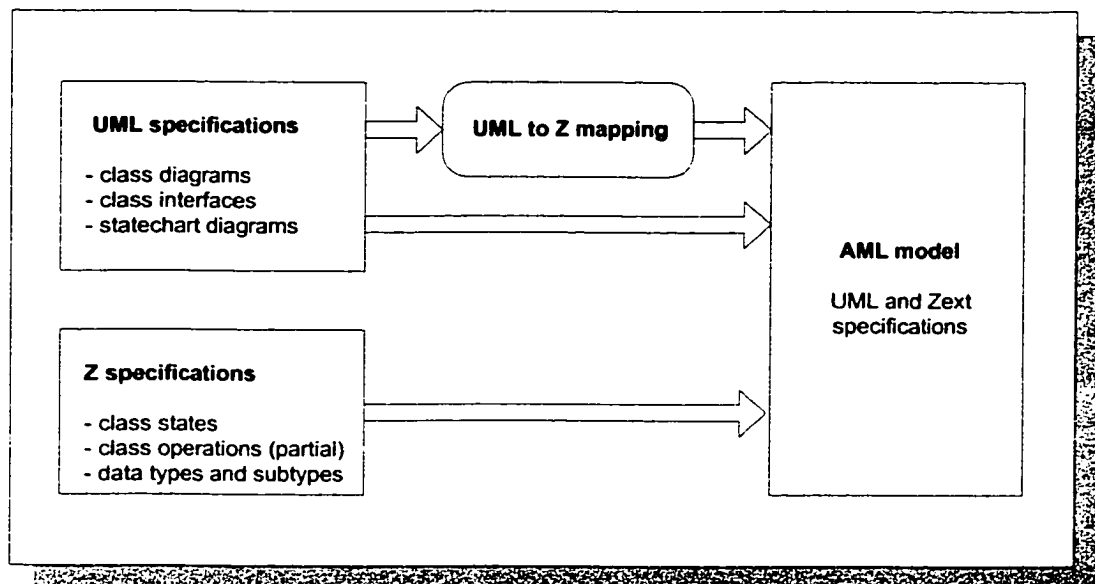


Fig. 4.3 Jia's AML-based Approach

Jia's AML-based approach can be related to the relatively new research direction of light-weight formal methods, succinctly discussed in Subsection 2.6.5. As described by Jia, the

integration of UML and Z is focused on compensating for the limitations of UML, primarily the fact that data types and operations are not formally specified. In essence, as shown in Fig. 4.3, adapted from [Jia97], the UML notation is used to specify the system's class organisation, the class interfaces, and the related state diagrams, while the Z language is employed to provide supplementary details, specifically class states, data definitions, and partial descriptions of operations.

While excellently addressing the practical barriers that hinder the large-scale use of formal methods in practice, Jia's approach differs from ours in a number of ways. Firstly, there is no particular emphasis on capturing time-related property of systems, thus making its application dependent on the modeling ability of UML and on the limited time-capturing capability of the regular Z. Secondly, even though the UML notation is employed, the formal part of the object-oriented model is expressed via a minimal set of extensions of Z, and we believe that by employing a full-fledged object-oriented version of Z additional modeling power would be available, without significant increase of the notation's complexity. Thirdly, it is not indicated whether the opposite translation, the mapping from Z to UML is included. The diagram on which we have based Fig. 4.3 indicates that Z specifications are only fed forward to the complete model, without a corresponding feedback from the integrated AML model to Z descriptions.

4.5.2 Noe and Hartrum's Extension of Rational Rose 98

More recently, Capt. Penelope Noe, from the Air Force Personnel Center, Randolph, Texas, and Prof. Thomas Hartrum, from the Air Force Institute of Technology (AFIT), Ohio, have proposed the extension of Rational Rose 98 for the inclusion of formal specifications [Noe00]. In summary, their approach is to exploit existing features of Rational Rose, specifically Rose's scripting language and available textual fields that can be used for embedding formal expressions, and produce a formalised model that can be fed into the AFITtool transformation system. Based on this input, the AFITtool is capable of generating Ada code (Fig.4.4). From Rose's set of graphical representations, only the class diagrams and the state diagrams are considered, and an additional non-Z and non-UML state transition

table is also employed in the semi-formal specification process. Using primarily the *Documentation* field associated with classes, operations, and state transitions, Z specifications that supplement the description of the system can be embedded into the extended Rose model. These formal specifications are partially written in the L^AT_EX format.

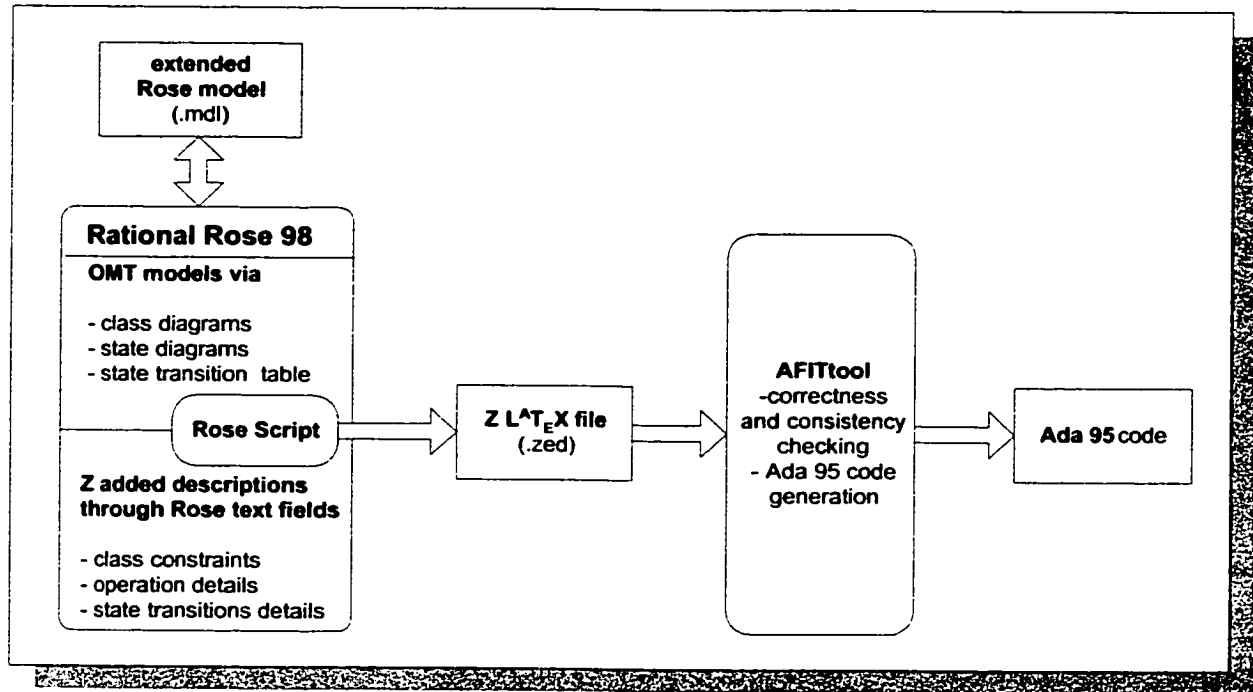


Fig. 4.4 Noe and Hartum's Approach

The approach follows the OMT methodology [Rumbaugh91] and consists of building three models: *the object model*, which defines the class structure of the application, *the functional model*, which describes the desired interaction of the system with its environment, and *the dynamic model*, which expresses the state changes of the system. In summary, the object model is created by augmenting the UML class diagram with Z-specified user defined data types and constraints on classes and attributes, the dynamic model is built using regular finite state machines whose states and events are represented in Z using static schemas, and the functional model is obtained through the description of class operations, details such as pre- and post-conditions being added in Z and the operations being represented by dynamic Z schemas. After the extended model of the system is completed a translation procedure (a

Rose script) is invoked in order to produce a Z file in L^AT_EX format, as entry for the AFITtool. Consistency and correctness checks are performed and Ada 95 code is produced.

This approach is well explained in the [Noe00] paper and its practical utility has obvious merits. In addition, as indicated by the authors, it suggests a viable line of work, that of developing Rose scripts for interfacing with other CASE tools. This approach is different from ours in several ways. Firstly, as in the [Jia97] approach discussed previously, RTS are not targeted explicitly. Secondly, Z is used again in its regular version, which has the advantage of keeping the notation simple and the potential of interfacing with a larger variety of analysis tools, but this solution is less direct than employing an OO variant of Z for OO specifications. Thirdly, many Z descriptions are entered in the L^AT_EX syntax, which is clearly not user-friendly. Fourthly, the internal format of the “.zed” file is custom-made (tailored to the AFITtool), and thus its usage in connection with other tools is restricted. In addition, as in the [Jia97] example, this approach also fits in the complementary formalisation category of integration, and ours proposes a tight-integration solution. Lastly, it can be noted that although Rose 98 acts as the sole front-end modelling tool, Noe and Hartrum’s integration of notations is not entirely monolithic. This is due to the fact that a separate program, the AFITtool, with its own set of commands and interface demands, is invoked outside the main environment, Rational Rose.

4.5.3 Blending Octopus and Z

The approach described by France et al. in [France 97] represents one of the relatively few attempts of integrating OOAD methods with formal specification techniques for developing RTS (work connected to this approach is described as well in [Bruel96], [Shroff97], and [Bruel98a]). The formal specification language used is Z, which was chosen, as indicated by the authors, because of its maturity and the availability of related analysis tools. Here, the combining of an OO approach with a formal specification technique consists of translating the three analysis models of the Octopus method [Awad96] into equivalent Z specifications. The formal specification language Z is used to enhance the modelling capability of Octopus

by allowing consistency checking across models (thus opening the way for the application of automated analysis tools) and by providing the developer with a better insight into the problem's requirements. Octopus analysis models are translated into Z constructs using procedures that could be partially automated. The formalisation process is applied to all three analysis models of Octopus: the *object model* is formalised using class schemata, while the derivation of the other two models, *dynamic* and *functional*, which capture system behaviour, involves four steps: definition of states, definition of subsystem responses to events, Z modelling of transitions described in the dynamic model, and description in Z of statechart actions and activities, including those represented in the functional model (Fig. 4.5, based also on the earlier [Bruel96] paper on FuZE, which combines Fusion and Z).

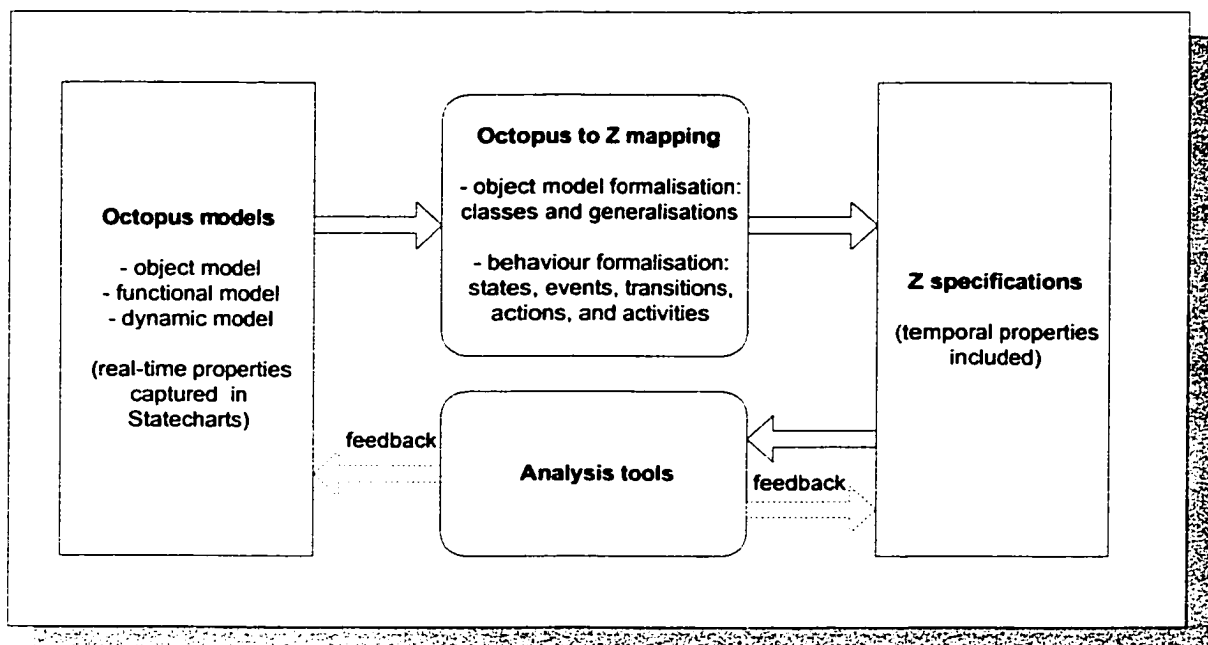


Fig 4.5 The Octopus and Z Integration Approach

Feedback from analysis tools to both the diagrammatic and the formal models is considered, but we can note however that in this approach the integration of notation is not tight in the sense defined in Section 4.2. Also, even though the object model is translated into regular Z constructs that model classes, a translation into an object-oriented version of Z would be more natural and direct, and the specifications would have similar structure in terms of

classes. Additionally, the OMT-based notation of Octopus has currently less exposure than UML, which has enjoyed a constantly growing expansion over the last few years.

4.5.4 Headway System's RoZeLink

Most probably, the only tool that has been developed commercially to support an object-oriented modeling approach and combine the advantages of graphical, semi-formal notations with those of formal notations is RoZeLink [RoZeLink99], produced by Headway Software Inc. as a bridge between the UML notation supported by the 1998 version of the Rational Software Corporation's Rose environment and the ZEST object-oriented formal

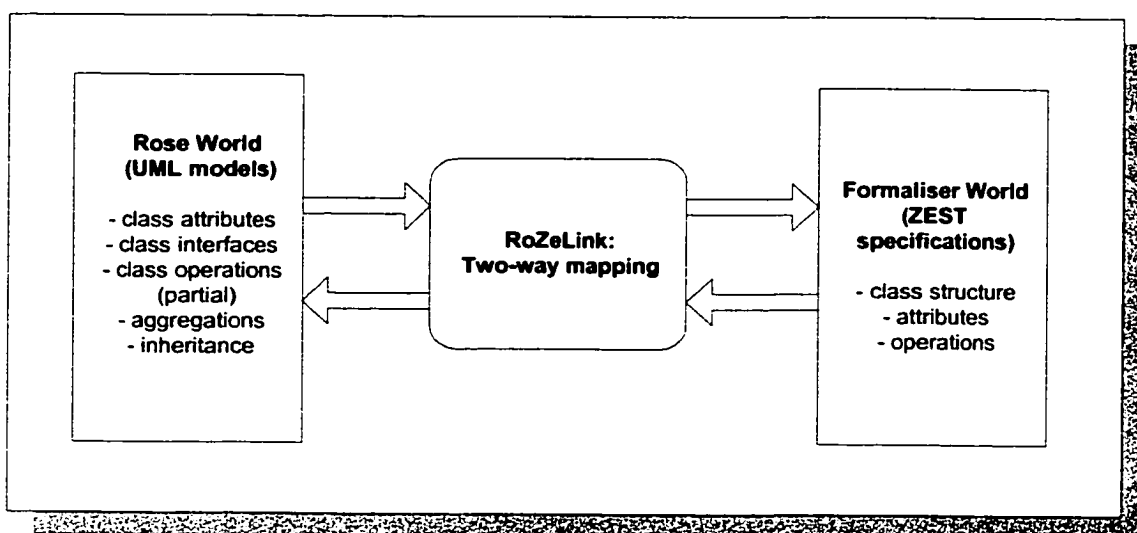


Fig. 4.6 The RoZeLink Tool

specification language supported by Logica's Formaliser [Formaliser01]. RoZeLink, which apparently has not been further developed since the producing company has changed its direction of work (see the web-site in the [RoZeLink99] reference), provides the necessary bi-directional link between the two notations and achieves the goal of maintaining specifications consistent between models. RoZeLink operates between a Rose UML model and a collection of Formaliser documents, and through a continuous translation mechanism maps elements from the semi-formal model into the formal model and vice-versa. This

implies that changes in one “world” are reflected during the modelling process in the other one (Fig. 4.6). We borrowed from RoZeLink this idea and also pursued a tight integration of notations.

Although practical and comprehensive in its dealing with structural aspects of the system, the Headway Systems’ approach has its limitations, primarily because there are no particular provisions for dealing with time-related properties of the systems. In addition, only the class structure of the system is involved in formalisation, the statecharts are not. Also, the RoZeLink approach does not propose a truly monolithic integrated environment, its role being to act as an intermediary that interconnects two already existing commercial software development tools. In order to work both “graphically” and “formally” on his specifications, a user must first start up three separate applications.

4.5.5 Object Z and UML

The closest approach to ours (it belongs in the same C3+ class) and also the most recent is presented in [Kim00b] (earlier work by the same authors on formalising UML diagrams is described in [Kim99a] and [Kim00a]). In many ways, our work is similar to Kim and Carrington’s alternative for integrating UML and an object-oriented variant of Z, but there are also some notable differences, as indicated below. First, however, we would like to indicate that we started to develop our approach in the form presented here sometime in 1998 and an early outline of the integration and of the proposed Harmony tool was presented by the author of this thesis in August 1999 as part of the requirements for the Visual Languages course taught by Prof. Phil Cox at Dalhousie University, Halifax, Nova Scotia [Dasalu99]. Therefore, we have worked independently in the same topic area, and only very recently have learned about Kim and Carrington’s approach.

In summary, as shown in Fig. 4.7, their proposal is to translate UML models into Object-Z models, temporal properties of the systems receiving adequate treatment via a time trace notation based on time refinement calculus. Kim and Carrington’s approach proposes not

only a formalisation of the class structure but also a formalisation of dynamic properties based on use case diagrams, sequence diagrams, and statechart diagrams. Very briefly, there is a direct correspondence from UML classes to Object-Z classes that “makes the semantic translation between the two languages less complex” [Kim00b, pp. 241], and the dynamic behaviour of the system is formalised using detailed translation rules for all elements of the statecharts (initial state, regular states, entry and exit actions and activities, events, and guards).

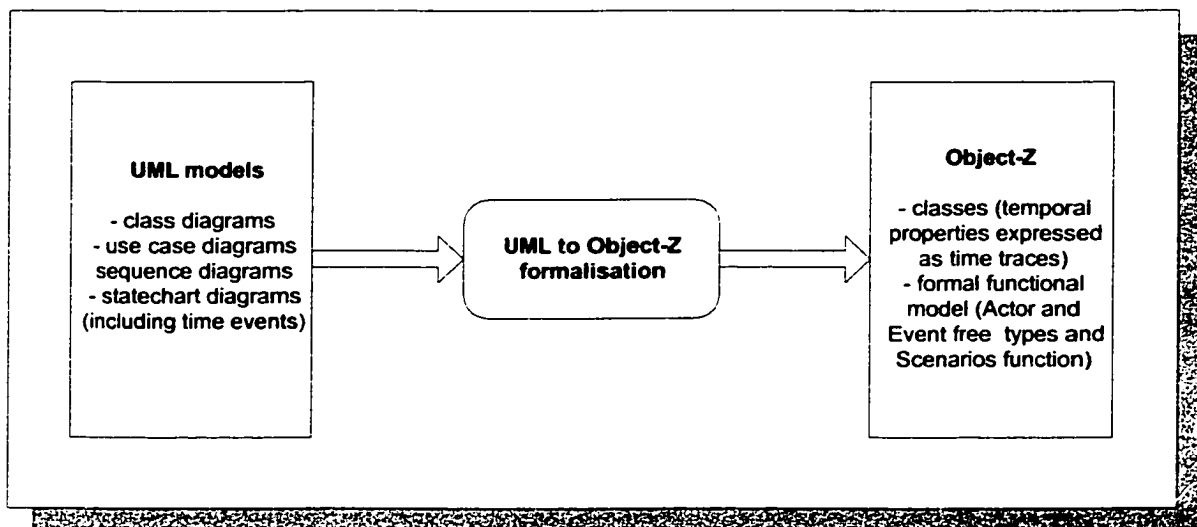


Fig. 4.7 The UML/Object-Z Combination

Kim and Carrington’s approach is one of the most mature solutions for UML and Z integration and has been developed in one of the strongest research groups on formal methods, the Software Verification Research Center at Queensland University, Brisbane, Australia (their web-site is mentioned in the [Cogito97] reference). It builds on extensive research developed over more than a decade by prominent scientists in the field, and benefits from a suite of tools and techniques that have been validated through numerous applications. Nevertheless, our approach also has its merits. It uses Z++ instead of Object-Z and we believe that Lano’s OOZ proposal fares better than Object-Z in some respects, specifically in the details included in the class specification (see class definition in Subsection 3.2.3) and in the integration of the RTL semantics and syntax for explicitly dealing with temporal

properties. In our view, RTL, described in Chapter 5 in conjunction with our formalisation process, has an intuitive and natural syntax and its semantics are easier to grasp by developers not trained in formal methods. Z++ and RTL offer therefore a friendlier user-interface and sustain our lightweight alternative for pragmatic TCS specification. Also, the approach presented by Kim and Carrington does not propose a tight integration of notations (there is not a two-way mapping between semi-formal and formal models) and there is no specific mention of a development tool in their paper, so we cannot ascertain its characteristics under the monolithic/non-monolithic environment criterion.

4.6 Modalities of Specifying Temporal Constraints in Z

Capturing time-related properties of systems is not a simple task. Actually, it is one of most demanding challenges faced by the developers of timed-constrained systems, as emphasised by the profusion of approaches proposed in this direction, including numerous variants of Temporal Logic (comprehensively reviewed in [Bellini00]) and all sorts of “timed” formalisms, including Timed Petri Nets [Ramchadani74], Timed CSP [Schneider92], Timed CCS [Moller92], Timed Statecharts [Kersten92], and Timed LOTOS [Léonard98], (Interestingly, although various alternatives of using Z for specifying RTS have been proposed, the term “timed Z” appears nevertheless in surprisingly few references –and in a rather general way,– so we cannot speak of an established Timed Z notation.) Since a general discussion of the various solutions proposed over the years for capturing temporal properties of systems exceeds the scope of this dissertation we summarise in this Section only some of the most important ways of tackling the “time issue” within Z-based specification approaches. In general, enhancement of Z with constructs and symbols borrowed from other formalisms have been proposed (e.g., [Mahony92, Fidge97, Mahony98, Yuan98]), much fewer being the approaches that attempt to capture timing constraints using exclusively the constructs of regular Z (e.g., [Evans97]). In all cases, special mechanisms for modelling temporal properties have proved to be necessary.

4.6.1 Time Refinement in Z

In one of the earlier approaches that employ Z for modelling time-constrained systems, Mahony and Hayes propose an extension of the notation and the use of refinement calculus to allow a unified treatment of both analog and discrete properties of such systems [Mahony92]. Three “notational devices” are introduced: topologically continuous functions for expressing both analog and discrete quantities, physical units attached to variables, and specification statements describing the assumptions made by the system about its environment and the effect the system is expected to achieve provided the assumptions are satisfied. In order to declare a variable such as the temperature at a given location (say, in an aquarium), a TEMPERATURE type with associated physical units can be first defined by resorting to the set \mathbf{R} of real numbers:

$$\text{TEMPERATURE} == \mathbf{R} \text{ [Celsius]} \quad (4.1)$$

Then, by introducing the type TIME through syntactic equivalence with the set of real numbers (and with an appropriate physical unit attached):

$$\text{TIME} == \mathbf{R} \text{ [Second]} \quad (4.2)$$

the evolution of our variable of interest can be modelled as a continuous total function:

$$| \quad \text{aquariumTemperature} : \text{TIME} \xrightarrow{\ominus} \text{TEMPERATURE} \quad (4.3)$$

On the other hand, discrete variables, such as the following one, which indicates whether or not fresh water is pumped into the aquarium, can be modelled as a partial continuous function from time to Boolean (“partial” because they are possibly times when the variable is undefined or its value is changing):

$$| \quad \text{waterIn} : \text{TIME} \xrightarrow{\oplus} \mathbf{B} \quad (4.4)$$

Other important concepts used in Mahony and Hayes' approach include the notion of open time interval ($\alpha \dots \beta$), the collection τ_{TIME} of all open intervals of time, the time topology T_{TIME} , which encompasses all periods of time consisting of open, disjoint intervals, the $\text{cov}(\text{Period})$ function, which gives the collection of disjoint, open intervals of time that comprise the Period set of time, and timed history predicates such as Pred on Period , Pred in Period , and $\text{Pred at } t$, where Pred is a logical predicate, Period a period of time consisting of a set of open intervals of time, and t a moment in the passage of time.

This approach allows the description of both analog and discrete properties of systems –two aspects of TCS that usually are modelled separately– within a unified framework that at the same time supports the capturing of temporal properties over intervals of time. By associating physical units to variables it helps both the understanding and the type compatibility checking of the specifications. While this approach is expressive and practical for specifying various properties of RTS, including concurrency aspects, it has been pointed out that by modelling variables as constant functions over all time the focus is shifted away from important features of Z , such as state schemas and operation schemas, which become “buried in the specification” [Dong97a, pp.26].

4.6.2 The Quartz Alternative

The Quartz approach [Fidge97] is similar to the previously described work of Mahony and Hayes in that it deals with time and functional behaviour in a unified way, and places equal emphasis on capturing temporal constraints and on specifying functional requirements. The proposed scope of the Quartz approach is however different since it aims at integrating RTS specifications (in a variant of Z) with program refining techniques leading to the generation of Ada-like high-level programs augmented with time constraints. In the words of its authors, “Quartz encompasses real-time software development from specifying the formal requirements through writing the high-level language code” [Fidge97, pp. 100]. The major principles of the method are that program development and verification are performed in lockstep at all levels of abstraction and the same rules are applied throughout the entire

refinement process. In outline, Quartz proceeds as follows: first, top-level specifications are created, defining the behaviours of the system via allowable traces of observable variables (histories indexed by the absolute time); then, the specifications are refined to a set of concurrent components that constitute the basis for a skeletal program design; next, each individual component is refined using sequential refinement rules, leading to the identification of low-level state changes and descriptions in an executable subset of the specification notation that correspond to constructs of the target high-level language; finally, since some time constraints may not be yet fully verified, they are subject to further analysis at the executable code level.

Conceptually, time is modelled in Quartz using an additional variable that in the refinement process receives the same treatment as the other variables of the system do. The time domain can be either discrete ($T == \mathbb{N}$) or continuous ($T == \mathbb{R}$) and the variable `now` can be introduced to model the passage of available processor time. The concepts of action systems are brought in to allow the expressing of concurrency and timing and, from the notational point of view, Z schemata are combined with guarded-command language constructs.

The goal of the authors of Quartz, that of proposing a formal development method that iteratively transforms top level RTS specifications into executable time-verified executable code is undoubtedly ambitious, but what strikes the reader of the [Fidge97] article is the complexity of the approach, a relatively simple example necessitating a rather long refinement and long explanatory descriptions. Of course, this is the general case with formal refinement and analysis, but questions can be raised regarding the applicability of the method in all but smaller-sized or highly critical applications.

4.6.3 Andy Evans' Approach

Andy Evans also shows that even though traditionally it has been considered that Z in itself is insufficient for specifying RTS, it is nevertheless possible to introduce extensions to the standard Z language that provide the capability of capturing the dynamic aspects of the

systems [Evans97]. In his approach, Evans proposes four extensions to Z addressing the issue of specifying reactive systems: genericity, generic operations being used as instruments for describing concurrent behaviour; real-time extension, allowing the specification of dynamic properties of the system, including timed computations; modularity, that permits the encapsulation of concurrent behaviours; and synchronized communication, which allows modules to communicate via a CSP-like mechanism. Notably, Evans uses only regular Z constructs, thus eliminating the need for specialised specification and analysis tools.

The key idea of Evans' approach is to specify the dynamic behaviour of the system as the set of allowable sequences of system states. States and operations are specified in the classical Z style, thus providing the static specification of the systems, while the dynamic specification is achieved using a model based on the notions of infinite computations, atomic events, and non-deterministic interleaving of atomic operations. Infinite sequences are specified using a new data type (in the following, X is a type):

$$\text{comp } X == \mathbf{N}_1 \rightarrow X \quad (4.5)$$

a next-state schema is introduced, and a generic operation `validcomp` is proposed as an extension to Z for specifying the valid behaviours (computations) of the system:

<pre> [STATE] _validcomp_ : comp STATE ↔ (P STATE × (STATE ↔ STATE)) </pre>	
<pre> ∀σ : comp STATE; I : P STATE; R : STATE ↔ STATE • σ validcomp (I, R) ⇔ σ(1) ∈ I ∧ (∀n : N₁ • σ(n) R σ(n+1) ∨ σ(n+1) = σ(n)) </pre>	(4.6)

Timed computations explicitly capturing temporal constraints imposed on the system are modeled using the notions of discrete time ($\text{Time} == \mathbf{N}$) and of infinite sequences of states with associated time values. A generic relation validcomp_t , similar to the one in (4.6), is proposed in order to specify the allowable behaviours of the system. Evans' approach makes an elegant use of generic constructs to provide Z with extensions for specifying real-time

systems. However, using infinite sequences of states to describe the system's dynamic properties brings a level of mathematical complexity that may hinder the adoption of the proposed extensions by the larger community of software developers.

4.6.4 RTOZ

Another approach that employs a variant of Z, specifically Object-Z, in a formalism aimed at specifying RTS is Periyasamy and Alagar's Real-Time Object Z (RTOZ) [Periyasamy97, Periyasamy98]. RTOZ addresses both time-dependent data and time-constrained processes, and allows for a separation between temporal constraints and functional specification. The philosophy of RTOZ is based on the notion of filter specifications (classes that specify timing constraints) and a model of time that relies on the history of data objects.

A specification in RTOZ is composed of two sets of classes: regular classes, that capture the structural and the behavioural requirements of the system without regard to temporal restrictions, and timing classes (filters) that model timing properties of the system. There is a one-to-one correspondence between regular classes and filter specifications, a filter specification describing the timing constraints imposed on the behaviour of its associated class. Each filter specification consists of several filter schemas, and each operation in a given class is restricted by a filter schema in its class' associated filter specification.

The approach described by Periyasamy and Alagar is novel in that it utilizes real-time filters in the context of object-oriented specifications, and makes a clear separation between the specification of the system's functional requirements and the description of its timing constraints. This demarcation between the timing aspects and the "time-abstracted" behaviour of the system brings a series of advantages, most notably the increased reusability of the functional specifications of the system, localisation of effects in the case changes in requirements are required (improved by the isolation of functional requirements from temporal constraints), and better understanding of both the operational characteristics and

the timing properties of systems. Also, RTOZ extends only minimally the syntax of Object-Z, thus preserving the capabilities of Object-Z without increasing its complexity.

Although RTOZ provides adequate support for the verification of properties such as safety and liveness, it can nevertheless be difficult to specify the characteristics of TCS only in a formalised way. A combination of diagrammatic and formal techniques would combine the advantages of both, essentially ease of use on the one hand and rigorous, verifiable descriptions on the other.

4.6.5 TCOZ

Another approach aimed at capturing RT requirements is presented in [Mahony98] and [Mahony00], and involves the combination of Object-Z and Timed CSP in a blended notation called Timed-Communicating Object-Z (TCOZ). The motivation of this notation is, as pointed out by its authors, to complement the expressive modelling power of Z regarding the static, single-threaded specification of systems with the capability of Timed CSP of capturing the behaviour of concurrent real-time systems. The integration of notations and techniques is actually multi-levelled; first, Object-Z extends Z with constructs suitable for object-oriented modelling, then the notion of time is added to Object-Z to obtain the enhanced notation Timed Object-Z. This enhancement is made possible by considering a global real-time clock, represented by the state attribute `now`, and by modelling environmental interactions as functions of time, included in the system state. On the other hand, CSP is extended with two primitives, `delay` and `timeout` that permit the specification of temporal aspects of sequencing and synchronisation. Finally, Timed Object-Z and Timed CSP are blended in the TCOZ notation, whose principal characteristic is to model operations as terminating CSP processes and objects as non-terminating processes.

The basic constructs of Timed CSP are sequencing, parallel composition of processes, and choice (internal and external). Sequencing has two forms, the first one describing the succession event-process behaviour as follows:

$$a @ t \rightarrow P(t) \quad (4.7)$$

where a is an event, t is the time parameter, and P the process.

The second one describes the sequential composition of processes, as in:

$$P;Q \quad (4.8)$$

where the sequential execution of processes P and Q is indicated by the operator “;”.

Parallel composition of processes is represented using the syntax:

$$P \parallel [X] \parallel Q \quad (4.9)$$

where P and Q are processes and X is a set of events enabled jointly by P and Q .

The external choice operator has the form:

$$a \rightarrow P \square b \rightarrow Q \quad (4.10)$$

and signifies that the above processes begins by enabling both a and b and then behaves (as P or Q) according to the event a or b that is actually enabled by the environment. The internal choice operator has a similar meaning, but the variation in behaviour is determined by the internal state of the process:

$$a \rightarrow P \sqcap b \rightarrow Q \quad (4.11)$$

The above operators are added to Z's set of operators and bring with them the semantics of CSP. The time-specific primitives `delay` and `timeout` are as well imported in the extended version of Object-Z.

In essence, the approach proposed by Mahony et al. makes use of the complementary semantics of the state-based behavioural model and of the event-based behavioural model and offers an excellent example of multi-integration of formal notations for software specification. However, the very combination of the two extended formalisms may raise a barrier that could prevent the wider acceptance of TCOZ in practice; the result is a rather complex notation, not easily accessible to developers who are not trained in formal methods. Also, oversized specifications may result from applying TCOZ to larger systems.

4.6.6 Other Approaches

Besides the approaches discussed above, other proposals for applying Z to TCS have been made over the years. Some of them are succinctly reviewed below.

In one of the earliest approaches, Duke and Smith suggest the integration of Z and TL for modelling TCS in a solution that allows the verification of properties such as liveness and safety [Duke89] but as indicated in [Johnson95] the application of temporal operators on both schemas and predicates can be confusing.

The work of Coombes and McDermid [Coombes93] can also be placed in the traditional line of research, that of enhancing the semantics of Z with semantics of other formalisms that are more suitable for specifying and verifying TCS. In essence, the authors consider constructs specific to a variant of TL, namely Interval Logic, employ the grid concept to allow the inclusion of multiple clocks (needed in distributed systems), and adapt to time intervals the CSP concept of trace. Although sound and thorough, Coombes and McDermid's approach seems too complex for practical application and can lead to oversized specifications.

The issue of capturing temporal properties using Z is also addressed by C.W. Johnson, this time in a less researched context, albeit very important, that of supporting user interface development in the construction of interactive safety-critical systems [Johnson95]. Johnson's proposal combines Z schemas with TL formulae, structured graphics, and generic input events. While Johnson's proposal successfully addresses a series of issues pertaining to the formal development of user interfaces (such as modelling of temporal properties that affect usability and synchronisation between the interface and the underlying application) and is supported by a prototyping system entitled Prelog there is still work needed regarding the refinement of specifications, as acknowledged by the author.

Dong and Zucconi suggest a framework for incorporating time in Z-based formal models [Dong97a] and propose the use of timed refinement and the ProCoS approach [He96] to capture the input environment and the Quartz approach to express the requirements of the core system, all within the frame of an extended version of Object-Z. This is one of the most flexible frameworks proposed to date for extending the modelling power of Z to the RT domain, since it allows the integration of a variety of time formalisms (not only the ones mentioned above) in an OO extension of Z. Nevertheless, the observations made previously regarding TCOZ can apply here as well.

In a similar line of research, involving the expression of time constraints in a Z-centered formalism, Bolognesi and Derrick introduce an ambitious concept, constraint- and object-oriented (C-O-O), in a highly innovative specification method that in essence combines object-oriented constructs (mapped to Object-Z) with constraints that define the time-ordering of operations (modelled as transition graphs) [Bolognesi98]. In our opinion this solution, although very original and interesting, is too complex and involves an adjustment of the OO paradigm that may appear too difficult to the larger community of software developers.

Also relatively recently the proposal of a Complete-Object-Oriented-Z (COOZ) has been made [Yuan98], relying on an object-oriented version of Z that integrates mechanisms and notations from Object-Z and OOZE and employing Duration Calculus (DC) [Chaochen91] for describing temporal properties of objects. Yuan et al.'s solution is one of the most complete proposals to date and its application is supported by a set of tools, entitled COOZ-Tools, that consists of an editor and viewer, a syntax and semantics checker, a refinement tool, a help system, and a project manager. Although DC is considered by the authors of COOZ more powerful than TL, it is the very complexity of DC and the particularities of its notation that can constitute an obstacle for the larger application of COOZ in practice.

4.6.7 The Z++ Alternative

As mentioned in Subsection 2.7.3, Z++ supports the modelling of TCS by incorporating a TL-based formalism. In essence, the HISTORY clause of the class specification describes the admissible sequences of execution, in the form of TL or RTL predicates. Because we rely on Z++ to achieve "time capturing," the Z++ way for dealing with time is described in more detail in Chapters 5 and 6 of the thesis. We mention here only that our time specification solution relies on Jahanian and Mok's RTL, whose constructs are incorporated in the larger frame of Z++ in the way proposed initially by Lano [Lano95]. This solution follows the general approach for extending Z to TCS modelling, that of incorporating constructs and symbols from other formalisms, and has been chosen for reasons outlined in Chapter 5 of the thesis.

4.7 Chapter Summary

In this chapter work related to our approach has been surveyed. The major directions of integrating notations in software specification have been investigated and a closer look at proposals aimed at dealing with systems characterised by complex temporal properties has

been taken. The major ways of dealing with time in software specification have been identified and several particular approaches have been analysed in greater detail. As the overall result of our survey, we found out that five reported projects come significantly close to the line of research we have pursued; they are, respectively, Jia's pragmatic approach based on AML, Noe and Hartrum's support for formal methods in Rational Rose, France et al.'s formalisation of Octopus, Headway Software's RoZeLink tool, and Kim and Carrington's integration of UML and Object-Z. The major characteristics of these approaches have been discussed and the main differences between them and our own approach have been pointed out.

5 FORMAL SPECIFICATION OF TEMPORAL CONSTRAINTS

“And then the clock collected in the tower/
Its strength and struck.”

[A. E. Housman, Eight O'Clock, *Last Poems*, 1922]

5.1 Introduction

In this chapter the formal resources employed in our approach for specifying temporal aspects of TCS are presented. Because various sorts of TCS behaviour can be described using the archetypal constraints identified some sixteen years ago by Dasarathy, we start by reviewing this author's classification [Dasarathy85], wrapping the original classes of constraints in the garments of a simple notation introduced for manipulation purposes. These classes of constraints will be used later (in Chapter 8) to illustrate our approach for capturing temporal properties of systems. Then, we emphasise the need for formality in describing timing properties of the systems and, because our Z++ formalism partially relies on Jahanian and Mok's Real-Time Logic (RTL) and its underlying event-action model [Jahanian86], we briefly survey the model and the RTL notation. Finally, we present the extensions proposed by Lano for employing RTL within the frame of Z++ [Lano95]. Throughout the chapter the concepts and notations are illustrated by short examples of daily-life extraction.

5.2 Dasarathy's Classification of Temporal Constraints

Since our modelling approach is aimed at TCS, particular attention is paid to specifying temporal restrictions placed on such systems. Dasarathy's landmark paper [Dasarathy85] on constructs for expressing timing constraints of RTS provides the reference for our way of dealing with time. The original classification introduced by Dasarathy was widely accepted by the researchers in the field because it covers in a simple yet extensive manner the various types of temporal constraints that can be imposed on systems. The basic notions on which the classification was built are those of *stimulus* (S), *response* (R), and *event* (E). The latter, as indicated by the author, can be either a stimulus received by the system from its environment or an externally observable response issued by the system.

However, in order to unify the terminology and subsequently make the transition to the event-action model underlying Jahanian and Mok's RTL [Jahanian86], we had to make some alterations to the original concepts of Dasarathy. Specifically, following Jahanian and Mok's approach and as opposed to Dasarathy's, we consider the events instantaneous and make use of the additional notion of *action* to describe an operation that has a non-zero duration (details about actions and events are given in Subsection 5.4.1). Consequently, the duration class of timing constraints identified by Dasarathy will no longer apply to events, but to actions, because in our approach events have no duration. This has however only a minor impact on the original classification of Dasarathy, since it affects only one of the nine classes of constraints ("classes of constraints" is our terminology). And, as in the original Dasarathy paper, both stimuli and responses continue to be considered events.

In the following, Dasarathy's classification of timing constraints, presented in our own notation, is briefly reviewed. The examples given by Dasarathy for the classes he proposed were from the field of telephony; in this section, we employ a microwave oven device to provide short illustrations for each class of temporal constraints. The classes of constraints are given in an informal manner, some implicit assumptions being made about the occurrences

of stimuli and responses. However, as discussed in the next section, a more rigorous specification of the constraints is necessary for developing reliable models of TCS.

Before reviewing the possible types of temporal constraints it is useful to note that, as Dasarathy points out, the timing restrictions placed on a system can be either *performance constraints*, which impose limits on the system's response time, or *behavioural constraints*, which specify restrictions on the rates of stimuli applied to the system. Both types of constraints can be described using three broad categories of timing constraints: *maximum*, *minimum*, and *durational*. A constraint of type *maximum* specifies an upper limit placed on the interval of time between two occurrences of events, a constraint of type *minimum* specifies a lower limit for the interval between two such occurrences, and a *durational constraint* indicates the amount of time required for the duration of an action. These three categories of temporal restrictions are not exclusive, in a more complex case being possible to have constraints of all three kinds placed on a particular behaviour of the system.

When possible combinations involving stimuli and responses come into consideration, the maximum and minimum categories expand in four subcategories (or classes) each. Because the duration category needs no further partitioning, a total of nine classes of temporal constraints are hence possible (the notation DC_x means "Dasarathy constraint class x", where x is a number we provide for easier referencing):

[DC1] MaxSS(S₁, S₂, t), specifies the maximum time t allowed between the occurrence of stimulus S₁ and the occurrence of the subsequent stimulus S₂.

Example: After the power level has been set, the microwave oven's start button should be pressed no later than 60 seconds, otherwise the attempt to use the heating feature of the microwave oven will be considered abandoned. This timing condition can be expressed as MaxSS(setPowerLevelCmd, startHeatingCmd, 60.0).

(Note that for documentation purposes the Cmd postfix is used in this chapter to indicate a stimulus event, as opposed to a response event, which has no specific postfix);

[DC2] $\text{MinSS}(S_1, S_2, t)$, specifies the minimum time t allowed between the occurrence of stimulus S_1 and the occurrence of the subsequent stimulus S_2 .

Example: After the current date and time has been set, the microwave oven's heating feature should not be started for at least 1 second. This can be expressed as $\text{MinSS}(\text{setDateTimeCmd}, \text{startHeatingCmd}, 1.0)$;

[DC3] $\text{MaxRS}(R, S, t)$, specifies the maximum time t allowed between the occurrence of response R and the occurrence of the subsequent stimulus S .

Example: After the countdown chronometer has been paused, the user should press the Resume button no later than 1800 seconds (otherwise the Countdown mode of operation will be considered abandoned). This timing constraint can be specified as $\text{MaxRS}(\text{chronometerPaused}, \text{resumeCountdownCmd}, 1800.0)$;

[DC4] $\text{MinRS}(R, S, t)$, specifies the minimum time t allowed between the occurrence of response R and the occurrence of the subsequent stimulus S .

Example: After the heating process has been completed, the user should wait at least one second before opening the door. This timing constraint can be expressed as $\text{MinRS}(\text{stopHeating}, \text{openDoorCmd}, 1.0)$.

[DC5] $\text{MaxSR}(S, R, t)$, specifies the maximum time t allowed between the occurrence of stimulus S and the occurrence of the subsequent response R .

Example: After the user has pressed the Pause button during a heating operation, the actual stopping of the heating process should occur no later than 0.5 seconds. This condition can be expressed as $\text{MaxSR}(\text{pauseHeatingCmd}, \text{stopHeating}, 0.5)$;

[DC6] $\text{MinSR}(S, R, t)$, specifies the minimum time t allowed between the occurrence of stimulus S and the occurrence of the subsequent response R .

Example: After the user has pressed the Start button for a heating operation, the actual heating process could start immediately. This can be expressed as $\text{MinSR}(\text{startHeatingCmd}, \text{startHeating}, 0.0)$;

[DC7] $\text{MaxRR}(R_1, R_2, t)$, specifies the maximum time t allowed between the occurrence of response R_1 and the occurrence of the subsequent response R_2 .

Example: When the heating process has been completed, the completion should be indicated by three successive beeps, the time interval separating every two consecutive beeps not exceeding 1.5 seconds. The timing condition imposed on the beeps can be specified broadly as $\text{MaxRR}(\text{endBeep}, \text{startBeep}, 1.5)$.

[DC8] $\text{MinRR}(R_1, R_2, t)$, specifies the minimum time t allowed between the occurrence of response R_1 and the occurrence of the subsequent response R_2 .

Example: When the heating process has been completed, the completion should be indicated by three successive beeps, the time interval that separates every two consecutive beeps being not less than 1.0 second. This timing condition imposed on the beeps can be specified as $\text{MinRR}(\text{endBeep}, \text{startBeep}, 1.0)$;

[DC9] $\text{Duration}(A, t_1, t_2)$, specifies the minimum time t_1 and the maximum time t_2 required for action A to last (t_1 may be 0, and t_2 may be omitted, in which case it will be interpreted as $+\infty$).

Example: The audio signals emitted by the microwave oven to indicate the completion of some operations (such as “done heating,” or “chronometer reached zero”) should be in the form of beeps whose duration, per beep, should be no less than 1.0 seconds and no more than 2.0 seconds). The last part of this requirement can be described by the expression $\text{Duration}(\text{Beep}, 1.0, 2.0)$.

Of course, in the S-S cases S_1 and S_2 may be the same type of stimulus, and in the R-R cases the responses R_1 and R_2 may be of the same nature. Another observation is that in classes [DC1] to [DC4] the timing constraints are imposed on the system’s users, and therefore it is necessary to specify the actions the system must take when these constraints are not satisfied. In such cases, Dasarathy proposes the use of an artificial stimulus, a timer to signal situations in which the user fails to apply the second stimulus within the requirements of classes [DC1] to [DC4]. If a maximum-time constraint is not satisfied the timer will signal the absence of the stimulus within the prescribed deadline and the system will be able to transition to a new state and/or issue a specific response. Similarly, if a minimum-time constraint is disobeyed (that is, the user applies the stimulus too soon) then the armed timer will not go off and this will be considered an undesirable situation, which requires specific treatment. Classes [DC5]

to [DC8] are conditions imposed on the system's performance, rather than on the user's behaviour, and therefore the use of a timer is not necessary.

5.3 On the Rigorous Specification of Temporal Constraints

Although essentially simple, Dasarathy's categories of temporal constraints are archetypal for they can be successfully used to specify of large variety of conditions involving time (more precisely, such conditions can be "reduced", or "translated," to combinations of Dasarathy constraints). However, the way the constraints have been described previously leaves room for interpretations. For instance, the constraint [DC6], defined as $\text{MinSR}(S, R, t)$ does not specify whether the occurrence of R is actually required (that is, should R always follow S , or it is possible to have instances of S without subsequent response R ?). In addition, as observed from the short examples provided in the previous section (e.g., for [DC4] and [DC8]), it is necessary to associate some temporal markings with the beginning and the end of actions and to take in consideration the actual number of occurrences of a stimulus or response. Moreover, parallel execution of actions is difficult to describe precisely without resorting to additional constructs.

For these reasons, while taking the Dasarathy constraints as a reference basis for formulating the requirements of TCS, we resort in our approach to a formal language, RTL, that unambiguously describes the temporal restrictions placed on such systems. To give only an example, in RTL the constraint [DC6] can be expressed in a more precise way, for instance as

$$\forall i \in \mathbb{N}_1 \quad @(s, i) + d \leq @(\uparrow R, i)$$

meaning that each event s (stimulus) is followed by the start of response R after at least d units of time and no response R can occur without being triggered by s (the notation of the Dasarathy constraint has been adapted for RTL). Other detailed predicates related to [DC6] are possible, for instance the response R can be allowed to occur without being triggered by s .

In fact, one of the reasons for using Z++ as counterpart of UML in our integrated modelling approach was its inclusion of RTL, a precise and easy to comprehend language for expressing time-related properties of the systems.

5.4 Real-Time Logic (RTL)

The dynamic aspects of systems are formally expressed in Z++ using statements written in an extended version of RTL. We introduce below the specification language Real-Time Logic, originally proposed by Jahanian and Mok [Jahanian86, Jahanian94] and in the next section indicate the extensions brought by Lano to the language. Because RTL is based on the event-action model, a brief presentation of the major components of the model is given first, followed by a summary overview of the notation.

5.4.1 The Event-Action Model

RTL, as described by its inventors, provides a uniform way for specifying both relative and absolute timing of events. The computational model on which RTL relies is centred around two key elements: the first is *action*, and the other is *event*. Additionally, the concepts of *state predicates* and *timing constraints* complete this model that allows the capturing of data dependencies and of temporal ordering of computations performed by the system in response to external and internal events.

An *action* is an operation that requires a bounded amount of system resources and is delimited by two events, one denoting its initiation, the other its completion (notational details are given in the next Subsection). An *event* is a temporal marker that has attached a time value, its time of occurrence, and imposes no requirements on the system's resources. Actions may be either *primitive* or *composite*. The former have atomic implementations, while the latter consist of two or more *subactions*, whose order of precedence can be specified using the sequential or parallel operators. Events can be classified in four categories:

- *External events*, stimuli received by the system from its surrounding environment, for instance the user pushes the microwave oven's Start button;
- *Start events*, marking the initiation of some action, for instance the beginning of the Heating operation;
- *Stop events*, marking the termination of some action, for instance, the end of the Heating operation;
- *Transition events*, signaling a change in the state of the system, for instance speedLimitReached, indicating the fact that a locomotive has reached the maximum allowed speed under some given conditions (e.g., 60 km/h on a bridge).

5.4.2 RTL Concepts and Notations

After the introduction of the two most important concepts of RTL, event and action, an overview of the notation is presented in the following. For practical reasons, we introduce some minor alterations to the notation. Specifically, we use combination of words for longer action names and capitalise each word in the combination, as opposed to Jahanian and Mok's original uppercase only convention. Also, when denoting events we use lowercase single-word identifiers or multiple-word identifiers with all the words of the combination except the first capitalised.

- **Actions**
 - are denoted by capital letters such as A, B, etc., capitalised words or combination of capitalised words such as Heating, MoveToNextFloor, or abbreviations such as TCD ("Timer Counting Down");
 - A.B denotes the subaction B of composite action A;
 - A.B_i signifies the i-th appearance of subaction B within composite action A;
 - B||C means that subactions B and C execute in parallel;
 - B;C signifies that subactions B and C execute in sequence, B followed by C;
 - !N indicates a synchronisation point N, and A!N together with !NB specifies that action A should be completed before action B starts its execution;

- **Events**

- external events are denoted using the convention for event identifiers described above and are prefixed by the symbol Ω . For instance, $\Omega_{\text{pushStartButton}}$ is the event corresponding to the user pressing the button Start;
- start events are indicated by the symbol \uparrow , for instance $\uparrow A$ represents the event associated to the start of action A;
- stop events are indicated by the symbol \downarrow , for instance $\downarrow A$ represents the event associated to the completion of action A;
- transition events indicate a modification in one of the system's state variables. The notation $(S := \text{true})$ denotes the event corresponding to the transition that makes the state variable S true and $(S := \text{false})$ denotes the transition event that makes the state variable S false.

- The **occurrence function**, denoted $@$, is introduced to capture the notion of real time:

$@(E,i)$ = time of the i -th occurrence of the event E , where $i \in \mathbb{N}_1$

Note that within $Z++$ the alternative symbol \clubsuit is used, as described in Subsection 5.5.3. Therefore, $\clubsuit(E,i)$ is employed in the following chapters of the thesis.

- **State predicates**, assertions about the state of the system. The value of a state predicate can change over time, as a result of external events and/or system responses. Depending on the boundary conditions, nine forms of state predicates are possible, from $s_{\langle t_1, t_2 \rangle}$, through $s_{(t_1, t_2)}$, to $s_{[t_1, t_2]}$. Informally " $\langle t$ " means before time t , " $(t$ " means "before or at t ", " $[t$ " signifies "at time t ", etc. For instance, $s_{\langle t_1, t_2 \rangle}$ specifies that the state predicate s is true before time t_1 and remains so until exactly at time t_2 . An example of state predicate is $\text{DoorsClosed} \langle \uparrow \text{Heating}, \downarrow \text{Heating} \rangle$.
- **RTL predicates** are formed using arithmetical relations ($=, \neq, <, \leq, >, \geq$) and algebraic expressions containing integer constants, variables, addition, subtraction, multiplication by constants, and the occurrence function.

- **RTL formulae** can be constructed using universal and existential quantifiers, equality and inequality predicates, and first order logical connectives. An example of an RTL formula is:

$$\forall i, @(\Omega\text{MailReceived}, i) < \uparrow(\text{DispatchMail}, i) \wedge \downarrow(\text{DispatchMail}, i) < @(\Omega\text{MailReceived}, i) + 60$$

The above can be interpreted as “action DispatchMail must be executed after the event MailReceived each time the event occurs and must be completed within 60 time units of the occurrence of the MailReceived event.”

- **Timing constraints** complete RTL's underlying model by providing assertional statements about the absolute timing of events that characterise the system's behaviour. Four types of constraints are considered of particular importance in RTL:

- *sequential constraints*, constraints on the sequential execution of actions. For instance to indicate that subaction B always precedes subaction C in the composite action A one can write $\forall i @(\downarrow A.B, i) \leq @(\uparrow A.C, i)$;
- *parallel constraints*, constraints on the parallel execution of actions. For instance to indicate that subaction B precedes the parallel execution of C and D within composite action A one can write $\forall i @(\downarrow A.B, i) \leq @(\uparrow A.C, i) \wedge @(\downarrow A.B, i) \leq @(\uparrow A.D, i)$;
- *sporadic timing constraints*, given as a requirement for action A to complete its execution within a deadline d after the occurrence of the event E, event for which a separation p between occurrences is required;
- *periodical timing constraints*, in the form “while S is true execute A with period p and deadline d” where S is a state variable and A an action (the longer RTL formulae for the last two categories of constraints can be found in [Jahanian86]).

5.5 Using RTL in Z++

The key idea of Lano's approach for expressing temporal properties of systems is to include in the HISTORY clause of Z++ classes an extended RTL predicate that defines the behaviour

of the objects of the class. This behaviour is seen as a continuous and infinite series of states segmented by occurrences of events, and the RTL predicate holds at all times.

In Z++, the domain TIME of time-valued terms is totally ordered, meaning that in additions to satisfying the axioms for partial order, the following properties also hold: (a) there is a designated element 0 , such that $0 \leq t$, for each element $t \in \text{TIME}$; and (b) for every pair of elements $(t_1, t_2) \in \text{TIME} \times \text{TIME}$, $(t_1 < t_2) \vee (t_1 = t_2) \vee (t_1 > t_2)$. The time domain satisfies the axioms of a set of non-negative elements of a totally ordered topological ring, with operation $+$ and $*$, and units 0 and, respectively, 1 . It can be considered that $\mathbb{N} \subseteq \text{TIME}$.

The following are summarised from [Lano95], only the concepts and notations needed later in the thesis being presented. Compared with Lano's description, we use the term operation instead of method, this choice being maintained throughout the entire thesis.

5.5.1 Lano's Key Extensions to RTL

The key concepts of Lano's extension of RTL are:

- *invocation instance*, which comprises the initiation, the execution, and the termination of operation op ;
- *request event*, in the form $\rightarrow op$, denoting the arrival at the current object of a request for the execution of operation op ;
- the *temporal operators* \Box^t "at all future times", \Box "at some future time", and \odot "holds at";
- *counters for operation events* $\#req(op)$, $\#act(op)$, and $\#fin(op)$, as defined in 5.5.3.

5.5.2 Events

Each operation op of class C has associated the following events:

- $\uparrow_{op}(x)$, the initialisation of an invocation instance of $op(x)$, $x \in X$, where X is the set of the operation's possible inputs;

- $\downarrow_{op(x)}$, the termination of the operation's invocation instance;
- $\rightarrow_{op(x)}$, the arrival at the object of a request for the invocation of the operation;

Other events are events of the form $\varphi := \text{true}$ or $\varphi := \text{false}$, where φ is a predicate without modal operators or occurrences of *now*, denoting that the events of this predicate are *true* (or, respectively, *false*), and events for a supplier object *s* of class *S*, in the form $\uparrow_{(ops(x),s)}$, $\downarrow_{(ops(x),s)}$, and $\rightarrow_{(ops(x),s)}$. In addition, $\leftarrow_{(ops(x),s)}$ signifies the sending from the current object of a request for *s* to execute the operation *ops* with input *x*.

5.5.3 Terms

The following terms can appear in a class *C*'s associated RTL formulae:

- variables v_i , $i \in \mathbb{N}$;
- attributes of the class, its ancestors, and supertypes;
- *n*-ary functions in the form $f(e_1, e_2, \dots, e_n)$;
- $\#e$, denoting the time at which *e* occurs, where *e* is an event occurrence (E, i) , $i \in \mathbb{N}_1$;
- $\rightarrow_{(op(x), i)}$, $\uparrow_{(op(x), i)}$, and $\downarrow_{(op(x), i)}$, where *op* is an operation of class *C*, *x* its input, and $i \in \mathbb{N}_1$;
- event occurrences in the form $\leftarrow_{(ops(x), s), i}$, where *s* is an object of *C*'s supplier class *S* and *ops* an operation of *S*;
- *self*;
- *now*;
- $e \oplus t$, which indicates the value of *e* at time *t*, where *e* is a term and *t* a time-valued term;
- O_e , which denotes the value of term *e* at the next operation initiation time;
- $\#act(op)$, the number of initiations of *op*'s execution, up to the present time;

- $\#req(op)$, the number of requests for op 's execution, received by the object up to the present time;
- $\#fin(op)$, the number of terminations of op 's execution, up to the present time.

5.5.4 Formulae

Considering a class C , the following are RTL formulae related to it:

- $P(e_1, \dots, e_n)$ for an n -ary predicate symbol P and terms e_1, e_2, \dots, e_n ;
- $\varphi \wedge \psi$, $\varphi \vee \psi$, $\varphi \Rightarrow \psi$, and $\neg \varphi$, for formulae φ and ψ ;
- $\varphi @ \tau$, which indicates that φ holds at time τ , where φ is a formula and τ a time-valued term;
- $\forall D \bullet \varphi$ and $\exists D \bullet \varphi$, for declarations D and formulae φ ;
- $\Box^t \varphi$, which denotes that φ holds at all future times (not related to C); $\Box \varphi$ which means that φ holds at each initiation time of an operation from the class; and $\bigcirc \varphi$, which is the value of φ at the next operation initiation time;
- $\Diamond^t \varphi$, which means that eventually φ will hold in the future, and $\Diamond \varphi$, which indicates that φ will eventually hold at the initiation time of an operation from the class;
- $enabled(op)$ and $enabled(op(x))$, where op is a method of class C and x an expression in the input type of op , indicating the condition that must hold at the operation's initiation.

5.5.5 Abbreviations

Lano also introduces a number of abbreviations, including:

- $\#active(op)$, for the number of execution instances of op that are currently executing; it abbreviates $\#act(op) - \#fin(op)$;
- $delay(op, i) = \hat{\rightarrow}(op, i) - \rightarrow(op, i)$, the delay between the i -th request for the execution of operation op and the actual i -th initialisation of the operation;

- $\text{duration}(\text{op}, i) = \downarrow_{\text{op}}(i) - \uparrow_{\text{op}}(i)$, the duration of operation op 's i -th execution;
- $\text{mutex}(\{\text{op}_1, \dots, \text{op}_n\})$, meaning that at any given moment a method op_k of the set $\{\text{op}_1, \dots, \text{op}_n\}$ has a number of active instances that is equal to the total number of active instances of all the operations in the set;
- $\text{self_mutex}(\{\text{op}_1, \dots, \text{op}_n\})$, meaning that each operation op_k in the set $\{\text{op}_1, \dots, \text{op}_n\}$ has at most one active instance at any given moment;
- $\underline{\text{op}}$, which is an abbreviation for $\#\text{active}(\text{op}) > 0$;

5.5.6 Axioms

A comprehensive set of axioms is included in Lano's book. For illustration purposes two are given below, but for full details we refer the reader to Appendix A of [Lano95]:

(a) At any given time, there cannot be more terminations of $\text{op}(x)$ than activations:

$$\forall i \in \mathbb{N}_1 \bullet \clubsuit(\uparrow_{\text{op}(x)}, i) \leq \clubsuit(\downarrow_{\text{op}(x)}, i)$$

(b) Event occurrences are indexed ordered on their time of occurrence:

$$\forall i, j \in \mathbb{N}_1 \bullet i \leq j \Rightarrow \clubsuit(E, i) \leq \clubsuit(E, j)$$

5.6 Chapter Summary

In this chapter the formal basis for expressing temporal properties of the systems has been presented. Since requirements on the behaviour of TCS can be described using the classes of constraints proposed by Dasarathy, a review and respecification of these classes using a simple notation and small examples related to a microwave oven application have been presented. Also, since, in our modelling technique, the capturing of timing constraints is performed using extended RTL formulae, an overview of the notational elements of RTL as well as a brief description of its underlying action-event model has been provided. Lano's extensions of RTL have also been presented. As a result, the preparation for the formalisation of UML models, including timing restrictions on the behaviour of systems, has been completed. The following chapter provides additional details on the specific use of RTL in our approach.

6 TRANSLATIONS BETWEEN UML AND Z++: FORMALISATION AND DEFORMALISATION

"Poetry is what gets lost in translation."

[attributed to Robert Frost (1874-1963)]

6.1 Introduction

This chapter presents the translation processes between UML models and their corresponding Z++ specifications. Emphasis is placed on the UML to Z++ translation, whose purpose is to increase the rigor of the system's description, but in order to make formal specifications easier to understand during the integrated modelling of the system the reverse translation, from Z++ to UML, is also considered. The first type of translation, alternatively referred to as formalisation, applies both to UML class diagrams, which capture structural aspects of the system, and to UML state diagrams, which describe the system's dynamics. The second translation, alternatively denoted deformalisation, produces UML classes from the information contained in Z++ specifications and thus can be considered "structure-oriented". The focus is on those parts of formalisation and deformalisation that can be performed automatically, a detailed set of translation principles and a translation algorithm based on these principles being presented for each process. The formalisation and deformalisation processes described in this chapter are included in the larger modelling frame of TCS that constitutes the subject of Chapter 7 and their application is illustrated in the Elevator Controller case study presented in Chapter 8.

6.2 Preliminary Remarks

The modelling approach described in this thesis relies on the combined use of UML and Z++. In Chapter 7 details are given on the complete UML/Z++ integrated modelling process proposed in the thesis, a process that consists of a number of activities such as definition of use cases, construction of UML class diagrams, and elaboration of Z++ specifications. In the present chapter the focus is on two key parts of this process, the formalisation and deformalisation activities. Before describing these two activities, which essentially consist of translations between UML models and Z++ specifications, some general observations are necessary.

First, a couple of remarks on terminology. Specifically, in the larger frame of the modelling approach described in Chapter 7 formalisation and deformalisation are denoted *activities* (or *subprocesses*), yet for simplicity in the present chapter we refer to them as *processes* (another possible generic term for formalisation and deformalisation, *procedure*, was avoided because it appears extensively in the pseudocode description of the algorithms presented later in this chapter). Also, the term *translation* (from UML to Z++, or from Z++ to UML), used as a substitute for formalisation and, respectively, deformalisation, should be seen as “selective translation” since in both cases only a partial mapping from one modelling space to the other is performed (in the case of deformalisation the term “truncated translation” would be even more accurate since significant informational content is possibly discarded when generating UML constructs from Z++ specifications).

In what regards formalisation, its main role in the approach presented in this thesis is to help both developers and their clients gain a better understanding of the system under construction by increasing the rigour of the system’s description. With an accurate insight into the system’s desired structure and behaviour those involved in the early stages of the system’s development will be able to avoid a significant number of potentially very costly specification errors. Also, since the formalisation process makes precise and amenable to formal reasoning and formal refinement the initially written in UML description of the

system, it opens the door for subsequent formal processing, but aspects regarding formal analysis of specifications and formal refinement of specifications to code are not dealt with in the present thesis.

Guidelines for formalising object-oriented semi-formal models have been proposed by Lano and Haughton in [Lano94c] and by Lano in [Lano95]. They represent the starting point for the semi-formal to formal translation process presented in this chapter but it should be pointed out that Lano and Haughton's work was concerned with the formalisation in Z++ of OMT models, so we have adapted and extended their approach to UML models. Also, in the present approach we have attempted to provide a systematic description of the formalisation, through detailed sets of principles and detailed algorithms, and have additionally tackled the reverse translation from formal specifications to graphical representations, translation that was not considered by Lano and Haughton.

As in the case of Lano and Haughton's work, the approach proposed in this thesis addresses the formalisation of both structural and behavioural aspects of the system. For the latter, the same RTL formalism proposed by Jahanian and Mok is employed but differences exist between the two approaches regarding the details of this employment, as shown in Section 6.4. In practical terms, the formalisation of UML constructs in Z++ consists of two components, formalisation of class diagrams (described in Section 6.3, and concerned primarily with structural aspects of the system), and formalisation of state diagrams (presented in Section 6.4 and dealing with behavioural characteristics of the system). The formalisation of UML models applies only to the core elements of the language (class diagrams, classes, relationships, and state diagrams) but, as shown in studies published by authors who have worked on similar formalisation approaches, these constructs provide good insights into the system and allow formal reasoning about its properties [Lano95, France99, Kim99a].

Additional reference for the formalisation processes described in this chapter has been provided by the work of Kim and Carrington on formalising UML models in Object-Z

[Kim99a, Kim00a, Kim00b]. In particular, their formal Z description of UML class diagram constructs, preliminary to the translation procedure from UML to Object-Z, has served us to better define and organise the rules for well-formed UML class diagrams presented in Subsection 6.3.1.

In what regards the reverse translation, from Z++ specifications to UML constructs, it should be noted that it has a secondary role in the modelling process, its purpose being to make easier the interpretation of the integrated model by developers and users not trained in formal methods. This feature may or may not be used within a particular modelling context, but its inclusion in the proposed approach allows a form of “reverse engineering,” from formal specifications to semi-formal graphical descriptions. In practice, it is thus possible to have some Z++ specifications developed first and then their class structure propagated into the UML space. This allows an improved communication between developers skilled in formal methods and developers and users that favour the graphical representation of the system. The deformatisation option is not a common feature in integrated approaches and its practical utility is smaller than that of formalisation. In fact, the only other approach that deals with the reverse propagation of models is Headway System’s RoZeLink [RoZeLink99], from which we have borrowed the idea. Nevertheless, the reverse translation suggested in Section 6.5 is significantly distinct from that used in RoZeLink, major differences stemming both from the quite dissimilar OO variants of Z used (ZEST in the case of RoZeLink, and Z++ in our case) and from the particular way the Formaliser structured editor used in conjunction with RoZeLink continually enforces the correct syntax of ZEST specifications [Formaliser01].

Since both formalisation and deformatisation processes can be partially automated we focus in this chapter on those translation operations that can be implemented by a computer program. For each process a set of translation principles is presented first and then, based on these principles, an algorithm that allows the automatic execution of parts of the translation is proposed. A number of issues pertaining to the practical utilisation of the formalisation

and deformalisation algorithms, in particular regarding their combined application, are discussed in Section 6.6.

6.3. Formalisation of UML Class Diagrams in Z+

The first part of formalisation addresses the translation of UML structural constructs to Z++. This formalisation applies to UML class diagrams and to the elements they contain (classes and relationships), the result being a set of corresponding Z++ classes. For the target language of the translation, Z++, it is useful to consider again the general form of a Z++ class, introduced in Chapter 2 and presented in more detail in Appendix A, and to notice that a supplementary clause, `PUBLICS`, has been included in the definition of Z++ classes. This clause allows better specification of member visibility, in the same way the \uparrow list of Object-Z classes declares the attributes and operations that are externally accessible through the dot notation [Duke94]. (The introduction of this clause is in agreement with the declared intention of Z++'s authors, who designed the language's syntax "to enable simpler extension of the notation by the addition of new clauses to a class definition" [Lano94d, pp. 138]). During the automatic translation the clauses of Z++ classes are partially filled in according to the information contained in UML class diagrams and then the formal specifications can be enhanced by developers with details of data structures, definition of operations, and more elaborate constraints. In this section, the input considered for the formalisation process is a single class diagram, a discussion regarding the application of the process to a set of class diagrams, as well as to a class or a group of selected classes being presented in Section 6.6.

6.3.1 Rules for Developing Well-Formed Class Diagrams

In order to reliably perform the translation of UML structural constructs into Z++ specifications a number of constraints on the syntactic structures of UML class diagrams must be enforced. These constraints ensure that the UML constructs are syntactically well-

formed and thus can be subjected to automatic translation to Z++. Many of them represent restrictions on the development of UML models that are due to the specifics of the target language of the translation, Z++ (they can be described as “compatibility constraints” between UML and Z++), for instance interfaces and abstract classes are not treated since there are no equivalent constructs for them in Z++ and, if parameters of operations are provided in UML, both the names and the types of parameters must be specified in order to allow the automatic formalisation of operation signatures. Other restrictions represent simplifications of UML in cases in which it has been considered that the burden on the formalisation process would not be compensated in practice by the inclusion of less frequently used features (e.g., only binary relationships are considered).

These constraints, given below in the form of rules for developing well-formed class diagrams, raise indeed the level of rigour required in the UML space and reduce to a certain degree the modelling options of the UML developer. However, this reduction in modelling flexibility is well compensated by the benefits of the more precise descriptions made possible by formalisation. Also, while rather large and detailed, the set of constraints described below is however not exhaustive, its purpose being to avoid the more common modelling errors that would prevent reliable automatic formalisation of class diagrams. In addition, minor constraints such as restrictions on the number of characters used in the names of UML constructs have been omitted for simplicity.

The rules for well-formedness presented in this section have been inspired primarily from [Kim99a], with additional observations drawn from [Lano95]. Many rules have been added (e.g., rules regarding attributes and operations, rules for generic classes) while some have been discarded (association classes are not considered). All rules are commented and organised in a manner intended to facilitate the subsequent description of the translation principles presented in Subsection 6.3.2 and of the formalisation algorithm AFCD (Algorithm for Formalising Class Diagrams) described in Section 6.3.3.

6.3.1.1 Rules for Class Diagrams

The following must be satisfied by each class diagram that is subjected to formalisation:

- The class diagram consists only of classes and binary relationships between classes; (6.1)
- There is a finite number of classes and a finite number of relationships in the class diagram; (6.2)
- Each relationship that belongs to the given class diagram involves two classes that also belong to the given class diagram; (6.3)

The first rule indicates that for formalisation purposes only classes and binary relationships between classes are considered, other structural elements of UML that in general can be included in class diagrams, such as interfaces and multiple relationships, being ignored (these are restrictions generally imposed in other similar formalisation approaches, e.g., [Brue196], [France99], [Kim99a]). However, in practice, some of the UML constructs that are not subjected to formalisation can still be present in the UML model, but in this case means to extract a representation of the class diagram suitable to formalisation should be devised. In addition, as indicated by rule (6.4) below, the classes can be of three kinds: regular, parameterised, and binding (classes that instantiate parameterised classes [Booch98]). The AFCD algorithm described in Subsection 6.3.3 assumes that rule (6.1) is satisfied, the class diagram that represents the input to AFCD being given as two sets, one of classes, and the other of binary relationships.

Rule (6.2) imposes limitations on the cardinality of the set of classes and, respectively, of the set of relationships that make up a diagram. Included here for the sake of completeness, it can serve for a formal description (e.g., in Z or Z++) of the formalisation algorithm.

Rule (6.3) makes sure that the input provided to AFCD is valid in the sense that no extraneous classes are involved in a relationship that belongs to the input class diagram. In

practice, this rule has an impact on the way two or more class diagrams can be related for translation purposes, as discussed in more detail in Section 6.6.

Some other rules presented later in Subsection 6.3.1 can also be seen as applied to class diagrams, for instance rule (6.37) that prevents more than one generalisation relationship between any two classes, but for presentation reasons they have been described as “rules for relationships,” after the description of the rules for classes and the introduction of the kinds of relationships considered for formalisation.

6.3.1.2 Rules for Classes

The following constraints apply to UML classes contained in the class diagram that provides the input of the formalisation process:

- Each class is either a regular class, a parameterised class, or a binding class; (6.4)
- Each class has a name, a finite number of attributes and a finite number of operations; (6.5)
- In addition to name, attributes, and operations, each parameterised class and each binding class has a finite number of class parameters (in the following, the parameters of parameterised classes are denoted formal class parameters while the parameters of binding classes are denoted actual class parameters). Regular classes do not have class parameters; (6.6)
- The name of each regular class is unique within the class diagram; (6.7)
- The name of each parameterised class is the same as the name of its binding classes but is distinct from the names of all other classes that belong to the class diagram; (6.8)
- The name of each binding class is the same as the name of the parameterised class it binds and the name of other binding classes that instantiate this parameterised class, but is distinct from the names of all other classes that

- belong to the class diagram; (6.9)
- Each parametrised class and each binding class has at least one class parameter; (6.10)
- Each formal class parameter and each actual class parameter is given only as a name; (6.11)
- Each instantiating class has the same number of parameters as the parameterised class it binds; (6.12)
- Each attribute has a name and, optionally, a type, a visibility, an initial value, and a property; (6.13)
- The name of each attribute of a class is distinct from the names of all attributes and operations that belong to the same class; (6.14)
- The visibility of an attribute is one of the following: public, protected, or private; (6.15)
- The property of an attribute is either changeable or frozen; (6.16)
- Each operation has a name and, optionally, a visibility, a finite list of parameters, a return type, and a property; (6.17)
- The name of each operation of a class is distinct from the names of all operations and attributes that belong to the same class; (6.18)
- The visibility of an operation is one of the following: public, protected, or private; (6.19)
- The property of an operation is either none or query; (6.20)
- Each parameter of an operation has a name, a type, and, optionally, a direction; (6.21)
- The parameters of an operation have unique names within the operation's list of parameters; (6.22)
- The direction of each operation parameter is one of the following: in, out, or inout; (6.23)
- The type of each attribute, class parameter, operation parameter, and the return type of each operation is either a basic type, a class type, or

an array type; (6.24)

- Each formal class parameter denotes a basic type or a class type that is not the type defined by a parameterised or binding class; (6.25)
- The name of the each formal parameter is different from all the names of types used in the class diagram outside the parameterised class to which the formal parameter belongs ; (6.26)
- The name of an actual class parameter is the name of a basic type or of a class type that is not the type defined by a parameterised or binding class. (6.27)

In the above, rule (6.4) specifies the types of classes that are subjected to formalisation. In essence, only the regular UML classes and the UML parameterised classes together with their binding classes are translated to Z++, which also allows parameterisation of classes (the parameterised classes are also referred to as template classes, or as generic classes, while the binding classes are alternatively denoted instantiating classes).

The structure of classes that is considered by the formalisation process is specified in rules (6.5) and (6.6), the former giving the regular class structure while the latter appending the requirement for class parameters in the case of parameterised and binding classes. As in the case of rule (6.2), the requirements for a finite number of items in rules (6.5), (6.6), and (6.17) are included for the sake of completeness. Evidently, the formalisation algorithm will work on a finite input.

Rules (6.7) to (6.9) provide constraints on the naming of classes. In general, within a class diagram the names of classes must be unique, but exceptions to this principle are necessary to accommodate binding of template classes such as `Queue[X]`, which can be instantiated as `Queue[Task]`, `Queue[Patient]`, etc. (this is denoted implicit binding). In UML there is a second way of instantiating parameterised classes, explicit binding, with the name of the binding class different from the name of the template class, but for simplification purposes the formalisation algorithm assumes only implicit binding is used in class diagrams. In practical

terms, to ensure efficient checking of class names, the AFCD will consider as names of generic and binding classes the string formed by concatenating the name of the class with the list of the class' parameters. As such, it is easier to automatically detect that, for instance, the class `Queue[Task]` is distinct from the class `Queue[Patient]`. Also, this internal representation is needed for the specification of relationship ends, as indicated in Subsection 6.3.1.3.

Rules (6.10) to (6.12) deal further with the well-formedness of template and instantiating classes. Obviously, the absence of parameters would contradict the concept of parameterised classes, hence rule (6.10), and the matching between formal class parameters and actual class parameters must also be enforced, as stated by rule (6.12). Rule (6.11) limits the format of class parameters to a single name, whose use is further restricted by rules (6.25) and (6.26).

Rules (6.13) to (6.16) are concerned with the well-formedness of attributes. Although the visibility and the property of an attribute are listed as optional in rule (6.14), the AFCD will assign default values for these two components if none is provided (public for visibility and changeable for property). Also, even though Z++ requires types for all the attributes, we decided to allow the AFCD to translate attributes without their types specified in UML, leaving to the developer the task of specifying in Z++, post translation, the missing types of attributes. Rule (6.14) requires unique names for attributes in a given class. Notably, the names of attributes must also be distinct from the names of operations, including inherited operations, a constraint that stems from the specifics of Z++ and from the addition of the `PUBLICS` clause, which lists attributes and operations without their type. Rule (6.15) specifies the possible kinds of attribute visibility and rule (6.16) gives details about allowable values for attribute property. The inclusion of rule (6.16) serves the formalisation process since the frozen (constant) attributes are included in Z++ in the clause `FUNCTIONS` while the changeable attributes are specified in the `OWNS` clause.

Regarding the visibility of attributes and operations addressed by rules (6.15) and, respectively, (6.19), public attributes and operations will be made visible in Z++ by their inclusion in the `PUBLICS` clause, while private attributes and operations will require the use

of an intermediary class and of a hiding operation applied to this class, as detailed in the formalisation algorithm. Following from the specifics of Z++ and from the introduction of the `PUBLICS` component in the definition of Z++ class, protected attributes and operations will not require any special treatment.

Rules (6.17) to (6.23) address syntactic aspects of operations. Regarding the uniqueness of an operation name in a class required by rule (6.18), considerations similar to those for rule (6.14) apply. Rule (6.20) has been included to support the translation process to since query operations, which do not change the state of the object, are listed separately (in the `RETURNS` clause) from the regular operations indicated by the `none` property (these operations are listed in the `ACTIONS` clause of the Z++ class). Rules for the parameters of operations are also necessary to help the automatic translation to Z++. In particular, both the name and the type of a parameter are required (6.21), since both are necessary in Z++ for declaring operations and an automatic assignment of parameter names by the AFCDD would complicate unnecessarily the translation. Also, unique names for the parameters of an operation are required in Z++ even though they may have distinct types, hence rule (6.22), and the provisions of rule (6.23) are used in specifying the signatures of operations in Z++. If unspecified, the direction of a parameter will be considered in.

Rule (6.24) indicates that three kinds of types are possible for attributes, parameters of template classes, parameters of operations, and the returns of operations. Class types are all the types whose name is identical with one of the names of classes that exist in the class diagram. For practical purposes, the formalisation algorithms will accept names of types given either as `T`, `T[]`, or `T[params]`, where `params` is a set of class parameters (more details are given in Subsection 6.3.2.1).

Rules (6.25) to (6.27) further restrict the use of class parameter names in order to avoid possible complications when formalising generic classes.

6.3.1.3 Rules for Relationships

The following rules apply to relationships between classes included in the class diagram:

- Each relationship between two classes is either an association, an aggregation, a composition, a generalisation, or an instantiation; (6.28)
- Each association relationship has a name; (6.29)
- Each relationship has two relationship ends; (6.30)
- Each end of a relationship is attached to a class; (6.31)
- Each end of a relationship has one of the following types, depending on the kind of relationship to which it belongs:
 - (a) *assoc* in the case of association;
 - (b) *aggreg*, if the end is attached to the “whole” class of the aggregation, and *none* if the end is attached to the “part” class;
 - (c) *comp*, if the end is attached to the “whole” class of composition, and *none* if it attached to the “part” class;
 - (d) *super*, if the end is attached to the superclass of a generalisation, and *none* if it is attached to the subclass;
 - (e) *generic*, if the end is attached to the parameterised (generic) class of an relationship, and *none* if it is attached to the binding class; (6.32)
- Each end of a relationship has a multiplicity constraint attached, which is expressed in the form of a finite sequence of ranges

$$a_1 .. b_1, a_2 .. b_2, \dots, a_K .. b_K$$
 where:

$$K > 0,$$

$$\forall i, 1 \leq i \leq K, a_i \geq 0, b_i > 0, a_i \leq b_i,$$

$$\forall i, 1 \leq i \leq K-1, b_i < a_{i+1},$$
 and b_K only may be $+\infty$ (denoted *) (6.33)
- The multiplicity of the relationship end that is attached to the “whole” part of a composition relationship is 1; (6.34)

- Both ends of a generalisation have multiplicity 1; (6.35)
- Both ends of an instantiation have multiplicity 1; (6.36)
- Between any two given classes, if more than one relationship exist,
the relationships are all either associations or aggregations/compositions; (6.37)
- The names of the associations that involve the same two classes are distinct; (6.38)
- Each generalisation involves two distinct classes; (6.39)
- Each instantiation is between a parameterised class and an instantiating class; (6.40)
- A class cannot be the superclass of any of its ancestors; (6.41)

Rule (6.28) specifies the kinds of relationships considered in the present approach. Compared with the types of UML relationships described in Section 3.3, the dependency and realisation relationships are not included (with the exception of the instantiation version of dependency). Also, it should be noted that the term instantiation relationship is not in the UML vocabulary, but we use it here to describe in a shorter way the dependency relationship between a parameterised class and a binding class.

The names of association relationships are needed for formalising purposes, hence rule (6.29).

Rules (6.30) and (6.31) enforce non-tangling relationships by requiring that each relationship be specified in terms of two relationship ends, each end being attached to a class.

Rule (6.32) specifies constraints on relationship ends for properly formed relationships. It avoids incorrect situations such as a relationship with both ends of type aggregation.

Rule (6.33) gives a general form for the multiplicity constraint attached to a relationship end. This form encompasses all cases normally used in UML, including the multiplicity 1, which can be represented as 1 .. 1, and the notation *, which can be represented 0 .. *.

Rule (6.34) makes sure that unshared containment, characteristic to composition, is properly specified in terms of multiplicity while rules (6.35) and (6.36) do the same for the instantiation of parameterised classes and, respectively, for generalisation.

Rule (6.37) gives the conditions under which multiple relationships between two classes are allowed, while rule (6.38) makes sure that duplicate associations can be mechanically formalised.

Rule (6.39) prevents a class to be its own superclass, while rule (6.40) defines more precisely the instantiation relationship;

Finally, rule (6.41) avoids invalid situations in which a class acts as superclass to one or more of its ancestors. Technically, rule (6.41) incorporates rule (6.39), but the latter was included for increased clarity. The AFCD will detect the existence of cycles in the graph whose nodes are the classes and whose links are the generalisation relationships contained in the input class diagram.

The set of rules for relationships described above need be completed with rules regarding the involvement of generic and binding classes in other types of relationships than instantiation. To keep things simple, the algorithm for automatic translation will assume that invalid situations such as a generic class at the “part” end of an aggregation whose “whole” end is attached to a regular class are resolved by the developer before the algorithm is applied.

6.3.2 Translation Principles for Class Diagrams

The automated translation of UML class diagrams to Z++ specifications follows a number of principles, as described below.

6.3.2.1 Translation of Types

In order to facilitate the mechanisation of the formalisation process restrictions are placed on the use of types, as indicated in rule (6.24). In the UML space the considered types of attributes, parameters of operations, and returns of operations (henceforth collectively denoted UML types) can be expressed in one of the following forms:

- (a) In “scalar form” T , where T is a string identifier denoting either a basic type or a regular class type (the latter means that a regular class with name T exists in the class diagram);
- (b) In “array form” $T[]$, where T is the name of a basic type or a regular class type (note the empty space within the square brackets, meaning that only one dimensional arrays are automatically processed and the information on array bounds, if any, is left to be formalised manually by the developer);
- (c) In “generic form” $T[\text{params}]$, where params is a list of parameters passed to a template class, each parameter in params denoting a basic type or a regular class type (array types and types in generic form are not allowed within params , as indicated by rule (6.27)).

With these restrictions, the mapping of types from UML to Z++ proceeds along the following lines:

- When the UML type is expressed in scalar form T , then:
 - if T is the name of a recognised basic type, specifically unsigned integer, integer or real then the corresponding Z++ type will be, respectively, N , Z , or R (variants such as byte, int, long, double, and float will also be treated as recognised basic types within the above three categories). Constraints on the range of the type, if needed, will be specified by the human formaliser. The Boolean type will be recognised for the returns of operations, but no explicit output variable and no output domain will be associated in Z++ to the operation’s return. Also, the type void of an operation’s return will be recognised and treated as a type that requires no specification of output domain in the

operation's signature and no specification of output parameter in the operation's definition;

- if T is the name of an existing regular UML class then the $Z++$ type will also be T ;
 - if $\text{uppercase}(T)$ is the name of an existing given set in $Z++$, then the $Z++$ type will be $\text{uppercase}(T)$ (by $\text{uppercase}(x)$ we denote the string obtained from the identifier x by promoting to uppercase all its lowercase letters, while keeping the others unchanged);
 - if T is neither the name of a recognised basic type, nor the name of an existing regular UML class or of an existing $Z++$ given set, it will be treated as the name of a unrecognised basic type and a new given set will be added in $Z++$, with the letters of the identifier T written in uppercase, as it is customary in Z . The $Z++$ type will therefore be $\text{uppercase}(T)$;
 - if T is used in the context of a parameterised class and it is identical with the name of a formal parameter of the class, the $Z++$ type will also be T ;
- When the UML type is an array type $T[]$, then T will be first checked as described above and then the operator seq will be applied to the $Z++$ type corresponding to the scalar type T . For instance, the UML type $\text{int}[]$ will become $\text{seq}(Z)$ in $Z++$ and the UML type $\text{Car}[]$ will be mapped either to $\text{seq}(\text{CAR})$, if no class with the name Car exists in the class diagram, or to $\text{seq}(\text{Car})$, if Car is the name of an existing UML class;
 - When the UML type is given in generic form $T[\text{params}]$ it will be assumed that the items of the params list represent actual parameters for the generic class T . If these parameters are provided by the formal parameters of the enclosing class, they will be left unchanged, otherwise each parameter P of params will be checked against recognised basic types, existing regular classes, and existing given sets, as outlined previously for types expressed in scalar form. It is possible therefore that a new given set will be created in $Z++$ if P is neither the name of an existing class, nor the name of a recognised basic type (e.g., the UML type $\text{Stack}[\text{Book}]$ will lead to the creation of the given set BOOK in $Z++$ if class Book does not exist in the class diagram).

Since in order to allow an earlier transfer of UML class diagrams to $Z++$ specifications the UML types can be left unspecified, the formalisation algorithm may produce incomplete definitions for attributes and operations in $Z++$. This means that after the automatic

translation is performed one of the first tasks of the human formaliser will be to complete the information on types if further development of formal specifications is intended.

6.3.2.2 Translation of Attributes

The following apply when translating to Z++ the attributes of UML classes :

- The names of UML attributes will be used as names for the corresponding Z++ attributes, for instance the attribute `size` in UML will be mapped into the same name attribute `size` in Z++;
- The property of the UML attribute will determine the clause in which the corresponding Z++ attribute is placed. Attributes that cannot be modified, declared `frozen` in UML, will be placed in the `FUNCTIONS` clause of the Z++ class, while all other attributes (`changeable`) will be included in the `OWNS` clause. Due to Z++'s specifics, it is assumed that `frozen` attributes are also declared `protected`, since the constants declared in the `FUNCTIONS` clause of a Z++ class are local to the class and to its subclasses;
- The initial value of the attribute, if provided in UML, will be used as follows:
 - if the attribute is listed as `changeable` the initialisation of the attribute will be performed using an assignment statement in the `init` operation of the Z++ class;
 - if the attribute is `frozen` the initialisation will be performed in the predicate part of an axiomatic box definition that will be included in the `FUNCTIONS` clause;

It is assumed that the type of the initial value of the attribute is the type of the attribute, which means that for array types the initial values must be given as sequences of the form

$\langle v_1, \dots, v_n \rangle, n \geq 0$;

- The visibility of an attribute `att` of a class `C` will be treated as follows by the translation algorithm:
 - if the attribute has `public` visibility the name of the attribute will be appended to the clause `PUBLICS` of the Z++ class `C`;
 - if the attribute has `protected` visibility no special measures will be taken since in Z++ all attributes are inherited automatically by the derived classes;

- if the attribute has private visibility it will be appended to the list of hidden features kept by the algorithm for each class. This list, if not empty after the processing of all the attributes and operations of the class, will require a hiding operation applied to the class, as detailed in Subsection 6.3.2.4.
- the type of the attribute will be determined according to the translation principles for types presented in Subsection 6.3.2.1.

6.3.2.3 Translation of Operations

The following principles apply for translating to Z++ the operations of UML classes:

- The names of UML operations will be used as names for the corresponding Z++ operations, for instance the operation `determineTrend` in UML will be mapped into the same name operation `determineTrend` in Z++;
- The property of an `op` operation of a UML class `C` will determine the clauses of the Z++ class `C` in which the signature and the definition of the corresponding Z++ operation `op` are placed. Operations declared `query`, which do not change the state of the object, will have their signatures specified in the `RETURNS` clause of the Z++ class `C`, while all other operations will have their signatures included in the `OPERATIONS` clause. For both `query` and non `query` operations, definitions specified as indicated below are included in the `ACTIONS` clause of the Z++ class;
- The parameters of the UML operation `op`, if any, are processed as follows:
 - the type of each operation parameter will be processed according to the translation principles for types described previously and the Z++ type of the parameter will be added to the Z++ operation's signature according to the direction of the parameter. Specifically, if the direction of the parameter is `in` then the type of the parameter will be added to the list of input domains, if the direction is `out` it will be added to list of output domains, and if the direction is `inout` it will be added to both lists;
 - the name of each operation parameter is used to construct the initial part of the operation's definition in Z++. If the type of the parameter is `in` the name of the

parameter post-fixed by the symbol ? (denoting an input variable in Z) will be appended to the operation's definition list of input parameters and if the type of the parameter is out, the name of the parameter postfixed by ! (denoting an output variable in Z) will be added to the operation's definition list of output parameters. If the direction of the parameter is inout, both the above operations will be performed;

- The return type of an UML operation, if present and different from void and Boolean, will be first processed according to the principles outlined for types in Subsection 6.3.2.1 and then placed as an item in the list of output domains of the corresponding Z++ operation's signature. If void or Boolean, no action will be taken;
- The visibility of each UML operation will be processed similarly to the visibility of UML attributes. The name of a UML public operation will be included in the PUBLICS clause of the Z++ class in which the corresponding Z++ operation has been created while the name of a private operation will be added to the list of hidden features maintained by the translation algorithm for each Z++ class for the purposes described in Subsection 6.3.2.4. Protected UML operations will not require any special treatment.

6.3.2.4 Translation of Classes

The following apply for automatic formalisation of UML classes in Z++:

- Only regular and generic classes will be translated, no action being necessary for binding classes, which simply instantiate generic classes. In fact, a particular instantiation of an existing generic class may not necessarily correspond to a binding class present in the class diagram (e.g., if the parameterised Producer[X] class exists in the class diagram, a variable can be declared as P:Producer[Car] in a UML class without having the Producer[Car] explicitly drawn in the class diagram);
- The names of UML classes will be used for their corresponding Z++ classes, each regular or generic UML class C being mapped into a class with the same name C in Z++;
- The class parameters of a generic UML class will be listed in the parameter list of the corresponding Z++ class;

- The names of all direct superclasses of a UML class will be listed in the EXTENDS clause of the corresponding Z++ class;
- All the attributes of a UML class will be processed according to the principles described previously in Subsection 6.3.2.2, information being placed in the PUBLICS, FUNCTIONS, OWNS, and ACTIONS clauses of the corresponding Z++ class, as well as in the list of hidden features maintained by the algorithm for the Z++ class. The list of given sets of the Z++ specification will be updated during this process based on the information contained in the types of UML attributes;
- All the operations of a UML class will be processed according to the principles described previously in Subsection 6.3.2.3, information being placed in the PUBLICS, RETURNS, OPERATIONS, and ACTIONS clauses of the corresponding Z++ class, as well as in the list of hidden features maintained by the algorithm for the Z++ class. The list of given sets of the Z++ specification will be updated during this process based on the information contained by the types of operation parameters and the type of operation return;
- After all the classes in the class diagram are processed as described above, the classes C with a non empty list of hidden features will be used for creating hiding classes, prefixed by the symbol H (from Hiding), classes needed for providing the desired visibility of attributes and operations. Specifically, for each class C with hidden features an operation $H_C \triangleq C \setminus [\text{hidden_features}_C]$ will be included in the Z++ specification and the class H_C will be used instead of C in the EXTENDS list of classes that have C superclass.

6.3.2.5 Translation of Relationships

The relationships included in a class diagram are formalised in Z++ as follows:

- Inheritance relationships (generalisations) are formalised during the translation of classes through the inclusion in the EXTENDS clause of each Z++ class of the names of the class' immediate superclasses;

- Instantiation relationships are formalised during the translation of classes by including the formal parameters of the class in the parameter lists of Z++ classes, as described in Subsection 6.3.2.4;
- Aggregation and composition relationships are formalised by adding to the container class of the relationship an attribute that indicates the contained object or objects. Specifically, if the aggregation or composition is between class W (“whole”) and the class P (“part”), then the attribute will be created in class W with a name and a type that depend both on the multiplicity of the “part” end of the relationship, as follows:
 - if the multiplicity is “one,” then the attribute will have the name p (the class name in lowercase) and its type will be P. For instance, given a one-to-one aggregation or composition between the classes Radio and Antenna, with Antenna the “part” class of the relationship, then the attribute `antenna : Antenna` will be created in the class Radio;
 - if the multiplicity is “many,” then the attribute will have the name $p+s$ and its type will be $\mathbb{P}P$. For instance, considering a one-to-many aggregation or composition between Radio and Button, with Button the “part” class of the aggregation, then the attribute `buttons : PButtons` will be created in the class Radio.

However, if attributes of type P or $\mathbb{P}P$ already exist in W, no additional attribute describing the aggregation/composition will be created in W.

- Associations relationships are formalised by creating a Z++ class that describes the association and by including in the System class of the Z++ specification an object of this class, with appropriate constraints attached. More precisely, considering a many-to-many association `assoc` between classes A and B, then:
 - a class with the name `AssocDescriptor` will be created in Z++;
 - the attributes `instancesOfA` of type $\mathbb{P}A$, `instancesOfB` of type $\mathbb{P}B$ and `assocInstances` of type $A \leftrightarrow B$ will be included in the `OWNS` clause of the `AssocDescriptor` class;
 - the constraint `dom assocInstances = instancesOfA \wedge ran assocInstances = instancesOfB` will be included in the `INVARIANT` clause of the `AssocDescriptor` class;
 - the object `theAssocDescriptor` of type `AssocDescriptor` will be included in the `OWNS` clause of the System class of the specification.

For instance, considering the many-to-many association `departs` between the classes `Flight` and `Airport`, then the class `DepartsDescriptor` will be created in `Z++` with attributes `instancesOfFlight: IPFlight`, `instancesOfAirport: IPAirport`, and `departsInstances: Flight ↔ Airport` placed in its `OWNS` clause (the names of the classes are underlined to indicate that instances of associations are created between existing objects of the classes). A single object `theDepartsDescriptor` of type `DepartsDescriptor` will also be created in the `System` class of the `Z++` specification.

If the association is one-to-one or many-to-one from `A` to `B` than the type of the attribute `assocInstances` will be `A → B` and if the association is one-to-many from `A` to `B` the attribute's type will be `B → A`.

6.3.3 Algorithm for Formalising Class Diagrams (AFCD)

Based on the rules for syntactically well-formed UML class diagrams, classes, and relationships presented in Subsection 6.3.1 and on the formalisation principles described in Subsection 6.3.2, an algorithm for translating the core structural UML constructs into `Z++` specifications is given below in a Pascal-like pseudocode. The structure of the algorithm's input as well as the format of the algorithm's output are given first and then the algorithm is detailed in top-down fashion. The code of a Java program that implements the algorithmic contents of ADFC and adapts its data structures for an OO solution is included in Appendix B. Details that have been omitted from the presentation that follows can be found in the code presented in this Appendix. As a matter of convention, in ADFC's pseudocode the basic structuring module employed, the procedure, is specified as follows:

```
procedure ProcedureName (<inputParams>; <outputParams>) (6.42)
```

where `<inputParams>` is a list of parameters given in the form `<ip1: T1, ip2: T2, ..ipM: TM>`, with each `ipi`, $1 \leq i \leq M$, an input parameter of type `Ti`, and `<outputParams>` is a list of the form `<op1: T1, op2: T2, .. opN: TN>`, with each `opj`, $1 \leq j \leq N$, an output parameter of type `Tj`. For simplicity, the implicit type of output parameters is considered to be `inout`, meaning that the calling

module passes them to the procedure, which returns them after execution in a possibly modified form.

6.3.3.1 AFCD Input

The input of the formalisation algorithm is a representation of a UML class diagram, denoted \mathcal{CD} , that consists of the tuple (C, \mathcal{R}) where C is the set of classes and \mathcal{R} is the set of binary relationships between the classes, $\mathcal{R}: C \leftrightarrow C$. In terms of the structure, the following are considered:

$$C = \{C_0, \dots, C_{N-1}\}, N \geq 0 \quad (6.43)$$

with $N = 0$ for the empty set of classes $C = \emptyset$. Similarly:

$$\mathcal{R} = \{R_0, \dots, R_{M-1}\}, M \geq 0 \quad (6.44)$$

Each class C in C has the following format:

$$C = (\text{name}, \text{ctype}, \text{atts}, \text{ops}, \text{cparams}) \quad (6.45)$$

where *name* is a string identifier and *ctype* one of the following: *reg*, *para*, or *bind*, while the other components have the form:

$$\begin{aligned} \text{atts} &= \{\text{atto}, \dots, \text{att}_{N_a-1}\}, N_a \geq 0 \\ \text{ops} &= \{\text{opo}, \dots, \text{op}_{N_o-1}\}, N_o \geq 0 \\ \text{cparams} &= \{\text{cp}_0, \dots, \text{cp}_{N_{cp}-1}\}, N_{cp} \geq 0 \end{aligned} \quad (6.46)$$

Each attribute *att* in *atts* has the form:

$$\text{att} = (\text{name}, \text{atype}, \text{vistype}, \text{initval}, \text{property}) \quad (6.47)$$

where *name* and *type* are string identifiers, *vistype* is either *public*, *protected*, or *private*, and *property* is either *changeable* or *frozen*. With respect to *initval*, this should be a value of type, but the formalisation algorithm does not perform type checking.

Each operation *op* in *ops* shown in (6.45) has the form:

$$op = (\text{name}, \text{vistype}, \text{params}, \text{rettype}, \text{property}) \quad (6.48)$$

where *name* is a string identifiers, *vistype* is either *public*, *protected*, or *private*, *property* is *none* or *query*, and *params* is a set:

$$\text{params} = \{p_0, \dots, p_{Np-1}\}, Np \geq 0 \quad (6.49)$$

where each parameter *p* in *params* has the form:

$$p = (\text{name}, \text{ptype}, \text{dir}) \quad (6.50)$$

with *name* and *ptype* string identifiers and *dir* one of *in*, *out*, or *inout*.

Each class parameter *cp* in *cparams* given in (6.46) is a string identifier and *attype* of (6.47), *rettype* of (6.48) and *ptype* a (6.50) are type identifiers given as *T*, *T[]* or *T[params]*, where *T* is a string identifier and *tparams* is a list:

$$\text{tparams} = \{tp_0, \dots, tp_{Ntp-1}\}, Ntp \geq 0 \quad (6.51)$$

with each $tp_i, 0 \leq i \leq Ntp-1$, a string identifier.

Each relationship *R* in \mathcal{R} of (6.44) has the form:

$$R = (\text{name}, \text{rend1}, \text{rend2}) \quad (6.52)$$

where *name* is a string identifier or the reserved word *null* and the two ends of the relationship have the structure:

$$\text{rend} = (\text{kind}, \text{classname}, \text{mult}) \quad (6.53)$$

with *kind* either *assoc*, *aggreg*, *comp*, *super*, *generic*, or *none*, the *classname* given as a string identifier, and *mult* specified in the form:

$$\text{mult} = (a_1 .. b_1, \dots, a_K .. b_K) \quad (6.54)$$

where *K* and the range limits a_i and b_i , $1 \leq i \leq K$, satisfy condition (6.33).

6.3.3.2 AFCD Output

The output of the algorithm is a Z++ specification $Z = (H, ZC, OC)$ that consists of a header H that precedes the class declarations, a set ZC of classes, and a set OC of operations on classes that gathers statements that represents operations applied on Z++ classes such as hiding and composition. A statement is considered to be a text consisting of one or more lines built according to the syntax of Z++. For AFCD purposes:

$$\mathcal{H} = (\text{GivenSets}) \quad (6.55)$$

meaning that only given sets are placed by the algorithm in the header specification, with:

$$\text{GivenSets} = \{GS_0, \dots, GS_{N_{GS}-1}\}, N_{GS} \geq 0 \quad (6.56)$$

where each GS is an uppercase string identifier.

The set of Z++ classes has the form:

$$ZC = \{ZC_0, \dots, ZC_{N_Z-1}\}, N_Z \geq 0 \quad (6.57)$$

where each ZC has the structure indicated in (6.62). The set of operation on classes is given as:

$$OC = (\text{HidingOperations}) \quad (6.58)$$

meaning that AFCD constructs only hiding operations on classes for inclusion in OC , the form of HidingOperations being:

$$\text{HidingOperations} = \{HO_0, \dots, HO_{N_{HO}-1}\}, N_{HO} \geq 0 \quad (6.59)$$

where each HO is a Z++ statement.

The form of each ZC in (6.59) is:

$$ZC = (\text{NAME, CPARAMS, EXTENDS, PUBLICS, TYPES, FUNCTIONS, OWNS, RETURNS, OPERATIONS, INVARIANT, ACTIONS, HISTORY}) \quad (6.60)$$

which corresponds to the structure of Z++ described in Appendix A. In the above NAME is a string identifier, CPARAMS, EXTENDS and PUBLICS are lists of string identifiers and all the other components of ZC are sets of Z++ statements. Notationally:

$$\begin{array}{lll}
 \text{CPARAMS} = & \{cp_0, \dots, cp_{N_{zcp}-1}\}, & N_{zcp} \geq 0 \\
 \text{EXTENDS} = & \{ext_0, \dots, ext_{N_{xt}-1}\}, & N_{xt} \geq 0 \\
 \text{PUBLICS} = & \{pb_0, \dots, pb_{N_{pb}-1}\}, & N_{pb} \geq 0 \\
 \text{TYPES} = & \{typ_0, \dots, typ_{N_{tp}-1}\}, & N_{tp} \geq 0 \\
 \text{FUNCTIONS} = & \{fun_0, \dots, fun_{N_{fun}-1}\}, & N_{fun} \geq 0 \\
 \text{OWNS} = & \{own_0, \dots, own_{N_{ow}-1}\}, & N_{ow} \geq 0 \\
 \text{RETURNS} = & \{ret_0, \dots, ret_{N_{ret}-1}\}, & N_{ret} \geq 0 \\
 \text{OPERATIONS} = & \{zop_0, \dots, zop_{N_{zo}-1}\}, & N_{zo} \geq 0 \\
 \text{INVARIANT} = & \{inv_0, \dots, inv_{N_{inv}-1}\}, & N_{inv} \geq 0 \\
 \text{ACTIONS} = & \{act_0, \dots, act_{N_{act}-1}\}, & N_{act} \geq 0 \\
 \text{HISTORY} = & \{hist_0, \dots, own_{N_{his}-1}\}, & N_{his} \geq 0
 \end{array} \tag{6.61}$$

From the AFCD point of view the above corresponds to the external representation of a Z++ class, but for implementation purposes additional components are used for modelling Z++ classes (they make up the “internal representation” of the Z++ class, which facilitates the translation and allows extensions of the algorithm). Specifically, a set of attributes, a set of operations and a list of hidden features are included, as shown in the AFCD code presented in Appendix B.

6.3.3.3 AFCD Pseudocode

The highest level, pseudocode description of the AFCD is given in Fig. 6.1. The input for the FCD procedure is a class diagram, and its output is a Z++ specification. The FCD procedure invokes first the CheckCDSyntax procedure to verify that the rules for well-formed class diagrams are satisfied and, if this is confirmed, proceeds with the translation of UML constructs to Z++ by calling the TranslateCD procedure. The errorFlag variable, visible across the FCD, is used to signal the detection of errors (violations of rules for well-formedness) at all levels of procedure nesting. Specific messages that indicate the kind of the errors detected are issued locally by the lower level procedures.

```

-- Top level UML to Z++ formalisation procedure

procedure FCD(CD:ClassDiagram)
  ZPPS:ZPPSpec;           -- Z++ specification to be generated
  errorFlag := false;    -- flag to signal well-formedness errors
  begin
    CheckCDSyntax(CD);    -- check correctness of the class diagram
    if (not errorFlag) then
      TranslateCD(CD;ZPPS) -- and translate only if no errors found
    endif;
    PrintZPPSpec(ZPPS);   -- print to file resulting Z++ specification
  end FCD;

```

Fig. 6.1 The Top Level FCD Procedure

In the following, the CheckCDSyntax procedure is described only through its high-level components, specific details of implementation being provided by the code included in Appendix B. Here, only the rules that involve more than preliminary checks of the input in terms of expected structures and valid items are covered (examples of such preliminary checks include verifying that two relationship ends have been provided for each relationship and checking that the property of an attribute is either changeable or frozen). The TranslateCD procedure is described after the high-level modules of CheckCDSyntax are presented.

```

-- Check the well-formedness of the input class diagram

procedure CheckCDSyntax(CD:ClassDiagram)
  begin
    CheckRelationships(CD); -- check constraints at relationship level
    if (not errorFlag) then
      CheckAcrossCD(CD);
    end if;
    if (not errorFlag) then
      CheckClasses(CD); -- check constraints at class level
    end if;
  end CheckCDSyntax;

```

Fig. 6.2 The CheckCDSyntax Procedure

The `CheckCDSyntax` procedure shown in Fig. 6.2 consists of three categories of checkings, each addressing a context (class, relationship, or class diagram) that corresponds from a notational point of view to the groups of rules presented in Subsection 6.3.1. However, due to practical considerations, the order of contexts has been changed and, as detailed later, the contents of each group of checkings match only loosely the contents of the associated group of rules (although globally all major rules are covered). More precisely, we have taken the approach of checking in a given context those rules that require (almost) exclusively information available in that context. For this reason, a rule such as (6.41) given previously as a relationship rule (a rule preventing a class to be the superclass of any of its ancestors) is verified in the `CheckAcrossCD` procedure and not in `CheckRelationships`. Regarding the order of checkings, the validation of the internal contents of classes (`CheckClasses` procedure), involving the inspection of lower-level structural details, is performed only if the other two categories of tests are passed. Also, the `CheckAcrossCD` procedure follows the internal checking of relationships since improperly formed relationships would preclude reliable verifications at the class diagram level.

To simplify the pseudocode descriptions that follow, the testing of the `errorFlag` indicator between procedures is no longer shown, but it should be considered that an error in a given procedure would generally preclude the meaningful execution of the procedures that follow. Thus, if a test fails, the execution of the algorithm will stop. With this approach, the UML developer is required to incrementally improve the well-formedness of the class diagram.

Also, since comments are included in the procedures given below, only brief indications on the correspondence between the FCD's procedures and the rules of well-formedness are given in conjunctions with the components of the `CheckCDSyntax` procedure.

As shown in Fig. 6.3, the internal verification of relationships consists of five tests, covering, in order, rules (6.32), (6.33), (6.29), (6.34), (6.35), and (6.36). The other rules listed as relationships rules in Subsection 6.3.1 are checked in the `CheckAcrossCD` procedure, shown in Fig. 6.5.

```

-- Check constraints on the relationships

procedure CheckRelationships (CD:ClassDiagram)
begin
    CheckRelationshipEnds (CD);           -- verify proper ends of the relationships
    CheckWellFormedMultiplicity (CD);    -- verify multiplicity at the two ends
    CheckAssociationsHaveName (CD);     -- verify names are given to associations
    CheckCompMultOne (CD);              -- the whole part of composition and
    CheckRelMultOne (CD, GEN)           -- both ends of generalisation and
    CheckRelMultOne (CD, INST)         -- instantiation must have multiplicity one
end CheckRelationships;

```

Fig. 6.3 The CheckRelationships Procedure

It is necessary to note that the organisation of tests shown in Fig. 6.3 for CheckRelationships was chosen over the faster alternative depicted in Fig. 6.4 because it allows a clear demarcation of tests and a clear separation of error messages.

```

-- Alternative testing of relationships (not used). Faster, but with no clear separation of messages.

procedure AlternativeCheckRelationships (CD:ClassDiagram)
begin
    for i = 0 to M-1 do
        CheckRelationshipEnds (CD.R[i]) -- verify all relationships
        CheckWellFormedMultiplicity (CD.R[i]) -- verify proper ends of the relationship
        CheckWellFormedMultiplicity (CD.R[i]) -- verify multiplicity at the two ends
        if (isAssociation (CD.R[i])) then
            CheckAssocHasName (CD.R[i]) -- associations must have names
        end if;
        if (isComposition (CD.R[i])) then
            CheckWholeMultOne (CD.R[i]) -- the whole part of composition
            -- must have multiplicity one
        end if;
        if (isGeneralisation (CD.R[i])) then
            CheckRelMultOne (CD.R[i], GEN) -- both ends of generalisation
            -- must have multiplicity one
        end if;
        if (isInstantiation (CD.R[i])) then
            CheckRelMultOne (CD.R[i], INST) -- and both ends of instantiation
            -- must have multiplicity one
        end if;
    end for;
end AlternativeCheckRelationships;

```

Fig. 6.4 Alternative CheckRelationships Procedure

More complex verification work is done by the `CheckAcrossCD` procedure, whose component tests are sequentially ordered based on their possible implications on other tests.

```

-- Check constraints across class diagram

procedure CheckAcrossCD (CD:ClassDiagram)
begin
  CheckEndRelClassesExist (CD); -- verify existence of classes involved in relationships
  CheckClassNamesUnique (CD); -- check constraints on names of classes
  CheckDistinctAssocNames (CD); -- distinct names of assoc. between the same two classes
  CheckDuplicateRelationships (CD); -- only assoc and aggreg/comp can be duplicated
  CheckInstantiationEnds (CD); -- verify instantiation ends attached correctly to classes
  CheckMatchingBindings (CD); -- classes in an inst. rel. must have same no. of params.
  CheckNoAncestorToSelf (CD); -- a class cannot be ancestor to itself
end CheckAcrossCD;

```

Fig. 6.5 The `CheckAcrossCD` Procedure

The rules verified by the `CheckAcrossCD` procedure are, in order (6.3), (6.7 to (6.9), (6.38), (6.37), (6.40), (6.41), (6.12), and (6.41).

The last procedure within `CheckCDSyntax` is `CheckClasses`, shown in Fig. 6.6, whose role is to ensure the uniqueness of names of attributes, operations, and parameters of operations, as required by rules (6.14), (6.18), and (6.22).

```

-- Check constraints at class level

procedure CheckClasses (CD:ClassDiagram)
begin
  for i = 0 to N-1 do -- verify all classes in the class diagram
    CheckAttributeNameUnique (CD.C[i]); -- verify names of attrib. within the class
    CheckOperationNamesUnique (CD.C[i]); -- verify names of ops. within the class
    CheckOpParamNamesUnique (CD.C[i]) -- verify names of op. parameters
  end for;
end CheckClasses;

```

Fig. 6.6 The `CheckClasses` Procedure

The translation part of the algorithm, coordinated from the CDTranslate procedure is described next (Fig. 6.7 to 6.20).

The top-level procedure CDTranslate performs the major tasks of translating the classes and the relationships (Fig. 6.7). In order to establish the required visibilities of attributes and operations, it also applies hiding operations on classes, an activity that can take place only after both classes and relationships are processed.

The TranslateClasses procedure (Fig. 6.8) subjects to translation all non-binding UML classes by invoking TranslateClass (Fig. 6.9). Here, detailed formalisation work on individual UML classes is performed. Based on the information available in the input UML class a corresponding Z++ class is created, with its “internal representation” filled according to the translation principles presented in Subsection 6.3.2. Essentially, translations of attributes (procedures TranslateAttributes of Fig. 6.10 and TranslateAttribute of Fig. 6.11) and operations (procedures TranslateOperations of Fig. 6.12 and TranslateOperation of Fig. 6.13) are performed first, followed by placement of information in the “externally visible representation” of the Z++ class. This preparation work for external representation is done by PlaceZPPAttributes and PlaceZPPOperations procedures (Fig. 6.16 and 6.17). Details on the processing of operations are shown in the procedures ProcessOPParameters (Fig. 6.14) and ProcessOpReturn (Fig. 6.15), which deal with the translation of the operation’s parameters and, respectively, of the operation’s return.

Since some of the relationships are implicitly processed during the formalisation of classes, only associations and aggregations/compositions receive special treatment, as indicated by the procedure TranslateRelationships (Fig. 6.18). Details on formalising aggregations and compositions are given in TranslateAggregation (Fig. 6.19), while the translation of association is described by TranslateAssociation (Fig. 6.20).

Further translation details are available from the code included in Appendix B.

– UML to Z++ translation of a class diagram

```

procedure CDTranslate(CD:ClassDiagram; ZPPS:ZPPSpec)
begin
  TranslateClasses(CD;ZPPS);           -- process classes
  TranslateRelationships(CD;ZPPS)      -- process relationships
  ResolveVisibility(;ZPPS)             -- apply hiding operations on Z++ classes
end CDTranslate;

```

Fig. 6.7 The CDTranslate Procedure

– Translation of classes

```

procedure TranslateClasses(CD:ClassDiagram; ZPPS:ZPPSpec)
begin
  for i = 0 to N-1 do                  -- inspect all classes in the class diagram
    if(CD.C[i].ctype /= bind) then    -- translate regular and parameterised
      TranslateClass(CD,CD.C[i];ZPPS) -- classes only (ignore binding classes)
    endif;
  end for;

end TranslateClasses;

```

Fig. 6.8 The TranslateClasses Procedure

– Translation of an individual class

```

procedure TranslateClass(CD:ClassDiagram,C:UMLClass; ZPPS:ZPPSpec)
  ZC:ZPPClass;                         -- Z++ class to be created
begin
  AppendClass(C.name; ZPPC, ZC);       -- create corresponding Z++ class
  if (C.ctype==para) then              -- if UML class is generic transfer formal
    TransferCParams(C; ZC)             -- class parameters to Z++ class
  endif;
  ProcessParents(CD,C; ZC);            -- process parents and fill EXTENDS clause
  TranslateAttributes(CD,C; ZPPS,ZC);  -- formalise attributes
  TranslateOperations(CD,C; ZPPS,ZC);  -- formalise operations
  PlaceAttributes(;ZC);                 -- fill FUNCTIONS, OWNS, and ACTIONS
  PlaceOperations(;ZC);                 -- fill FUNCTIONS, OWNS, and ACTIONS
end TranslateClass;                    -- work done on this class

```

Fig. 6.9 The TranslateClass Procedure

```

-- Translation of attributes

procedure TranslateAttributes(CD:ClassDiagram, C:UMLClass;
                             ZPPS:ZPPSpec, ZC: ZPPClass)

begin
  for i = 0 to Na-1 do
    TranslateAttribute(CD,CD.att[s[i]];ZPPS,ZC) -- inspect all attributes of the class
                                                -- and save info in Z++ class
  end for;
end TranslateAttributes;

```

Fig. 6.10 The TranslateAttributes Procedure

```

-- Translation of an attribute

procedure TranslateAttribute(CD:ClassDiagram, att:UMLAtt;
                             ZPPS:ZPPSpec, ZC:ZPPClass)

  zatt: ZPPAtt; -- Z++ attribute to be created
begin
  zatt.name = att.name; -- take name,
  zatt.visibility = att.visibility; -- visibility,
  zatt.initval = att.initval; -- and initial value from UML attribute
  if (att.property == changeable) then -- determine place of attribute in Z++
    zatt.clause = OWNS -- class depending on property
  else
    zatt.clause = FUNCTIONS
  end if;
  if (zatt.visibility == public) then -- make provisions for attribute visibility
    Append(zatt.name; ZC.Publics)
  else if (att.visibility == private) then
    Append(zatt.name; ZC.HiddenFeatures)
  end if;
  ProcessType(att.type,CD,ZC;ZPPS,zatt.ztype); -- determine type of Z++ att. and
                                                -- possibly add to given sets of Z++ spec.
  Append(zatt;ZC); -- finally, add attribute to Z++ class
end TranslateAttribute;

```

Fig. 6.11 The TranslateAttribute Procedure

```

-- Translation of operations

procedure TranslateOperations(CD:ClassDiagram, C:UMLClass;
                           ZPPS:ZPPSpec, ZC: ZPPClass)

begin
  for i = 0 to No-1 do
    TranslateOperation(CD,CD.op[i]; ZPPS,ZC); -- inspect all operations of the class
    -- and save info in Z++ class
  end for;
end TranslateOperations;

```

Fig. 6.12 The TranslateOperations Procedure

```

-- Translation of an operation

procedure TranslateOperation(CD:ClassDiagram, op:UMLOp;
                           ZPPS:ZPPSpec, ZC: ZPPClass)
  zop: ZPPOp;
begin
  zop.name = op.name;
  zop.visibility = op.visibility;
  if (zop.visibility == public) then
    Append(zop.name; ZC.Publics)
  else if (zop.visibility == private) then
    Append(zop.name; ZC.HiddenFeatures)
  end if;
  if (op.property == query) then
    zop.clause = RETURNS
  else
    zop.clause = OWNS
  end if;
  ProcessOPParameters(CD,op; ZPPS, ZC, zop);
  ProcessOpReturn(CD, op; ZPPS, ZC, zop);
  Append(zop; ZC);
end TranslateOperation;

```

Fig. 6.13 The TranslateOperation Procedure

```

-- Translation of parameters of operations

procedure ProcessOPParams (CD:ClassDiagram, op:UMLop, ZC:ZPPClass;
                          ZPPS:ZPPSpec, zop:ZPPOp)
    ztype:Ztype;                                -- helper variables
    name,dir:String;
begin
    for i = 0 to Npo-1 do                        -- process all operation parameters
        name = op.p[i].name;                    -- take name and
        dir = op.p[i].name;                    -- direction of parameter
        ProcessType(op.p[i].ptype, CD, ZC; ZPPS, ztype); -- determine Z++ type and
                                                -- possibly add to given sets of Z++ spec
        if (dir == in) then                    -- if direction of parameter is in
            Append (ztype; zop.sign.InputDomain); -- append type to input domain
            Append (name+"?"; zop.def.InputList); -- and decorated name to input list
        else if (dir == out) then             -- if direction of parameter is out
            Append (ztype; zop.sign.OutputDomain); -- append type to output domain
            Append (name+"!"; zop.def.OutputList); -- and decorated name to input list
        else                                  -- otherwise, direction is inout
            Append (ztype; zop.sign.InputDomain); -- and therefore do both
            Append (name+"?"; zop.def.InputList);
            Append (ztype; zop.sign.OutputDomain);
            Append (name+"!"; zop.def.OutputList);
        end if;
    end for;
end ProcessOpParams;

```

Fig. 6.14 The ProcessOpParams Procedure

```

-- Interpretation of operation return

procedure ProcessOPReturn (CD:ClassDiagram, op:UMLop, ZC:ZPPClass;
                          ZPPS:ZPPSpec, zop:ZPPOp)
    ztype:Ztype;
begin
    ProcessType(op.rettype, CD, ZC; ZPPS, ztype); -- determine Z++ type and
                                                -- possibly add to given sets of Z++ spec
                                                -- if type neither boolean nor void
    if ((op.rettype /= boolean) &&
        (op.rettype /= void)) then
        Append (ztype; zop.sign.OutputDomain); -- append to output domain
        Append ("result!"; zop.def.OutputList); -- and append result param. to output list
    end if;
end ProcessOpReturn;

```

Fig. 6.15 The ProcessOpReturn Procedure

```

-- Placement of Z++ attribute descriptions in appropriate clauses
procedure PlaceZPPAttributes( ;ZC:ZPPClass;)
  stmtA:String;           -- two statements needed per attribute, one for attribute definition
  stmtB:String;           -- the other for intialisation assignment (if an init value is provided)
  initop:ZPPOp;          -- a Z++ operation that may be needed for the initialisation of attributes
  axiomDef:String;       -- representation for the predicate part of a Z axiomatic definition
begin
  for i = 0 to Nza-1 do           -- process all attributes
    AssembleZPPAttDef(ZC.att[i];stmtA); -- form att. def. from data in Z++ class
    if(ZC.att[i].clause == OWNS) then
      Append(stmtA;ZC.OWNS)      -- place attribute def. in OWNS clause
      if(ZC.att[i].initval not null) then -- if initial value exists
        AssembleZPPAttAssign(ZC.att[i]; stmtB); -- form assignment statement
        if (initop not in ZC.ops) then
          AddInitOp(;ZC.ops)      -- create init op. in Z++ class if needed
        endif;
        Append(stmtB;ZC.initop.code) -- and add initialisation assignment to it
      endif;
    else
      Append(addBar(stmtA);ZC.FUNCTIONS); -- place att. def. in FUNCTIONS clause
      if(ZC.att[i].initval not null) then -- and if initial value exists
        AssembleZPPAttAssign(ZC.att[i]; stmtB); -- form assignment statement
        Append(stmtB;axiomDef)      -- and append it to pred. part of ax. def.
      endif;
    endif;
  endfor;
  Append(schemaPred; ZC.FUNCTIONS); -- complete Z schema in FUNCTIONS
end ProcessOpReturn;

```

Fig. 6.16 The PlaceZPPAttributes Procedure

```

procedure PlaceZPPOperations ( ;ZC:ZPPClass;) -- place op. descriptions in clauses
  stmt: String;           -- statement that can be used for both signature and definition
begin
  for i = 0 to Nzo-1 do           -- process all operations
    AssembleZPPOpDef(ZC.op[i];stmt); -- form op. def. from data in Z++ class
    Append(stmt; ZC.ACTIONS);      -- and place it in ACTIONS clause
    AssembleZPPOpSign(ZC.op[i];stmt); -- form op. signature
    if(ZC.op[i].clause == RETURNS) then
      Append(stmt;ZC.RETURNS)      -- and place it either in RETURNS clause
    else
      Append(stmt;ZC.OPERATIONS)   -- or in OPERATIONS clause
    endif;
  endfor;
end PlaceZPPOperations;

```

Fig. 6.17 The PlaceZPPOperations Procedure

```

-- Translation of relationships

procedure TranslateRelationships (CD:ClassDiagram; ZPPS:ZPPSpec)
begin
  for i = 0 to M-1 do
    if (IsAggreg(CD.R[i]) or IsComp(CD.R[i])) then -- inspect all relationships
      TranslateAggregation(CD.R[i]; ZPPS) -- translate aggregs/comps
    else if (IsAssoc(CD.R[i])) then
      TranslateAssociation(CD.R[i]; ZPPS) -- and associations
    end if;
  end for;
end TranslateRelationships;

-- gen. and instantiations are
-- processed during the
-- translation of classes

```

Fig. 6.18 The TranslateRelationships Procedure

```

-- Translation of aggregation and composition

procedure TranslateAggregation (rel:UMLRelationship; ZPPS:ZPPSpec)

  whole, part: String; -- names of classes in aggreg/comp relationships
  mp: boolean; -- multiplicity of component (one/many as F/T)
  watt: ZPPAtt; -- attribute to be added to container (by default
                -- protected, without initial value, and with clause OWNS)
  cmp = "component"; -- constant string used in def. of attributes

begin
  getEndsDescription (rel; whole, part, mp); -- get info from relationship
  -- and then assign name and type to attribute
  if (!mp) then -- depending on the multiplicity of the part class
    Assign (cmp+part, part; watt) -- multiplicity of part one
  else
    Assign (cmp+part+"s", "P"+part; watt) -- multiplicity of part many
  end if;
  addAttToZPPClass (watt, whole; ZPPS); -- add attributes to container class
end TranslateAggregation;

```

Fig. 6.19 The TranslateAggregation Procedure

```

-- Translation of association

procedureTranslateAssociation(rel:UMLRelationship; ZPPS:ZPPSpec)

    one,two: String;           -- names of the two classes in association
    zatt: ZPPAtt;             -- helper ZPP attribute to be added to Z++ classes
                              -- (protected, without initial value, and with clause OWNS)
    line: String;            -- local variable
    zcls: ZPPClass;          -- Z++ class to be created
    dscr: ="Descriptor";     -- constant strings used in the creation of the new class
    instOf: ="instancesOf";
    inst: ="instances";

begin

    zcls.name = rel.name + dscr; -- the name of new class is derived from the name of assoc.
    getEndsDescription(rel; one, two); -- get the names of the two classes in association
    formInvariantConstraint(one, two; line); -- create predicate for INVARIANT clause
    Append(line; zcls.INVARIANT); -- and append to new class
    AddClassToZPPSpec(zcls; ZPPS); -- append class to Z++ spec.
    Assign(instOf + one, "P " + one; zatt);
    AddZPPAttToClass(zatt, zcls.name; ZPPS); -- add first attribute to the new class
    Assign(instOf + two, "P " + two; zatt)
    AddZPPAttToClass(zatt, zcls.name; ZPPS) -- add second attribute to the class
    FormInstancesType(one,two,ZPPS; stmt)
    Assign(rel.name + inst, line; zatt)
    addAttToZPPClass(zatt, zcls.name; ZPPS); -- add third attribute
    updateSystemDescriptors(zcls; ZPPS); -- update descriptors of associations

endTranslateAssociation;

```

Fig. 6.20 The TranslateAssociation Procedure

6.4 Formalisation of UML State Diagrams in Z++

The second part of formalisation is concerned with the translation of UML dynamic constructs to Z++. More precisely, this formalisation applies to UML state diagrams that are associated to individual classes, the result consisting in information appended to the Z++ classes created previously during the formalisation of the structural aspects of the system. As in the case of formalising class diagrams, the focus is on those parts of the translation process that can be automatically performed. The structure of the present section is similar to that of Section 6.3, but instead of a set of rules for syntactically correct state diagrams the expected format of states and transitions is given in a descriptive manner. Also, the Algorithm for Formalising State Diagrams (AFSD) is not presented at the same level of details as AFCD and it does not have an example of implementation included in the thesis' appendices (due to space considerations only the code for AFCD is provided in Appendix B). However, an example of formalising a state diagram is given in Section 6.4.

6.4.1 Constraints on the Contents of State Diagrams

In Subsection 3.3.2 the notions of event, finite state machine and statechart diagram were discussed and the description of states and transitions was given. Compared with that description, the AFCD uses a slightly different version of state machine, some elements being ignored while other are added. In Fig. 6.21 the general form of a transition is presented, showing the modelling elements used in state diagrams that are accepted by the formalisation algorithm (the structure of these elements is reflected in the format of the AFSD's input detailed in Subsection 6.4.3.1).

As can be seen from Fig. 6.21, the AFSD takes into consideration timed transitions, in the sense described in [Lano95], but internal transitions of states (which do not cause state changes) and deferred events (that could be handled by the object in different states) are not dealt with during the mechanised translation to Z++. Also, signal events are omitted but all other possible types of trigger events, namely call event, passage of time event, and change

event, are considered. A further simplification is that composite states are not covered, although their possible treatment is briefly discussed in Section 6.6.

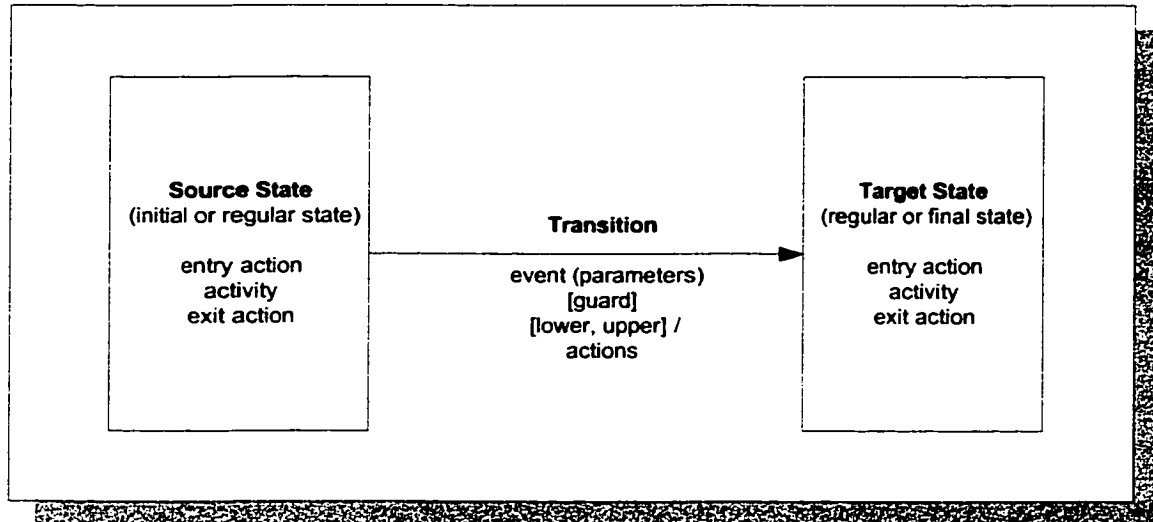


Fig. 6.21 General Form of a State Transition

A state diagram consists of a finite number of states and a finite number of transitions between states. Each state is of one of the following kinds: *initial*, *final*, or *regular* (we introduce the last term to denote a state that is neither initial nor final). Exactly one of the states is the initial state of the diagram, and zero or more final states can be included in the state diagram. Each regular state has a unique name within the state diagram and may contain an entry action, an activity, and an exit action. Initial and final states, which are in fact pseudostates, do not have names and do not contain actions or activities.

Each transition connects a source state to a target state and is either *triggerless* (automatic transition) or has a trigger event of the kind indicated below. A guard condition that can enable or disable the transition, an additional condition denoted *initiation timing condition* (expressed as an interval of time [lower, upper]), and a set of actions can optionally be attached to the transition. The source state and the target state of the transition may be the same, and each transition has only one trigger event. The same event, however, may serve as trigger for several transitions. The trigger event is of one of the following kinds: *call event*, denoted by a

name, *passage of time event*, specified in the form *after* (duration), or *change event*, given as *when* (condition). A call event may have a number of formal parameters, with types indicated. The guard condition is a Boolean expression that when evaluated as true enables the firing of the transition, provided the object is in the source state of the transition. When not indicated on the transition, the guard condition is assumed to be true. The timing limits *lower* and *upper*, if present, indicate the requirements for the transition's initiation time, more precisely after the transition is enabled its execution must be initiated no earlier than *lower* units of time and no later than *upper* units of time. The actions attached to the transition as well as the actions and activities included in states are specified as method invocations, using a name and optionally a list of formal parameters, with types indicated (as in the case of the call events, the requirement for explicit types of parameters is needed for automated translation purposes, although usually the types of parameters are not specified in state diagrams). Actions may represent invocations of operations from supplier classes, in which case the name of an object of the supplier class precedes the name of the action (the dot notation is used, for instance in the state diagram for class C an action *a.op()* denotes the invocation of method *op* of object *a*, where *a* is an object of C's supplier class A). Activities of states are assumed to be operations of the class for which the state diagram was drawn, so the dot notation need not be used (they are methods invoked on *self*).

Depending on the type of their trigger event, the transitions can be classified as *externally invoked* if the trigger is a call event or *internally invoked* if the trigger is a change or passage of time event, or the transition is triggerless. For formalisation purposes triggerless transitions are assimilated to transitions caused by "change events" *when(true)*. Anonymous transitions with guarding condition *guard* are assimilated to transitions triggered by change events *when(guard)*. Normally, when a change event *when(condition)* triggers a transition the guard component of the transition should be omitted (included in condition), although the AFSD processes it properly by appending the guard to the condition of the transition. In order to simplify the translation procedure, it is assumed that transitions from the initial state are triggerless, with no guarding condition, execution timing condition, or actions attached.

Also, it is assumed that the same call event appears throughout the entire state diagram with the same formal parameters, including names and types, as do actions and activities.

6.4.2 Translation Principles for State Diagrams

Before detailing the formalisation of the principal components of state diagrams, the states and the transitions, a number of preliminary observations on the approach taken for formalising state diagrams are necessary.

6.4.2.1 General Principles and Terminology

First of all, we need to recall that while a transition has a single trigger event an event may serve as trigger for several transitions (for the time being the point of view is sequential, meaning that at each occurrence time a trigger event triggers a single transition, but the transition it triggers may be different over the lifetime of the object). As pointed out by Kim and Carrington, who cite [Douglass98], each trigger event must have an associated event *acceptor operation* in the class for which the state diagram has been drawn [Kim00b]. Since an event may trigger more than one transition, this operation may in fact describe several transitions. Because it indicates the effects of the event in terms of transitions triggered and because of notational reasons that will become apparent in Subsection 6.4.2.3, we chose to use the term *transit operations* for these event acceptor operations.

However, using a single transit operation to cover all transitions possibly triggered by a certain event can be difficult to formalise mechanically, mainly because of the potential complexity of the timing constraints included in the `HISTORY` clause of the `Z++` class. In our approach, we resort to the notion of *transition signature* for avoiding excessively long temporal formulae in the `HISTORY` clause, while keeping reasonably small the number of transit operations associated to a trigger event. The use of transition signatures, defined below, provides an intermediary solution between two opposite alternatives: the alternative of using a transit operation for each trigger event, which may lead to complex formulae, and

the alternative of using a transit operation for each transition, which may lead to a large number of operations included in the Z++ class.

By *transition signature* we denote the compound resulting from the concatenation of the following components associated to a transition, starting from the source state: the exit actions of the source state, the trigger event of the transition, the guard condition of the transition, the initiation timing constraint of the transition, and the actions attached to the transition (the parameters of events and actions are also part of the signature). In short, the signature of a transition includes all the components of the transition depicted in Fig. 6.1, prefixed by the exit action of the source state of the transition. This signature serves the purpose of identifying transitions that behave similarly but differ in the states they connect, transitions with identical signatures being described by the same transit operation. For example, in the state diagram of Fig. 3.12, reproduced in a simplified form in Fig. 6.29, there are three shared transition signatures, namely “when (limited_reached)/stop(),” “goSpeedOne,” and “off”. (Fig. 6.29 is used in Subsection 6.4.4 for exemplifying the application of the AFSD).

The above definition of transition signature also hints to the fact that while we attach exit actions of states to outgoing transitions and include them in transit operations, the entry actions of the states are not formalised using transit operations. This is further explained in Subsection 6.4.2.2.

For formalisation purposes, a number of additional conventions are introduced, as follows:

- A transition triggered by a call event is said to be a *simple transition* if its signature consists exclusively of the name of the trigger event and, if provided, of the names and types of the parameters of the event (in other words, the source state of the transition has no exit action and the transition itself has no guard condition, no initiation timing condition, and no actions). The notion of simple transition describes a non-guarded asynchronous method call with no restrictions on initiation time and no appended

actions (examples of such simple transitions in Fig. 6.29 are `reverseDirection`, `goSpeedOne`, `goSpeedTwo`, and `off`);

- Since several transit operations may be created for the same trigger call event, a *basic name* for the transit operations associated with the call event is needed. The basic name is the name of the event, for instance if the call event is `sendCharacter(c: char)` the basic name for the transit operations will be `sendCharacter`, and if more than one transit operation will be created, they will be denoted `sendCharacter1`, `sendCharacter2`, etc. (an exception applies if one of the transit operations describes the simple transition associated with the event –in this case the name `sendCharacter` will be used for it, without an index appended). To distinguish between the operations that model the transitions and the event that triggers the transitions, in the Z++ specification the name of the event will be prefixed by ω , for instance the call event in the case described above will be denoted ω `sendCharacter`.

A note on the creation of Z++ operations describing transitions, actions, and activities is also necessary. During the formalisation of the state diagram, when such an operation is to be created, an operation with the same name may exist as the result of previously applying the AFCD. In this case, it is no longer necessary to create another operation, but an error message will be generated if the input and output domains, as well as the input and output lists of the existing operation do not match the ones that would be generated for the new operation.

6.4.2.2 Translation of States

The formalisation of states proceeds as follows:

- An enumerated type `CState` will be created in the `TYPE` clause of the Z++ class `C` corresponding to the UML class associated to the state diagram (e.g., a type `DisplayState` in the class `Display`). The elements of this type are the names (in lowercase) of the regular states included in the state diagram plus the names `finalk`, $k \geq 1$, generated incrementally for each final state present in the state diagram (final states are included here for the sake of

completeness, although they appear rarely in RTS). In addition, an attribute state of type CState denoting the current state of the object will be created in the OWNS clause. The state attribute, local to the class, will not be listed in the PUBLICS clause of the Z++ class;

- The name of the target state of the transition outgoing from the initial state will be used as initial value for the state attribute. The initialisation of state will be performed in the init operation of the Z++ class;
- The names of the regular states and the generated names of the final states will be used to construct predicates in the HISTORY clause of the Z++ class, along the lines proposed in [Lano95]. Specifically, the following categories of predicates will be generated: permission predicates, definition of transition effects, and reachability properties. Delay, duration, and other timing constraints will also be included in the HISTORY, and the names of the states will be used in these constraints as well, as detailed later in the description of translations of state actions, state activities, and transitions (the last category of HISTORY predicates, describing mutual exclusion properties, involves only the names of transitions). For the first three categories of predicates, the following apply:
 - the permission predicates relate transitions with their source states and will be given in the form

$$\square(\text{transit_operation} \Rightarrow \text{state} = \text{sourcestate}_1 \vee \dots \vee \text{state} = \text{sourcestate}_N);$$
 - the predicates describing the effect of transitions relate transitions with their target states and will be given as

$$\square(\text{transit_operation} \Rightarrow \bigcirc(\text{state} = \text{targetstate}_1 \vee \dots \vee \text{state} = \text{targetstate}_M));$$
 - the predicates for reachability indicate the relationships between source states and their outgoing transitions, and will be specified in the form

$$\square(\text{state} = \text{sourcestate} \Rightarrow \text{transit_operation}_1 \vee \dots \vee \text{transit_operation}_P)$$

The names of regular and final states will be placed accordingly in the above predicates, as will be the names of transit operations created as detailed in Subsection 6.4.2.3.

- The entry action of each state, as well as the activity of the state will be formalised as local operations of the Z++ class, if not already declared otherwise in the class. The principles of translating UML operations described in Subsection 6.3.2.3 apply here as well, the names and the types of the parameters of the actions and activities being

processed in the same way the names and the parameters of operations are processed by the AFCD. A distinction occurs however if an entry action represents the invocation of a method on an instance of a supplier class. Since the class of this supplier object is not specified in the format of entry actions, no generation of operation will take place and no verification will be made to ensure that the method invoked actually exists, but a reminder in the generated Z++ specification will be included as a comment (e.g., `// >> check invocation heater.raiseTemp(delta) is valid <<`). This remainder will help the specifier to complete the formalisation of the state diagram after the AFSD is applied. If an operation with the same name already exists in the Z++ class as the result of previously applying the AFCD no action will be taken, the idea being that entry actions and activities may be operations already declared in the UML description of the class included in the class diagram provided as input to the AFCD. Temporal specifications on the entry action and the activity of the state will be appended in the HISTORY clause of the Z++ class as follows:

- if the entry action $\text{entry_action}(\text{params}_E)$ exists in state S , where params_E are the names of the action's parameters, then the predicate

$$\forall i \in \mathbb{N}_1 \bullet \uparrow(\text{entry_action}(\text{params}_E), i) = \clubsuit((\text{state} = S) := \text{true}, i)$$

will be added to indicate that the entry action initiates its executions as soon as the state is entered;

- if the entry action $\text{entry_action}(\text{params}_E)$ is followed by an activity $\text{activity}(\text{params}_A)$, where params_A are the names of the activity's parameters, then temporal chaining between the two will be indicated as

$$\forall i \in \mathbb{N}_1 \bullet \downarrow(\text{entry_action}(\text{params}_E), i) = \uparrow(\text{activity}(\text{params}_A), i)$$

meaning that the termination of the entry action coincides with the initiation of the activity;

- if the state has only $\text{activity}(\text{params}_A)$ but no entry action, the predicate

$$\forall i \in \mathbb{N}_1 \bullet \uparrow(\text{activity}(\text{params}_A), i) = \clubsuit((\text{state} = S) := \text{true}, i)$$

will be included to indicate that the state's activity commences its executions as soon as the state is entered.

For both the entry action and the activity the precondition $state = S$ will be added to the definition of the operations that describe them;

- The exit actions of the states will be covered by transit operations created to formalise translations, as described in the next subsection.

6.4.2.3 Translation of Transitions

Each transition will be formalised using a transit operation declared in the `OPERATIONS` clause and defined in the `ACTIONS` clause of the in the `Z++` class. As previously stated, a transit operation describes several transitions with the same signature. Differences exist between the formalisation of externally invoked transition (transition whose triggers are call events) and internally invoked transitions (transitions triggered by change or passage of time events), as follows:

- If the transition is triggered by a call event denoted $call$, then for formalisation purposes the basic name of the transit operation will be $call$ and the event itself will be denoted ω_{call} . For each such transition:
 - an operation $call$ with the signature included in the `OPERATIONS` clause and definition included in the `ACTIONS` clause of the `Z++` class will be created using the information provided by the parameters of the event ω_{call} for defining the input and output domains of the operation's signature and the input and output lists of the operation's definition. The name of this operation will be included in the `PUBLIC` clause of the class;
 - if this is the only transition in the state diagram triggered by ω_{call} , or if all the transitions triggered by ω_{call} have the same signature, then the above is the only transit operation associated with ω_{call} . Information extracted from the transitions that have the same signature will be appended to the `Z++` class as follows:
 - if the guard condition $guard$ is specified then a predicate of the type

$$(\text{enabled}(\text{call}) \equiv (\text{state} = S_1 \vee \dots \vee \text{state} = S_K) \wedge \text{guard})$$

will be included in the HISTORY clause of the Z++ class. In this predicate the states S_1, \dots, S_K are the source states of the transitions that share the same signature. Since the well-formedness of the guard condition is not verified, a reminder for the human specifier to check the condition will be included as a comment, in the form // >> check condition [guard] is well-formed <<; The inclusion of this predicate in the HISTORY clause allows further specification by the human formaliser of detailed temporal constraints regarding the execution of transition, for instance in the case of a transit operation that corresponds to a single guarded transition it is possible to write

$$\begin{aligned} (\text{enabled}(\text{call}) &\equiv (\text{state} = \text{sourcestate}) \wedge \text{guard}) \wedge \\ \forall i \in \mathbb{N}_1 \bullet \exists j, j_1, j_2 \in \mathbb{N}_1 \bullet &((\text{state} = \text{sourcestate}) \wedge \text{guard}) \odot \clubsuit (\omega_{\text{call}}, j) \wedge \\ \clubsuit (\omega_{\text{call}}, j) = \rightarrow (\text{call}, i) \wedge &((\text{state} = \text{sourcestate}) \wedge \text{guard}) \odot \uparrow (\text{call}, i) \wedge \\ \downarrow (\text{call}, i) = \clubsuit ((\text{state} = \text{sourcestate}) &:= \text{false}, j_1) \wedge \\ \downarrow (\text{call}, i) = \clubsuit ((\text{state} = \text{targetstate}) &:= \text{true}, j_2) \end{aligned}$$

The above indicates the conditions under the operation call is enabled, shows that the enabling condition holds at the time of the j-th occurrence of the trigger event ω_{call} and that the operation is requested as soon the trigger event occurs. It also indicates that the enabling condition still holds at the initiation of the operation and details the change of state at the termination of the operation (the assumption is that sourcestate and targetstate are distinct, otherwise the last two lines should be omitted);

- if specified, the timing condition [lower, upper] will be used for including in the HISTORY clause the predicate

$$\forall i \in \mathbb{N}_1 \bullet \text{fires}(\text{call}, i) \Rightarrow \text{lower} \leq \text{delay}(\text{call}, i) \leq \text{upper}$$

which indicates that the execution of call initiates sometime between lower and upper units of time after the request for execution is made;

- in the definition of the operation `call` predicates relating the source state with the target state of all transitions covered by the operation will be included in the form

$$(\text{state} = \text{sourcestate}_1 \wedge \text{state}' = \text{targetstate}_1) \vee \dots \vee (\text{state} = \text{sourcestate}_k \wedge \text{state}' = \text{targetstate}_k)$$

unless there is only one target state involved, in which case the inclusion of the predicate `state' = targetstate` will suffice (conditions on source states will be included in permission and reachability predicates);

- the state exit action and the actions attached to transitions are formalised as class operations declared in the `OPERATIONS` clause and defined in the `ACTIONS` clause of the `Z++` class. These operations, which are local to the class, will have their invocations appended in sequence in the definition of the `call` operation (the order is the exit action first, followed by the actions attached to transitions in the order they are written on the transitions).
- if there are several distinct signatures for the transitions triggered by ω_{call} , then for each distinct signature a transit operations will be created in the `OPERATIONS` clause and defined in the `ACTIONS` clause of the `Z++` class. These operations will be declared public. If one of the transition signatures is the signature of a simple transition, then the corresponding transit operation is the `call` operation created previously, the remaining operations being named `call1`, `call2`, etc. If there is no simple transition signature among the signatures of transitions triggered by ω_{call} , then the names of the operations will be `call1`, `call2`, etc.;
 - for each transit operation `callk` ($k \geq 1$), information extracted from the transitions that have the same signature will be appended to `Z++` class in the manner described above for processing guards, initiation timing constraints, and source and target states. However, the state exit action and the actions attached to the transitions are appended in the following order to the body of the transit operation: state exit action first, followed by the invocation of the simple operation `call`, and then by the actions attached on transitions, in the order they are specified on transitions. Operations for

state exit action and transition actions are created in the Z++ class in the way described previously;

- If the transition's trigger event is a change event `when(condition)` then the formalisation proceeds in a way similar to the one described for transitions triggered by call events, the difference being that no operation for the simple transition is created and that internal (spontaneous) transit operations with the name τ_k , $k \geq 1$, will be generated incrementally, one for each group of transitions that have the same signature. These internal operations are local to the class, therefore their names will not be included in the `PUBLICS` clause of the Z++ class. The `condition` of the event will be appended to the guard condition of the transitions, if any, and will be used in the above given formulae in the place of `guard`;
- If the transition's trigger event is a passage of time event `after(time_expression)` then the formalisation is similar to that of transitions triggered by change events, internal transit operations with the name τ_k being generated incrementally by the algorithm for each group of transitions that have the same signature. The only difference resides in the way the temporal condition is handled. For each such condition the predicate

$$\forall i \in \mathbb{N}_1 \bullet \text{enabled}(\tau_k) \wedge \uparrow(\tau_k, i) = \clubsuit((\text{state} = \text{sourcestate}) := \text{true}, i) + \text{time_expression}$$

will be appended to the `HISTORY` clause of the Z++ class meaning that the operation is initiated after `time_expression` units of time from the moment the state is entered, provided the transition is enabled. This predicate need be checked by the human specifier, since no verification of the validity of the time expression is performed by the AFSD.

The translation of transitions continues until all trigger events present in the state diagram are processed, each trigger event leading to the creation of one or more transit operations. Then, all the transit operations created in the translation process will be used to generate mutex and self-mutex predicates, permission predicates, effect of transition predicates, and

reachability predicates, all included in the `HISTORY` clause of the `Z++` class as indicated in Subsection 6.4.2.2. For the first category, it is assumed that transitions in UML state diagrams are both mutually exclusive and mutually self exclusive (see definition of these properties in Chapter 5), therefore the names of all transit operations will be included in both the `mutex` and `self_mutex` expressions appended to the `HISTORY` clause.

6.4.3 Algorithm for Formalising State Diagrams (AFSD)

In the same way the AFCD was described in Section 6.3, the AFSD is presented in this section through the structure of its input and output and through the pseudocode description of its executable contents. For separation of concerns purposes it is assumed that AFSD is invoked after AFCD, although they can be merged in an implementation, as discussed in Section 6.6. With this assumption, the `Z++` class structure corresponding to the one developed in the UML space is already available, thus the AFSD only appends information to `Z++` classes and is not concerned with the creation of classes.

6.4.3.1 AFSD Input

The input for the AFSD is provided by the `Z++` specification resulted from the execution of the AFCD, specification given in the format presented in Subsection 6.3.3.2, and by a finite state diagram \mathcal{SD} that consists of the tuple (S, \mathcal{T}) , where S is a set of states and \mathcal{T} a set of transitions between states, $\mathcal{T}: S \longleftrightarrow S$. In terms of the structure, the following are considered:

$$\begin{aligned} S &= \{S_0, \dots, S_{N-1}\}, N \geq 0 \\ \mathcal{T} &= \{T_0, \dots, T_{M-1}\}, M \geq 0 \end{aligned} \tag{6.62}$$

Each state S in S has the following format:

$$S = (\text{name}, \text{kind}, \text{entry_action}, \text{activity}, \text{exit_action}) \tag{6.63}$$

where `name` is a string identifier (null if the state is not regular), and `kind` is one of the following: `initial`, `regular`, or `final`. The components `entry_action` and `exit_action` can be null, if not provided, or actions given in the form:

$$\text{action} = (\text{name}, \text{params}) \quad (6.64)$$

while activity is either null (if not provided) or an action prefixed by the name of an object, which is a string identifier, possibly null:

$$\text{activity} = (\text{objectname}, \text{action}) \quad (6.65)$$

In (6.64) *params* are given in the format indicated for operation parameters in (6.49) and (6.50).

Each transition T in \mathcal{T} of (6.62) has the form:

$$T = (\text{source}, \text{target}, \text{trigger}, \text{guard}, \text{time_range}, \text{actions}) \quad (6.66)$$

where *source* and *target* are states that belong to S , *guard* is a Boolean expression including the default value *true*, *time_range* is either null or given as an interval [*lower* .. *upper*] with *lower* and *upper* numerical values such that $\text{lower} \leq \text{upper}$, and *actions* has the form:

$$\text{actions} = \{\text{action}_0, \dots, \text{action}_{N_{\text{act}}-1}\}, N_{\text{act}} \geq 0 \quad (6.67)$$

with each action given in the format (6.65). The last component of a transition, the trigger event has the following form:

$$\text{trigger} = (\text{kind}, \text{body}) \quad (6.68)$$

where *kind* is one of the following: *none* (used only for the transition from the initial state), *call*, *change*, or *timing*. If the kind of the trigger event is *none*, then its body is null, and if the kind of the trigger is *call*, then its body has the form:

$$\text{body} = (\text{name}, \text{params}) \quad (6.69)$$

where *name* is a string identifier and *params* a list of parameters with the structure specified in (6.49) and (6.50). If the kind of the trigger is *change*, its body has the form:

$$\text{body} = (\text{condition}) \quad (6.70)$$

where condition is a Boolean expression. If the kind of the trigger is timing, then its body has the form:

$$\text{body} = (\text{duration}) \quad (6.71)$$

where duration is a timed-valued expression.

6.4.3.2 AFSD Output

The output of the AFSD is a Z++ specification having the structure described in Subsection 6.3.3.2. Under the assumption indicated at the beginning of Subsection 6.4.3, this output is generated by appending information to the Z++ specification provided as input to the AFSD.

6.4.3.3 AFSD Pseudocode

Using the convention (6.42) for the representation of procedures, the pseudocode description of AFCD is given in Figures 6.22 to 6.28. These figures show the higher level modules of the AFCD, designed according to the principles of translation outlined in Subsection 6.4.2. Since comments are included in procedures only some brief explanations are given below.

The `SDTranslateProcedure` of Fig. 6.22 coordinates the entire formalisation work. Its three major components are the `TranslateStates`, `TranslateTransitions`, and `WriteHistoryPredicates` procedures. The `TranslateStates` procedure shown in Fig 6.23 has two roles: the first of creating the enumerated type `State` and the attribute state of this type (with proper initialisation), and the second of coordinating the individual formalisation of states. Each state is processed individually by the `TranslateState` procedure (Fig. 6.24), which appends the name of the state to the members of the `State` type and formalises the entry action and the activity of the state, if available.

Transitions are processed based on their trigger event by the `TranslateTransitions` procedure (Fig. 6.25). Details on the formalisation of transitions triggered by call events are given in Fig. 6.26, which contains the pseudocode of the `ProcessCallTrans` procedure. Since call events are asynchronous method calls a simple transit operation is generated in any case for the event, based on the name and parameters of the call event. If there is a single transition signature for this event, it is assumed that the simple transit operation is the only such method needed by the developers of the state diagram, hence the additional work on the simple transit operation done by procedure `CompleteUniqueTransitOperation` (not detailed in the AFSD pseudocode). In fact, if there are no guards, time range, state exit action and transition actions in this single transition signature, the procedure does nothing else other than appending the simple transition operation created previously to the list of transit operations maintained by the state diagram. Since the processing of translations is driven by the trigger events present in the state diagram, it is necessary to mark as “processed” the transitions covered in each invocation of the `ProcessCallTrans` procedure.

If there are several transition signatures for the same call event, the `GenerateTransitOperation` is invoked for each such signature, as shown in Fig. 6.27. Formalisation work involving the processing of state exit action, of the guard condition, of the initialisation timing condition, and of the actions attached to transitions is performed here.

The last procedure shown for the AFSD, the `WriteHistoryPredicates`, appends to the `HISTORY` clause of the Z++ class a number of predicates, as indicated in Subsection 6.4.2.2.

```

-- UML to Z++ translation of a state diagram

procedure SDTranslate(SD:StateDiagram, zcls:String; ZPPS:ZPPSpec)
begin
  TranslateStates(SD, zcls; ZPPS);           -- process states
  TranslateTransitions(SD, zcls; ZPPC);     -- process transitions
  WriteHistoryPredicates(SD, zcls; ZPPC)    -- add predicates to the HISTORY clause
end SDTranslate;

```

Fig. 6.22 The `SDTranslate` Procedure

```

-- Translation of states

procedure TranslateStates (SD: StateDiagram, zcls: String; ZPPS: ZPPSpec)
  zppET: ZPPEnumType;           -- enumerated type to be created
  state: ZPPAtt;                -- and an attribute of this type
begin
  for i = 0 to N-1 do           -- inspect all states in the state diagram
    TranslateState (SD, SD.S[i], zcls; ZPPS, zppET) -- translate each of them and
                                                    -- create the enumerated State type
  end for;
  AddTypeToZPPClass (zppET, zcls; ZPPS);          -- add type to Z++ class
  Assign ("state", zcls+"STATE"; zatt);           -- create attribute state:ClassState
  AddZPPAttToClass (zatt, zcls; ZPPS);            -- add it to the class
  InitialiseStateAtt (SD, zcls; ZPPS);            -- and initialise the state attribute
end TranslateStates;

```

Fig. 6.23 The TranslateStates Procedure

```

-- Translation of an individual state

procedure TranslateState (SD: StateDiagram, S: State, zcls: String;
                        ZPPS: ZPPSpec, zppET: ZPPEnumType)
begin
  if (S.kind == final) then      -- incrementally generate names of final
    AppendFinalState (; zppET, S.name) -- states and append them to STATE type
  else if (S.kind = regular) then
    AppendState (; zppET);       -- append name of reg. state to type
    if (S.entry_act /= null) then
      ProcessEntryAct (S, zcls; ZPPS); -- formalise entry action
    if (S.activity /= null) then
      ProcessActivity (S, zcls; ZPPS); -- formalise activity
    endif;
  end if;
end TranslateState;

```

Fig. 6.24 The TranslateState Procedure


```

-- Translation of transitions

procedure TranslateTransitions(SD:StateDiagram,zcls:String;
                             ZPPS:ZPPSpec)

begin
  for i = 0 to M-1 do
    if (not Processed(T[i].trigger)) then
      if(T[i].kind == call) then
        ProcessCallTrans(SD,T[i],zcls;ZPPS)
      else if (T[i].kind == change) then
        ProcessChangeTrans(SD,T[i],zcls;ZPPS)
      else if (T[i].kind == timing) then
        ProcessTimingTrans(SD,T[i],zcls;ZPPS)
      end if;
    end if;
  end for;
end TranslateTransitions;

```

-- inspect all transitions
 -- if not already processed
 -- process the transition
 -- based on its trigger event:
 -- call event trigger,
 -- change event trigger, or
 -- passage of time trigger
 -- (the transition from the
 -- initial state is not processed)

Fig. 6.25 The TranslateTransitions Procedure

```

-- Translation of transitions triggered by a call event

procedure ProcessCallTrans(SD:StateDiagram,T:Transition,
                          zcls:String;ZPPS:ZPPSpec)
  tsigns[]: TransSign;
  postfixNo: int := 1;
begin
  GenerateSimpleTransitOperation(T.trigger,zcls;ZPPSpec);
  FormTransitionSignatures(SD,T.trigger;tsigns)
  if (tsigns.size == 1) then
    CompleteUniqueTransitOperation(SD,tsigns[0],zcls;ZPPC)
  else
    for i = 1 to tsigns.size do
      GenerateTransitOperation(SD,tsigns[i],zcls;ZPPC)
    end for;
  end if;
  MarkTransitionsProcessed (T.trigger;SD);
end ProcessCallTrans;

```

-- holder for transition signatures
 -- number to be appended to op. names
 -- create simple operation for this trigger
 -- determine all trans. signatures
 -- if one only, update simple op.
 -- otherwise generate a trans. op
 -- for each signature
 -- mark "processed" all transitions with this trigger

Fig. 6.26 The ProcessCallTrans Procedure

```

-- Creation of operations for a transition signature

procedure GenerateTransitOperation (SD:StateDiagram,tsign:TransSign,
                                   zcls:String;ZPPS:ZPPSpec)
  zop: ZPPOp;                                -- transit op. to be created
begin
  if (isSimpleTransSignature(tsign)) then
    CompleteUniqueTransitOperation (SD,tsign,zcls;ZPPC)
  else
    SetName (T.trigger.name+getPostfix;zop);    -- assign postfix number
    ProcessExitAction (SD,tsign,zcls;zop,ZPPC); -- process exit action
    ProcessGuard (SD,tsign,zcls,zop.name;ZPPC); -- use guard for HISTORY
    ProcessTimeRange (SD,tsign,zcls,zop.name;ZPPC); -- use time range for HISTORY
    RelateStatesInOperation (SD,tsign;zop);    -- relate source and target in
                                                -- operation body
    AppendActions (SD,tsign,zcls;zop,ZPPC);    -- create operations as needed
                                                -- and append actions to op.
    AddOperation (zop,zcls;ZPPC);              -- finally, attach op. to class
  end if;
end GenerateTransitOperation;

```

Fig. 6.27 The GenerateTransitOperation Procedure

```

-- UML to Z++ translation of a state diagram

procedure WriteHistoryPredicates (SD:StateDiagram, zcls:String;
                                  ZPPS:ZPPSpec)
begin
  WriteMutexSelfMutex (SD, zcls;ZPPS);    -- write mutex and self-mutex predicates,
  WritePermissions (SD, zcls;ZPPC);       -- permission predicates,
  WriteTransEffects (SD, zcls;ZPPC);      -- transition effects predicates,
  WriteReachability (SD, zcls;ZPPC)       -- and reachability predicates in HISTORY clause
end WriteHistoryPredicates;

```

Fig. 6.28 The WriteHistoryPredicates Procedure

6.4.4 Example of Formalising a State Diagram

In order to illustrate the proposed approach for formalising state diagrams the state diagram shown in Fig. 3.12 is reproduced here in a reduced form, stripped of annotations and with

shorter names for some of its states (Fig. 6.29). By applying the AFSD described in Subsection 6.4.3, the Z++ class presented in Fig. 6.30 is obtained.

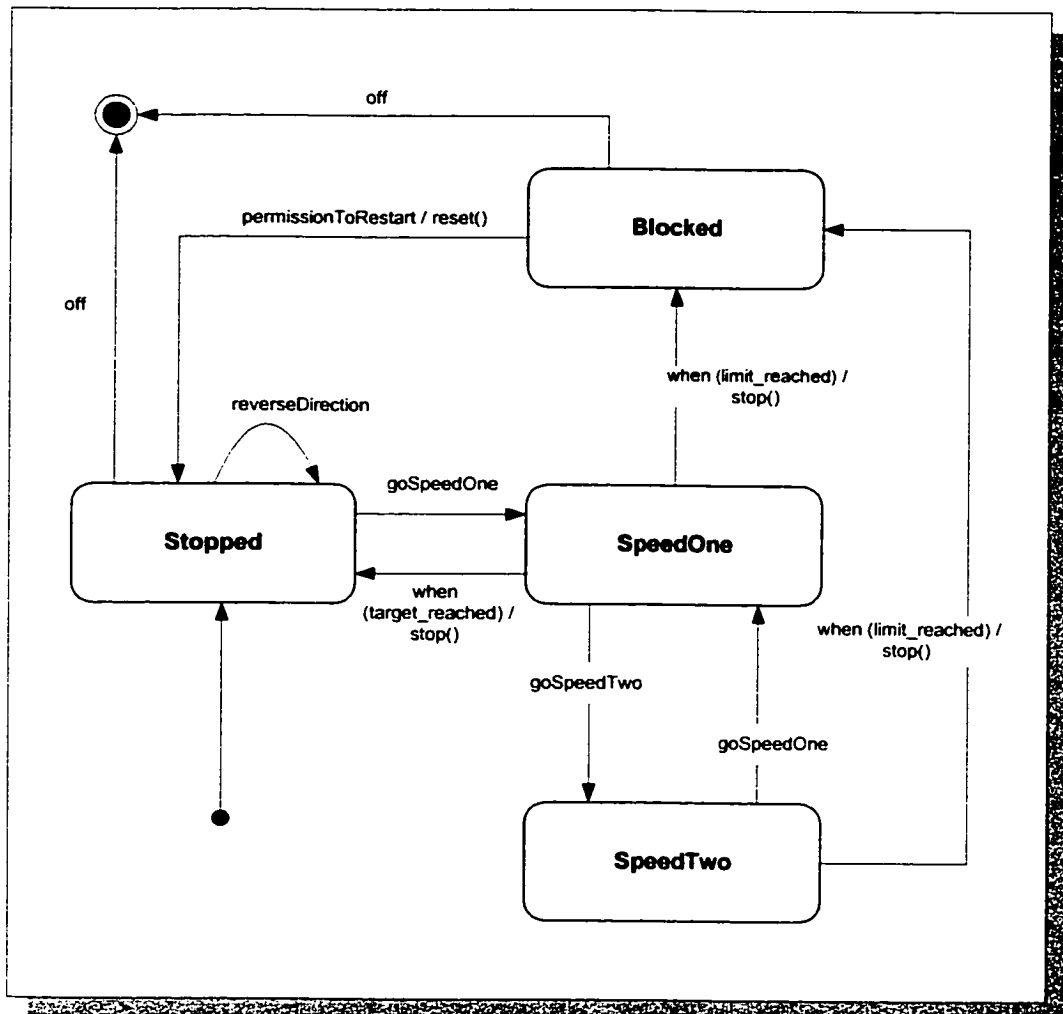


Fig. 6.29 DCMotor State Diagram from the ACTS

The notions of transition signature and transit operation can be easily related to the particular context of the DCMotor state diagram and of the DCMotor Z++ class obtained from it. To further describe the two notions, let us assume that another transition `permissionToRestart`, this time with two actions attached, `stop()` and `reset()`, is added to the state diagram, connecting the states `SpeedTwo` and `Stopped` (the latter being the target state of the transition).

```

CLASS DCMotor EXTENDS Motor
PUBLICS

    permissionToRestart, reverseDirection, off, goSpeedOne, goSpeedTwo

TYPES

    DCMotorState ::= stopped | blocked | speedone | speedtwo | final

FUNCTIONS
OWNS

    state : DCMotorState

RETURNS
OPERATIONS

    permissionToRestart: → ;
    reverseDirection: → ;
    off: → ;
    goSpeedOne: → ;
    goSpeedTwo: → ;
    *τ1: → ;
    *τ2: → ;
    stop: → ;
    reset: →

INVARIANT
ACTIONS

    init ==> state' = stopped;
    permissionToRestart ==> reset;
                                state' = stopped;
    reverseDirection ==> state' = stopped;
    off ==> state' = final;
    goSpeedOne ==> state' = speedone;
    goSpeedTwo ==> state' = speedtwo;
    *τ1 ==> stop;
            state' = blocked;
    *τ2 ==> state' = stopped;
    stop ==> ;
    reset ==>

HISTORY

    // mutual exclusion properties

    mutex({permissionToRestart, reverseDirection, off, goSpeedOne,
           goSpeedTwo, τ1, τ2}) ^
    self_mutex({permissionToRestart, reverseDirection, off, goSpeedOne,
               goSpeedTwo, τ1, τ2}) ^

```

Fig. 6.30 Z++ Class DCMotor Generated by the AFSD (continued on next page)

```

// permission predicates
□(permissionToRestart ⇒ state = blocked) ∧
□(reverseDirection ⇒ state = stopped) ∧
□(off ⇒ state = blocked ∨ state = stopped) ∧
□(goSpeedOne ⇒ state = stopped ∨ state = speedtwo) ∧
□(goSpeedTwo ⇒ state = speedone) ∧
□(I1 ⇒ state = speedone ∨ state = speedtwo) ∧
□(I2 ⇒ state = speedone) ∧

// definition of transition effects
□(init ⇒ O(state = stopped)) ∧
□(permissionToRestart ⇒ O(state = stopped)) ∧
□(reverseDirection ⇒ O(state = stopped)) ∧
□(off ⇒ O(state = final)) ∧
□(goSpeedOne ⇒ O(state = speedone)) ∧
□(goSpeedTwo ⇒ O(state = speedtwo)) ∧
□(I1 ⇒ O(state = blocked)) ∧
□(I2 ⇒ O(state = stopped)) ∧

// reachability properties
□(state = stopped ⇒ reverseDirection ∨ off ∨ goSpeedOne) ∧
□(state = blocked ⇒ permissionToRestart ∨ off) ∧
□(state = speed_one ⇒ goSpeedTwo ∨ I1 ∨ I2) ∧
□(state = speed_two ⇒ goSpeedOne ∨ I1) ∧

// delay, duration, and other constraints

(enabled(τ1) ≡ (state = speedone ∨ state = speedtwo)
  ∧ limit_reached) ∧

  // >> check [limit_reached] is well-formed <<

(enabled(τ2) ≡ (state = speed_one) ∧ target_reached)

  // >> check [target_reached] is well-formed <<

END CLASS

```

Fig. 6.30 Z++ Class DCMotor Generated by the AFSD (continued from the previous page)

In this situation two distinct transition signatures would exist for the transitions triggered by the call event `permissionToRestart`. In terms of operations, the `permissionToRestart` would still be generated as an operation (corresponding to a “simple” transition), but it would not be in fact a transit operation. Thus, it would no longer be included in `HISTORY` predicates, and its

body would be empty. The two distinct transition signatures would have associated two transit operations, `permissionToRestart1` and `permissionToRestart2`, which would be used to describe state changes. In their bodies, an invocation to `permissionToRestart` would be included before the invocation of their specific actions. During the enhancement of the Z++ specification, the human formaliser could decide whether these three operations can be replaced by a single (but more complex) operation.

6.5 Deformalisation: From Z++ Specifications to UML Representations

As discussed in Section 6.2, the reverse mapping, from Z++ to UML, can be useful in certain situations. As in the case of formalisation, this “reverse” translation can be partially mechanised, but it should be noted that relevant information included in the Z++ specification can be lost (in particular, various types, constraints, and bodies of operations). In this section a number of guiding principles for deformalisation are suggested and the outline of an Algorithm for Deformalisation (ADF) is presented.

6.5.1 Principles of Deformalisation

In the following, it is considered that a Z++ specification with the structure given in Section 6.3.3.2 is available, based on which a class diagram together with a set of state diagrams associated to individual classes can be obtained. For the ADF the structure of the output class diagram is the one given in Subsection 6.3.3.1, while the state diagrams are represented as described in Subsection 6.4.3.1.

6.5.1.1 Assigning Types for UML Attributes, Parameters of Operations, and Operation Returns

Due to the specifics of Z++, not all attributes, parameters of operations, and returns of operations present in the Z++ specification will have their types translated to UML. Only

attributes specified as `att:typespec` in $Z++$, with `typespec` detailed as below, and only parameters of operations and returns of operations that correspond to input or output operation domains specified as `typespec` will have their types mapped to UML. The format of `typespec` that allows an automated translation of type to UML is one of the following:

- (a) T (“scalar form”), where T is the name of a given set, or of an enumerated type, or of a regular $Z++$ class, or of a predefined Z type (\mathbb{N} , \mathbb{Z} , or \mathbb{R}). If T is \mathbb{N} the corresponding UML type will be unsigned int, if T is \mathbb{Z} the type in UML will be int, if T is \mathbb{R} the type in UML will be real, and in all other cases the type in UML will be T ;
- (b) $\text{seq}(T)$, PT , or FT (“array form”), with T given as in (a) above. In this case, if T is \mathbb{N} the type used in UML will be unsigned int[], if T is \mathbb{Z} the UML type will be int[], if T is \mathbb{R} the type will be real[], and in all other cases the corresponding UML type will be T [];
- (c) $T[\text{params}]$ (“generic form”), where T is the name of a generic class included in the $Z++$ specification and `params` a list of names denoting actual parameters whose types are assumed to be of form (a) (parameters of generic classes may not be arrays or instances of generic or binding classes). In this case, the translated type in UML will be $T[\text{params}]$.

In practical terms, the above restrictions on `typespec` signify that more complex $Z++$ specifications of types (e.g., involving functions, relations, or Cartesian products) are not mapped automatically to UML.

6.5.1.2 Generating Attributes for UML Classes

The following apply for obtaining the attributes of a UML class C , whose correspondent $Z++$ class is C (for easier referencing the latter will be denoted ZC in the following):

- Each attribute `att` included in the `OWNS` clause of the ZC class will have a corresponding attribute `att` in the C class, provided that the type of the attribute is not a class type (attributes of class type will lead to the creation of associations and aggregations, as shown in Subsection 6.5.1.5). The property of this attribute will be `changeable`, the type of the attribute will be assigned according to the principles presented in Subsection 6.5.1.1

for the translation of types, and the visibility of the attribute will be `public` if `att` is included in the clause `PUBLICS` of class `ZC`, `private` if it is used in the hiding operation defining the `Z++` class `H_C`, and `protected` otherwise. The initial value `initval` will be given to the attribute in the `C` class if an assignment statement `att = initval` exists in the `init` operation of class `ZC`;

- From the `FUNCTIONS` clause of `ZC`, each attribute `att` will be extracted and included in the UML class `C` if the definition `att:typespec` is present in a axiomatic definition included in the clause. The property of this attribute will be `frozen`, the type of the attribute will be assigned according to the principles for translating types presented in Subsection 6.5.1.1, and the visibility of the attribute will be `private` if the name of the attribute is used in the hiding operation defining the `Z++` class `H_C`, and `protected` otherwise (attributes declared in the `FUNCTIONS` clause cannot be `public`). The initial value `initval` will be given to the attribute in the `C` class if a statement `att = initval` exists in the predicate part of the axiomatic definition of the `FUNCTIONS` clause.

6.5.1.3 Generating Operations for UML Classes

The following apply for obtaining the operations of a UML class `C` whose correspondent `Z++` class is `ZC`:

- Internal operations of class `ZC` (operation prefixed by the symbol `*`) and the `init` operation of the class will not be translated to UML;
- All other operations of `ZC` will be treated as follows:
 - The name of the operation in `ZC` will be used as the name of the corresponding operation in `C`;
 - The visibility of the operation will be `public` if the name of the operation is included in the `PUBLICS` clause of `ZC`, `private` if the name appears in the hiding operation defining the class `H_C`, and `protected` otherwise;
 - The property of the operation will be `query` if the operation is declared in the `RETURNS` clause of `ZC` and `none` if it is declared in the `OPERATIONS` clause;

- The return type of the operation will be assigned according to the principles described in Subsection 6.5.1.1, based on the output domain of the operation specified in either the RETURNS or the OPERATIONS clause of the ZC class;
- The parameters of the operation in class C will receive the names used in the definition of the operation included in the ACTIONS clause of ZC. For each parameter, the direction of the parameter will be in if the name of the parameter is decorated with the symbol `?`, out if it is decorated with the symbol `!`, and inout if the parameter appears in both the input and the output lists of the operation. The type of each operation parameter will be assigned as described in Subsection 6.5.1.1, based on the input and output domains of the operation, which are listed in either the RETURNS or the OPERATIONS clause of ZC;
- The precondition of the operation as well as the body of the operation will not be translated to UML. However, assignment statements included in the init operation will be used for assigning initial values to attributes in UML, and predicates involving the state attribute, if available, will be inspected when generating state diagrams.

6.5.1.4 Generating UML Classes

The following apply for obtaining UML classes from a Z++ specification:

- Each class C in Z++ that is not a descriptor of an association (association descriptor classes were introduced in Subsection 6.3.2.5) will have a correspondent class C in UML. If the Z++ class C has an associated hiding class H_C in Z++, the list of hidden features used in the hiding operation that defines H_C will be employed to assign the visibility private to the corresponding features (attributes and operations) of the UML class C, as described in Subsections 6.5.1.2 and 6.5.1.3;
- Each generic class G in Z++ will be translated to generic class G in UML, the names of the formal class parameters of the Z++ class G being used as names for the formal class parameters of the UML class G;

- A binding UML class $G[\text{actual_params}]$ will be created whenever a type $G[\text{actual_params}]$ is encountered in the Z++ specification, with G matching the name of an existing generic class G in Z++ and the number of actual parameters actual_params equal to the number of the formal parameters of the Z++ class G (however, the names of the actual_params should not be the same with the names formal_params of the generic class). If not already present, a binding relationship between the binding class and the generic class will be drawn in the class diagram, with the names of the actual parameters used to differentiate the binding class from other possible classes that instantiate the same generic class (see also Subsection 6.5.1.5 on generating relationships);
- The attributes and the operations of each regular or parameterised UML class will be obtained as indicated in Subsections 6.5.1.2 and 6.5.1.3, based on the inspection of the corresponding Z++ class.

6.5.1.5 Generating Relationships

Relationships will be generated in UML class diagrams as follows:

- Generalisation relationships will be obtained based on the information included in the EXTENDS clause of Z++ classes. For each class P (parent) included in the EXTENDS clause of the Z++ class C (child) a generalisation relationship between P and C will be created in the class diagram. If the EXTENDS clause of C includes a hiding class H_P , the relationship in the class diagram will be nevertheless between P and C ;
- Instantiation relationships will be obtained based on the attributes of generic type $G[\text{actual_params}]$, where G is the name of a generic Z++ class. A binding class $G[\text{actual_params}]$ will be created for each different set of actual parameters actual_params encountered for G , and a instantiation relationship between this class and the generic UML class G will be included in the class diagram;
- Associations will be obtained in two ways:
 - (a) From association descriptor classes that exist in the Z++ specification (their description was given in Subsection 6.3.2.5). For each such descriptor class an

association relationship will be created in the class diagram between the classes A and B included in the definition of instancesOf attributes of the association descriptor class;

(b) From attributes of the type D, seq(D), PD, or FD where D is the name of a Z++ class.

For each such attribute encountered in a Z++ class C an association relationship between UML classes C and D will be created in the class diagram. The attribute may indicate in fact an aggregation or a composition relationship, but the human formaliser will be required to change the type of the relationship if necessary;

- Aggregations and compositions will not be generated automatically by the ADF but, as mentioned above, some of the association relationships produced by the ADF may in fact be aggregations or compositions. It will be left to the human specifier to make the necessary changes.

6.5.1.6 Generating State Diagrams

State diagrams will be created by the ADF only for those Z++ classes C that have an enumerated CState (or State) type defined in their TYPE clause and an attribute state of this type declared in their OWNS clause. For each such Z++ class a state diagram “C’s State Diagram” will be generated as follows:

- The names of the enumerated type State’s members will be used as names of the states created in the state diagram (however, final states, which will be created as well, will not receive names);
- If an initialisation assignment state = entrystate exists in the init operation of the Z++ class, an initial state will be created and an anonymous, non guarded and actionless transition from the initial state to entrystate will be created;
- Based on the predicates included in the HISTORY clause of the Z++ class and on the predicates included in the transit operations of the class (specifically, predicates that relate source states with target states) transitions will be created in the state diagram. For each transition, the name of the transit operation that describes the transition in class C will be attached to the transition in the state diagram.

6.5.2 Outline of the Algorithm for Deformalisation (ADF)

Based on the principles proposed in Subsection 6.5.1 for the generation, starting from a Z++ specification, of a UML model consisting of a class diagram and of a set of state diagrams associated to classes, an outline for a deformalisation algorithm is presented in Fig. 6.31 to 6.33. This outline describes the ADF only in terms of its high level components, but it covers nevertheless all the significant aspects of the Z++ to UML translation process.

As a matter of general approach, the mapping of the Z++ specification to a UML model can be tackled in (at least) two ways. One alternative is to design the algorithm in a manner that allows the successive generation of the major modelling elements of the UML space, namely the classes, the relationships, and the state diagrams. This approach would require however a triple processing of the individual Z++ classes, the first for creating the UML class structure that mirrors the one present in the formal specification, the second for generating the relationships between classes, and the third for creating state diagrams for those classes in which state changes are explicitly described in Z++ via a state attribute. While this approach allows a better separation of concerns, an incremental development of the UML model in terms of major kinds of artefacts, and a less complex structure of the algorithm, it is however less efficient in terms of implementation.

Since this alternative involves a repeated treatment of each Z++ class and we envisage the possibility of applying the deformalisation process on an individual class or a group of selected classes, we have opted for a second approach, that of generating all types of UML elements –classes, relationships, and state diagrams– through a single inspection (processing loop) of the Z++ classes, each class being mapped to UML elements based on the information contained in its definition and on the information provided by the context of the Z++ specification. While this approach allows the complete treatment of an individual Z++ class in a single processing step, it has the disadvantage that the generation of some UML elements is “buried” in modules whose primary purpose is different, more precisely binding classes and association relationships are created, if necessary, during the processing of

attributes (this is nevertheless in agreement with the translation principles described in Subsection 6.5.1.5).

The approach we have taken is apparent in the top-level ADF procedure, presented in Fig. 6.31.

```

-- Z++ to UML translation

procedure ADF(ZPPS:ZPPSpec;CD:ClassDiagram,SDS:StateDiagrams)

begin
  for i = 0 to Nz-1 do
    TranslateZPPClass(ZPPS,ZPPS.ZC[i];CD,SDS);
  end for;
  PrintClassDiagram(CD);
  PrintStateDiagrams(SDS);
end ADF;

```

-- process all Z++ classes
-- show/save results: class diagram
-- and state diagrams

Fig. 6.31 The ADF Procedure

The particular treatment of a Z++ class is handled by the `TranslateZPPClass` procedure, which coordinates the generation of the UML class, the processing of generalisations, and, if appropriate, the generation of the state diagram associated with the class (Fig. 6.32). The last procedure shown for the ADF, `GenerateUMLClass`, describes the work needed for the completion of the UML class (Fig. 6.33). It is here, in the procedures called by `GenerateUMLClass`, where the possible generation of associations and binding classes can take place, while dealing with the types of attributes (processing the types of parameters of operations and of operation returns may also prompt the creation of binding classes).

Nevertheless, as shown in Chapter 9, this organisation of the ADF suits better our modelling purposes. In fact, the closely related generation of the UML class and of the state diagram associated with the class in the `TranslateZPPClass` procedure forecasts the combined use of the regular UML class specification and of the state diagram associated with the class in the integrated modelling approach proposed in Chapter 7.

```

-- Translate individual Z++ class to UML

procedure TranslateZPPClass (ZPPS:ZPPSpec, ZC:ZPPClass;
                           CD:ClassDiagram, SDS:StateDiagrams)

begin
  if (isAssocDescriptor(ZC)) then
    GenerateAssociation(ZPPS, ZC; CD)
  else
    GenerateUMLClass(ZPPS, ZC; CD);

    ProcessGeneralisations(ZPPS, ZC; CD)

    if (hasStateAtt(ZC)) then
      GenerateStateDiagram(ZC; SDS)
    end if;
  end if;
end TranslateZPPClass;

```

-- if the class describes an association
-- simply add association to class diagram;
-- otherwise
-- generate the corresponding UML class
-- (in the process, create associations
-- and binding classes, if detected)
-- process list of ancestors and
-- update relationships in class diagram
-- if there is a 'state' attribute in the Z++
-- create state diagram and add to
-- the collection of state diagrams

Fig. 6.32 The TranslateZPPClass Procedure

```

-- Generate UML Class from Z++ class ; in the process, generate associations and binding classes from type information
-- contained in the definition of attributes

procedure GenerateUMLClass (ZPPS:ZPPSpec, ZC:ZPPClass;
                           CD:ClassDiagram)

  C:UMLClass;
begin
  SetNameAndType(ZC; C);
  if (C.ctype == para) then
    SetClassParameters(ZC; C);
  end if;
  GenerateAttributes(ZPPS, ZC; C, CD);
  GenerateOperations(ZPPS, ZC; C, CD);
  AppendClassToClassDiagram(C; CD);
end GenerateUMLClass;

```

-- UML class to be completed
-- name the class and establish its
-- type (regular or parameterised)
-- if generic, provide parameters
-- attach attributes
-- attach operations
-- then append class to the class diagram

Fig. 6.33 The GenerateUMLClass Procedure

6.6 Notes on the Application of Formalisation and Deformalisation Algorithms

At the conclusion of this chapter, several notes regarding the application of the three proposed algorithms for formalisation and deformalisation are necessary.

First of all, while the focus in this chapter was on those aspects of translations between UML and Z++ that can be automated, it is necessary to mention that the proposed algorithms are intended only to serve as aids during the modelling process, and in no way to substitute the human developer. In fact, we cannot stress enough the importance of the human factor in the process of formalisation (and, generally, in the development process), the quality of the software product depending essentially on the skills of its developers. Also, as shown in the next chapter, while we assign a prominent role in the modelling process to the activities of formalisation and deformalisation, the emphasis is not on automated translations between UML and Z++, but on the combined, efficient use of the two notations.

In practical terms, the three algorithms need be further refined in several aspects. In particular, in conjunction with the integrated specification environment described in Chapter 9, an environment whose design incorporates the mechanics of translation presented in this chapter, the following issues need be tackled (we suggest below solutions for each of them):

- While the AFCD applies to class diagrams, for practical purposes it is necessary to allow the formalisation of a single class or of a selected group of classes. The solution for this is to allow the AFCD to continue to operate within the context of the class diagram and to visually mark in the generated Z++ specification the references made from within the group of formalised classes to classes outside this group (e.g., by including a comment listing the names of referenced but not formalised classes). This would allow the developer to decide if additional classes need be formalised;

- Also regarding the AFCD, its application to two or more related class diagrams need be considered. This is not so much an issue of the algorithm itself as it is an issue of combining and representing the related class diagrams in the environment that uses the AFCD. The problem resides in classes included in one diagram that are in relationships with classes from another class diagram. The suggested solution is to attach a description to the class (similar to a property sheet) indicating the relationships in which the class is involved, irrespective of the class diagram;
- Although not a major issue, the combined use of the AFCD and of the AFSD can also be improved. At this point in time, AFCD is applied first, followed by the AFSD, the latter algorithm only appending information in a Z++ class created by the former. The AFSD can be extended without difficulty to create itself the target Z++ class and, more generally, the work of both algorithms can be integrated in a single formalisation algorithm. Since the same translation principles apply and the data structures used by the algorithms is already in place this integration should be straightforward;
- Regarding the AFSD, its extension to composite and concurrent states is a topic that deserves investigation. The first thing in such extension is to create an enumerated type for each composite state in the state diagram, with an attribute of this type describing the current local state. Then, more complex descriptions of transitions are necessary. Parallel executions can be expressed via the || operator available in RTL;
- Finally, the combined use of the three algorithms, the AFCD, the AFSD, and the ADF is to be considered in an integrated environment (see Chapter 9). The main issue is the “update problem,” which arises when a model is switched back and forth between the two spaces, UML and Z++. The solution, similar to the one used in version control systems, is to let the developer decide on committing the changes. To help his or her decision, things to be added can be marked in a specific way (e.g., with indicators such as “>>>>,” meaning “in,” or new information) and things to be removed in a different way (e.g., with “<<<<<,” meaning “out,” or information to be discarded).

6.7 Chapter Summary

In this chapter translations between structural and dynamic UML model elements and Z++ specifications have been discussed. The focus has been on the formalisation process, which has the role of generating formal specifications from UML class diagrams and state diagrams but the auxiliary reverse process, denoted deformalisation, has also been considered. Detailed principles and algorithms have been presented for the automated UML to Z++ translation and guidelines for the reverse translation have been proposed. In Chapter 7 the activities of formalisation and deformalisation are included in a larger procedural frame that is aimed at guiding the development of the integrated UML/Z++ model of TCS and in Chapter 8 the application of the formalisation algorithms are illustrated through an Elevator Controller case study.

7 A PROCEDURAL FRAME

“Arithmetic is where the answer is right and everything is nice
and you can look out of the window and see the blue sky
--or the answer is wrong and you have to start over
and try again and see how it comes out this time.”

[Carl Sandburg, Arithmetic, *Complete Poems*, 1950]

7.1 Introduction

The translation principles described previously are included in this chapter in a procedural frame whose aim is to make systematic the elaboration of the integrated semi-formal/formal model of the system. Although given as a series of interconnected steps and although a “regular” sequence of steps is proposed, this procedural frame is intended only to guide the development of the model, and not to insist on a pre-established sequence of modelling activities. As shown in this chapter, the frame is flexible enough to accommodate various specification strategies and to support the iterative development of the model. The artefacts obtained in the modelling process are described, including a key modelling element, denoted class compound and introduced primarily for supporting the formalisation process. This new construct represents an extension of the fundamental concept of class and encompasses the traditional UML class and the UML state diagram associated with the class. The specific modelling activities are also described and comments on the various elaboration paths that can be followed during the development of the combined UML/Z++ model are included. In addition to the suggested “regular” sequence of modelling activities an example of an alternative scenario for the modelling process is given. The regular modelling scenario proposed in this chapter is applied on the case study described in Chapter 8 and the entire procedural frame is supported by the Harmony environment presented in Chapter 9.

7.2 Modelling Focus

As indicated in Section 3.1, the approach presented in this thesis is focused on the structural and behavioural aspects of TCS and is aimed at developing OO models in a rigorous, pragmatic, and efficient way. For this reason, a number of modelling activities supported by UML are not included in the procedural frame described in this chapter and their corresponding artefacts (specifically, diagrams) are not included in the integrated UML/Z++ model. This simplification is justified by the fact that the above diagrams are either parallel to some already incorporated (specifically, collaboration diagrams are essentially re-writings of sequence diagrams), can be ignored without losing significant insight into the system (activity diagrams), or can be deferred to later development stages that are beyond the scope of the approach proposed in this thesis (component diagrams and deployment diagrams).

By considering the 4+1 architectural views shown in Fig. 3.4, only the User View, the Structural View, and the Behavioural View are dealt with in the proposed approach, and from the diagrams that support them only the use case diagrams, the class diagrams, the sequence diagrams, and the state diagrams are employed. In addition to the discarded diagrams indicated in the previous paragraph, object diagrams are not utilised either, the reason being twofold: firstly, they bring relatively little information about the system in addition to that already contained in class diagrams and sequence diagrams (“object diagrams show instances instead of classes; they are useful for explaining small pieces of complicated relationships, especially recursive relationships” [TogetherSoft00a]), and, secondly, the objects do appear in sequence diagrams in a more important role, that of describing behaviour (the system structure being sufficiently expressed by classes).

In short, we look at a 2+1 views architecture of the system, a reduction of the generic 4+1 views approach that nevertheless allows a reliable description of the system. It is worth noting that many of the UML applications described in the recent literature focus typically on use cases, scenarios, class diagrams, and statecharts diagrams, e.g., [Howerton99, Barrios99, Xie99, Jigorea00, Xu00] and less frequently other types of diagrams are also presented, e.g.,

[Bell99, Fernandes00]. In fact, having the class organisation completed in terms of both attributes and operations allows the further development of the system possibly up to and including implementation (partition and deployment of components may or may not be necessary, depending on the application).

7.3 Artefacts

During the modelling of the system, a series of diagrams are drawn, modelling constructs are completed, including both UML and Z++ specification of classes, and the formalisation and deformalisation processes are performed. Starting from a set of requirements that describe the desired properties of the system, the following five categories of artefacts (products) are obtained, making up the combined semi-formal/formal model of the system:

- *Use case diagrams*, describing the intended high-level behaviour of the system as seen from the point of view of external entities (actors) that interact with the system. These are typical UML use case diagrams, each capturing a portion of the system's externally visible behaviour (its "functionality"), and each containing a number of use cases that further detail this behaviour. The regular UML notation is used to develop both use case diagrams and use cases. Abbreviations for these constructs, introduced for easier referencing and used as prefix denominations within Harmony's Project Pane described in Chapter 9 are UC for use cases and UCD for use case diagrams;
- *Scenarios*, specific sequences of actions involving the system and the actors that interact with it. Scenarios, as pointed out by Booch, "are to use cases what instances are to classes, meaning that a scenario is basically one instance of a use case" [Booch98, pp. 225]. UML provides sequence and collaboration diagrams for representing scenarios; however, these diagrams involve a high level of detail (they require the designation of classes and objects for carrying out the scenarios) so we felt necessary to introduce a distinction between a scenario and a sequence diagram. Specifically, we see a scenario as an informal, analysis-level description of a particular sequence of actions encompassed by a use case, while a sequence diagram is a detailed, design-level description of the same thing (in sequence

diagrams responsibilities for carrying out actions are assigned to individual classes and objects, as opposed to the system as a whole.) In our approach, another major difference between scenarios and sequence diagrams is that no specific notation is required to represent scenarios, while sequence diagrams are developed using the UML notation. From a development point of view, a refinement step is thus introduced between the elaboration of scenarios and that of sequence diagrams. Scenarios can be written in natural language, possibly as a series of numbered steps [Schach99], captured in decision tables [Davis93], shown using custom-made, application-specific graphical aids (Chapter 8 provides an example), or described in a notation similar to that of sequence diagrams (e.g., event traces [Rumbaugh91]). In order to provide a modality to relate scenarios with their encompassing use case, the notion of *group of scenarios*, a basic structuring mechanism, is introduced. The abbreviation associated to a scenario is SC and the one for a scenario group is SCG;

- *Sequence diagrams*, developed using the UML notation and providing a design-level representation of scenarios. As discussed above, they are also “instances of use cases” and capture the externally visible behaviour of the system but in addition they show internal interactions among objects. The abbreviation for sequence diagrams is SQD and, by symmetry with scenarios, the notion of *group of sequence diagrams* is introduced, with the associated abbreviation SQDG;
- *Class diagrams*, defining the high level architecture of the system and consisting of classes, relationships among classes, and additional structural constraints expressed as multiplicity values. For formalisation purposes, only UML classes and the usual types of relationships indicated in Section 6.3.1 are considered. Class diagrams are represented using their dedicated UML notations, and are abbreviated as CD;
- *Class compounds*, each class compound, denoted COMP, being a simple syntactic extension of class, grouping the regular class description (CLS) and the state diagram associated with the class (CLSTD). The notion of class compound is introduced primarily for supporting the needs of the formalisation process but it represents in general a simple yet useful extension of the concept of class. The idea of a class compound comes naturally from Z++, but it has also been inspired from the approach of

Howerton and Hinchey [Howerton99], who propose the annexation of the Z specification of the class state diagram to the UML description of the class. The intention of Howerton et Hinchey is to directly combine UML descriptions and Z specifications for describing classes in an approach that advocates different notations for modelling different aspects of the system. However, they do not envisage the syntactical concatenation of the UML class and state diagram constructs and do not propose a denotation for their solution. As a brief remark, it is only natural to add when necessary the state diagram of the class (defining possible sequences of executions) to the two traditional sets of elements encapsulated in a class: data (defining structure) and operations (defining behaviour). Thus, the class compound concept can be viewed as a class with enhanced description of behaviour. Of course, not all classes need a state diagram, so the CLSTD section of COMP can be empty. A summary of the abbreviations introduced above is given in Table 7.I.

- *Z++ specification (ZSPEC)*, consisting of a set of Z++ classes (ZPPCs), each Z++ class corresponding to a class from the UML space. The Z++ specification as a whole is the formal counterpart of the combined contents of the class diagrams that make up the UML component of the integrated model of the system.

Table 7.I Abbreviations for Modelling Artefacts

Element	Abbreviation	Element	Abbreviation
Use Case	UC	Class Diagram	CD
Use Case Diagram	UCD	Class Compound	COMP
Scenario	SC	Class Description	CLS
Scenario Group	SCG	Class State Diagram	CLSTD
Sequence Diagram	SQD	Z++ Specification	ZSPEC
Sequence Diagram Group	SQDG	Z++ Class	ZPPC

7.4 Activities

Fig. 7.1 gives a diagrammatic description of the procedural frame proposed in this thesis for modelling TCS. The figure shows both the modelling activities (steps) performed and the artefacts obtained as the result of each activity. Since the artefacts have been discussed in the preceding section, the focus is here on the activities. Before discussing them, a number of conventions and simplifications used in Fig. 7.1 are indicated.

7.4.1 Conventions in the Diagrammatic Representation of the Procedural Frame

Several conventions are used in Fig. 7.1, as follows:

- Activities are represented by rounded rectangles;
- Modelling products (the artefacts) are represented by regular rectangles;
- Continuous, arrow-ended flow lines connect activities with their output products and products with activities that use them as input;
- Dashed, arrow-ended lines represent a change from an activity to another and, in contrast with the continuous flow lines, do not require that artefacts are obtained in the originating activity (the decision to move to another activity may be based on the inspection of the already existing artefacts associated with the current activity). These dashed lines are used in two situations: in the process of iterative development of the model (feedback links), and when moving from one activity to another without necessarily providing new input to the newly initiated activity;
- The steps are numbered and organised in five *stages* (or *levels*), their ordering suggesting the typical flow of activities within the modelling process. Same level activities can be performed in a parallel fashion, including an interleaved form of parallelism;
- The set of diagrams obtained as a result of a specific modelling activity in stages 1 to 3 are generically denoted *collection*, e.g. the Use Case Collection is created in the Definition of Use Cases step.

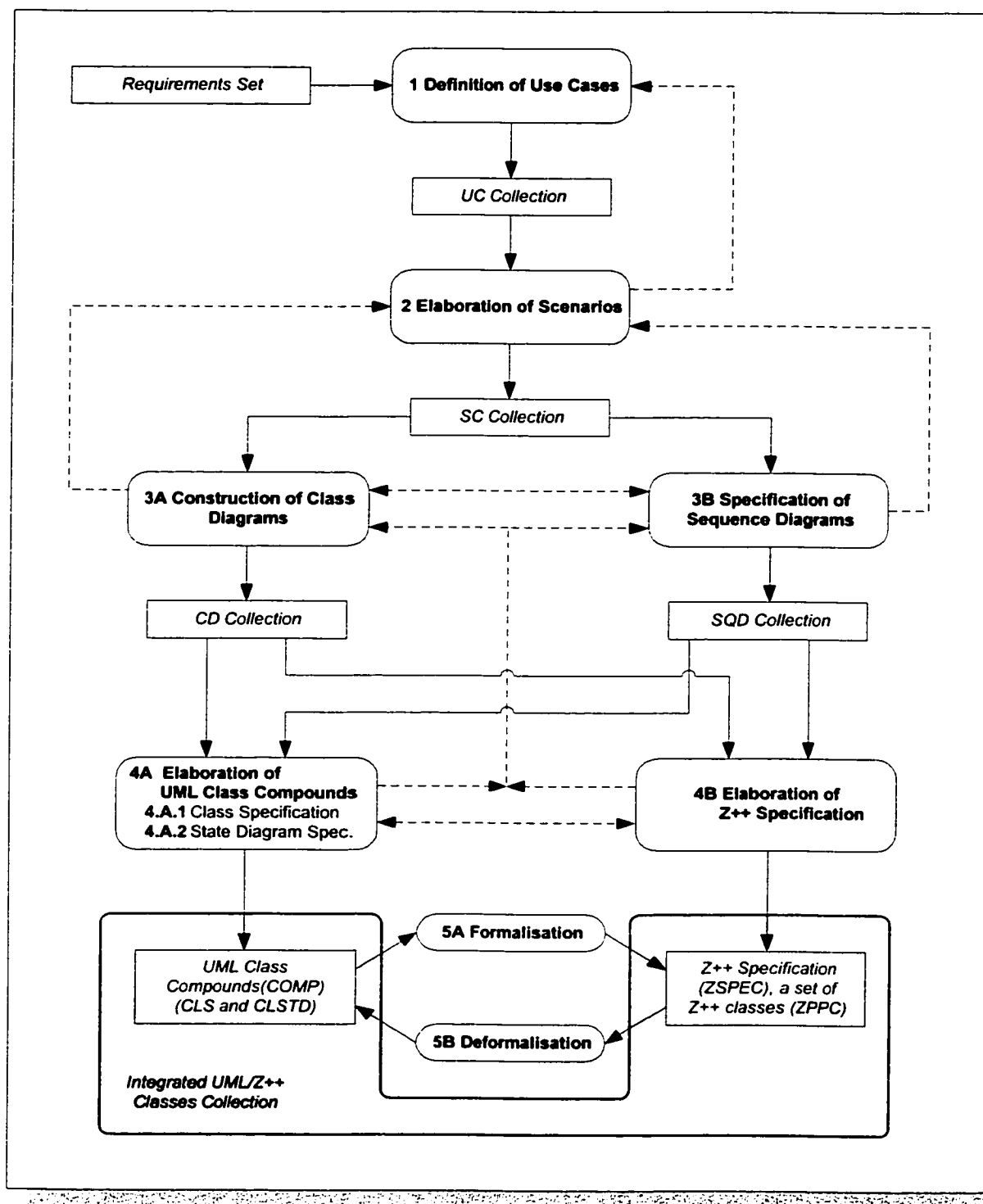


Fig. 7.1 The Procedural Frame

7.4.2 Simplifications in the Diagrammatic Representation of the Procedural Frame

In order to keep the diagram readable a number of simplifications have been made regarding aspects of the modelling process that are less common and therefore less emphasised in the proposed procedural frame. Firstly, while in principle it is possible to come back several levels at a time, for instance from step 4A to step 2 or even to step 1, the revision links are drawn however only from one level to the immediately preceding one (this would be the regular, more frequent way of refinement). Secondly, the CD Collection can serve as input for the Specification of Sequence Diagrams activity (3B) and, vice-versa, the SQD Collection can be used as reference for the Specification of Class Diagrams activity (3A), but more important is their respective input for activities 4B (Elaboration of UML Class Compounds) and, respectively, 4A (Elaboration of Z++ Classes). Thirdly, there is an implicit feedback from activities 5A (Formalisation) and 5B (Deformalisation) to activities 4A and 4B (in fact, so strong a feedback that activities on levels 4 and 5 can be aggregated on a single level, but we needed to highlight the steps of formalisation and deformalisation), which again is not shown for keeping the diagram readable. Lastly, the input for the entire process is represented by the Requirements Set, whose elaboration is not of our concern (we assume that a workable collection of requirements is available). Yet, in practice there is a continuous need for revising the requirements, so links back from modelling activities to the definition of requirements (an activity not shown in Fig. 7.1) should be considered implicit in the diagram.

7.4.3 Stages and Steps

The procedural frame outlined in Figure 7.1 serves only as a guide for modelling TCS, the most important thing being to correctly and completely develop all the artefacts of the integrated UML/Z++ model. The diagram presented in Fig. 7.1 is flexible enough to accommodate various specification strategies and encompasses diverse modelling paths, as discussed more in the next Subsection. In the following, we highlight the modelling activities included in our procedural approach and present them as organised in five stages. The

ordering of the modelling stages corresponds roughly to the typical sequence of activities so the discussion that follows is somewhat biased towards the “regular” modelling scenario proposed in the next Subsection and shown in Fig. 7.2. However, possible variations in the sequencing of activities are also indicated, and are further illustrated in Subsection 7.4.5.

The specific activities performed in each stage are the following:

- At stage 1, starting from the Requirements Set that describes the desired system, a number of use cases that capture segments of externally visible system functionality are identified, making up the Use Cases Collection of the integrated model. During this activity actors interacting with the system are also identified;
- At stage 2 use cases are used to instantiate a number of scenarios that will serve later for the identification of classes. There is no restriction on the way scenarios are represented since they are expected to produce “a rough cut” of the externally visible behaviour of the system and provide high-level insight into the application. Normal scenarios (most likely to occur) as well as abnormal scenarios (or exceptional scenarios, describing situations that diverge from the normal case) are developed and possibly tied together in a Scenario Group that corresponds to a particular use case. Taken together, groups of scenarios as well as individual scenarios (that is, scenarios not yet related to an already defined use case) make up the Scenarios Collection of the model. Although initially individual scenarios as well as groups of scenarios not bounded to uses cases are possible, it is recommended that through iterative revision of specifications the final model should contain only bounded groups of scenarios, a one-to-one correspondence use case-scenario group being desirable;
- At stage 3, using the available Scenarios Collection two possibly intertwined activities can take place: Specification of Class Diagrams (3A) and Specification of Sequence Diagrams (3B). In practice, one needs to develop concurrently the system’s model on both directions, structure (classes) and behaviour (primarily, operations included in classes). Only by simultaneously considering the class structure and the responsibilities of classes and class instances, as captured in sequence diagrams, can the catch-22 type of problem

at this level be resolved (what classes and objects to include in the sequence diagrams if the class diagram is not defined, and what classes make up the high-level architecture of the system if the internal behaviour is not known? –recall that scenarios describe externally visible behaviour). However, in practice, the specification of sequence diagrams is the one that can be deferred since in general it is easier to construct the class diagrams by exploiting the information contained in scenarios (class diagrams may contain only the names of the classes, without any other details, while the sequence diagrams necessarily include both classes and their operations). Thus, step 3A is normally performed first and step 3B follows. In fact, the specification of sequence diagrams performed in step 3B can be omitted all together, as shown in Subsection 7.4.5. The best thing, however, is not to ignore it, but to use it at least as a “revision checkpoint,” with input from all subsequent levels. In short, from our point of view, on stage 3 the development of class diagrams is compulsory while the development of sequence diagrams is recommended;

- At stage 4 the CD Collection as well as the SQD Collection (if available) provide the basis for the detailed specification of classes. An argument can be raised about the development of classes represented separately from the development of class diagrams and, indeed, there is a blurred line between these two activities. We separate them for systematisation purposes and view the Specification of Class Diagrams as an activity in which the rough sketch of the system’s class structure is drawn (in terms of classes, relationships, and cardinality constraints) while the subsequent activities of UML and Z++ class elaboration are concerned with the specification of class details (attributes, operations, and constraints). And, indeed, stages 3 and 4 are the closest related stages in the “stratification” suggested in Fig. 7.1. Regarding the “parallel” steps 4A, Elaboration of UML Class Compounds, and 4B, Elaboration of Z++ Classes, they can be started and performed simultaneously (this is the reason for placing them on the same level) but the typical way is to perform step 4A first or to perform only the step 4A and rely on the subsequent formalisation of class compounds (step 5A) to obtain Z++ specifications of classes. In the regular flow of activities shown in Subsection 7.4.4 we actually use step 4B as a refinement activity, which follows step 5A. The Elaboration of UML Class

- Compounds on stage 4 consists in: (a) establishing the attributes and the operations of the classes as well as the constraints attached to classes in the regular UML construct of class (CLS); and (b), in drawing state diagrams (CLSTD) for those classes that require them, thus completing for each class in CDs its corresponding class compound COMP;
- At stage 5 the formalisation of selected UML class compounds takes place in step 5A by applying initially the rules for automated translation described in Chapter 6 and then by manually adding the necessary details to the formal specification. This activity has the role of producing rigorous descriptions of the system, captured in the Z++ specification. It provides the strongest basis for refining the model, many ambiguities, omissions and inconsistencies being detected here. At the same level of modelling, deformalisation of classes initially written in Z++ (step 5B) can be performed according to the guidelines suggested in Chapter 6. However, as shown in the next Subsection, the “regular” flow of activities includes only step 5A at this level of modelling and the procedural frame treats the activity of deformalisation as a “variation” of the modelling process.

It is important to note that iterative refinements of the products obtained so far need be performed. We envisage essentially two categories of iterative development, one expectedly more intensive (more frequently performed), which can be called “short range revision” because it involves backwards stages 5 to 3 only, and the second performed less frequently yet reaching farther, which can be called “long range revision,” potentially affecting backwards all the stages of activities, including the (not shown in the diagram) elaboration of requirements. As already mentioned the iterative development of the artefacts that constitute the integrated model of system is primarily propelled by the process of formalisation, a key idea of our approach being to use formalisation of UML constructs for improving the chances of detecting errors.

7.4.4 The Regular Sequence of Modelling Activities

A graph-like representation of the regular sequence of modelling activities, which for simplicity omits the products of each activity, is represented in Fig. 7.2. (In UML terms, this

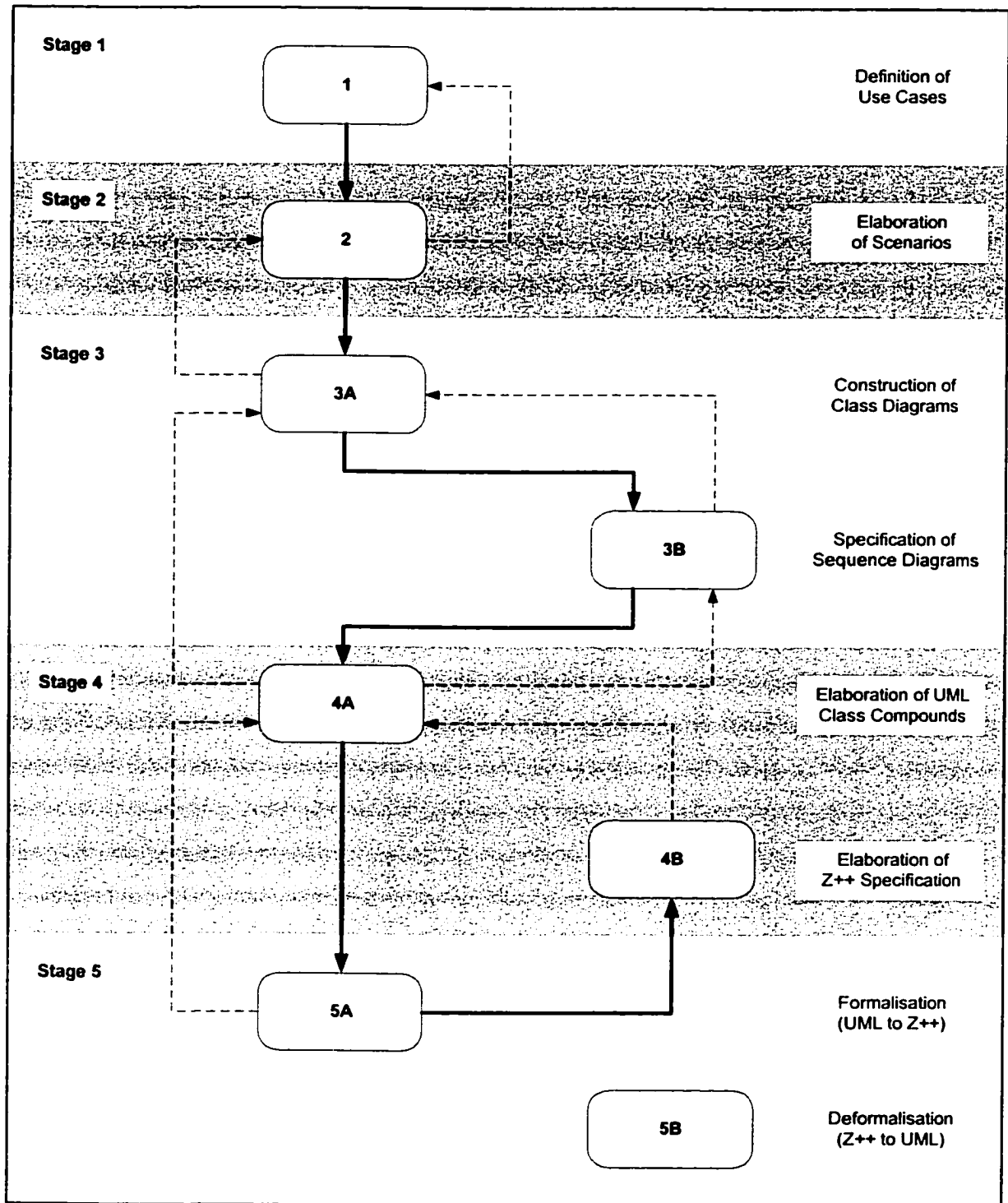


Fig. 7.2 Regular Sequence of Modelling Activities

can be assimilated with the normal scenario of the use case represented by the procedural frame described in Fig. 7.1) The modelling stages are highlighted, the direct flow of activities is emphasised by a thicker, continuous line and the iterative revisions of specifications are indicated by a dashed line. This scenario, which in its “forward segment” (that is, not including feedback links) does not encompass the deformalisation activity (reserved for “irregular” modelling scenarios), can be succinctly described by the <1, 2, 3A, 3B, 4A, 5A, 4B> sequence, where the numbers are associated with activities as indicated in Fig. 7.1.

7.4.5 Alternative Flows of Modelling Activities

The procedural frame presented in Fig. 7.1 encompasses different orderings of activities and we do not claim that the “regular” flow suggested in the previous Subsection represents the unique or the most effective way of developing the integrated UML/Z++ model of the system. There are other alternatives possible, and depending on the particular application, on the experience of the development team, as well as on an a series of other factors, including project priorities and deadlines, one of them may be considered better suited for the particular development needs of a given application. Our “regular” chaining of modelling activities represents only a reference procedure which we believe can be applied in the general case, but nevertheless we do not constrain the ordering of the steps in the modelling process, more important being the correct completion of the integrated model.

Among the other alternatives of sequencing the modelling activities, the one presented in Fig. 7.3 is described here because it highlights a specific strategy that deserves further examination. More precisely, this example of “irregular” scenario for the modelling process can be described in its “forward segment” as <1, 2, 3A, 4A||4B, 5A||5B, 3B>, where the symbol || describes parallel activities (notice that in order to show that 3B comes after 5A and 5B a compromise regarding the notation has been made in Fig. 7.3, where thick dashed lines are used as part of the “forward segment”; they are however different from the regular feedback connections, which continue to be represented as thin dashed lines). Two elements are special in this scenario: first, the fact that the description of classes proceeds in parallel in UML and Z++ and, second, that step 3B comes last in the forward part of the scenario.

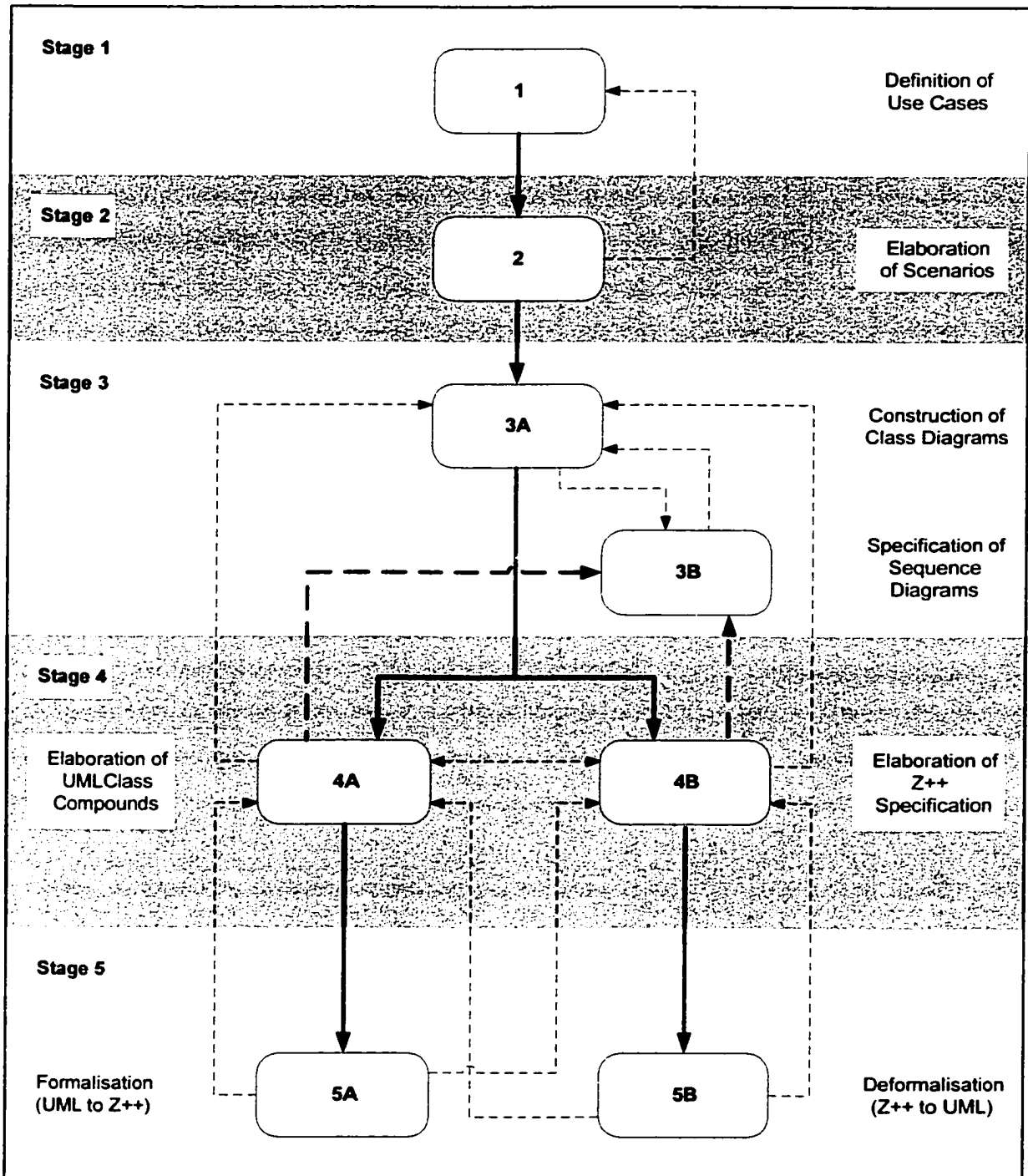


Fig. 7.3 An Example of "Irregular" Flow of Modelling Activities

The first element highlights the idea that various teams of specifiers may have various backgrounds and while some would favour the use of UML, some may prefer employing Z++ as specification notation. In fact, there is the possibility that the specification of classes may proceed first in Z++ and then in UML (and, by extrapolation, it is theoretically possible to have all classes specified in Z++ and not at all in UML). The second element illustrates the idea previously mentioned in Subsection 7.4.3 that sequence diagrams can be used as a tool for fine-tuning the specification, and thus can be the last set of artefacts developed in the modelling process. Of course, additional refinements for improving the accuracy of the model follow in any case.

Another example of an irregular modelling scenario, which stresses rapid development is, in its “forward segment,” <2, 3A, 4A, 5A, 4B>, meaning that the definition of use cases (step 1) and the specification of sequence diagrams (step 3B) are omitted. In short, this modelling alternative takes a “fast-track route” and, after the elaboration of scenarios, class diagrams are developed, UML classes are detailed, the formalisation process takes place, and the detailed specification of Z++ classes is completed. It represents in fact a shorter version of the regular flow of modelling activities suggested earlier. Interestingly, perhaps due to space limitation, in many papers describing the use of UML only the artefacts of steps 2, 3A and 4A are described, in some cases step 2 being skipped as well. While we recommend the regular alternative described in 7.4.3 the above modelling scenario can nevertheless work well in various application contexts.

7.5 Chapter Summary

In this chapter a procedural frame for pragmatic, efficient and reliable modelling of TCS have been presented. The artefacts produced in the modelling process, specifically various sets of UML diagrams organised in collections, the set of UML class compounds, and the formal specification consisting of a set of Z++ classes have been presented. The class compound, a simple yet useful construct serving primarily the purpose of formalising UML

classes in Z++ and representing a practical extension of the fundamental notion of UML class, has been introduced. Specific abbreviations that are later used in the development of the Harmony environment have been associated with the modelling artefacts. The modelling activities, included in the proposed procedural frame and organised in five stages have also been described and the modelling process has been discussed in terms of both regular and irregular chainings of activities. The application of the suggested modelling process, relying on the combined use of UML and Z++, is illustrated on an Elevator System case study presented in the next chapter. It has also driven the design of the supporting tool Harmony described in Chapter 9.

8 AN APPLICATION: THE CASE OF THE ELEVATOR SYSTEM

“In order to get some kind of limit to this enormous subject [elevators] it seems sensible to restrict this study to those devices which have land as their starting point, leaving aside the larger question of aviation and rocketry.”

[Jean Gavois, in the Preface to his *Going Up: An Informal History of the Elevator from the Pyramids to the Present*, Otis Elevator Co., New York, 1983]

8.1 Introduction

Our modelling approach is illustrated in this chapter by a fairly complex application, an Elevator System (ELS). A brief review of this frequently used case study starts the presentation, then the application is defined in terms of general and temporal requirements. In particular, the timing constraints imposed on ELS are shown to provide a comprehensive coverage of the Dasarathy constraints discussed in Chapter 5. Following the specification steps presented in Chapter 7 the elevator system is subjected first to UML modelling, then the formalisation algorithms described in Chapter 6 are applied. The need of enhancing the formal specification and of precisely expressing the temporal requirements placed on systems is emphasised and, based on this application, observations regarding the modelling process proposed in Chapter 7 are included.

8.2 On the Elevator Case Study

The elevator example constitutes one of the preferred case studies of software engineering authors, its extraction from a daily life reality (everybody knows what an elevator is) doubled by its intrinsic complexity --which allows the illustration of various modelling concepts and techniques, including the treatment of temporal constraints-- accounting for its popularity and frequent employment. The origins of this case study can be traced back to Donald Knuth's first volume on the Art of Computer Programming, where a simulation program of Caltech's Mathematics building's elevator was included to exemplify coroutine-based implementation techniques [Knuth73, pp. 280-295]. Since then, many other authors have resorted to the elevator problem as a means of illustrating new software development approaches; to point out only a few reports focused on an elevator system, we refer to Glenn Coleman et al's paper on simulating concurrent systems using Statemate specifications and automatic prototyping [Coleman90], Zhang and Mackworth's formal description of embedded real-time systems using Constraint Nets [Zhang93], Dong et al's approach on specifying parallel and distributed systems in Object-Z [Dong97b], Duval and Cattel's PROMELA and Synchronous C++-based method for developing safe process control applications [Duval97], and Schach's textbook on software engineering [Schach99]. The latter author points out that the elevator case study is non-trivial ("the problem is by no means as simple as it looks" [Schach99, pp. 347]), and can be of great value when illustration of software development techniques is intended. In fact, Schach makes the elevator system one of the two main case studies recurrent in his book. Recently, a detailed, comprehensive Object-Z description of an elevator has been proposed [Mahony00] and although it is one of the few that employs an object-oriented variant of Z, it differs from ours in several major aspects: firstly, it is purely formal, and we combine semi-formal graphical descriptions with formal specifications; secondly, they use CSP and we employ RTL as primary instrument for capturing temporal properties of systems; and thirdly, the OO extension of Z they employ is different from ours. In addition, our goal in this chapter has been to illustrate the steps and the artefacts of the modelling process proposed in Chapter 7, without giving comprehensive

details on a specific application. Thus we have been less ambitious with our Elevator System: it is not a multiple elevator-system (although the modelling resources employed in our approach, in particular UML and RTL, allow the expressing of concurrent behaviours), and it is not specified in all details.

The starting point for our example was provided by the multiple-elevator system presented by Robert Holibaugh in his special report on Joint Integrated Avionics Working Group's Object-Oriented Domain Analysis Method (briefly denoted JODA) [Holibaugh93]. However, in order to make it illustrative for our purposes, we modified the problem statement in numerous places, in some cases by adding new requirements or by providing supplemental details to the existing ones, while in others by eliminating stipulations that would have had only limited significance for exercising our approach (for instance, we renounced providing the elevators with back doors). For the same illustration purposes, we have added a set of temporal constraints (time conditioning was non-existent in Holibaugh's case study) and described the solution of the problem in a fair level of detail. In this way, our example has departed significantly from its starting point, and acquired a "personality" of its own. Although fictive, without a precise correspondent in real life, the elevator described below is sufficiently general to be easily imagined working around the clock in the concrete, shadowing high-rise office building across the street.

8.3 The Problem

In our initial source of inspiration, the [Holibaugh93] report, the elevator system was part of an Office Building Transportation System that also encompassed an escalator system and a set of staircases. Since the focus of this application is on the elevator system, detailed requirements on the building's escalators and stairs are not considered below. The general requirements for the elevator are denoted R_x (where x is a number provided for casier referencing) while the requirements that explicitly impose timing constraints on the system are denoted T_x . The correspondence between the temporal constraints placed on the Elevator

System and the basic Dasarathy timing constraints to which they can be related is given in Subsection 8.3.3 and both types of requirements are consequently treated by the problem's solution, presented in Sections 8.4.

8.3.1 General Requirements for the Elevator System

The general, non time-related requirements for the Elevator System are the following:

- [R1] The elevator serves two or more floors;
- [R2] The elevator contains on board a set of destination buttons (car buttons), one for each floor served by the elevator. When pressed, a destination button becomes illuminated and remains so until the elevators arrives at the corresponding floor;
- [R3] The elevator has on board a set of lights (floor indicators), in one-to-one correspondence with the floors. At any given moment, exactly one of these indicators is lit, showing the floor the elevator is currently at;
- [R4] The elevator has on board two door buttons (a close door button and, respectively, an open door button) which, when the elevator is stopped at a floor, can be pressed by the passengers to close the door earlier than otherwise done automatically and, respectively, to keep the door open longer than otherwise allowed by the elevator's preset timeout;
- [R5] The elevator contains an Alarm button that, when pressed, will generate an intense audio signal (further details are given in constraint [T5]);
- [R6] Each floor except the top and bottom floors has two request buttons, one for requesting the elevator to go up, and the other to go down. When pressed, such a button becomes illuminated and remains so until the elevator arrives at the floor and then moves in the requested direction. The terminal floors (the bottom floor and the top floor) have only one request button;

- [R7] The elevator has a single elevator door, which is either closed or open. The door cannot be opened while the elevator is moving and, reciprocally, when open it will prevent the elevator from moving;
- [R8] On each floor, there is a floor door that will work in tandem with the elevator's door and, for safety reasons, a floor door can be open only if the elevator is stopped at that particular floor;
- [R9] When an elevator has no requests, it will remain idle at the last visited floor (the last target floor at which the elevator has stopped), with its door closed;
- [R10] On board of the elevator there is a special Stop button, which when pressed will stop the elevator's movement. It will not be possible to open the doors when the elevator is stopped in between the floors.

8.3.2 Temporal Constraints for the Elevator System

The elevator is also required to satisfy the following timing constraints:

- [T1] <Open Floor Door> After the elevator has stopped at a particular floor, the elevator's door will open no sooner than OPEN_MIN_TIME seconds and no later than OPEN_MAX_TIME. Practical values for these constants can be, for instance, 1.0 seconds and, respectively, 3.0 seconds;
- [T2] <Stay Open Floor Door> After the elevator has stopped at a given floor the elevator's door will normally stay open for a STAY_OPEN_NORMAL_TIME period of time (e.g., 12.0 seconds). However, if the Close Door button on board of the elevator is pressed before this timeout expires, the door will close but no sooner than STAY_OPEN_MIN_TIME (e.g., 2.0 seconds);
- [T3] <Resume Elevator Movement> After the door is closed, the movement of the elevator can resume, but no sooner than CLOSE_MIN_TIME seconds and no later than CLOSE_MAX_TIME seconds (possible values can be, for instance, 1.0 seconds and, respectively, 3.0 seconds);

- [T04] <Elevator Speed Constraints> The movement of the elevator between destination floors should be continuously monitored, and a minimum and maximum speed limits should be considered. Two preset values, `SPEED_LIMIT_LOW` and `SPEED_LIMIT_HIGH` will serve the detection of abnormal moving conditions (too slow or too fast). In such cases, the elevator will be stopped immediately and an alarm signal will be issued. Practical values for the above constraints can be expressed in seconds per floor, for instance the lower limit can be 5.0 seconds/floor and the higher limit can be 3.0 seconds/floor (during continuous movement). It can be considered that floor sensors are available to detect the presence of the elevator by any given floor;
- [T5] <Stop Request> If the Stop button on board the elevator is pressed, the moving elevator will stop as soon as possible, in any case no later than `STOP_MAX_TIME` seconds (e.g., 2.0 seconds). The floor doors will not open if the elevator is not positioned at a floor, and the elevator will remain in this state of emergency stop for `STAY_STOPPED_TIME` seconds (e.g., 20.0 seconds) unless the Stop button will restart the above timeout from zero. Before the elevator resumes its movement, the Stop button will be illuminated for a sequence of several consecutive visual signals (timing requirements for both audio and visual signals are specified by [T7]);
- [T6] <Alarm Triggered> If the Alarm button inside the elevator is pressed, then the elevator will stop immediately according to the timing condition `STOP_MAX_TIME` (from T5) and a continuous, highly audible alarm signal will be issued (T7 gives details on timing characteristics of these signals). In contrast to T5, the elevator will not resume its movement after `STAY_STOPPED_TIME` seconds, and will stay stopped until authorization for moving is given by a designated staff member and the alarm system is set-off;
- [T7] <Signal Timing> The audio alarm will consists of a sequence signals, each of a duration no less than `MIN_SIGNAL_DUR` (e.g., 1.0 seconds) and no greater than `MAX_SIGNAL_DUR` (e.g., 2.0 seconds). The separation between signals, `SIGNAL_SEPARATION`, should be preset to a given value, e.g. 1.0 seconds. The same constants can be used in the case of the visual signals mentioned by [T5];

8.3.3 Coverage of Dasarathy Constraints by the Elevator's Timing Requirements

The timing constraints T1-T8 placed on the Elevator System's behaviour are intended to illustrate the way our proposed approach deals with a variety of temporal requirements. Since, as indicated in Section 5.2, Dasarathy's classification of timing restrictions offers a reference basis for such requirements, it is useful to notice that eight out of nine classes presented in Subsection 5.2.1 are covered in our case study. The correspondence between the elevator's timing constraints [TC1]-[TC7] and the corresponding Dasarathy classes of temporal constraints [DC1]-[DC9] to which they can be related is given in Table 8.I, and shows that all DC classes except [DC4], which is a constraint placed on external stimuli, are covered by at least one of the elevator timing requirements TC. This table, together with the solution of the problem presented in the remaining of this chapter, demonstrates that our proposed approach can deal with a large variety of timing restrictions placed on TCS.

Table 8.I Correspondence between ELS's Timing Requirements and Dasarathy's Constraints

Elevator Timing Constraint	Corresponding Dasarathy Constraints
T1 <Open Floor Door>	DC7 (MaxRR), and DC8 (MinRR)
T2 <Stay Open Floor Door>	DC6 (MinSR), DC7 (MaxRR), and DC8 (MinRR)
T3 <Resume Elevator Movement>	DC7 (MaxRR) and DC8 (MinRR)
T4 <Elevator Speed Constraints>	DC1 (MaxSS) and DC2 (MinSS)
T5 <Stop Request>	DC3 (MaxRS) and DC5 (MaxSR)
T6 <Alarm Triggered>	DC5 (MaxSR)
T7 <Beep Timing>	DC9 (Duration)

8.4 The Modelling Solution

The steps of the “regular” flow of modelling activities described in Chapter 7 are illustrated below using the Elevator case study. The role of each modelling step is highlighted and examples of artefacts obtained in each step are given.

8.4.1 Definition of Use Cases

A single use case diagram is sufficient to describe the externally visible behaviour of the elevator system, as shown in Fig. 8.1. Within this diagram, two use cases are considered, Inside Request and Outside Request, and there are only two actors that interact with the systems, the User, a person that issues a command to the elevator, and the Elevator itself.

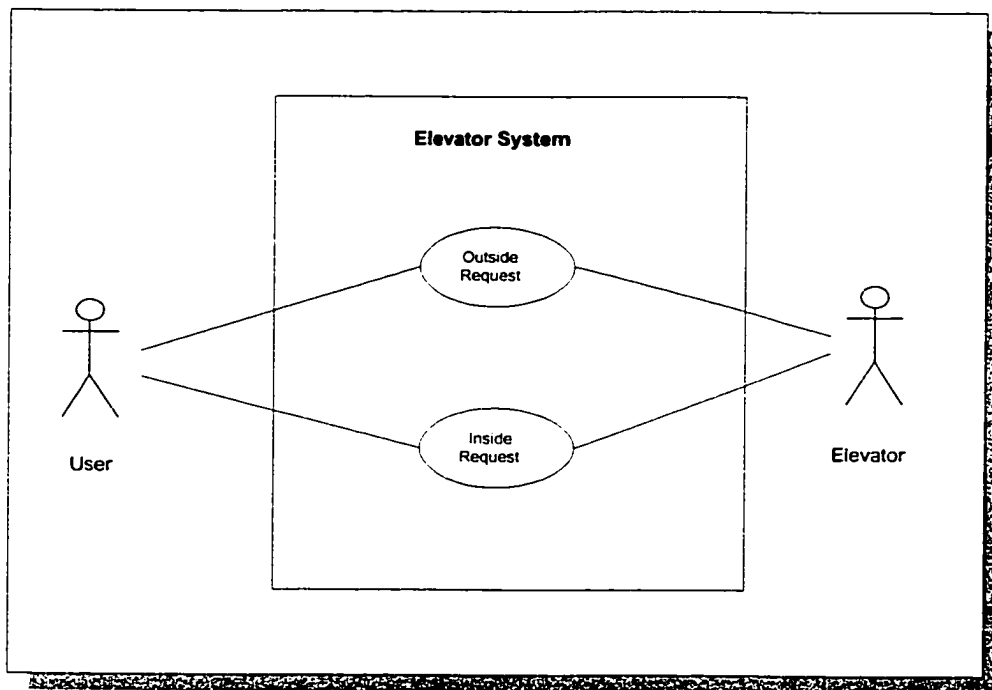


Fig. 8.1 ELS Use Case Diagram

8.4.2 Elaboration of Scenarios

The two use cases represented in Fig. 8.1 are next detailed through the use of scenarios, four such scenarios being presented in Fig. 8.2 to 8.5. As mentioned in Chapter 7, the scenarios can be described in various ways, one of which being to use application-tailored visual descriptions. In our case, a number of graphical symbols are used, as indicated in the legend attached to the figures, allowing a more elaborate description of the system's externally visible behaviour.

While developing the scenarios it has been observed that there is not a clear cut line between the two use cases considered initially, in real-life situations combinations of internal and external requests being issued for the elevator's service. For this reason, the scenarios that follow are "attached" to the two use cases considered in Fig. 8.1 based on the type of the first issued request shown in the scenario. For illustration purposes only a segment of the building in which the elevator operates is considered (levels 2 to 6), sufficient however to describe the most important aspects of the elevator's operation.

While developing the scenarios, a number of rules regarding the functioning of the elevator have been established. To describe them, the notions of *direction-changing* and *direction-keeping requests* need be introduced. While a direction-keeping request is simply not a direction-changing request, the latter can be either an internal request (for instance, someone presses the car button number 4 while the elevator is at floor 6 and moving up) or an external request (for instance when the elevator is at floor 3 and moving down a request is issued at floor 5, no matter for what direction). Using these two terms, the "rules of the elevator" can be formulated as:

Rule #1: "Maintain direction as long as possible". This rule means basically that if more service requests exist, the elevator will serve first the direction-keeping requests. If, at a given time, there are only direction-changing requests, than the stipulations of Rule#3 below have to be followed;

Rule#2: “When maintaining the direction, go to the closest floor from which a direction-keeping request has been issued”;

Rule #3: “If direction has to be changed, change direction and then (a) try to apply rules Rules #1 and #2 or (b) if this is not applicable, go to the farthest floor from which a direction-changing request has been issued.” This rule prevents the situation of an “infinite loop” in the elevator’s traveling, as described in the scenario shown in Fig. 8.5.

The first scenario, presented in Fig.8.2, is a normal instance of the Outside Request use case. Specifically, while the elevator is waiting at floor 6, a request from floor 3 is issued for movement up to floor 5. No other requests are issued while the elevator services this request. The basic behaviour of the elevator is captured in this scenario.

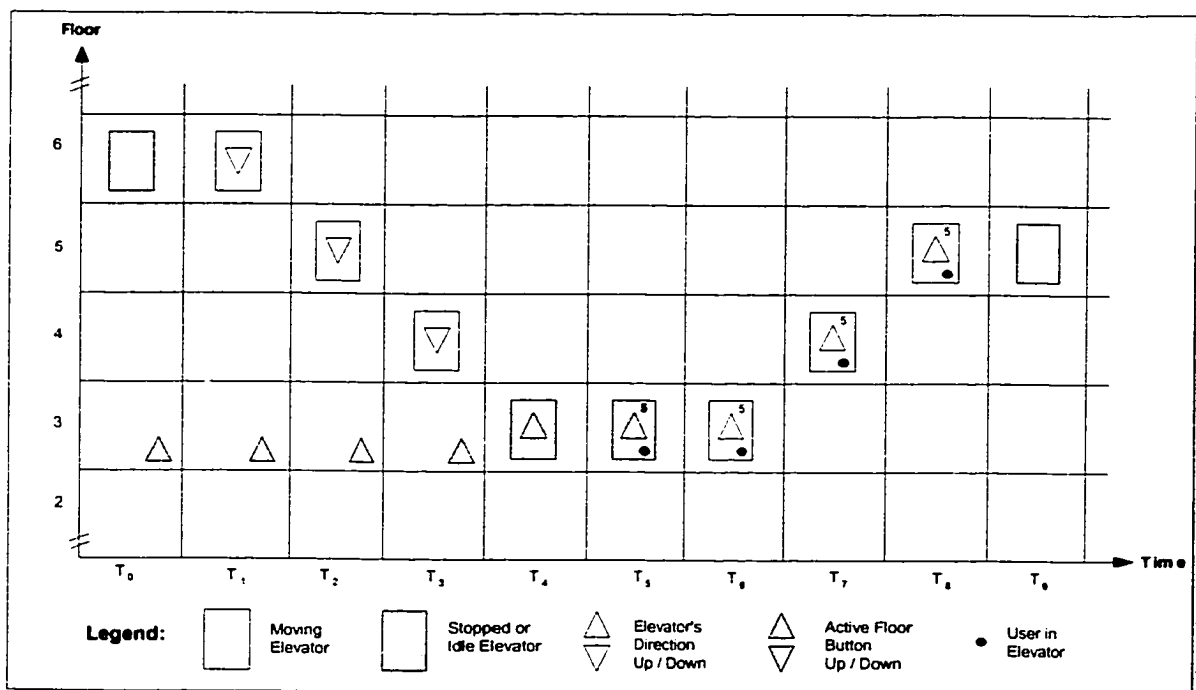


Fig. 8.2 ELS Scenario: Outside Request A

The second scenario, shown in Fig. 8.3, is another instance of the Outside Request use case. Its can be described summarily as “Sorry, but I changed my mind!,” because in it the issuer of the request from floor 3 is no longer taking the elevator after it arrives at the floor. The elevator, not having any other internal or external requests to serve, becomes idle at floor 3.

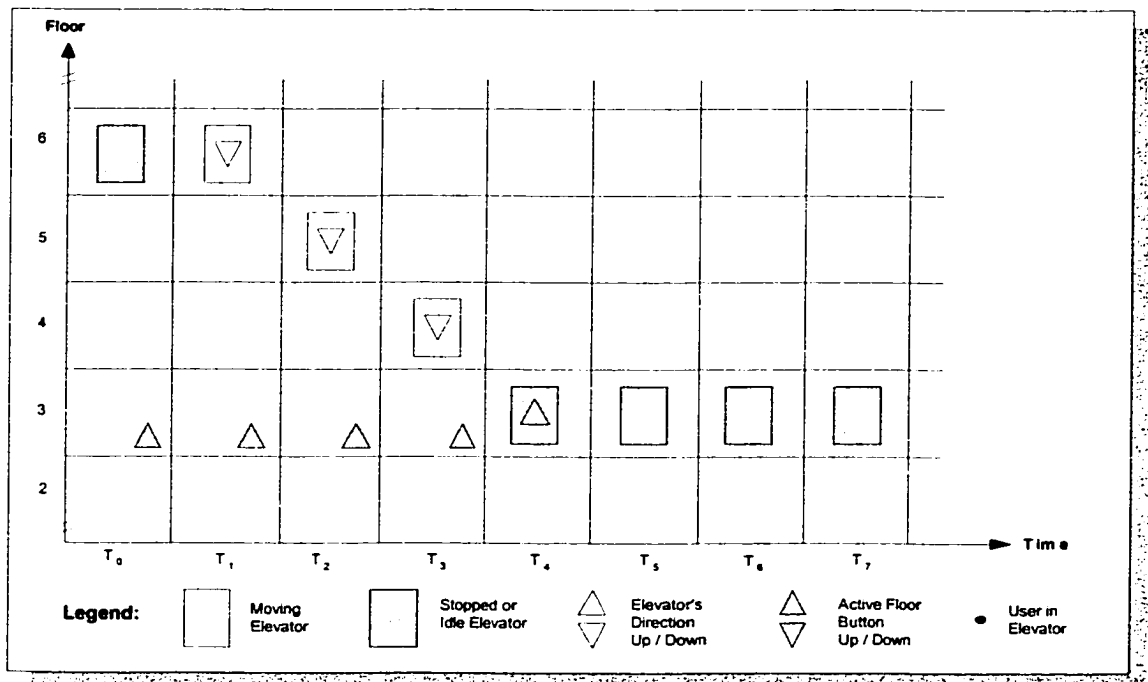


Fig. 8.3 ELS Scenario: Outside Request B

The third scenario, presented in Fig. 8.4, also an instance of the Outside Request use case, describes a situation in which two requests are issued at the same floor for different directions. By applying Rules #1 and #2 of the elevator, the Up request at floor 4 is served before the Down request at the same floor, although the latter was issued first.

The fourth scenario, depicted in Fig. 8.5, is an instance of the Inside Request use case that serves for illustrating the solution for a situation in which a user attempts to use the elevator in a rather mischievous way. The scenario, which can be denoted “You cannot be the exclusive user of the elevator!” shows that a user that repeatedly tries to travel between floors

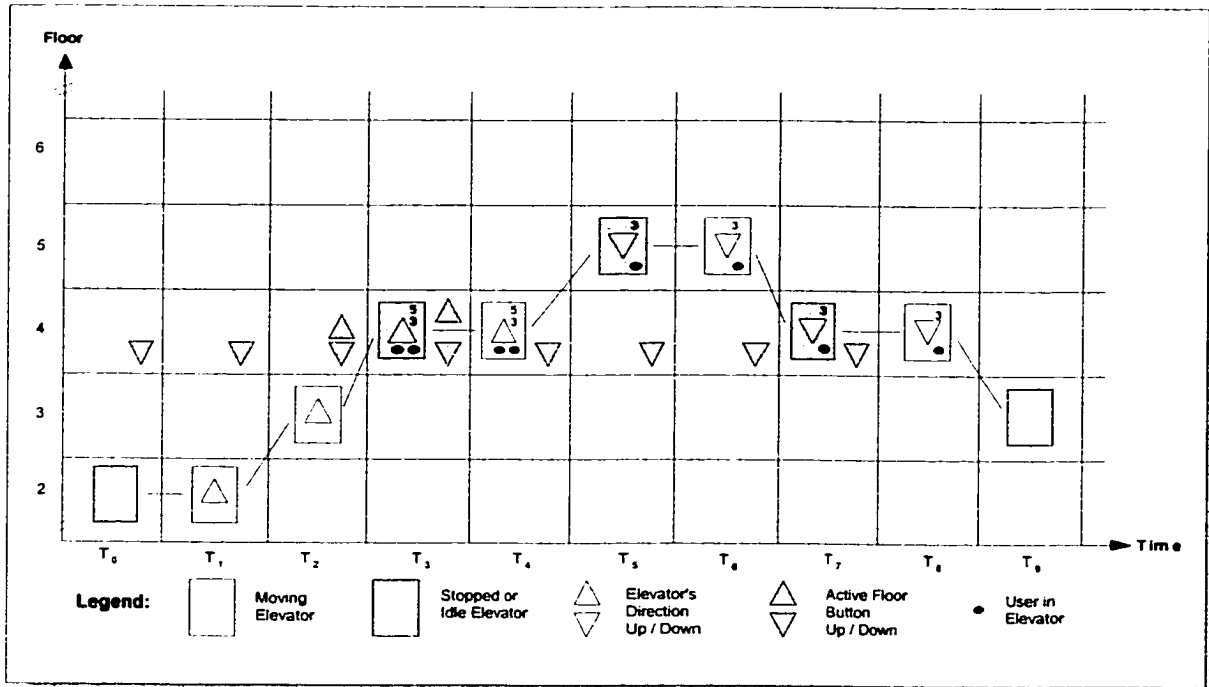


Fig. 8.4 ELS Scenario: Outside Request C

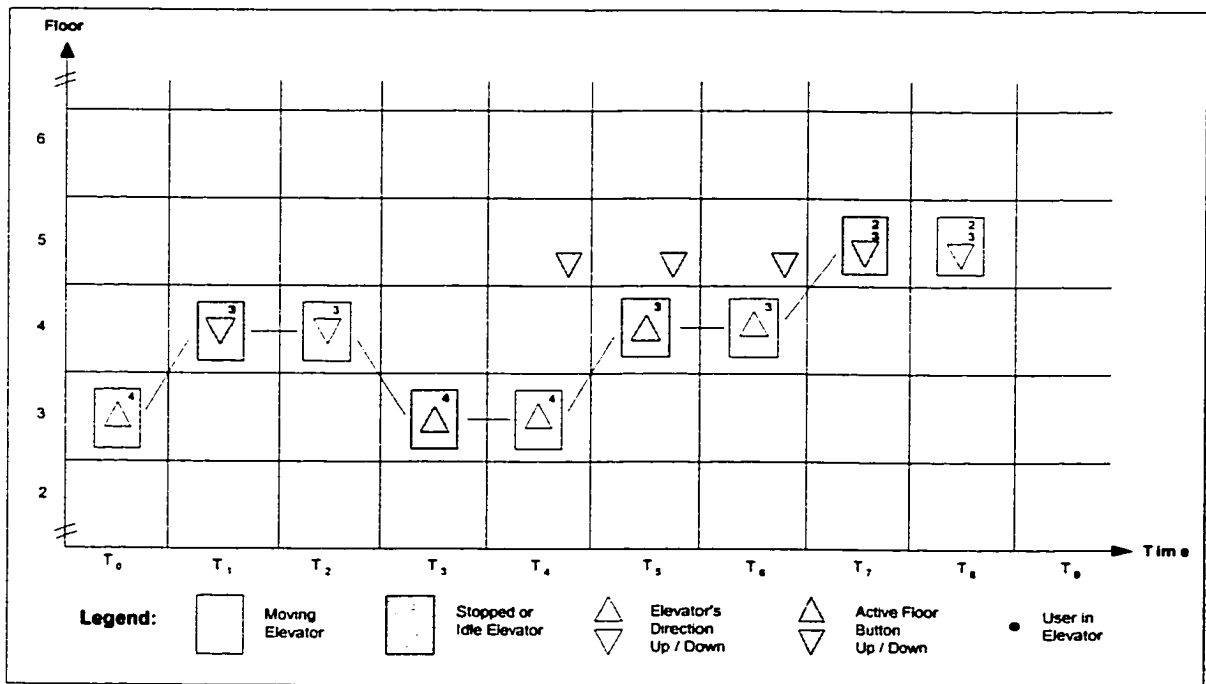


Fig. 8.5 ELS Scenario: Inside Request A

3 and 4 (by pressing car button number 3 when moving from floor 3 to 4 and car button 4 when moving from floor 4 to 3) is interrupted in this action when another request is issued at floor 5. This solution is made possible by Rule #3b.

8.4.3 Construction of the Class Diagram

The scenarios shown previously serve not only for establishing a set of rules for the intended behaviour of the elevator, but they also help the construction of the system's a class structure. The class diagram resulted from the information gathered while developing scenarios, as well from the inspection of the problem's requirements is shown in Fig. 8.6. This diagram is given only in terms of component classes and relationships between classes, and provides no details about the contents of the classes (attributes and operations are not specified yet).

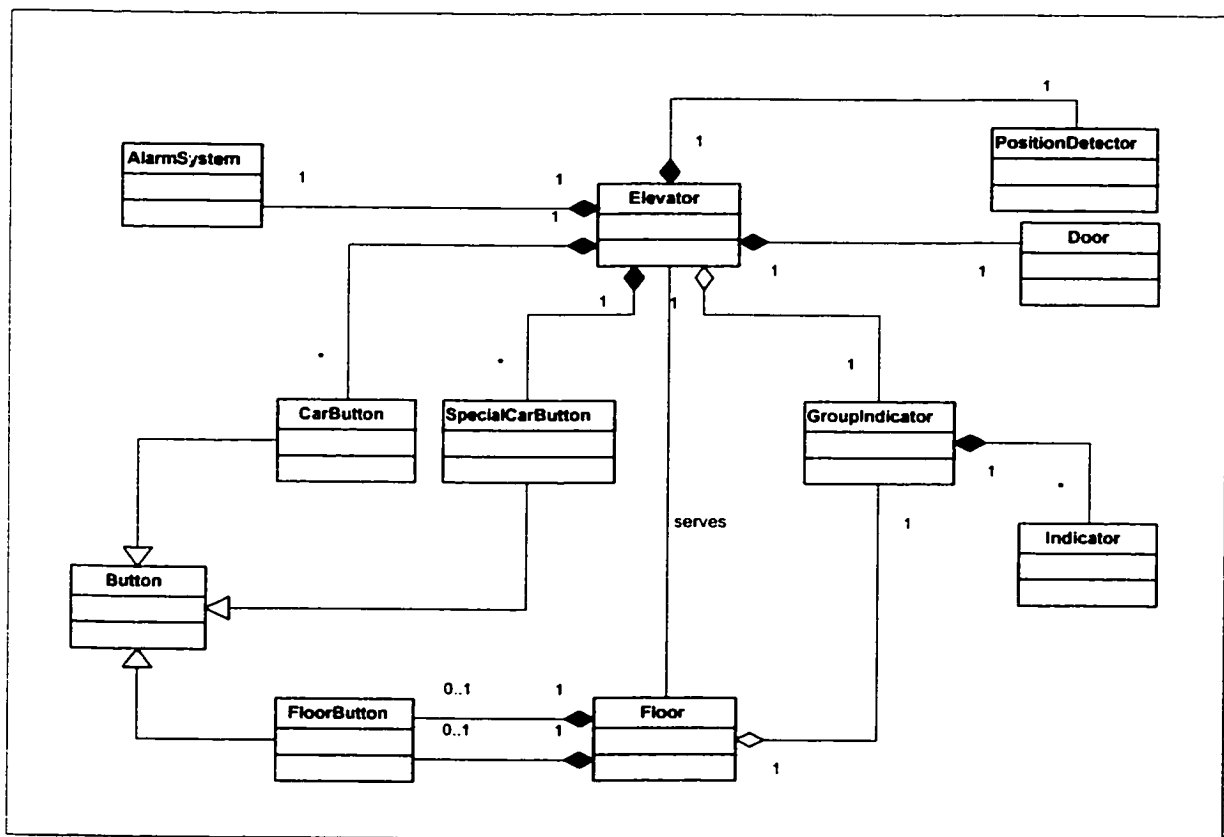


Fig. 8.6 ELS Class Diagram: Initial Structure

In this class diagram the obvious components of the elevator system are represented: the elevator itself, the floors, the buttons, and the indicators. In order to distribute the functionality of the elevator, the classes *AlarmSystem*, responsible with issuing audio and visual signals in exceptional situations, and *PositionDetector*, intended to take care of monitoring the position and the speed of the elevator, are also included. Since conceptually both the floor indicators and the indicators present in the elevator have always the same state (they show the same thing, the floor at which the elevator is currently at), a single class *GroupIndicator* has been introduced. In addition, buttons are modelled by several classes, based on their specific use (regular car buttons, used for accepting internal requests from the user, special car buttons such as *Alarm*, *Stop*, *Open Door* and *Close Door*, and floor buttons, which indicate the direction of external requests). Only a class *Floor* has been included in the diagram to keep the graphical representation simple, although two classes *TopFloor* and *BottomFloor* from which a *MiddleFloor* inherits would more accurately describe the floors in an object-oriented way (see [Dong97] solution in this respect).

8.4.4 Specification of Sequence Diagrams

Additional insight into the elevator system is obtained by developing sequence diagrams. In particular, while trying to assign responsibilities to the objects of the classes the internal behaviour of the system becomes more clear. Fig. 8.7 shows a sequence diagram that corresponds to the scenario depicted in Fig. 8.2 (Outside Request A Scenario). In fact, the sequence diagram describes only half of this scenario, yet the analysis of the elevator's behaviour indicates that the key design principles of the Elevator System are captured in this diagram. Specifically, the diagram describes both the situation in which the elevator is checking its requests at a floor ("idle" or "stopped"), and the situation in which the elevator is moving towards a destination floor ("target floor").

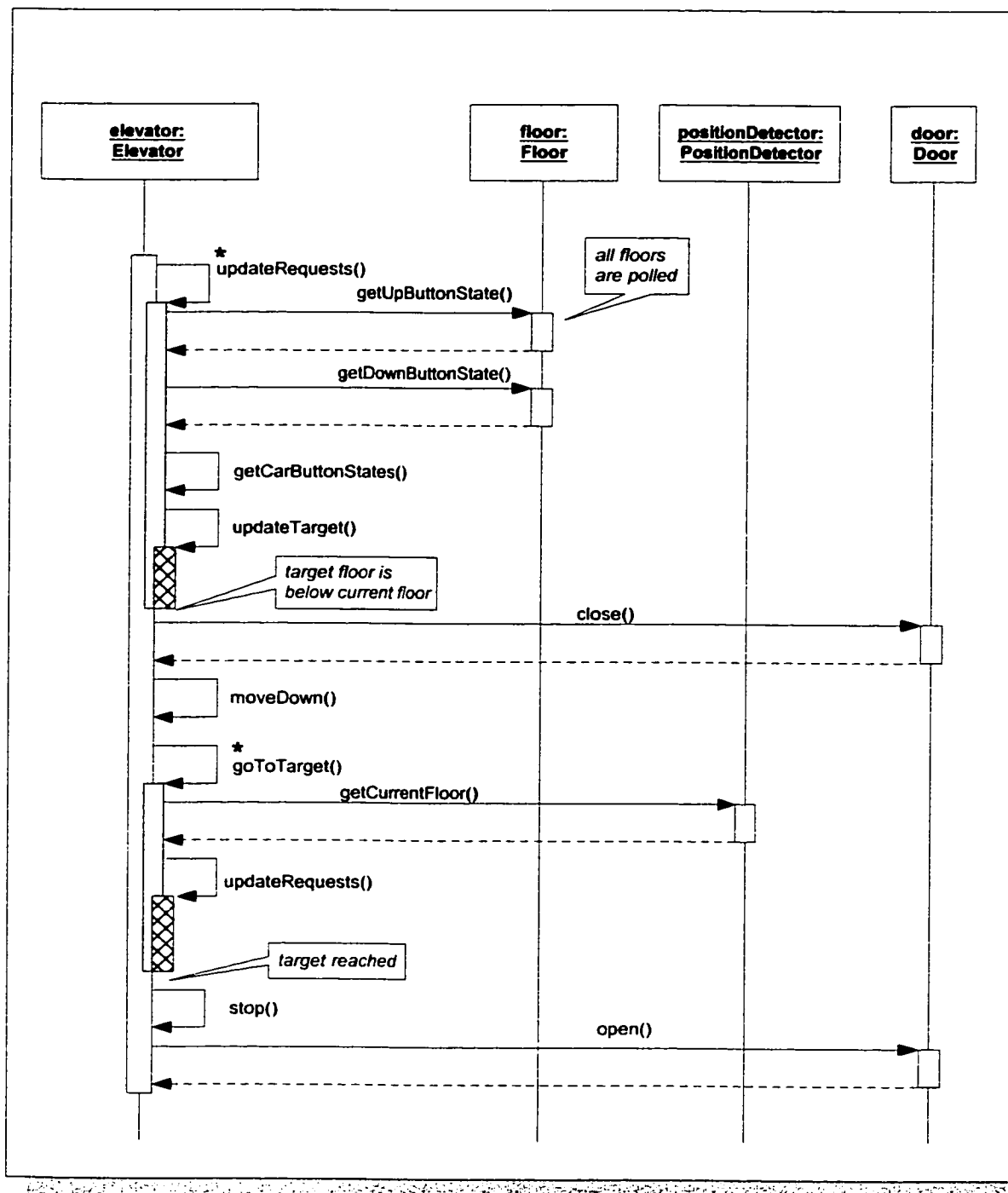


Fig. 8.7 ELS Sequence Diagram: Outside Request A

The design solution considered, apparent in this sequence diagram, is to continuously poll both the floor buttons and the internal car buttons in order to permanently maintain a list of existing requests (the repeated operation `updateRequests` in the diagram). When the need for a movement is detected by the idle (or stopped) elevator, the target floor is set (`updateTarget` operation) and the elevator starts moving but not before closing its door (to be precise, if the elevator is idle, the door is already closed). While moving towards the target the list of requests need be continuously updated (within the `goToTarget` operation, which is labelled “repeated” to indicate the repetitive nature of its major two components: `getCurrentFloor` and `UpdateRequests`, the latter performing the same task as in the idle or stopped situations). The sequence diagram also stresses the central role the Elevator class has in the system.

8.4.5 Elaboration of Class Compounds

With a better insight into the system’s structure and behaviour, the class compounds can be next detailed. Two class compounds are described in this Subsection in terms of both class specification (CLS) and state diagram associated with the class (CLSTD). One class is very simple (the Button class), while the second is quite complex, bearing the responsibility of much of the work done by the system (the Elevator class). The Button class compound, with its two components shown in Fig. 8.8 (the class specification) and 8.9 (the state diagram) is included here because in this particular application other classes present in the class diagram have a behaviour similar to that of the Button (these classes are Indicator and Door).

The Elevator class includes the operations deemed necessary in the sequence diagram drawn in the previous modelling step and has also its attributes specified. These attributes include the state descriptor for the objects of the class (the state attribute), an attribute for denoting the elevator’s current direction, another for keeping information about the elevator’s current floor, and three attributes for the groups of possible requests (internal requests, external requests for up movement, and external requests for down movement).

The CLSTD of the Elevator class compound, shown in Fig. 8.11, is constructed based on the following states:

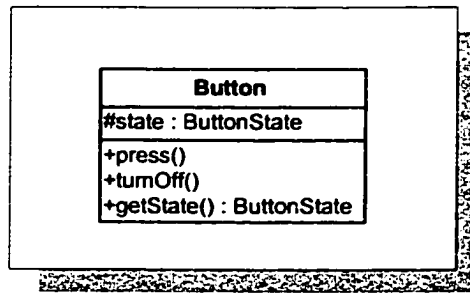


Fig. 8.8 ELS Class Button

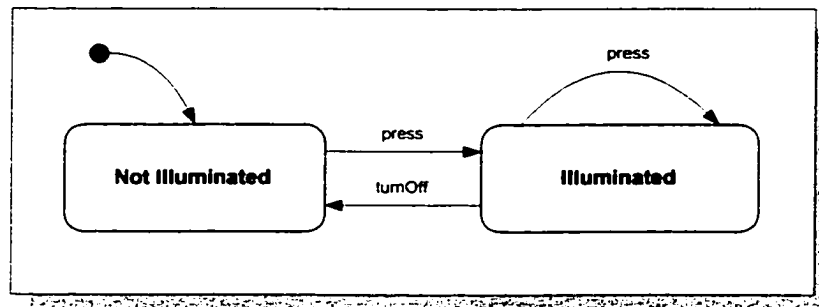


Fig. 8.9 ELS State Diagram for the Button Class

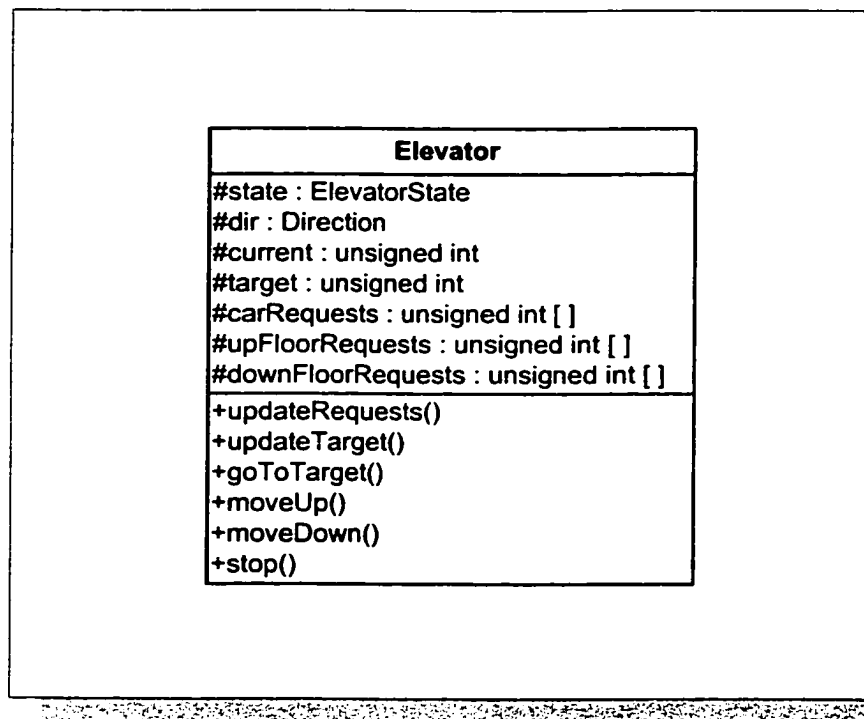


Fig. 8.10 ELS Class Elevator

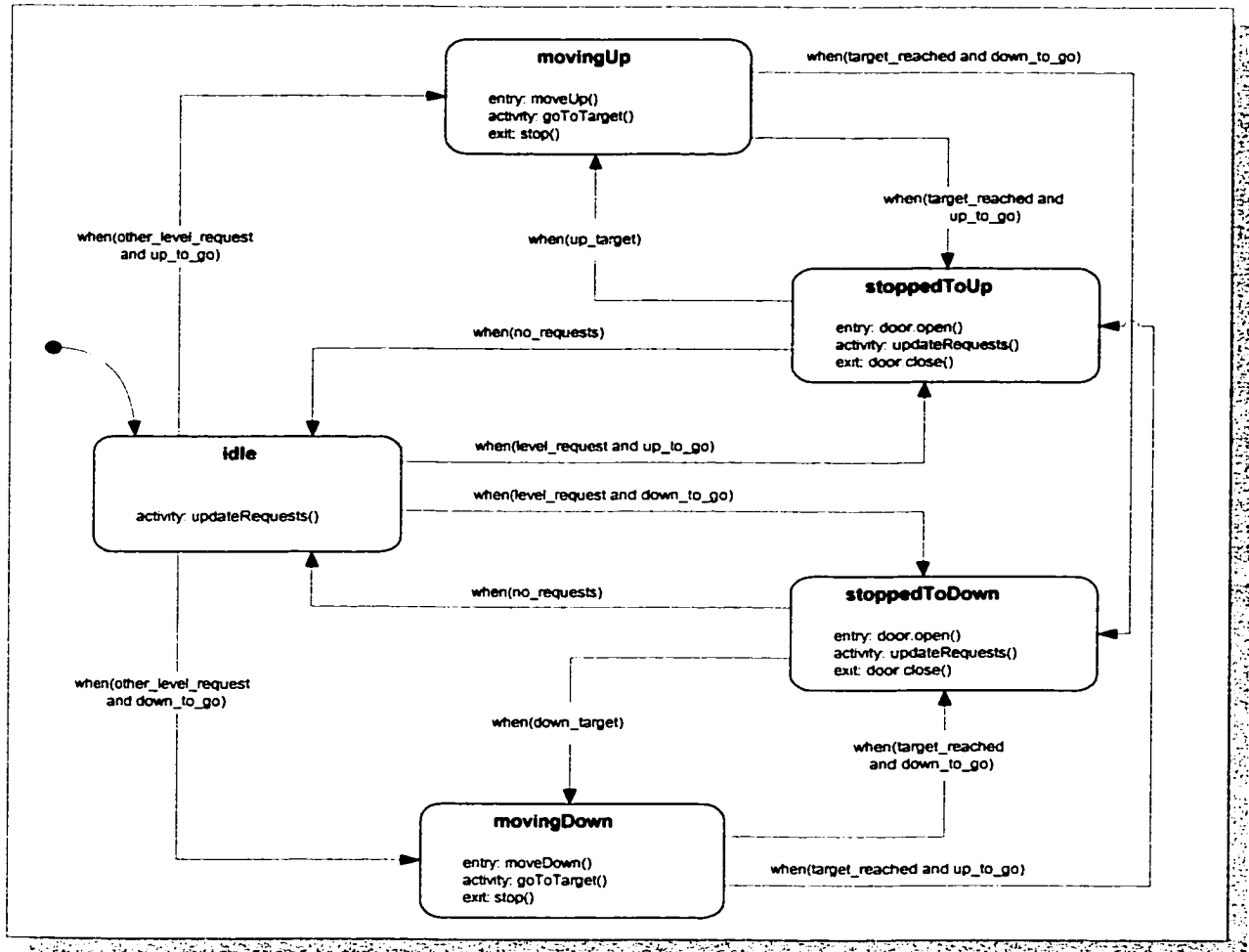


Fig. 8.11 ELS State Diagram for the Elevator Class

- idle, describing the situation in which the elevator has no requests, neither internal nor external (to be more precise, the elevator enters this state after stopping at a floor, waiting for a period of time, and still not having requests);
- movingUp, which is the state of the elevator moving upwards to its current target floor;
- movingDown, same as above, but for the opposite direction;
- stoppedToUp, which denotes the state in which the elevator is stopped at a floor and either (a) has pending requests, the analysis of which indicating that the next movement of the elevator is an up movement, or (b) has no pending requests but it

has just completed its last service coming from a lower floor and has not yet entered yet the idle state;

- stoppedToDown, same as above, but with either (a) a down “next direction to take,” or (b) a last moving direction “down.”

The Alarm and Stop Elevator situations (triggered by special inside car requests) correspond to two abnormal states of the elevator that for simplicity have been omitted from the diagram. In the diagram, specific change conditions lead to the transition of the elevator from state to state. Detailing of these conditions is best done in Z++, as shown later in the chapter.

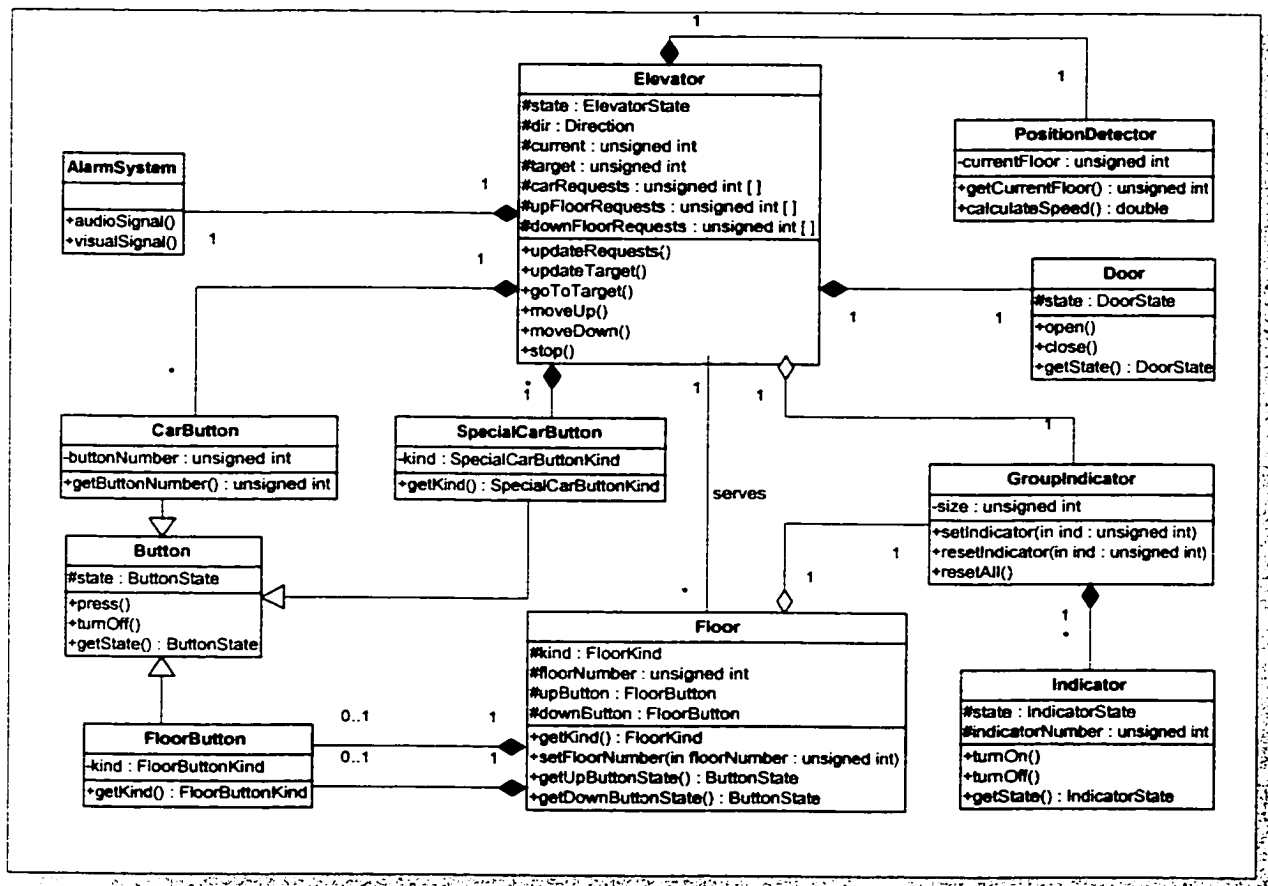


Fig. 8.12 ELS Class Diagram with Attributes and Operations Attached to Classes

With all the classes of the initial class diagram shown in Fig. 8.6 detailed in terms of attributes and operations the resulting class diagram is the one presented in Fig. 8.12.

8.4.6 Formalisation through the AFCD and the AFSD

Having the classes of the class diagram specified in detail in terms of attributes and operations and having the state diagram also specified for some classes, it is possible to apply the automated translation processes from UML to Z++ described in Chapter 6.

First, the class diagram is translated to Z++ by applying the AFCD algorithm detailed in Section 6.3, the result being shown in Fig. 8.13 (the text file generated by the Java program presented in Appendix B has been manually edited with Z specific symbols, the generation of such symbols directly from the program being one of the intended near future enhancements of the AFCD's implementation).

Next, the state diagrams associated with the classes are formalised via the AFSD algorithm presented in Section 6.4. In the case of the Button class the result, shown in Fig. 8.14, reflects the simplicity of the state diagram (it has been included here primarily for showing the groups of predicates generated in the HISTORY clause of the Z++ class), while in the case of the Elevator class presented in Fig. 8.15 it reflects the complexity of both the class' structure and behaviour. Because the Elevator state diagram is specified using transitions triggered by changed events, numerous internal operations have been created. The quite arid nature of these operations (that need be further processed by the human specifier, at least in what regards their proper renaming) has prompted us to add comments for them in the Elevator Z++ class.

In both the case of the complete Z++ specification shown in Fig. 8.13 and of the detailed Z++ class Elevator shown in Fig. 8.15, the intervention of the human formaliser after the application of the AFCD and AFSD algorithms is necessary, as described in more detail in the next Subsection.

```
[ELEVATORSTATE, DOORSTATE, INDICATORSTATE, SPECIALCARBUTTONKIND,
BUTTONSTATE, FLOORBUTTONKIND, FLOORKIND, DIRECTION]
```

```
-----
CLASS System
PUBLICS
TYPES
FUNCTIONS
OWNS
    theServesDescriptor : ServesDescriptor;
RETURNS
OPERATIONS
INVARIANT
ACTIONS
HISTORY
END CLASS
// -----
CLASS Elevator
PUBLICS
    setTarget, updateRequests, moveUp, moveDown, stop, alarm
TYPES
FUNCTIONS
OWNS
    state : ELEVATORSTATE;
    dir : DIRECTION;
    current : N;
    target : N;
    carRequests : seq(N);
    upFloorRequests : seq(N);
    downFloorRequests : seq(N);
    door : Door;
    carButtons : PCarButton;
    specialCarButtons : PSpecialCarButton;
    groupIndicator : GroupIndicator;
    positionDetector : PositionDetector;
    alarmSystem : AlarmSystem;
RETURNS
OPERATIONS
    updateRequests : → ;
    updateTarget : → ;
    goToTarget : → ;
    moveUp : → ;
    moveDown : → ;
    stop : → ;
INVARIANT
ACTIONS
    setTarget ==> ;
    updateRequests ==> ;
    moveUp ==> ;
    moveDown ==> ;
    stop ==> ;
    alarm ==> ;
HISTORY
END CLASS
```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD

```

// -----
CLASS AlarmSystem
PUBLICS
    audioSignal, visualSignal
TYPES
FUNCTIONS
OWNS
RETURNS
OPERATIONS
    audioSignal : → ;
    visualSignal : → ;
INVARIANT
ACTIONS
    audioSignal ==> ;
    visualSignal ==> ;
HISTORY
END CLASS
// -----
CLASS PositionDetector
PUBLICS
    getCurrentFloor, calculateSpeed
TYPES
FUNCTIONS
OWNS
    currentFloor : N;
RETURNS
OPERATIONS
    getCurrentFloor : → N;
    calculateSpeed : → R;
INVARIANT
ACTIONS
    getCurrentFloor result! ==> ;
    calculateSpeed result! ==> ;
HISTORY
END CLASS
// -----
CLASS Door
PUBLICS
    open, close, getState
TYPES
FUNCTIONS
OWNS
    state : DOORSTATE;
RETURNS
OPERATIONS
    open : → ;
    close : → ;
    getState : → DOORSTATE;
INVARIANT
ACTIONS
    open ==> ;
    close ==> ;
    getState result! ==> ;
HISTORY
END CLASS

```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```

// -----
CLASS CarButton
PUBLICS
    getButtonNumber
TYPES
FUNCTIONS
OWNS
    buttonNumber : N;
RETURNS
OPERATIONS
    getButtonNumber : → N;
INVARIANT
ACTIONS
    getButtonNumber result! ==> ;
HISTORY
END CLASS
// -----
CLASS SpecialCarButton
PUBLICS
    getKind
TYPES
FUNCTIONS
OWNS
    kind : SPECIALCARBUTTONKIND;
RETURNS
OPERATIONS
    getKind : → SPECIALCARBUTTONKIND;
INVARIANT
ACTIONS
    getKind result! ==>
HISTORY
END CLASS
// -----
CLASS Button
PUBLICS
    press, turnOff, getStatus
TYPES
FUNCTIONS
OWNS
    state : BUTTONSTATE;
RETURNS
OPERATIONS
    press : → ;
    turnOff : → ;
    getStatus : → BUTTONSTATE;
INVARIANT
ACTIONS
    press ==> ;
    turnOff ==> ;
    getStatus result! ==> ;
HISTORY
END CLASS

```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)


```

// -----
CLASS FloorButton
PUBLICS
    getKind
TYPES
FUNCTIONS
OWNS
    kind : FLOORBUTTONKIND;
RETURNS
OPERATIONS
    getKind : → FLOORBUTTONKIND;
INVARIANT
ACTIONS
    getKind result! ==> ;
HISTORY
END CLASS
// -----
CLASS Floor
PUBLICS
    getKind, setFloorNumber, getUpButtonState, getDownButtonState
TYPES
FUNCTIONS
OWNS
    kind : FLOORKIND
    floorNumber : N;
    upButton : FloorButton;
    downButton : FloorButton;
    groupIndicator : GroupIndicator;
RETURNS
OPERATIONS
    getKind : → FLOORKIND;
    setFloorNumber : N → ;
    getUpButtonState : → BUTTONSTATE;
    getDownButtonState : → BUTTONSTATE;
INVARIANT
ACTIONS
    getKind result! ==> ;
    setFloorNumber floorNumber? ==> ;
    getUpButtonState result! ==> ;
    getDownButtonState result! ==> ;
HISTORY
END CLASS
// -----
CLASS GroupIndicator
PUBLICS
    setIndicator, resetIndicator, resetAll
TYPES
FUNCTIONS
OWNS
    size : N;
    indicators : PIndicator;
RETURNS
OPERATIONS
    setIndicator : N → N;

```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```

    resetIndicator : N → ;
    resetAll : → ;
INVARIANT
ACTIONS
    setIndicator ind? result! ==> ;
    resetIndicator floorNumber? ==> ;
    resetAll ==>
HISTORY
END CLASS
// -----
CLASS Indicator
PUBLICS
    turnOn, turnOff, getState
TYPES
FUNCTIONS
OWNS
    state : INDICATORSTATE;
    indicatorNumber : N;
RETURNS
OPERATIONS
    turnOn : → ;
    turnOff : → ;
    getState : → INDICATORSTATE;
INVARIANT
ACTIONS
    turnOn ==> ;
    turnOff ==> ;
    getState result! ==> ;
HISTORY
END CLASS
// -----
CLASS ServesDescriptor
PUBLICS
TYPES
FUNCTIONS
OWNS
    instancesofElevator : PElevator;
    instancesofFloor : PFloor;
    servesInstances : Floor → Elevator;
RETURNS
OPERATIONS
INVARIANT
    dom servesInstances = instancesofFloor
    ran servesInstances = instancesofElevator
ACTIONS
HISTORY
END CLASS
HPositionDetector ≐ PositionDetector \ [currentFloor]
HDoor ≐ Door \ [state]
HCarButton ≐ CarButton \ [buttonNumber]
HSpecialCarButton ≐ SpecialCarButton \ [kind]
HFloorButton ≐ FloorButton \ [kind]
HGroupIndicator ≐ GroupIndicator \ [size]
HIndicator ≐ Indicator \ [state, indicatorNumber]

```

Fig. 8.13 ELS Z++ Specification Generated by the AFCD (continued from the previous page)

```

CLASS Button
PUBLICS

    press, turnOff

TYPES

    ButtonState ::= notilluminated | illuminated

FUNCTIONS
OWNS

    state : ButtonState

RETURNS
OPERATIONS

    press: → ;
    turnOff: → ;

INVARIANT
ACTIONS

    init ==> state' = notilluminated;
    press ==> state' = illuminated;
    turnOff ==> state' = notilluminated;

HISTORY

    // mutual exclusion properties
    mutex({press, turnOff}) ^ self_mutex ({press, turnOff}) ^

    // permission predicates
    □(press ⇒ state = notilluminated ∨ state = illuminated) ^
    □(turnOff ⇒ state = illuminated) ^

    // definition of transition effects
    □(press ⇒ O(state = illuminated)) ^
    □(turnOff ⇒ O(state = notilluminated)) ^

    // reachability properties
    □(state = notilluminated ⇒ press) ^
    (state = illuminated ⇒ press ∨ turnOff)

END CLASS

```

Fig. 8.14 ELS Z++ Class Button Updated by the AFSD

```

CLASS Elevator
PUBLICS

    press, turnOff

TYPES

    ElevatorState ::= idle | movingup | movingdown | stoppedtoup |
                    stoppedtodown

FUNCTIONS
OWNS

    state : ElevatorState;
    dir : Direction;
    current : N;
    target : N;
    carRequests : seq(N);
    upFloorRequests : seq(N);
    downFloorRequests : seq(N);
    door : Door;
    carButtons : PCarButton;
    specialCarButtons : PSpecialCarButton;
    groupIndicator : GroupIndicator;
    positionDetector : PositionDetector;
    alarmSystem : AlarmSystem;

RETURNS
OPERATIONS

    updateRequests : → ;
    updateTarget : → ;
    goToTarget : → ;
    moveUp : → ;
    moveDown : → ;
    stop : → ;
    *τ1 : → ;           // condition triggered operations
    *τ2 : → ;           // describing the transitions of
    *τ3 : → ;           // the Elevator state diagram
    *τ4 : → ;           // shown in Fig. 8.12
    *τ5 : → ;
    *τ6 : → ;
    *τ7 : → ;
    *τ8 : → ;
    *τ9 : → ;

```

Fig. 8.15 ELS Z++ Class Elevator Updated by the AFSD (continued from previous page)

**INVARIANT
ACTIONS**

```
// [...]
// definitions of Elevator operations as in Fig.8.12

*τ1 ==> state' = movingup;           // idle to movingup
*τ2 ==> state' = movingdown;        // idle to movingdown
*τ3 ==> stop;                        // movingup or movingdown to stoppedtoup
      state' = stoppedtoup;
*τ4 ==> stop;                        // movingup/movingdown to stoppedtodown
      state' = stoppedtodown;
*τ5 ==> door.close;                 // stoppedtoup to movingup
      state' = movingup;
*τ6 ==> door.close;                 // stoppedtodown to movingdown
      state' = movingdown;
*τ7 ==> state' = stoppedtoup;        // idle to stoppedtoup
*τ8 ==> state' = stoppedtodown;      // idle to stoppedtodown
*τ9 ==> state' = idle;               // stoppedtoup/stoppedtodown to idle
```

HISTORY

```
// mutual exclusion properties, permission predicates, definition
// of transition effects and reachability properties omitted for
// simplicity (examples are available in Fig. 6.30 and 8.14)

// enabling conditions:

(enabled(τ1) ≡ (state = idle) ∧ other_level_request_and_up_to_go) ∧
(enabled(τ2) ≡ (state = idle) ∧ other_level_request_and_down_to_go) ∧
(enabled(τ3) ≡ (state = movingup ∨ state = movingdown) ∧
              target_reached_and_up_to_go) ∧
(enabled(τ4) ≡ (state = movingup ∨ state = movingdown) ∧
              target_reached_and_down_to_go) ∧
(enabled(τ5) ≡ (state = stoppedtoup) ∧ up_target) ∧
(enabled(τ6) ≡ (state = stoppedtodown) ∧ down_target) ∧
(enabled(τ7) ≡ (state = idle) ∧ level_request and up_to_go) ∧
(enabled(τ8) ≡ (state = idle) ∧ level_request and down_to_go) ∧
(enabled(τ9) ≡ (state = stoppedtodown ∨ state = stoppedtoup) ∧ norequests
```

END CLASS

Fig. 8.15 ELS Z++ Class Elevator Updated by the AFSD (continued from the previous page)

8.4.7 Enhancement of the Formal Specification

After the automated translation from UML to Z++ takes place, the results obtained by applying the AFC D and the AFSD must be checked since modifications and additions may be necessary. In the case of the ELS Z++ specification example shown in Fig. 8.13, it can be observed that the “state types” (e.g., ElevatorState) are translated by the AFC D as given sets, although they should be defined as enumerated sets. Also, in the case of the Z++ class Elevator shown in Fig. 8.15, the attributes of array type denoting the internal and external requests of the elevator are translated as $\text{seq}(\mathbb{N})$, although a more suitable representation in this particular case is \mathbb{PN} , since these attributes are better modelled as sets.

The work on the formal specification, aimed at its enhancement, encompasses various aspects. In particular, all sorts of constraints on both the structure and the behaviour of the class’ instances, as well as the bodies of the operations can be specified. Without entering in too many details, we exemplify this aspect of formalisation by considering the Elevator Z++ class of Fig. 8.15 and by defining more precisely the conditions that trigger the transitions between the elevator’s states. Using the modified definitions:

```
carRequests:  $\mathbb{PN}$ 
upFloorRequests:  $\mathbb{PN}$ 
downFloorRequests:  $\mathbb{PN}$ 
```

and the equivalences:

```
floorRequests  $\hat{=}$  upFloorRequests  $\cup$  downFloorRequests
requests  $\hat{=}$  carRequests  $\cup$  floorRequests
```

some of the conditions of the internal transit operations τ_K included in the Elevator class can be written as follows (for each condition the transition it triggers is shown on the right-hand side of the formula):

- (a) $\text{other_level_requests_and_up_to_go} \hat{=} \text{current} \in \text{floorRequests} \wedge$
 $\exists x \in \text{floorRequests} \bullet x > \text{current}$ [τ_1]
- (b) $\text{target_reached_and_down_to_go} \hat{=} (\text{current} = \text{target}) \wedge$
 $(\text{dir} = \text{down} \wedge \bar{\exists} x \in \text{requests} \bullet x > \text{target}) \vee (\text{dir} = \text{up} \wedge$
 $\bar{\exists} x \in \text{requests} \bullet x > \text{target} \wedge \exists y \in \text{requests} \bullet y < \text{target})$ [τ_4]
- (c) $\text{up_target} \hat{=} \text{target} > \text{current}$ [τ_5]
- (d) $\text{level_request_and_up_to_go} \hat{=} \text{current} \in \text{upFloorRequests}$ [τ_7]
- (e) $\text{no_requests} \hat{=} \text{requests} = \emptyset$ [τ_9]

Of course, the above are a very small part of the work needed in the Z++ space, a significant amount of detail being necessary to describe the system in a complete and precise way. In particular, the modelling of the complex Z++ class Elevator is a laborious task, in which the fine interplaying of conditions and operations need be carefully specified. During the enhancement of the formal specification the deformalisation process can also take place, modifications performed in the Z++ space being reflected partially in the UML space.

Attention to temporal properties of the system is also necessary. A detailed, elaborate specification of these properties is possible by writing RTL formulae in the HISTORY clause of the Z++ classes. Taking into consideration the temporal constraints placed in Section 8.2 on the Elevator System, solutions for expressing them in Z++ can involve the following expressions:

- (a) For the temporal constraint [T1] the condition for the door to open within a given interval of time after the elevator stops at a floor can be expressed as:

$$\forall i:N_1 \bullet \exists j:N_1 \bullet \uparrow(\text{door.open}, j) - \downarrow(\text{stop}, i) \geq \text{OPEN_MIN_TIME} \wedge \\ \downarrow(\text{door.open}, j) - \downarrow(\text{stop}, i) \leq \text{OPEN_MAX_TIME}$$

if the interpretation of the constraint is that the door starts to open and completes this action within the specified time bounds, or as:

$$\forall i:N_1 \bullet \exists j:N_1 \bullet \downarrow(\text{stop}, i) = \rightarrow(\text{door.open}, j) \\ \wedge \text{OPEN_MIN_TIME} \leq \text{delay}(\text{door.open}, j) \leq \text{OPEN_MAX_TIME}$$

if the requirement is that the door only starts its opening within the specified time bounds.

- (b) For the temporal constraint [T2], the condition for the door to stay open at floor for a specific period of time provided the CloseButton is not pressed during this period of time can be expressed as:

$$\forall i:N_1 \bullet (\exists j:N_1 \bullet \downarrow(\text{door.open}, i) \leq \clubsuit((\text{CloseButton.state} = \text{on}) := \text{true}, j) \leq \\ \downarrow(\text{door.open}, i) + \text{STAY_OPEN_NORMAL_TIME}) \Rightarrow \\ \uparrow(\text{door.close}, i+1) = \downarrow(\text{door.open}, i) + \text{STAY_OPEN_NORMAL_TIME}$$

- (c) For the temporal constraint [T3] the correlation between the closing of the door and the start of the elevator movement (either up or down) can be expressed as:

$$\forall i:N_1 \bullet \exists j:N_1 \bullet \text{CLOSE_MIN_TIME} \leq \uparrow(\text{move}, i) - \downarrow(\text{door.close}, j) \\ \leq \text{CLOSE_MAX_TIME}$$

- (d) For the temporal constraint [T7] the details regarding the audio and visual signals in case of emergency can be written as:

$$\forall i:N_1 \bullet \text{MIN_SIGNAL_DUR} \leq \text{duration}(\text{signal}, i) \leq \text{MAX_SIGNAL_DUR} \wedge \\ \uparrow(\text{signal}, i+1) - \downarrow(\text{signal}, i) \leq \text{SIGNAL_SEPARATION}$$

The constraint [T4] can be modelled using a variable that records the value of now at “new floor” occurrences during the elevator’s movement, while conditions [T5] and [T6] can be expressed with predicates similar to those presented above.

Using the specification capabilities of RTL, including the extensions proposed by Lano for its use within the frame of Z++, detailed time-related requirements placed on the system can be rigorously expressed.

8.5 Chapter Summary

In this chapter the modelling approach proposed in the thesis has been exemplified using an Elevator System on which a number of general and temporal constraints have been placed. Examples of artefacts for all the steps of the regular modelling process presented in Chapter 7 have been provided and observations on the role of each step have been included. Examples of applying the AFCD and AFSD algorithms described in Chapter 6 have also been provided, the class diagram of the ELS being translated into a Z++ specification. Remarks on the need for enhancing the formal specification, as well as on the need of precisely describing the temporal aspects of the systems have also been included.

9 TOWARDS AN INTEGRATED ENVIRONMENT: A PROTOTYPE FOR HARMONY

“Build me straight, O worthy Master!
Staunch and strong, a goodly vessel,
That shall laugh at all disaster,
And with wave and whirlwind wrestle!”

[H. W. Longfellow, *The Building of the Ship*, 1849]

9.1 Introduction

In this chapter the overall design of the specification environment entitled Harmony is presented and details are given regarding both its operational capabilities and its GUI appearance. This tool is intended to fully support the modelling process proposed in Chapter 7, with special attention paid to provisions for sustaining the formalisation of UML classes and state diagrams described in Chapter 6. Deformalisation is also supported and aids are included for easy manipulation of Z++ symbols in the process of writing formal specifications. A “tandem” mode of operation is introduced, consisting essentially in the synchronised presentation of a UML class compound and its corresponding Z++ class specification. A more complete description of this CASE tool, currently evolving into a prototype, is provided in Appendix C. In this chapter, the general principles of Harmony are presented first, followed by an overall view of the environment, and then by successive descriptions of Harmony’s main components: the Project Pane, the UML Space, and the Z++ Space. A number of additional features of the environment, including specific toolboxes and buttons, are also presented.

9.2 General Principles

Because we attempt to combine the benefits of “both worlds” (“formal” and, respectively, “informal,” as well put by Alexander in the title of his paper [Alexander95]) and to “harmonise” the use of the UML and Z++ notations we have assigned the name Harmony to the environment that supports our modelling approach. In its present form, Harmony is intended to fully support the modelling process outlined in Chapter 7, with particular emphasis on the formalisation and deformalisation activities described in Chapter 6. Both UML and Z++ specifications can be “simultaneously” developed in Harmony and a bi-directional link exists between the formal and the graphical representations of the system, ensured by the translation mechanisms described previously in the thesis. This allows changes in the graphical representations to be reflected into the formal specifications, as well as the modifications of the formal part to be fed back into the diagrammatic description of the system.

Reflecting our philosophy for a rigorous yet pragmatic modelling approach, our goal for this stage of Harmony's development was to keep things simple and focus on those aspects of TCS that must be completely and correctly captured during the early stages of software development. Further extensions for the environment are possible, some of them outlined in the Conclusions of this thesis, and provisions for interfacing with external tools are included. Presently, Harmony's designed capabilities are adequate for the development of the Elevator case study described in Chapter 8, as well as for modelling other TCS. Of course, software systems in which the focus is not on temporal restrictions (“non TCS”) can also be specified using Harmony.

The environment operates on *specification projects*, which are sets of specifications represented in diagrammatical (UML) and/or mathematical (Z++) forms. For this reason Harmony is referred to as an *integrated specification environment* (ISE). The combined result of the specification activities supported by Harmony, more exactly the sum of the artefacts (model elements) produced in these activities within the procedural frame presented

in Chapter 7, constitutes the integrated model of the system. Since a total formalisation of the system is typically not required, the environment can be used for a partial formalisation within a complete specification of the system. In practical terms, this means that it is not necessary that a one-to-one correspondence between UML and Z++ components is achieved –some parts of the system will be described both in UML and Z++, while other only in UML or only in Z++. In principle, it is possible to have the entire system specified solely in Z++, but this can be efficient only in the case of small-sized systems (it also comes against our idea of combining formality and informality in software specification).

One of the distinguishing characteristics of Harmony is that it is monolithic in the sense defined in Section 4.2. By itself, it is sufficient to sustain a complete specification of the system and through its provisions for interfacing with external tools it is also capable in principle of supporting further development (in particular, we envisage interfacing with external tools for formal proof and formal refinement). Thus, it can be used as the “root” instrument from which other applications can be started –this is in contrast with other approaches, for instance RoZeLink’s, where three applications need be separately started from the operating system in order to perform the formalisation and deformalisation processes [RoZeLink99]. As shown in next section, it also “monolithically” integrates two “worlds,” the visual world of UML, and the textual, intensively symbolical world of Z++.

Another point that needs be further highlighted about Harmony is that it is designed for producing pragmatic, efficient, and precise models through the combined use of semi-formal diagrams and formal specifications. In this respect, the use of Z++ has primarily the role of supporting better understanding of the system and, generally speaking, of enhancing the intellectual control over the system. As discussed in Chapter 2, this is one of the main advantages of using formal methods and, consequently, although more intricate features for formal processing can be added later through Harmony’s “add-ins” feature, they nevertheless are beyond the scope of this thesis. Only a syntax and consistency checker is envisaged to be included directly in Harmony, this being a useful tool for enhancing the developer’s confidence in the accuracy of his or her Z++ specifications.

As a matter of overall organisation of the two modelling spaces it is necessary to mention that while a project may consist of several class diagrams there is a unique Z++ specification for the system. Because a Z++ specification encompasses not only classes but also statements external to classes (such as definitions of global types and operations on classes) and because we have attempted to ensure a consistent way of accessing the groups of artefacts within the project, a Global Spec component has been included in the Z++ Space. When fully expanded, as detailed in Section 9.6, it shows the entire text of the formal specification.

On a more detailed level, support for the class compound construct introduced in Chapter 7 is available, the idea being that the key classes of TCS need be detailed not only in respect with their attributes and their operations, but also with the sequencing of their operations (as captured in state diagrams). From a GUI point of view, a simple splitter bar between the space in which the UML class is represented and the one that corresponds to the class' state diagram is introduced, the two regular UML constructs being syntactically associated in the graphical model (they are also inherently associated in Z++ class declarations, where, within the CLASS construct, the HISTORY clause describes the temporal properties of the class' objects).

Also, since an intense work on UML class compounds and on their corresponding Z++ classes is expected, a synchronisation mechanism of on-screen presentation of the corresponding COMP and ZPPC constructs is proposed (the abbreviations introduced in Table 7.I are used again in this chapter). This mechanism defines a mode of operation that can be viewed as a manifestation, in our terminology, of *the tandem principle* (simply put, it means that two entities are working together for accomplishing a common goal). This mode of operation that allows in essence the “simultaneous” development (or the simple inspection) of a class in both its UML and Z++ forms is further described in the next section.

Before describing further the organisation of Harmony, we need to point out that only the design of the environment has been completed, but not its implementation. Therefore, the screen shots that follow are only aids for further development and do not represent actual

captures of the running environment. The examples of UML and Z++ model elements included in Fig. 9.1, showing the Elevator class compound and its corresponding Z++ class specification have been pasted into the environment panes, and have not been developed with Harmony. The design of Harmony's user-interface presented in this chapter and further described in Appendix C, a "mock-up" prototype written in Java, includes nevertheless the necessary details for fully sustaining Harmony's implementation.

9.3 Overall Organisation

Fig. 9.1 presents Harmony in a typical situation, in which a project is loaded and work is undergoing in the UML and Z++ spaces on several modelling elements, specifically a use case, a scenario, a sequence diagrams, two UML class compounds, and two Z++ classes. As seen in the figure, the environment consists of a main window (or browser), divided into several panes and containing other GUI elements such as a menu bar and toolbars. Since a systematic description of Harmony's user interface is included in Appendix C, we focus in this chapter only on those aspects that distinguish the most this ISE. From this perspective, it is notable that Harmony has three main panes, referred to, respectively, as the Project Pane, The UML Space and the Z++ Space. In addition to these panes, to the menu bar, and to the environment's toolbars, a message console and a status bar are also included.

In short, the entire organisation of the project can be viewed in the Project Pane, which shows the collections of artefacts grouped as indicated in Chapter 7. Work on the semi-formal model is performed in the UML Space, and formal specifications are written in the Z++ Space. It is important to note that in this organisation the three panes can all coexist on the screen at any given moment, but they can also be individually turned off, as shown in the View Menu presented in Fig. 9.2. Thus, work can proceed either in parallel in the semi-formal and the informal spaces, or can be focused on one of the to modelling "worlds". The specifier can turn off either "world" and use only half of Harmony's capabilities, either for

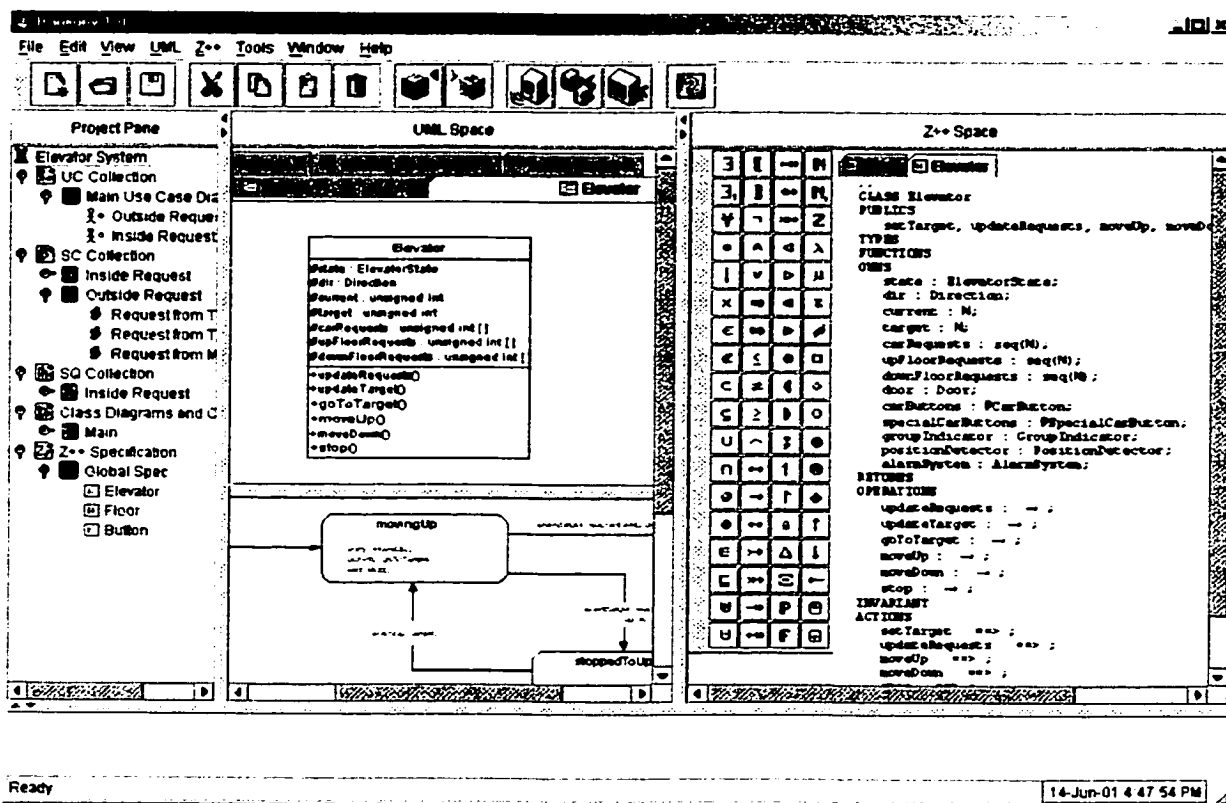


Fig. 9.1 Harmony's Look

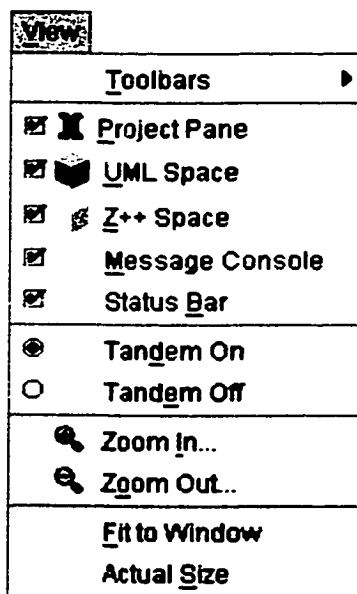


Fig. 9.2 The View Menu

developing solely UML models or for exclusively representing the system formally. As indicated in Fig. 9.2, all the panes of Harmony can be shown or hidden, although it will be of little value to have both the UML Space and the Z++ Space turned off simultaneously.

The same figure also allows the further description of the tandem mode of operation. In short, this mode of operation brings to the front of both modelling spaces (UML and Z++) the pair of corresponding COMP and ZPPC descriptions, irrespective on which space the developer is actually working. As such, all the relevant information about a class, specifically its UML structure in CLS, the UML state diagram CLSTD, and the formal ZPPC representation are visible at the same time on the screen provided that the “tandem option” is turned on and, of course, both UML and Z++ panes are open. The tandem mode of operations extends to class diagrams and their counterpart, the entire Z++ specification as reflected by Global Spec.

9.4 The Project Pane

The Project Pane, shown in Fig. 9.3 with the ELS project loaded and partially completed, is one of the three principal areas of the Harmony window. Its role is to visually present the project's structure in terms of artefacts and groups of artefacts as described in Chapter 7 and to support a number of operations that allow the gradual development and organisation of the project. These operations consist of creating a new artefact or group of artefacts, moving an element from a group to another, and deleting an artefact or a group of artefacts. They are invoked by mouse actions within the pane's area, for instance a right-mouse click on the empty space of the pane opens the New Model Element Selector shown in Fig. 9.4. Two of the above operations are also available through other interface elements of Harmony, more precisely New is included in the File, UML, and Z++ Menus, and both New and Delete have icons on the environment's main toolbar (again, we refer to Appendix C for further details).

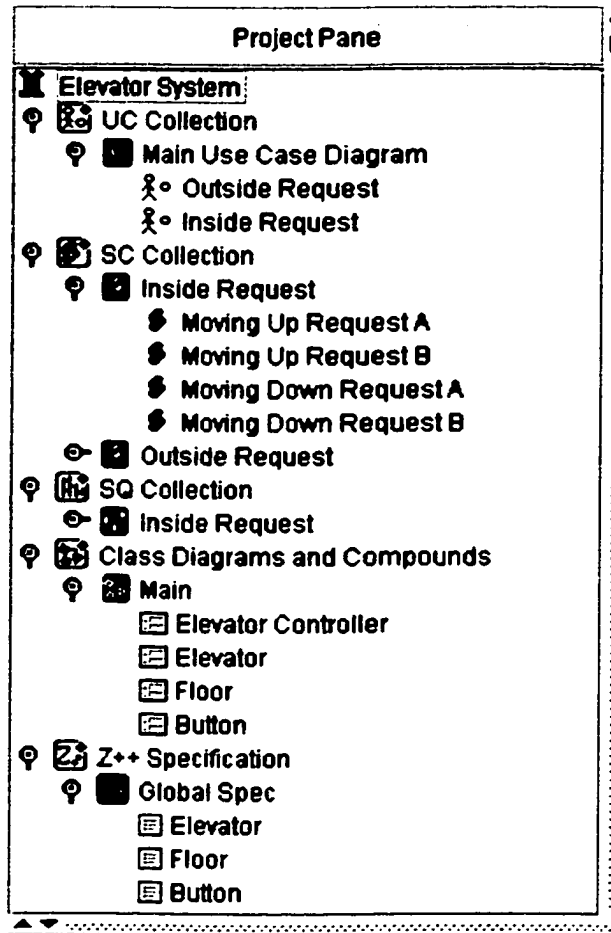


Fig. 9.3 The Project Pane

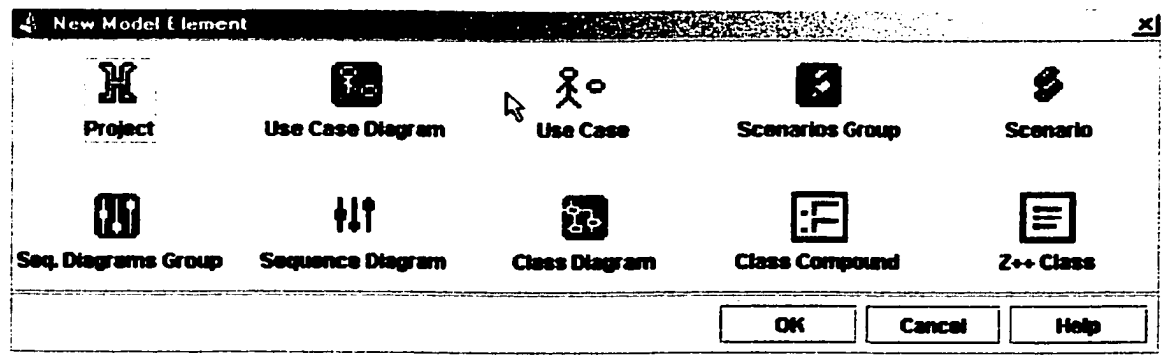


Fig. 9.4 The New Model Element Selector

From an operational point of view, immediately after Harmony is started all the environment's panes are empty, including the Project Pane. If from this state a new project is created, the view of Harmony is the one indicated in Fig. 9.5, which highlights the initial structure attributed by default to any project. This structure, following the guidelines of the procedural frame presented in Chapter 7, consists of the five major groups of artefacts considered there: the UC Collection, the SC Collection, the SQD Collection, the Class Diagrams and Compounds Section, and the Z++ Specification. (We prefer not to use the term collection for UML diagrams and compounds, since it may hint to an unstructured type of organisation, which is acceptable in the case of use cases, scenarios, and sequence diagrams –which need not, and typically cannot be completely specified,– but is not well suited for classes, which must be fully and correctly defined and organised).

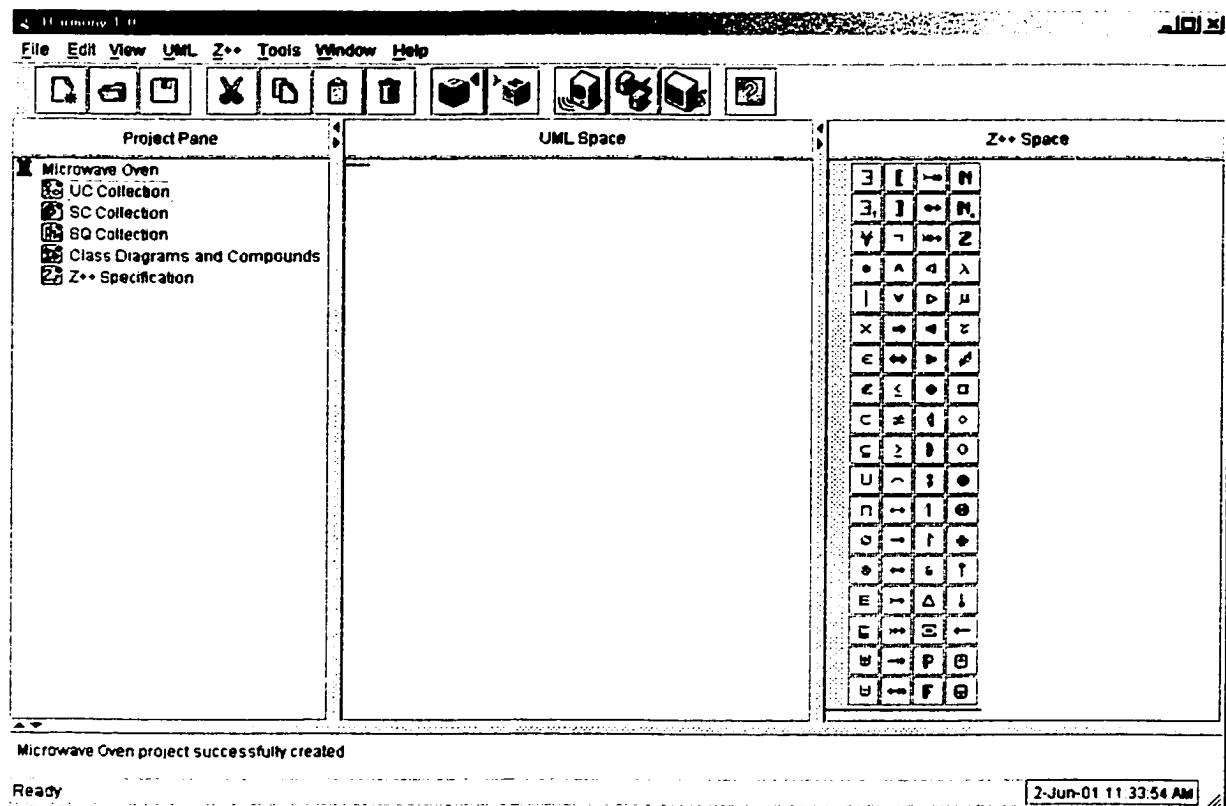


Fig. 9.5 Harmony with New Project Just Created

9.5 The UML Space

In the UML Space the specifier can work on one or more model elements, each model having its own tabbed-pane within this space. In Fig. 9.1, it can be seen that several such elements are opened at the same time, the one active being the Floor Class compound, with both its class specification and state diagram shown. Various options for working on the UML Space are available through the environment's menus and its toolbar. Among other things, the UML menu shown in Fig. 9.6 indicates that it is possible to disable the current UML toolbox (e.g., for inspection purposes, its elimination resulting into an increase of UML Space's visible area), as well as the state diagram part of a class compound (there are classes that have a trivial state diagram).

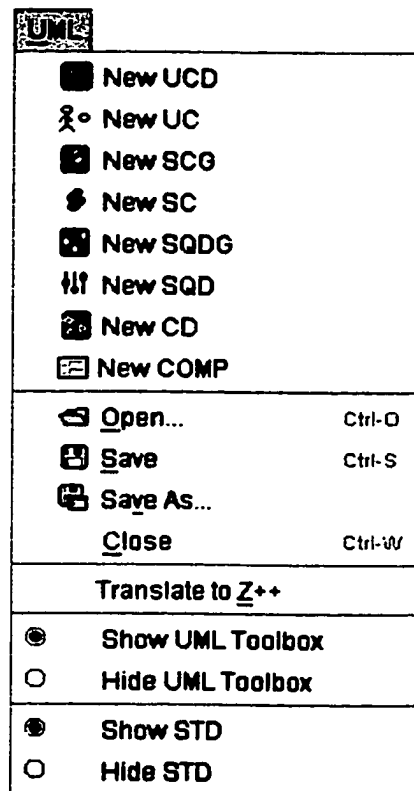


Fig. 9.6 The UML Menu

The same figure shows that creating new model elements, opening existing ones, saving them, or closing them are operations also possible through the UML Menu. An important function accessible via this menu is “Translate to Z++,” which allows the user to propagate new UML specifications or changes to existing ones into the Z++ Space. The rules for automated formalisation described in Chapter 6 are applied in this process.

The UML Space also contains a toolbox specific to each type of artefact created using the modelling approach proposed in this thesis. Since there are five distinct types of such artefacts, five types of UML toolboxes are provided, the one visible at a given moment corresponding to the type of artefact currently shown in the front tabbed-pane of the UML Space. One of these toolboxes is presented in Fig. 9.7, and all five are included in Appendix D. Some general symbols, such as “select item,” “text,” and “annotation” (the first three on the left-hand side of Fig. 9.7) are a common presence in most if not all UML toolboxes. The SQD Toolbox presented below includes additionally the “state,” “activation bar,” “message,” “message to self,” “asynchronous message,” “return message,” and “destroy object” symbols.



Fig. 9.7 A UML Toolbox

9.6 The Z++ Space

In Harmony, the Z++ space is the equal partner of the UML space and as such its dedicated menu has an organisation similar to that of the UML menu, as shown in Fig. 9.8. For instance, the reverse process of formalisation, the transfer of information from Z++ to UML, with its inherent simplifications discussed in Chapter 6, is invoked via the “Translate to UML” option, which has the counterpart “Translate to Z++” in the UML Menu.

Additionally, there are several functions specific to the Z++ space, all available through the Z++ menu.

Firstly, there is the “Analyse” option, which is intended to allow the syntax and consistency checking of the formal specifications. Secondly, due to the specific organisation of the Z++ specification, options for the presentation of the Global Spec as well of the Z++ classes are provided. More precisely, the Global Spec can show only the Z++ contents extraneous to classes, such as names of global types and hiding operations on classes (in case option Classes Hidden is selected); this contents together with the names of Z++ classes (if option Classes Collapsed is chosen); or the full text of the Z++ specification, i.e. the Z++ statements extraneous to classes and the detailed description of classes (if option Classes Expanded is selected). Also, an individual Z++ class can be presented within the UML space either alone (Hide Context option selected) or accompanied by the Z++ contents not included in classes (Show Context option selected).

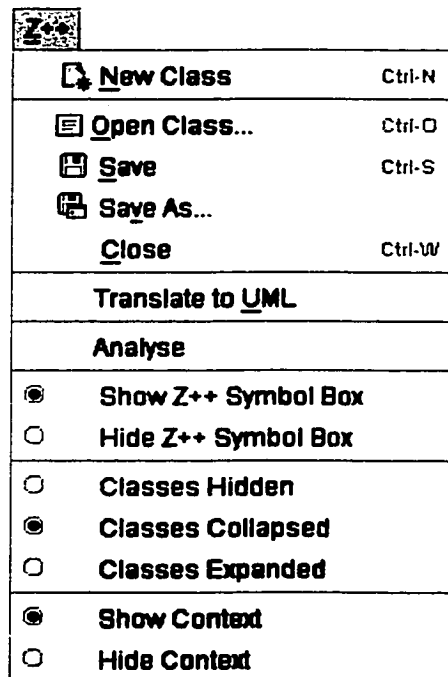


Fig. 9.8 Harmony's Z++ Menu

A further distinction from the UML counterpart comes from the fact that a Symbol Box instead of a Toolbox is available when working in this space. This “palette of mathematical symbols” provides a practical alternative to the use of combinations of keystrokes for inserting special symbols in the formal specification (a similar Symbol Palette is available in Logica’s Z Formaliser [Formaliser01]). The Symbol Box for the Z++ Space, with its comprehensive set of items is presented in Fig. 9.9. The Z and Z++ specific symbols have been compiled from the indexes available in [Spivey92, pp. 153] and [Lano95, pp. 417-418] and the Symbol Box includes only those items that cannot be written using a standard font such as Courier or Times New Roman. For instance, in order to keep the Z++ Symbol Box as small as possible the arithmetic operators +, -, *, and / are not included, nor are Z specific notational elements such as ::=, < .. >, or >> that can be represented using regular fonts. Because in Z++ there is a need for subscripts (and sometimes for superscripts) two non-Z symbols, the superscript and the subscript indicators are also included as the last two elements of the Z++ Symbol Box. As for the organisation of this toolbar, a “topic related” criterion has been applied, the symbols being grouped according to their use: existential and universal quantifiers first, followed by statement separators, then by operators pertaining to sets and bags, then by function related symbols, etc. The nine elements in the Symbols Box that precede the superscript and the subscript indicators on the last row are Z++ specific (do not pertain to the regular Z). Further details on the Z++ Symbol Box are available in Appendix C.

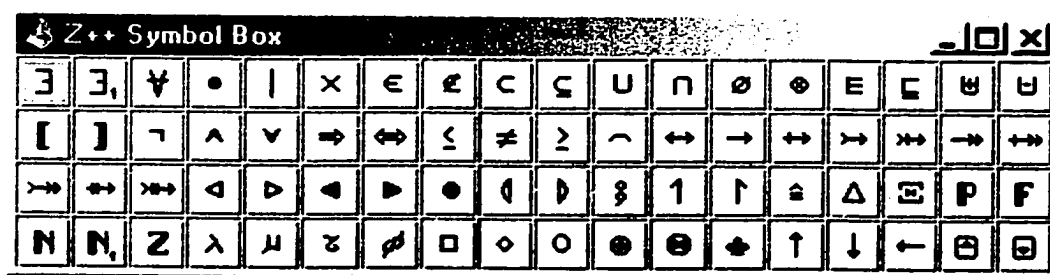


Fig. 9.9 The Z++ Symbol Box

9.7 Other Features

There are other features available in the Harmony environment, which is nevertheless kept as simple as possible without jeopardising either its ease of use or its full support for the formalisation activities described in Chapter 6 and for the combined UML/Z++ modelling process proposed in Chapter 7. Some of these features are briefly described below, while additional details are available in Appendix C.

For instance, there are five environment-specific buttons visible on Harmony's main toolbar, namely the Translate to Z++, Translate to UML, Tandem Off, Tandem On, and Analyse buttons. In addition, the logo used for Harmony (taken from [RogersGifs01] Clipart Gallery), can also be considered environment specific. In order to exploit a bit the harmony metaphor, the first four symbols presented in Fig. 9.10 have icons related to the acoustic domain, specifically a metronome for the Harmony logo, a single (mono) audio-speaker for the Tandem Off button, a pair of speakers (a stereo system) for Tandem On, and a sound analyser for the Analyse Z++ Specifications function. The last three icons have been downloaded from [LeosIcons01] while all the other icons present in Harmony have been either taken from the "Java Look and Feel Graphics Repository" (standard symbols such as New, Open, Delete, etc.) [JavaLook01] or created by us from the scratch (all the symbols for the artefacts and all the elements of the UML and Z++ toolboxes). For translation operations between the "worlds" of UML and Z++ two new symbols have been designed, both using "transfer arrows" and cubes in their representation, the latter suggesting complex, well defined "worlds" (of modelling, in our case). These translation buttons are represented on the last two positions of Fig. 9.10.



Fig. 9.10 Harmony Specific Symbols

Regarding the Harmony logo it is interesting to note that it can be viewed as conveying a combination of suggestions about the two most distinguishing characteristics of our modelling approach: focused on smooth integration of notations (“harmony”, suggested by an instrument associated with the rhythm of music), and focused on temporal properties of systems (“the metronome,” a device which punctuates the passage of time).

Other icon-centred elements of Harmony’s user interface include a Legend Pane for the symbols used in the modelling process, accessible via the Help menu. One of the tabbed-pane of the Legend Pane is shown Fig. 9.11.

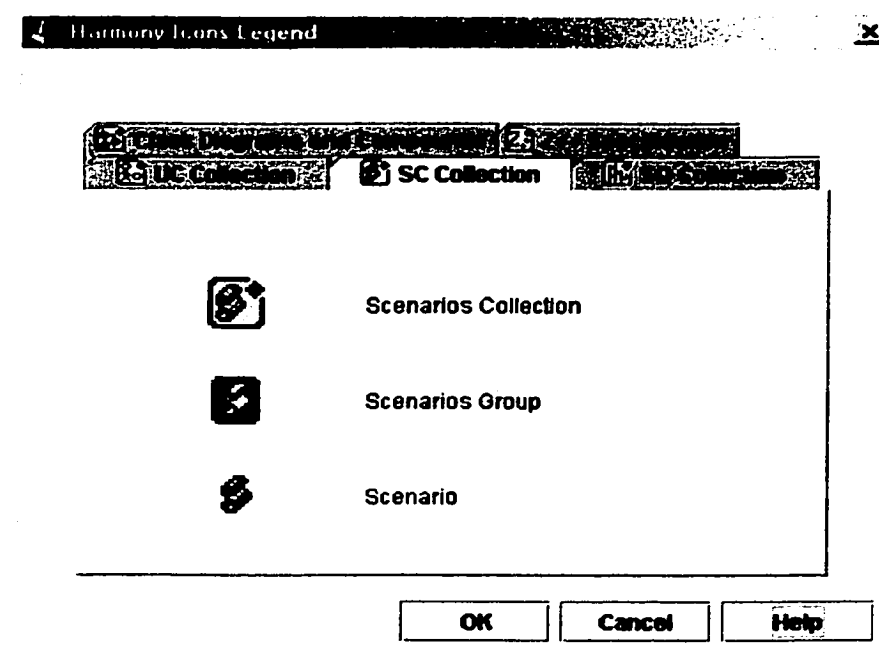


Fig. 9.11 The Legend Pane

Beside these detailed aspects of Harmony’s design there are some other, higher-level functional features that deserve to be mentioned. As shown in Appendix C, they include tools for customising the properties of the editor, of the project, or of the environment as whole, provisions for “add-ins,” zoom-in and zoom-out features, options for the export and the import of Z++ specifications, and creation of additional Harmony windows. In

particular, through the Add-Ins feature present in the Tools Menu, connections with external software tools are envisaged, and through the Export Z++ Specification included in the File Menu an independent file containing solely the Z++ description of the system can be generated for the purpose of being used in a separate development context. The counterpart of the latter feature, the Import Z++ Specification, has the role of allowing the inclusion of Z++ specifications developed externally into a Harmony project. This capability would permit the subsequent generation of the corresponding UML class structure through deformatisation.

9.8 Chapter Summary

In this Chapter the Harmony integrated specification environment has been introduced through the description of its user-interface and of the functionality available through this interface. This environment is intended to provide a monolithic integration of UML and Z++ notations by fully supporting the formalisation and deformatisation activities presented in Chapter 6 as well as the modelling process of TCS proposed in Chapter 7. The principles that permeate Harmony's design, the environment's general organisation, as well as its three major components, the Project Pane, the UML Space, and the Z++ Space have been described in a fair level of detail. Remarks on some secondary aspects of Harmony, such as specific icons and symbols, have also been included. Although Harmony is only in the design stage, the description presented in this chapter provides a good foundation for its implementation and allows the consideration of possible enhancements, some of them outlined in the next chapter.

10 CONCLUSIONS

“What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from.”

[T.S. Elliot, Little Gidding, *Four Quartets*, 1942]

10.1 Introduction

At the conclusion of this thesis, we first look back and highlight the merits and the limitations of our approach and then look forward to point out the work that remains to be done. To evaluate our work in the context of current research, a summary comparison with the closely related approaches discussed in Chapter 4 is included. The contributions of our work are presented by dividing them in two categories, principal and secondary, and the limitations of the present work are briefly reviewed. Needed improvements to the work described here and connected research paths that can be beneficially pursued in the future finalise the chapter.

10.2 Summary Comparison with Closely Related Approaches

In Chapter 4 five research studies were classified as “closely related approaches” to our work. Although they were examined in some detail in that chapter, we resort again to them in order to provide a brief comparison with our work. This comparison is based on a number of criteria, specifically:

- Type of translation from diagrammatic notations to formal specifications, which can be either an OO to an OO or an OO to non-OO formalisation (the latter means that

constructs of a non object-oriented formal language such as Z are adapted to represent OO constructs from the semi-formal counterpart);

- Provisions for modelling RT systems (either included or not);
- Type of integration of notations, based on the classification introduced in Section 4.2. Under this criterion, we denote simple formalisation (or derivation) by F, complementary formalisation by CF, and tight integration of notations (which involves two-way translations) by TF;
- The characteristic that can be referred to as the monolithic construction of the supporting specification environment (the definition of a monolithic environment has also been introduced in Section 4.2);
- Capability of applying tool-supported processing techniques on the formal specifications, including syntax and consistency checking, formal verification, and refinement. This capability describes the present situation and refers to the connection with tools that already exist;
- The usage of the formal notation involved, reflecting its popularity and the number of applications in which it has been employed.

The results of this comparison are shown in Table 10.I. From this Table it can be seen that our approach has its merits as well as its limitations. While the merits are stressed in Sections 10.3 and 10.4, about the two main limitations highlighted in the table we need to point out that although one is more difficult to overcome (specifically, it is difficult to match the popularity enjoyed by Z, ZEST, or Object-Z), the other (connection to tools for further formal processing) can be surmounted through the continuation of the work presented in this thesis (Section 10.6 describes our intentions in this respect). In addition, there are several other limitations, discussed in Section 10.5, which also can be overcome through additional work. Nevertheless, we believe that our approach provides a viable alternative for combined, semi-formal/formal software specification, and introduces a fresher presence in the landscape of pragmatic development of TCS through synergetic use of semi-formal and formal techniques.

Table 10.1 Summary Comparison with Closely Related Approaches

Approach	Criteria					
	OO to OO formalisation	RT specification capability	Type of integration of notations	Monolithic specification environment	Processing of formal specifications (analysis, refinement, etc.)	Usage of formal notation
[Jia97]	no	No	CF	no	Yes	high (Z)
[Noe00]	no	No	CF	partial	Yes	high (Z)
[France97]	no	Yes	F	no	Yes	high (Z)
[RoZeLink99]	yes	No	TF	no	Yes	high (ZEST)
[Kim00b]	yes	yes	CF	N/A	Yes	high (Object-Z)
Harmony	yes	yes	TF	yes	no	low (Z++)

10.3 Main Contributions

The main contributions of our work are the following:

- Pragmatic integration of two notations, one graphical and semi-formal (UML) and the other textual and formal (Z++), in a specification approach that attempts to reap the benefits of both;
- The advanced formalisation of UML constructs in Z++, both in terms of structure and behaviour. It is worth noting that although Lano describes ways of formalising OO models in Z++ [Lano95] this was nevertheless done in the context of the OMT notation

and, while we have not covered all the minute aspects of the formalisation process, our translation from UML to Z++ is performed in a more pragmatic and systematic way, with detailed algorithms being proposed. Also important to note, very few formalisation approaches look at both structure and behaviour, notably [France97] and [Kim00b]), and practically only one within the vicinity of our topic location [RoZeLink99], takes into consideration the reverse propagation, from formal (textual) specifications to semi-formal (diagrammatic) models;

- Rigorous and pragmatic treatment of TCS through the use of a formalism, RTL, whose notation is easy to comprehend and apply. The usability and coverage of our modelling approach stem from its capability of capturing various time-related properties of systems, as discussed in Chapters 5 and 8;
- Lightweight, practical specification process allowing for both reliable specifications and rapid development of software. The main idea of our approach is to provide a rigorous and usable alternative for OO specification of TCS. In order to achieve this, we have focused on the most critical aspects of modelling (in terms of consequences in the life-cycle of the product) and covered the earlier phases of software construction, in particular the OO analysis phase;
- Design of the Harmony ISE, aimed at fully supporting the technique proposed in this thesis. What particularly distinguishes this specification environment is its monolithic construction, support for tight-integration of notations, balanced inclusion of both functions and notational elements (we have attempted to keep things simple, yet still operationally powerful), provisions for easier manipulation of formal symbols, and capacity for extension.

10.4 Other Contributions

There are also a number of aspects of our work that can be listed as secondary contributions. They do not play principal roles in the discourse of this thesis, yet they support it and also represent bits of original work that can be further employed and further investigated:

- Development of a non-trivial example, the ELS. Through this application, which can be added to the rather large collection of elevator case studies recorded in the literature, the most relevant particularities of our approach have been illustrated;
- Classification of integrations of notations. We needed it to provide a basis of comparison with other approaches, but it can be usefully employed or adapted for comparing alternatives of integrating notations in other contexts (e.g., hardware design);
- A zoom-in technique of investigation. The technique has of course been employed in numerous other cases (it is an embodiment of the classical top-down method of investigation), yet there are no reports in literature that present it under the “zoom-in” metaphor;
- Proposal of a class compound construct that encompasses both structure, expressed in the class construct, and behaviour, captured in the state diagram (in addition to the one defined by the operations of the class construct). This pairing of UML constructs (class and state diagram) although quite simple in its idea is nevertheless powerful in that it extends the basic OO concept of encapsulation (data + operations) to a stronger appendage of the type data + operations + allowable sequences of execution;
- Several proposals regarding the terminology: TCS, ISE, tight-integration of notations, monolithic approach, transition signature, transit operation, and the set of terms and abbreviations used to denote the artefacts and steps of our modelling approach;
- A comprehensive review of the research space. Compulsory part of a PhD thesis, of course, but we extended our survey to cover aspects such as UML perspectives and exemplification of UML constructs through the ACTS specification “theme”. Both the survey of UML and the modelling of ACTS can evolve in fully-fledged studies on their own rights.

10.5 More On the Limitations of the Proposed Approach

Besides the two main limitations indicated in Section 10.2, namely Z++’s lack of exposure (due primarily to its lack of tools) and Harmony’s lack of connection to tools for formal analysis and refinement, there are several other limitations of the proposed approach that

need be addressed in order to enhance the work presented in this dissertation. In particular, the treatment of state diagrams is rather limited, confined to “flat,” non-composite structures and to sequential executions, which reduces the applicability of the translation algorithms to modelling TCS (such systems need treatment of concurrency, synchronisation, etc.). Also, the treatment of the timing constraints needs significant improvement, since we have not tackled the automated translation of timing constraints attached to UML structural constructs, and provided only a limited translation of such constraints in the case of state diagrams (the burden of formalising temporal constructs lies too heavily on the human formaliser). In addition, a more concise and precise description of the translation algorithms can be obtained if meta-models for UML and Z++/RTL are used. The deformalisation process also needs improvement; it has been described only by a set of principles and the outline of an algorithm, hence further work on details is needed, as it is needed on dealing with the particular aspects of applying the translations algorithms discussed in Section 6.6.

10.6 A Look Forward

The work presented here is neither complete nor free of errors. We are aware, as indicated in the previous section, of some of its limitations and know that further work is needed in several directions. In particular, our intentions for future work encompass:

- Enhanced automated formalisation and deformalisation. Due to the importance of these processes in producing reliable specifications further studies are necessary, especially regarding the translation of dynamic UML models into precise Z++ specifications;
- Syntax and consistency checking of Z++. We consider the alternative of translating Z++ to Z insufficient, and in order to achieve one of the primary goals of our approach, that of increased intellectual control over the software being developed, automated syntax and consistency checking of formal specifications can play an important role;
- The complete implementation of Harmony. At the time these lines are written, we have completed the design of Harmony. Nevertheless, only by implementing it and exercising

it on various case studies we will be able to both improve its design and gain additional insight about the ways our approach can be efficiently put to work in practice;

- Development of tools for formal analysis and refinement. Although we have not aimed at covering aspects of formal proof and formal refinement, the development of such tools is necessary to support the wider application of our approach;
- More applications. In addition to improving the design of Harmony, the application of the approach on more case studies will be beneficial for fine-tuning the technique of specifying TCS proposed in this thesis.

The above is work that we intend to pursue further in order to develop Harmony into a tool usable on large scale. But, predictably, while working on a given topic ideas for other subjects spring into one's mind, some related and some not so related to the original topic of investigation (and some relatively clear and some decidedly vague). Some of the more related and the slightly more well-formed such ideas that occurred to us are:

- Visualisation of Z constructs in the sense proposed by Kim and Carrington for Z in [Kim99a];
- Usage of the CSP formalism instead of RTL for alternative, enhanced modelling of parallelism;
- Animation of a subset of specifications. A look at solutions such as the Z-based Sum language [Utting95] can provide a starting point;
- Integration of our modelling technique with code generation tools aimed at exploiting the RT capabilities of high-level programming languages;
- Formalisation of modelling patterns and their utilisation in various contexts, for instance for developing Web applications.

At any given time, our Elevator's door may or may not be open depending on a series of factors, as discussed in Chapter 8, but the door of further work and further improvements should always be open.

Bibliography

- [Abernethy00] Abernethy, K., Kelly, J., Sobel, A., Kiper, J.D., Powell, J., "Technology Transfer Issues for Formal Methods of Software Specification," *Proc. of the 13th Conf. on Software Engineering Education and Training*, March 2000, pp. 23-31.
- [Abrial80] Abrial, J.-R., Schuman, S., and Meyer, B., "Specification Language," in McKeag R.M., and MacNaghten, A.M. (editors), *On the Construction of Programs: An Advanced Course*, Cambridge University Press, 1980, pp. 343-410.
- [Abrial96] Abrial, J.-R., *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
- [Alagar00] Alagar, V.S., and Muthiayen, D., "Towards a Mechanical Verification of Real-Time Reactive Systems Modeled in UML," *Proc. of the 7th Intl. Conf. on Real-Time Computing Systems and Applications*, Dec. 2000, pp. 245-254.
- [Alderson98] Alderson, A., Hull, M.E.C., Jackson, K., and Griffiths, L. E., "Method Engineering for Industrial Real-Time and Embedded Systems," *Information and Software Technology*, vol. 40, no. 8, Aug. 1998, pp. 443-454.
- [Alemán00] Alemán, J.L.F., and Álvarez, A.T., "Can Intuition Become Rigorous? Foundations for UML Model Verification Tools," *Proc. of the 11th Intl. Symposium on Software Reliability Engineering (ISSRE 2000)*, Oct. 2000, pp. 344-355.
- [Alencar94] Alencar, A.J., and Goguen, J.A., "Specification in OOZE with Examples," in Lano, K., and Houghton, H. (editors), *Object-Oriented Specification Case Studies*, Prentice Hall International, 1994, pp. 158-183.
- [Alexander95] Alexander, P., "Best of Both Worlds: Combining Formal and Semi-Formal Methods in Software Engineering," *IEEE Potentials*, vol. 14, no. 5, Dec. 1995/Jan. 1996, pp. 29-32.
- [Alloy00] "The Alloy Constraint Analyzer" web-site, Alloy version 2000, Software Design Group, Massachusetts Institute of Technology, accessed Feb. 6, 2001 at <http://sdg.lcs.mit.edu/alloy/>
- [Audsley96] Audsley, N.C., Burns, A., Davis, R.I., Scholefield, D.J., and Wellings, A.J., "Integrating Optional Software Components into Hard Real-Time Systems," *Software Engineering Journal*, vol. 11, no. 3, May 1996, pp. 133-140.
- [Aujla94] Aujla, S., Bryant, T., and Semmens, L., "Applying Formal Methods Within Structured Development," *IEEE Journal On Selected Areas in Communications*, vol. 12, no. 2, Feb. 1994, pp. 258-264.
- [Avnur90] Avnur, A., "Finite-State Machines for Real-Time Software Engineering," *Computing and Control Engineering Journal*, vol. 1, no. 6, Nov. 1990, pp. 273-278.
- [Awad96] Awad, J.Z.M., Ziegler, J., and Kuusela, J., *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice-Hall, 1996.

- [Bailin89] Bailin, S.C., "An Object-Oriented Requirements Specification Method," *Communications of the ACM*, vol. 32, no. 5, May 1989, pp. 608-623.
- [Barden94] Barden, R., Stepney, S., and Cooper, D., *Z in Practice*, Prentice-Hall International, 1994.
- [Barnes96] Barnes, J., *Programming in Ada '95*, Addison-Wesley Longman, 1996.
- [Barrett89] Barrett, G., "Formal Methods Applied to A Floating Point Number System," *IEEE Transactions on Software Engineering*, vol. 15, no. 5, May 1989, pp. 611-621.
- [Barrios99] Barrios, S.D., and Lopez J.C., "Heterogeneous Systems Design: a UML-based Approach." *Proc. of the 25th EUROMICRO Conf.*, Sep. 1999, vol. 1, pp. 386-389.
- [Becker00] Becker, L.B., Pereira, C.E., Dias, O.P., Teixeira, I.M., and Teixeira, J.P., "MOSYS: A Methodology for Automatic Object Identification from System Specification," *Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC-2000)*, pp. 198-201.
- [Bell99] Bell, A.E., and Schmidt, R.W., "UMLoquent Expression of AWACS Software Design," *Communications of the ACM*, vol. 42, no. 10, Oct. 1999, pp. 55-61.
- [Bellini00] Bellini, P., Mattolini, R., and Nesi, P., "Temporal Logic for Real-Time System Specification," *ACM Computing Surveys*, vol. 32, no. 1, March 2000, pp. 12-42.
- [Björkländer00] Björkländer, M., "Graphical Programming Using UML and SDL," *IEEE Computer*, vol. 33, no. 12, Dec. 2000, pp. 30-35.
- [Bjørner78] Bjørner, D., and Jones, C.B. (editors), *The Vienna Development Method: The Meta-Language*, Lecture Notes in Computer Science, vol. 61, Springer-Verlag, 1978.
- [Bloesch94] Bloesch, A., Kazmierczak, E., Kearney, P., and Traynor, O., "The Cogito Methodology and System," *Proc. of the First Asia-Pacific Software Engineering Conf.*, Dec. 1994, pp. 345-355.
- [Boehm84] Boehm, B.W., "Verifying and Validating Software Requirements and Design Specifications," *IEEE Software*, vol. 1, no. 1, Jan. 1984, pp. 75-88.
- [Bollella00] Bollella, G., Gosling, J., and Brosgol, B., *Real-Time Java Specification*, Addison-Wesley, 2000.
- [Bolognesi98] Bolognesi, T., and Derrick, J., "Constraint-Oriented Style for Object-Oriented Formal Specification," *IEE Proc. Software*, vol. 145, no. 2-3, April/June 1998, pp. 61-69.
- [Booch86] Booch, G., "Object-Oriented Development," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, Feb. 1986, pp. 211-221.
- [Booch94] Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin/Cummings Publishing Company, 1994.
- [Booch98] Booch, G., Rumbaugh, J., and Jacobson, J., *The Unified Modeling Language: User Guide*, Addison-Wesley Longman, 1998.

- [Bordbar00] Bordbar, B., Giacomini, L., and Holding, D.J., "UML and Petri Nets for Design and Analysis of Distributed Systems," *Proc. of the 2000 IEEE Intl. Conf. on Control Applications*, Sep. 2000, pp. 610-615.
- [Bowen95a] Bowen, J.P., and Hinchey, M.G., "Ten Commandments of Formal Methods," *IEEE Computer*, vol. 28, no. 4, April 1995, pp. 56-63.
- [Bowen95b] Bowen, J.P., and Hinchey, M.G., "Seven More Myths of Formal Methods," *IEEE Software*, vol. 12, no. 4, July 1995, pp. 34-41.
- [Bruel96] Bruel, J.M., France, R.B., and Benzekri, A., "A Z-Based Approach to Specifying and Analyzing Complex Systems," *Proc. of the 2nd Intl. Conf. on Engineering of Complex Computer Systems*, Oct. 1996, pp. 336-343.
- [Bruel98a] Bruel, J-M., and France, R.B., "Transforming UML Models to Formal Specifications," In *Proc. of the First Intl. Conf. on UML - Beyond the Notation, Lecture Notes in Computer Science*, Springer-Verlag, vol. 1618, 1998, accessed Feb. 10, 2001 at <http://www.cs.york.ac.uk/puml/papers/brueluml98.pdf>
- [Bruel98b] Bruel, J.M., Cheng, B., Easterbrook, S., France, R., and Rumpe, B., "Integrating Formal and Informal Specification Techniques. Why? How?," *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Oct. 1998, pp. 50-57.
- [Bucci94] Bucci, G., Campanai, M., Nesi, P., and Traversi, M., "An Object-Oriented Dual Language for Specifying Reactive Systems," *Proc. of the 1st Intl. Conf. on Requirements Engineering*, April 1994, pp. 6-15.
- [Buhr90] Buhr, R.J.A., *Practical Visual Techniques in System Design with Applications to Ada*, Prentice-Hall, 1990.
- [Burns95] Burns, A., and Wellings, A., *HRT-HOOD: A Structured Design Method for Hard Real-Time Systems*, Elsevier, 1995.
- [Burns97] Burns, A., and Wellings, A., *Real-Time Systems and Programming Languages*, Second Edition, Addison-Wesley Longman, 1997.
- [Cau98] Cau, A., Zedan, H., and Moszkowski, B., "'Lean' Formal Methods in the Development of Provably Correct Real-Time Systems," *IEE Colloquium on Real-Time Systems*, Digest no. 1998/306, April 1998, pp. 6/1-6/5.
- [Chaochen91] Chaochen, Z., Hoare, C.A.R., and Ravn, A.P., "A Calculus of Duration," *Information Processing Letters*, vol. 40, no. 5, May 1991, pp. 269-276.
- [Cheesman00] Cheesman, J., and Daniels, J., *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2000.
- [Chen76] Chen, P., "The Entity-Relationship Model—Toward Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, March 1976, pp. 9-36.
- [Chen98] Chen, Z., Cau, A., Zedan, H., Liu, X., and Yang, H., "A Refinement Calculus for the Development of Real-Time Systems," *Proc. of the 1998 Asia Pacific Software Engineering Conference*, Dec. 1998, pp. 61-68.

- [Cheng94] Cheng, B.H.C., Wang, E.Y., and Bourdeau, R.H., "A Graphical Environment for Formally Developing Object-Oriented Software," *Proc. of the 6th Intl. Conf. on Tools with Artificial Intelligence*, Nov. 1994, pp. 26-32.
- [Ciapessoni99] Ciapessoni, E., Coen-Porisini, A., Crivelli, E., Mandrioli, D., Mirandola, P., and Morzenti, A., "From Formal Models to Formally Based Methods: An Industrial Experience," *ACM Transactions on Software Engineering and Methodology*, vol. 8, no. 1, Jan. 1999, pp. 79-113.
- [Clarke96] Clarke, E.M., and Wing, J.M., "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, Dec. 1996, pp. 626-643.
- [Coad90] Coad, P., and Yourdon, E., *Object-Oriented Analysis*, Prentice-Hall, 1990.
- [Coad91] Coad, P., and Yourdon, E., *Object-Oriented Design*, Prentice-Hall, 1991.
- [Cogito97] "Cogito, Ergo Sum: Methodology and Toolset for the Formal Development of Software," (Cogito version 1997), The Cogito web-site, Software Verification Research Center, University of Queensland, Brisbane, Australia, accessed Feb. 5, 2001 at <http://svrc.it.uq.edu.au/Cogito/>
- [Coleman90] Coleman, G.L., Ellison, C.P., Gardner, G.G., Sandini, D., and Brackett, J.W., "Experience in Modeling a Software System Using STATEMATE," *Proc. of the IEEE Intl. Conference on Computer Systems and Software Engineering (COMPEURO'90)*, May 1990, pp. 104-108.
- [Coleman94] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P., *Object-Oriented Development: the Fusion Method*, Prentice-Hall, 1994.
- [Conallen99a] Conallen, J., "Modeling Web Application Architectures with UML," *Communications of the ACM*, vol. 42, no. 10, Oct. 1999, pp. 63-70.
- [Conallen99b] Conallen, J., and Bebick, K., *Building Web Applications with UML*, Addison-Wesley, 1999.
- [Coombes92] Coombes, A.C., and McDermid, J. A., "Using Diagrams to Give A Formal Specification of Timing Constraints in Z," in Bowen, J.P., and Nicholls, J.E. (editors), *Proc. of the Z User Workshop*, London, UK, Dec. 1992, Workshops in Computing, Springer, 1992, pp. 119-130.
- [Coombes93] Coombes, A.C., and McDermid, J. A., "Specifying Temporal Requirements for Distributed Real-Time Systems in Z," *Software Engineering Journal*, vol. 8, no. 5, Sep. 1993, pp. 273-283.
- [Cox93] Cox, P.T., "Picture the Future," *Object Magazine*, July-Aug. 1993.
- [D'Almeida92] D'Almeida, J., Achutan, R., Radhakrishnan, T., and Alagar, V.S., "Transformation of a Semi-Formal Specification to VDM," *Proc. of the 7th Knowledge-Based Software Engineering Conf.*, Sept. 1992, pp. 40-49.
- [D'Souza98] D'Souza, D.F., and Wills, A.C., *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison-Wesley Longman, 1998.

- [Dasarathy85] Dasarathy, B., "Timing Constraints of Real-Time Systems: Construct for Expressing Them, Methods of Validating Them," *IEEE Transactions on Software Engineering*, vol. 11, no. 1, Jan. 1985, pp. 80-86.
- [Dascalu89] Dascalu, S.M., "Architectural and Functional Features of the Computing, Measuring, and Control Subsystem of the ESMC-04 Automatic Camshaft Testing Machine," (in Romanian), *Proc. of the 2nd Symposium on Structures, Algorithms, and Equipment for Process Control*, Iasi, Romania, Oct. 1989, pp. 545-550.
- [Dascalu99] Dascalu, S.M., "Towards the Integration of Two Software Specification Notations: UML and Z++," paper submitted in partial fulfillment of the requirements for the Visual Languages course, Dalhousie University, Halifax, NS, Canada, Aug. 1999.
- [Davis93] Davis, A.M., *Software Requirements: Objects, Functions & States*, Prentice Hall, 1993.
- [Davis98] Davis, A. M., "Predictions and Farewell," *IEEE Software*, vol. 15, no. 4, July/Aug. 1998, pp. 6-9.
- [Day00] Day, N., "A Framework for Multi-Notation Requirements Specification and Analysis," *Proc. of the 4th Intl. Conference on Requirements Engineering*, June 2000, pp. 39-48.
- [Delisle90] Delisle, N., and Garlan, D., "A Formal Specification of An Oscilloscope," *IEEE Software*, vol. 7, no. 5, Sep. 1990, pp. 29-36.
- [Dill96] Dill, D.L., and Rushby, J., "Acceptance of Formal Methods: Lessons From Hardware Design," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 23-24.
- [Dillon94] Dillon, L.K., Kutty, G., Moser, L.E., Melliar-Smith, P.M., and Ramakrishna, Y.S., "A Graphical Interval Logic for Specifying Concurrent Systems," *ACM Transactions on Software Engineering and Methodology*, vol. 3, no. 2, April 1994, pp. 131-165.
- [Ding93] Ding, S., and Katayama, T., "Specifying Reactive Systems with Attributed Finite State Machines," *Proc. of the 7th Intl. Workshop on Software Specification and Design*, Dec. 1993, pp. 90-99.
- [Dong97a] Dong, J. S., and Zucconi, L., "A Framework for Adding Time into Formal Object Models," *Proc. of the 3rd Intl. Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 1997, pp. 26-31.
- [Dong97b] Dong, J. S., Zucconi, L., and Duke, R., "Specifying Parallel and Distributed Systems in Object-Z: The Lift Case Study," *Proc. of the 2nd Intl. Workshop on Software Engineering for Parallel and Distributed Systems*, May 1997, pp. 140-149.
- [Douglass98] Douglass, B.P., *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley Longman, 1998.
- [Douglass99] Douglass, B.P., *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*, Addison-Wesley Longman, 1999.

- [Duke89] Duke, R., and Smith, G., "Temporal Logic and Z Specifications," *Australian Computer Journal*, vol. 21, no. 2, May 1989, pp 62-69.
- [Duke94] Duke R., Rose G., and Smith, G., "Object-Z: A Specification Language Advocated for the Description of Standards," TR 94-45, Software Verification Research Centre, School of Information Technology, The University of Queensland, Brisbane, Australia, Dec.1994, accessed September 1998 at <http://svrc.it.uq.edu.au/Bibliography/svrc-tr.html?94-45>
- [Duval97] Duval, G., and Cattel, T., "From Architecture Down to Implementation of Safe Process Control Applications-The Lift Case Study," *Proc. of the 13th Hawaii Intl. Conf. on Systems Sciences*, Jan. 1997, pp. 24-32.
- [Easterbrook98] Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D., "Experiences Using Lightweight Formal Methods for Requirements Modeling," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, Jan. 1998, pp. 4-14.
- [Evans97] Evans, A., "An Improved Recipe for Specifying Reactive Systems in Z," *Z User Workshop*, Reading, UK, 1997, accessed Feb. 15, 1999 at <http://www.student.comp.brad.ac.uk/~asevans1/z.html>
- [Evans98] Evans, A., "Reasoning with UML Class Diagrams," *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Oct. 1998, pp. 102-113.
- [Evans99] Evans, A., and Wellings, A.J., "UML and the Formal Development of Safety Critical Real-Time Systems," *IEE Colloquium on Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, Jan. 1999, pp. 2/1-2/4.
- [Everett95] Everett, W., and Honiden, S., "Reliability and Safety of Real-Time Systems," *IEEE Software*, vol. 12, no. 3, May 1995, pp. 13-16.
- [Favre99] Favre, L., and Clerici, S., "Integrating UML and Algebraic Specification Techniques," *Proc. of the 32nd Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-32)*, Nov. 1999, pp. 151-162.
- [Feather98] Feather, M., "Rapid Application of Lightweight Formal Methods for Consistency Analysis," *IEEE Transactions on Software Engineering*, vol. 24, no. 11, Feb. 1998, pp. 949-959.
- [Fernandes00] Fernandes, J.M., Machado, R.J., and Santos, H.D., "Modeling Industrial Embedded Systems with UML," *Proc. of the 8th Intl. Workshop on Hardware/Software Codesign (CODES 2000)*, May 2000, pp. 18-22.
- [Fidge97] Fidge, C., Kearney, P., and Utting, M., "A Formal Method for Building Concurrent Real-Time Software," *IEEE Software*, vol. 14, no. 2, March/April 1997, pp. 99-106.
- [Formaliser01] "Formaliser," accessed Feb. 7, 2001 at Logica's web-site <http://public.logica.com/~formaliser/formlsr/formlsr.htm>
- [France97] France, R.B., Bruel, J.-M., and Raghavan, G., "Taming the Octopus: Using Formal Models to Integrate the Octopus Object-Oriented Analysis Models," *Proc. of the High-Assurance Engineering Workshop*, Aug. 1997, pp. 8-13.

- [Fraser94] Fraser, M.D., Kumar, K., and Vaishnavi, V.K., "Strategies for Incorporating Formal Specifications in Software Development," *Communications of the ACM*, vol. 37, no. 10, Oct. 1994, pp. 74-85.
- [Gaudel94] Gaudel, M.C., "Formal Specification Techniques," *Proc. of the 16th Intl. Conf. on Software Engineering*, Sorrento, Italy, May 1994, pp. 223-227.
- [Gerhart94] Gerhart, S., Craigen, D., and Ralston, T., "Experience with Formal Methods in Critical Systems," *IEEE Software*, vol. 11, no. 1, Jan. 1994, pp. 21-28.
- [Gibbs94] Gibbs, W.W., "Software's Chronic Crisis," *Scientific American*, Sep. 1994, pp. 86-95.
- [Glass96] Glass, R.L., "Formal Methods Are a Surrogate for a More Serious Software Concern," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 19-20.
- [Gomaa00] Gomaa, H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, 2000.
- [Gosling96] Gosling, J., and Steele, G., *The Java Language Specification*, Addison-Wesley, 1996.
- [Graw00] Graw, G., Herrmann, P., and Krumm, H., "Verification of UML-based Real-Time Systems by Means of cTLA," *Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, March 2000, pp. 86-95.
- [Green96] Green, T.R.G., and Petre, M., "Usability Analysis of Visual Environments: A 'Cognitive Dimensions' Framework," *Journal of Visual Languages and Computing*, vol. 7, no. 2, June 1996, pp. 131-174.
- [Gutttag85] Gutttag, J.V., Horning, J.J., and Wing, J., "The Larch Family of Specification Languages," *IEEE Software*, vol. 2, no. 3, May 1985, pp. 24-36.
- [Gutttag93] Gutttag, J.V., and Horning, J.J. (editors), *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
- [Hall90] Hall, A., "Seven Myths of Formal Methods," *IEEE Software*, vol. 7, no. 5, Sep. 1990, pp. 11-19.
- [Hall96] Hall, A., "What Is the Formal Methods Debate About?," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 22-23.
- [Hall98] Hall, A., "What Does Industry Need from Formal Specification Techniques?," *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Oct. 1998, pp. 2-7.
- [Harel87] Harel, D., "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming*, vol. 8, no. 3, June 1987, pp. 231-274.
- [Harel88] Harel, D., "On Visual Formalisms," *Communications of the ACM*, vol. 31, no. 5, May 1988, pp. 514-530.

- [Harel96] Harel, D., "The STATEMATE Semantics of Statecharts," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 4, Oct. 1996, pp. 293-333.
- [He96] He, J., Hoare, C.A.R., Muller-Olm, M., Olderog, E.-R., Schenke, M., Hansen, M.R., Ravn, A.P., and Rischel, H., "The ProCoS Approach to the Design of Real-Time Systems: Linking Different Formalisms," *Tutorial Papers, Formal Methods Europe '96*, April 1996, accessed April 4, 2001 at <http://www.cs.auc.dk/~apr/pub/pub.html>
- [Hinchey96] Hinchey, M.G., "To Formalize or not To Formalize?," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 18-19.
- [Hoare78] Hoare, C.A.R., "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, p. 666-677.
- [Hoare85] Hoare, C.A.R., *Communicating Sequential Processes*, Prentice Hall, 1985.
- [Holibaugh93] Holibaugh, R., "Joint Integrated Avionics Working Group (JIAWG) Object-Oriented Domain Analysis Method (JODA)," Special Report CMU/SEI-92-SR-3, version 3.1, Software Engineering Institute, Carnegie Mellon University, Nov. 1993.
- [Holloway96] Holloway, C.M., and Butler, R.W., "Impediments to Industrial Use of Formal Methods," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 25-26.
- [Howerton00] Howerton, W.G., and Hinchey, M.G., "Using the Right Tool for the Job," *Proc. of the 6th IEEE Intl. Conf. on the Engineering of Complex Computer Systems*, Sept. 2000, pp. 105-115.
- [Ionescu93] Ionescu, T., and Dascalu, S.M., "Algorithm and System for Automatic Camshaft Testing," in P. Kopacek (editor), *A Cost Effective Use of Computer-Aided Technologies and Integration Methods in Small and Medium Sized Companies: IFAC Workshop, Vienna, Austria, Sep. 1992*, Pergamon Press, 1993, pp. 131-136.
- [ISO89] International Standard ISO 8807, *LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, ISO, 1989.
- [Jackson96a] Jackson, D., "Requirements Need Form, Maybe Formality," *IEEE Software*, vol. 13, no. 2, March 1996, pp. 20-22.
- [Jackson96b] Jackson, D., and Wing, J., "Lightweight Formal Methods," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 21-22.
- [Jackson00a] Jackson, D., Schechter, I., and Shlyakhter, I., "Alcoa: The Alloy Constraint Analyzer," Laboratory of Computer Science, Massachusetts Institute of Technology, March 27, 2000, accessed Feb. 6, 2001 at <http://sdg.lcs.mit.edu/~dnj/pubs/alcoa-overview.pdf>
- [Jackson00b] Jackson, D., "Alloy: A Lightweight Object Modelling notation," Laboratory of Computer Science, Massachusetts Institute of Technology, July 28, 2000, accessed Feb. 7, 2001 at <http://sdg.lcs.mit.edu/~dnj/pubs/alloy-journal.pdf>
- [Jacky97] Jacky, J., *The Way of Z: Practical Programming with Formal Methods*, Cambridge University Press, 1997.

- [Jacobson94] Jacobson, I., *Object-Oriented Software Engineering: A Use-Case Driven Approach*, Addison-Wesley, 1994.
- [Jacobson99] Jacobson, J., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, 1999.
- [Jahanian86] Jahanian, F. and Mok, A.K., "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 12, no. 9, Sep. 1986, pp. 890-904.
- [Jahanian88] Jahanian, F., Lee, R., and Mok, A.K., "Semantics of Modecharts in Real-Time Logic," *Proc. of the 21st Annual Hawaii Intl. Conf. on System Sciences*, Software Track, Jan. 1988, vol. 2, pp. 479-489.
- [Jahanian94] Jahanian, F. and Mok, A.K., "Modechart: A Specification Language for Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 12, Dec. 1994, pp. 933-947.
- [JavaLook01] "Java Look and Feel Graphics Repository," a Sun Microsystems web-page, accessed April 10, 2001 at <http://developer.java.sun.com/developer/techDocs/hi/repository>
- [Jia97] Jia, X., "A Pragmatic Approach to Formalizing Object-Oriented Modeling and Development," *Proc. of the 21st Annual Intl. Conf. on Computer Software and Application (COMPSAC'97)*, Aug. 1997, pp. 240-245.
- [Jia98a] Jia, X., "ZTC: A Type-Checker for Z notation, User's Guide," version 2.03, Aug. 1998, accessed Feb. 7, 2001 at <http://se.cs.depaul.edu/fm/Papers/guide.ps>
- [Jia98b] Jia, X., "A Tutorial of ZANS -- A Z Animation System," Release 0.31, July 1998, accessed Feb. 7, 2001 at <http://se.cs.depaul.edu/fm/Papers/zanstut3.ps>
- [Jia98c] Jia, X., and Skevoulis, S., "Code Synthesis Based on Object-Oriented Design Models and Formal Specifications," *Proc. of the 22nd Annual Intl. Conf. on Computer Software and Applications*, Aug. 1998, pp. 393-398.
- [Jigorea00] Jigorea, R., Manolache, S., Eles, P., and Peng, Z., "Modelling of Real-Time Embedded Systems in an Object-Oriented Design Environment with UML," *Proc. of the 3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, March 2000, pp. 210-213.
- [Johnson95] Johnson, C.W., "Using Z to Support the Design of Interactive Safety-Critical Systems," *Software Engineering Journal*, vol. 10, no. 2, March 1995, pp. 49-60.
- [Johnson96] Johnson, C.W., "Literate Specifications," *Software Engineering Journal*, vol. 11, no. 4, July 1996, pp. 225-237.
- [Johnson00] Johnson, R.A., "The Ups and Downs of Object-Oriented Systems Development," *Communications of the ACM*, vol. 43, no. 10, Oct. 2000, pp. 69-73.
- [Jones90] Jones, C.B., *Systematic Software Development Using VDM*, Second Edition, Prentice-Hall, 1990.

- [Jones96] Jones, C.B., "A Rigorous Approach to Formal Methods," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 20-21.
- [Joyce94] J. Joyce, N. Day, and M. Donat, "S: A Machine Readable Specification Notation Based on Higher Order Logic," *Proc. of the 1994 Intl. Meeting on Higher Order Logic Theorem Proving and its Applications*, Lecture Notes in Computer Science, vol. 859, Springer-Verlag, 1994, pp. 285-299.
- [Kapur00] Kapur, D., "The Use of Formal Methods in Hardware and Software Cannot Be Abandoned," *Proc. of the 5th Symposium on High Assurance Systems Engineering*, Nov. 2000, pp. 142-143.
- [Kelley-Sobel00] Kelley-Sobel, A., E., "Empirical Results of A Software Engineering Curriculum Incorporating Formal Methods," *Proc. of the 31st SIGCSE Technical Symposium on Computer Science Education*, 2000, pp. 157-161.
- [Kent97] Kent, S., "Constraint Diagrams: Visualizing Invariants in Object-Oriented Models," *Proc. of OOPSLA97*, ACM SIGPLAN Notices, vol. 32, no. 10, ACM Press, Oct. 1997, pp. 327-341.
- [Kent98] Kent, S., and Gil, Y., "Visualizing Action Constraints in Object-Oriented Modelling," *IEE Proceedings: Software*, vol. 145, no. 2-3, April 1998, accessed March 3, 2001 at <http://www.cs.ukc.ac.uk/pubs/1998/786/index.html>, pp. 70-78.
- [Kent99] Kent, S., Gaito, S., and Ross, N., "A Meta-model Semantics for Structural Constraints in UML," Chapter 9 in Kilov, H., Rumpe, B., and Simmonds, I. (editors), *Behavioral Specifications for Businesses and Systems*, Kluwer Academic Publishers, Sep. 1999, pp. 123-141.
- [Kesten92] Kersten, Y., and Pnueli, A., "Timed and Hybrid Statecharts and Their Textual Representation," in J. Vytopil (editor), *Formal Techniques in Real-Time and Fault-Tolerant Systems, 2nd Intl. Symposium*, Lecture Notes in Computer Science, vol. 571, Springer-Verlag, Berlin, 1992, pp. 591-620.
- [Kim99a] Kim, S.-K., and Carrington, D., "Formalizing the UML Class Diagram Using Object-Z," in France R., and Rumpe, B. (editors), *2nd Intl. Conf. on UML*, Lecture Notes in Computer Science, vol. 1723, Springer-Verlag, Berlin, Oct. 1999, pp. 83-98.
- [Kim99b] Kim, S.-K., and Carrington, D., "Visualization of Formal Specifications," *Proc. of the 6th Asia Pacific Software Engineering Conf. (ASPEC '99)*, Dec. 1999, pp. 102-109.
- [Kim00a] Kim, S.-K., and Carrington, D., "A Formal Mapping between UML Models and Object-Z Specifications," in Bowen, J.P., Dunne, S., Galloway, A., and King, S. (editors), *Intl. Conf. of B and Z Users ZB2000*, Lecture Notes in Computer Science, vol. 1878, Springer-Verlag, Berlin, Feb. 2000, pp. 2-21.
- [Kim00b] Kim, S.-K., and Carrington, D., "An Integrated Framework with UML and Object-Z for Developing A Precise and Understandable Specification: The Light Control Case Study," *Proc. of the 7th Asia-Pacific Software Engineering Conf. ASPEC 2000*, Dec. 2000, pp. 240-248.

- [Kishida96] Kishida, K., "Managing Megaprojects: A Free-Form Approach," *IEEE Software*, vol. 13, no. 4, July 1996, pp. 28-30.
- [Knuth73] Knuth, D.E., *The Art of Computer Programming*, Second Edition, Addison-Wesley, 1973.
- [Kobryn99] Kobryn, C., "UML 2001: A Standardization Odyssey," *Communications of the ACM*, vol. 42, no. 10, Oct. 1999, pp. 29-37.
- [Kopetz97] Kopetz, H., *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.
- [Kortright97] Kortright, E.V., "Modeling and Simulation with UML and Java," *Proc. of the 30th Annual Simulation Symposium*, April 1997, pp. 43-48.
- [Laleau00] Laleau, R., and Mammar, A., "An Overview of a Method and its Support Tool for Generating B Specifications from UML Notations," *Proc. of the 15th IEEE Intl. Conf. on Automated Software Engineering*, Sep. 2000, pp. 269-272.
- [Lano91] Lano, K., "Z++, an Object-Oriented Extension to Z," in *Z User Workshop, Oxford 1990*, Springer-Verlag Workshops in Computing, 1991, pp.151-172.
- [Lano94a] Lano, K., and Haughton, H. (editors), *Object-Oriented Specification Case Studies*, Prentice Hall, 1994.
- [Lano94b] Lano, K., and Haughton, H., "A Comparative Description of Object-Oriented Specification Languages," Chapter 2 in Lano, K., and Haughton, H. (editors), *Object-Oriented Specification Case Studies*, Prentice Hall, 1994, pp. 20-54.
- [Lano94c] Lano, K., and Haughton, H., "Object-Oriented Specification Languages in the Software Life Cycles," Chapter 3 in Lano, K., and Haughton, H. (editors), *Object-Oriented Specification Case Studies*, Prentice Hall, 1994, pp. 55-79.
- [Lano94d] Lano, K., and Haughton, H., " Specifying A Concept-recognition System in Z++ ," Chapter 7 in Lano, K., and Haughton, H. (editors), *Object-Oriented Specification Case Studies*, Prentice Hall, 1994, pp 137-157.
- [Lano94e] Lano, K., and Haughton, H., "The Z++ Manual," version Oct. 25, 1994, accessed Jan. 10, 2001 at www.dc.uba.ar/people/materias/isoft1/Z/papers/z++.pdf
- [Lano95] Lano, K., *Formal Object-Oriented Development*, Springer-Verlag, 1995.
- [Lano98] Lano, K., and Bicarregui, J., "Formalising the UML in Structured Temporal Theories," *Proc. of the 2nd ECOOP Workshop on Precise Behavioral Semantics*, July 1998, pp. 105-121.
- [Larsen96] Larsen, P.G., Fitzgerald, J., Brookes, T., "Applying Formal Specification in Industry," *IEEE Software*, vol. 13, no. 3, May 1996, pp. 48-56.
- [Lawrence96] Lawrence, B., "Do You Really Need Formal Requirements?," *IEEE Software*, vol. 13, no. 2, March 1996, pp. 20-22.

- [Lee95] Lee, J., Pan, J.I., and Huang, W.T., "Integrating Object-Oriented Requirements Specifications with Formal Notations," *Proc. of the 7th Intl. Conf. on Tools with Artificial Intelligence*, Nov. 1995, pp. 34-41.
- [Léonard98] Léonard, L., and Leduc, G., "A Formal Definition of Time in LOTOS," *Formal Aspects of Computing*, vol. 10, no. 3, June 1998, pp. 248-266.
- [LeosIcons01] "Leo's Icons Archive," accessed Jan. 10, 2001 at <http://www.iconarchive.com/> (> Computer Icons > Misc. Comp. Icons I)
- [Leung96] Leung, K.R.P.H., and Chan, D.K.C., "Extending Statecharts with Duration," *Proc. of the 20th Intl. Conf. on Computer Software and Applications (COMPSAC'96)*, Aug. 1996, pp. 246-251.
- [Leveson86] Leveson, N.G., "Software Safety: Why, What, and How?," *ACM Computing Surveys*, vol. 18, no. 2, June 1986, pp. 125-163.
- [Lin92] Lin, K.J., and Burke, E.J., "Coming to Grips with Real-Time Realities," *IEEE Software*, vol. 9, no. 5, Sep. 1992, pp. 12-15.
- [Lin94] Lin, K.J., and Son, S.H., "Real-Time Databases: Characteristics and Issues," *Proc. of the First Workshop on Object-Oriented Real-Time Dependable Systems*, Oct. 1994, pp. 113-116.
- [Liu97] Liu, X., Yang, H., and Zedan, H., "Formal Methods for the Re-Engineering of Computing Systems: A Comparison," *Proc. of the 21st Intl. Conf. on Computer Software and Applications (COMPSAC '97)*, Aug. 1997, pp. 409-414.
- [Logica01] "Formal Methods Tools and Services," a Logica web-site, last updated Feb. 15, 2001, accessed April 10, 2001 at <http://public.logica.com/-formaliser/>
- [Lu99] Lu, M., Zhao, Z., and Li, M., "Object-Oriented Requirements Modeling Based on UML," *Proc. of the 31st Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-31)*, Sep. 1999, pp. 133-140.
- [Mahony92] Mahony, B.P., and Hayes, I.J., "A Case-Study in Timed Refinement: A Mine Pump," *IEEE Transactions on Software Engineering*, vol. 18, no. 9, Sept. 1992, pp. 817-826.
- [Mahony98] Mahony, B., and Dong, J.S., "Blending Object-Z and Timed CSP: An Introduction to TCOZ," *Proc. of the 1998 Intl. Conf. on Software Engineering*, April 1998, pp. 95-104.
- [Mahony00] Mahony, B., and Dong, J.S., "Timed Communicating Object Z," *IEEE Transactions on Software Engineering*, vol. 26, no. 2, Feb. 2000, pp. 150-176.
- [Manna81] Manna, Z. and Pnueli, A., "Verification of Concurrent Programs: The Temporal Framework," in Boyer, R.S., and Moore, J.S. (editors), *The Correctness Problem in Computer Science*, Academic Press, New York, 1981, pp. 215-273.
- [Mathai96] Mathai, J., (editor), *Real-Time Systems: Specification, Verification and Analysis*, Prentice-Hall, 1996.

- [McUmbler99] Mc-Umbler, W.E., and Cheng, B.H., "UML-Based Analysis of Embedded Systems Using a Mapping to VHDL," *Proc. of the 4th IEEE Intl. Symposium on High-Assurance Systems Engineering*, Nov. 1999, pp. 56-63.
- [Meisels97] Meisels, I., "Software Manual for Windows Z/EVES Version 1.5 and the Z Browser," Technical Report 97-5505-04e, ORA Canada, Sep. 1997.
- [Meyer97] Meyer, B., "The Next Software Breakthrough," *IEEE Computer*, vol. 30, no. 7, July 1997, pp. 113-114.
- [Meyer99] Meyer, B., "A Really Good Idea," *IEEE Computer*, vol. 32, no. 12, Dec. 1999, pp. 144-147.
- [Milner80] Milner, R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, 1980.
- [Moller92] Moller, F., and Tofts, C., "An Overview Of TCCS," *Proc. of the 4th EUROMICRO Workshop on Real-Time Systems*, June 1992, pp. 98-103.
- [Morgan94] Morgan, N.W., and Schahczenski, C., "Transitioning to Rigorous Software Specification," *Proc. of the First Intl. Conf. on Requirements Engineering*, April 1994, pp. 6-15.
- [Moszkowski86] Moszkowski, B., *Executing Temporal Logic Programs*, Cambridge University Press, 1986.
- [Mrva97] Mrva, M., "Reuse Factors in Embedded Systems Design," *IEEE Computer*, vol. 30, no. 8, Aug. 1997, pp. 93-95.
- [Mughal00] Mughal, K.A., and Rasmussen, R., *A Programmer's Guide to Java Certification: A Comprehensive Primer*, Addison-Wesley, 2000.
- [Muller98] Muller, R., *Database Design for Smarties: Using UML for Data Modeling*, Morgan Kaufman, 1998.
- [Narayan93] Narayan, S., and Gajki, D.D., "Features Supporting System-Level Specification in HDLs," *Proc. of the European Design Automation Conf. (EURO-DAC'93)*, Sep. 1993, pp. 540-545.
- [Narayan96] Narayan, S., "Requirements for Specification of Embedded Systems," *Proc. of the 9th IEEE Intl. ASIC Conf. and Exhibit*, Sep. 1996, pp. 133-137.
- [Natarajan92] Natarajan, S., and Zhao, W., "Issues in Building Dynamic Real-Time Systems," *IEEE Software*, vol. 9, no. 5, Sep. 1992, pp. 16-21.
- [Neil98] Neil, M., Ostrolenk, G., Tobin, M., and Southworth, M., "Lessons from Using Z to Specify a Software Tool," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, Jan. 1998, pp.15-23.
- [Nguyen96] Nguyen, K., "Towards a Practical Formal Method for Object-Oriented Modelling," *Proc. of the Asia-Pacific Software Engineering Conf.*, Dec. 1996, pp. 226-237.

- [Niemann99] Niemann, T., "Nuts to OOP!," *Embedded Systems Programming*, vol. 12, no. 8, Aug. 1999, accessed Feb. 12, 2001, at <http://www.embedded.com/1999/9908/9908feat1.htm>
- [Nix88] Nix, C., J., and Collins, B.P., "The Use of Software Engineering, Including the Z Notation, in the Development of CICS," *Quality Assurance*, vol. 14, no. 3, Sep. 1988, pp. 103-110.
- [Noe00] Noe, P.A., and Hartrum, T.C., "Extending the Notation of Rational Rose 98 for Use with Formal Methods," *Proc. of the IEEE National Aerospace and Electronics Conf. (NAECON 2000)*, Oct. 2000, pp. 43-50.
- [Oldevik98] Oldevik, J., and Berre, A.-J., "UML-Based Methodology for Distributed Systems," *Proc. of the Second Intl. Workshop on Enterprise Distributed Object Computing (EDOC'98)*, Nov. 1998, pp. 2-13.
- [Ostroff89] Ostroff, J.S., *Temporal Logic for Real Time Systems*, John Wiley and Sons, 1989.
- [Page-Jones99] Page-Jones, M., *Fundamentals of Object-Oriented Design in UML*, Addison-Wesley, New-York, 1999.
- [Paige98] Paige, R. F., "Heterogeneous Notations for Pure Formal Method Integration," *Formal Aspects of Computing*, vol. 10, no. 3, June 1998, pp. 233-242.
- [Paige99] Paige, R.F., "When Are Methods Complementary?," *Information and Software Technology*, vol. 41, no. 3, Feb. 1999, pp. 157-162.
- [ParadigmPlus01] "Paradigm Plus – Enterprise Component Modeling," Computer Associates International, Inc. web-site, accessed Jan. 27, 2001 at http://www.cai.com/products/alm/paradigm_plus.htm
- [Parnas96] Parnas, D.L., "Mathematical Methods: What We Need and Don't Need," *IEEE Computer*, vol. 29, no. 4, April 1996, pp. 28-29.
- [Periyasamy97] Periyasamy, K., and Alagar, V.S., "Extending Object-Z for Specifying Real-Time Systems," *Proc. of the 23rd Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-23)*, July 1997, pp. 163-175.
- [Periyasamy98] Periyasamy, K., and Alagar, V.S., "Adding Real-Time Filters to Object-Oriented Specification of Time Critical Systems," *Proc. of the 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, Oct. 1998, pp. 28-39.
- [Petri62] Petri, C.A., *Kommunikation mit Automaten* (in German), PhD Dissertation, University of Bonn, Germany, 1962.
- [Pnueli77] Pnueli, A., "Temporal Logics of Programs," *Proc. of the 18th IEEE Annual Symposium on Foundations of Computer Science*, Oct. 1977, pp. 46-57.
- [Polack92] Polack, F., "Integrating Formal Notations and Systems Analysis: Using Entity Relationship Diagrams," *Software Engineering Journal*, vol. 7, no. 5, Sep. 1992, pp. 363-371.
- [Price99] Price, R., Srinivasan, B., and Ramamohanarao, K., "Extending the Unified Modeling Language to Support Spatiotemporal Applications," *Proc. of the 32nd Intl.*

- Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-32)*, Nov. 1999, pp. 163-174.
- [pUML01a] "The Precise UML Group – Main Details," pUML Group web-site, accessed Jan. 28, 2001 at <http://www.cs.york.ac.uk/puml/maindetails.html>
- [pUML01b] "The Precise UML Group – Publications," pUML group web-site, accessed Jan. 28, 2001 at <http://www.cs.york.ac.uk/puml/publications.html>
- [Quatrani98] Quatrani, T., *Visual Modeling with Rational Rose and UML*, Addison Wesley Longman, 1998.
- [Ramchadani74] Ramchadani, C., *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets*, PhD thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering, Feb. 1974.
- [RationalRose01] "Rational Rose, A Rational Suite Product," Version 2001, Rational Software Corporation web-site, accessed Jan. 27, 2001 at <http://www.rational.com/products/rose/index.jsp>
- [RationalRoseRT01] "Rational Rose Real Time," Version 2001, Rational Software Corporation's web-site, accessed Jan. 27, 2001 at <http://www.rational.com/products/rosert/index.jsp>
- [Reisig85] Reisig, W., *Petri Nets: An Introduction*, Springer-Verlag, 1985.
- [Rescher71] Rescher, N., and Urquhart, A., *Temporal Logic*, Springer-Verlag, 1971.
- [Rhapsody01] "Rhapsody Overview," I-Logix, Inc. web-site, accessed Jan. 25, 2001 at http://www.ilogix.com/fs_about.htm, <Rhapsody> link.
- [RogersGifs01] "Creative Design Icon Archive," part of the Creative Design Clipart Gallery, RogersGifs.com, accessed Jan. 15, 2001 at <http://www.rogersgifs.com/iconmaster>, (Gallery Index 35).
- [Roman96] Roman, G.C., Hart, D., and Calkins, C., "Visual Presentation of Software Specifications and Designs," *Proc. of the 8th Intl. Workshop on Software Specification and Design*, March 1996, pp. 115-124.
- [RoZeLink99] "RoZeLink, Product Description," 1999 version, Headway Software Inc.'s web site, accessed May 1999 at <http://indigo.ie/-chrisc>
- [Rumbaugh91] Rumbaugh, J., Blaha, M., Premerlani, W., Frederick, E., and Lorenzen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [Rushby00] Rushby, J., "Disappearing Formal Methods," *Proc. of the 5th Symposium on High Assurance Systems Engineering*, Nov. 2000, pp. 95-96.
- [Sahraoui92] Sahraoui, N, and Delfieu, D., "Zaman, A Simple Language for Expressing Timing Constraints," *Proc. of the 18th IFIP-IFAC Workshop on Real-Time Programming*, June 1992, pp. 19-24.
- [Sahraoui97] Sahraoui, N., "Applying Specification Methods to Complex Systems," *Proc. of the IEEE Intl. Conf. on Systems, Man, and Cybernetics*, Oct. 1997, vol. 5, pp. 4488-4491.

- [Salek94] Salek, A., Sorenson, P.G., Tremblay, J.P., and Punshon, J.M., "The REVIEW System: From Formal Specifications to Natural Language," *Proc. of the First Intl. Conf. on Requirements Engineering*, Apr. 1994, pp. 220-229.
- [Schach99] Schach, S., *Classical and Object-Oriented Software Engineering with UML and Java*, WCB/McGraw-Hill, 1999.
- [Schneider92] Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., and Roscoe, A.W., "Timed CSP: Theory and Applications," *Lecture Notes in Computer Science*, vol. 600, Springer-Verlag, 1992, pp. 640-675.
- [Scholefield92] Scholefield, D.J., and Zedan, H.S.M., "The Refinement of Real-Time Systems," *Proc. of the 4th EUROMICRO Workshop on Real-Time Systems*, June 1992, pp. 122-127.
- [Scogings01] Scogings, C., and Phillips, C., "A Method for the Early Stages of Interactive System Design Using UML and Lean Cuisine+," *Proc. of the Second Australasian User Interface Conf. (AUIC 2001)*, Jan. 2001, pp. 69-76.
- [Scott99] Scott, L.P., and da Graça Pimentel, M., "An Object-Oriented Model for HyTime Using UML," *Proc. of the Third Intl. Conf. on Computational Intelligence and Multimedia Applications*, Sep. 1999, pp. 393-398.
- [Selic94] Selic, B., Gullekson, G., and Ward, P.T., *Real-Time Object-Oriented Modeling*, John Wiley & Sons, 1994.
- [Selic96] Selic, B., "Modeling Real-Time Distributed Software Systems," *Proc. of the 4th Intl. Workshop on Parallel and Distributed Real-Time Systems*, April 1996, pp. 11-18.
- [Selic98] Selic, B., "Animating Structures: Real-Time, Objects, and the UML," Keynote Talk, *Proc. of the 19th IEEE Real-Time Systems Symposium*, Dec. 1998, pp. 165.
- [Selic99a] Selic, B., "Turning Clockwise: Using UML in the Real-Time Domain," *Communications of the ACM*, vol. 42, no. 10, Oct. 1999, pp. 46-54.
- [Selic99b] Selic, B., "Using UML for Modeling Complex Real Time System Architectures" (a 1999 PowerPoint presentation), ObjectTime Limited web-site, accessed Jan. 8, 2001 at <http://www.objecttime.com/otl/technical/umlrt.html>
- [Shaw92] Shaw, A.C., "Communicating Real-Time State Machines," *IEEE Transactions on Software Engineering*, vol. 18, no. 9, Sep. 1992, pp. 805-816.
- [Shlaer88] Shlaer, S., and Mellor, S.J., *Object-Oriented Systems Analysis: Modeling the World in Data*, Prentice-Hall, 1988.
- [Shlaer91] Shlaer, S., and Mellor, S.J., *Object Lifecycles: Modeling the World in States*, Prentice-Hall, 1991.
- [Shroff97] Shroff, M., and France, R.B., "Towards a Formalization of UML Class Structures in Z," *The 21st Annual Intl. Conf. on Computer Software and Applications (COMPSAC'97)*, Aug. 1997, pp. 646-651.

- [Si-Alhir98] Si Alhir, S., *UML In A Nutshell: A Desktop Quick Reference*, O'Reilly & Associates, 1998.
- [Simons99] Simons, A.J.H., "Use Cases Considered Harmful," *Proc. of the 29th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-29 Europe)*, June 1999, pp. 194-203.
- [Sommerville95] Sommerville, I., *Software Engineering*, Fifth Edition, Addison-Wesley, 1995.
- [Sowmya98] Sowmya, A., and Ramesh, S., "Extending Statecharts with Temporal Logic," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, March 1998, pp. 216-231.
- [Spivey92] Spivey, J.M., *The Z Notation: A Reference Manual*, Second Edition, Prentice-Hall International, UK, 1992.
- [Stankovic88] Stankovic, J.A., and Ramamritham, K., *Tutorial: Hard-Real Time Systems*, Computer Society Press of the IEEE, 1988.
- [Stankovic96a] Stankovic, J.A., "Real-Time and Embedded Systems," *ACM Computing Surveys*, vol. 28, no. 1, March 1996, pp. 205-208.
- [Stankovic96b] Stankovic, J.A., et al., "Strategic Directions in Real-Time and Embedded Systems," *ACM Computing Surveys*, vol. 28, no. 4, Dec. 1996, pp. 751-763.
- [Steggles94] Steggles, P., and Hulance, J., "Z Tools Survey," June 1994, accessed February 1998 at <ftp://ftp.ist.co.uk/pub/doc/zola/ztool-survey.ps>
- [Stepney92a] Stepney, S., Barden, R., and Cooper, D. (editors), *Object-Orientation in Z*, Workshops in Computings, Springer-Verlag, 1992.
- [Stepney92b] Stepney, S., Barden, R., and Cooper, D., "A Survey of Object-Orientation in Z," *Software Engineering Journal*, vol. 7, no. 2, March 1992, pp. 150-160.
- [Stoeklin98] Stoeklin, S., Williams, D.D., and Swain, R., "Understanding Object-Oriented Systems Specifications Using Familiar Systems," *Proc. of the Intl. Conf. on Software Engineering: Education & Practice*, Jan. 1998, pp. 10-15.
- [Stroustrup97] Stroustrup, B., *The C++ Programming Language*, Third Edition, Addison-Wesley, 1997.
- [SystemArchitect01] "System Architect 2001," Popkin Software web-site, accessed Jan. 27, 2001 at <http://www.popkin.com/products/sa2001/systemarchitect.htm>
- [Taentzer99] Taentzer, G., "Adding Visual Rules to Object-Oriented Modeling Techniques," *Proc. of the 29th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-29 Europe)*, June 1999, pp. 275-284.
- [Taylor99] Taylor, D.A., "Programming for Everyone," *IEEE Computer*, vol. 32, no. 5, May 1999, pp. 50-51.
- [TogetherSoft00a] "Practical UML: A Hands-on Introduction for Developers," TogetherSoft Corporation web-site, Oct. 16, 2000 revision, accessed Jan. 9, 2001 at <http://www.togethersoft.com/services/UMLShortCourse/index.html>

- [TogetherSoft00b] "Together Product Feature Chart," TogetherSoft Corporation's web-site, Dec. 18, 2000 update, accessed Jan. 26, 2001 at <http://www.togethersoft.com/together/matrix.html>
- [UML00] *OMG Unified Modeling Language Specification*, version 1.3, Object Management Group web site, published March 1, 2000, accessed Sep. 23, 2000 at <ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf>
- [Utting95] Utting, M., "Animating Z: Interactivity, Transparency and Equivalence," *Proc. of the 1995 Asia Pacific Conf.*, Dec. 1995, pp. 294-303.
- [Vishnuvajjala96] Vishnuvajjala, R.V., Tsai, W.-T., Mojdehbakhsh, R., and Elliott, L., "Specifying Timing Constraints in Real-Time Object-Oriented Systems," *Proc. of the High-Assurance Systems Engineering Workshop*, Oct. 1996, pp. 32-39.
- [Visio00] "Microsoft Visio Overview Tour," Microsoft Corporation web-site, accessed Nov. 20, 2000 at <http://www.microsoft.com/office/visio/overview.htm>
- [Warmer98] Warmer, J., and Kleppe, A., *The Object Constraint language: Precise Modeling with UML*, Addison-Wesley, 1998.
- [Watkins98] Watkins, S., Dick, M., and Thompson, D., "From UML to IDL: A Case Study," *Proc. of the 28th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-28)*, Nov. 1998, pp. 141-153.
- [Wing90] Wing, J.M., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, vol. 23, no. 9, Sep. 1990, pp. 8-24.
- [Wizard01] "Wizard, A Type-Checker for Object-Z Specifications," Software Verification Research Center's web-site, University of Queensland, Brisbane, Australia, accessed Feb. 6, 2001 at <http://svrc.it.uq.edu.au/Object-Z/pages/Wizard.html>
- [Wordsworth92] Wordsworth, J.B., *Software Development With Z*, Addison-Wesley, 1992.
- [Xie99] Xie, Z., Yu, J., and Liu, J., "Applying UML to Gas Turbine Engine Simulation," *Proc. of the 31st Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-31)*, Sep. 1999, pp. 458-464.
- [Xu00] Xu, R., Masaru, Z., and Zhang, H.-Q., "Object-Oriented AGVS Modeling with UML," *Proc. of the 39th SICE Annual Conference, Intl. Session Papers*, July 2000, pp. 261-264.
- [Yang96] Yang, S.M., Yoon, T.M., and Kim, M.H., "System Development Based On A Real-Time Object Model," *Proc. of the 2nd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'96)*, Feb. 1996, pp. 152-159.
- [Yuan98] Yuan, X., Hu, D., Hao, Xu, H., Li, Y., and Zheng, G., "Complete Object-Oriented Z and Its Supporting Environment COOZ-Tools," *Proc. of 27th Intl. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS-27)*, Sept. 1998, pp. 206-213.
- [Zadeh96] Zadeh, H.B., and Stepney, S., "ZEST - Z Extended with Structuring: A User's Guide," British Telecommunications 7004.0.20.13, Logica's PROST web-site, Jan. 22, 1996, accessed Feb. 7, 2001 at <http://public.logica.com/~prost/ps/d13.ps>

- [ZANS98] "ZANS Animations," Formal Methods Research Group's web-site, De Paul University, Chicago, IL, last updated Feb. 25, 1998, accessed Feb. 7, 2001 at <http://se.cs.depaul.edu/fm/zans.html>
- [Zave96] Zave, P., and Jackson, M., "Where Do Operations Come From? A Multiparadigm Specification Technique," *IEEE Transactions on Software Engineering*, vol. 22, no. 7, July 1996, pp. 508-528.
- [Zed01] "The Z Notation," The World Wide Web Virtual Library, last updated by Jonathan Bowen on Jan. 13, 2001, accessed Feb. 6, 2001 at <http://www.afm.sbu.ac.uk/z/>
- [ZETA00] "The ZETA System: Overview," release 1.5, Technische Universität Berlin's web-site, last modified July 18, 2000, accessed Feb. 6, 2001 at <http://uebb.cs.tu-berlin.de/zeta/>
- [ZEVES00] "Z/EVES," an ORA, Canada, web-site, last updated July 12, 2000, accessed Feb. 5, 2001 at <http://www.ora.on.ca/z-eves/welcome.html>
- [Zhang93] Zhang, Y., and Mackworth, A.K., "Design and Analysis of Embedded Real-Time Systems: An Elevator Case Study," TR-93-04, Department of Computer Science, University of British Columbia, accessed July 1998 at <ftp://ftp.cs.ubc.ca/ftp/local/techreports/1993/TR-93-04.ps.gz>
- [Zimmerman00] Zimmerman, M., Rodriguez, M., Ingram, B., Katahira, M., de Villepin, M., and Leveson, N., "Making Formal Methods Practical," *Proc. of the 19th Conf. on Digital Avionics Systems*, Oct. 2000, vol. 1, pp. 1B2/1-1B2/8.
- [ZTC98] "Z Type Checker," Formal Methods Research Group's web-site, De Paul University, Chicago, IL, last updated Aug. 12, 1998, accessed Feb. 7, 2001 at <http://se.cs.depaul.edu/fm/ztc.html>

Appendix A Summary Overview of Z++

The following is a summary presentation of Z++, based on [Lano94b], [Lano94e] and [Lano95] (throughout the entire thesis the later was used as primary reference whenever it was necessary to resolve differences between various Z++ materials). For documentation and manipulation purposes several identifiers have been modified, e.g., we write `RTLFormula` instead of `FmlRTL`, and use `⊆` instead of `⊂`. Also, a `PUBLICS` clause, listing externally visible attributes and operations has been appended to the structure of the Z++ class presented in Section A.1. The recommended place for `PUBLICS` is between the `EXTENDS` and `TYPES` clauses of the Z++ class (more details about this new clause can be found in Chapter 6).

A.1 BNF Syntax of the Z++ Class Declaration

```

ZPP_Class ::= CLASS Identifier [TypeParams]
           [EXTENDS Ancestors]
           [TYPES Types]
           [FUNCTIONS AxiomaticDefs]
           [OWNS Locals]
           [RETURNS OpTypes]
           [OPERATIONS OpTypes]
           [INVARIANT Predicate]
           [ACTIONS Actions]
           [HISTORY History]
           END CLASS

```

where:

```

TypeParams ::= [ "[" Parlist "]" ]
Parlist    ::= Identifier [, Parlist] |
             Identifier << Identifier [, Parlist]
Ancestors  ::= Idlist

```

```

Types      ::=  TypeDeclarations
Locals     ::=  Identifier : Type ; Locals | Identifier : Type
OpTypes    ::=  [*] Identifier : Idlist → Idlist; OpTypes |
                [*] Identifier : Idlist → Idlist
Actions    ::=  [*] [Predicate &] Identifier Idlist ==> Code; Actions
                | [*] [Predicate &] Identifier Idlist ==> Code
History    ::=  LTLFormula | RTLFormula

```

Briefly, about the clauses in the class declaration:

- **CLASS** is followed by an identifier and a possibly empty list of generic type parameters. In this list, the notation $A \ll B$ signifies that class parameter A is the descendent of class B ;
- **EXTENDS** contains the list of classes inherited by this class;
- **TYPES** contains definitions of types used in the declarations of local variables. Classes can be used as types in this clause and in the clauses that follow;
- **FUNCTIONS** is followed by axiomatic definitions of constants;
- **OWNS** is followed by attribute declarations, for each attribute the name and the type being given;
- **RETURNS** includes the signatures of operations that do not change the attributes of the class instances. These operations represent pure enquiry accesses to the state;
- **OPERATIONS** includes the signatures of the operations that can change the state of the objects. The operations are specified here and in the **RETURNS** clause as functions from a sequence of input domains to a sequence of output domains;
- **INVARIANT** specifies a property of the internal state that must remain unchanged between the executions of operations. The default invariant of a class is `true`;
- **ACTIONS** contains definitions of operations that can be performed on instances of the class. For an operations op of class C the input parameters x are listed before the output parameters y . The body of the operation $Code(op, C)$ contains Z statements. The operation has either the implicit precondition `true` or an explicit precondition $Pre(op, C)$. The general form for an operation's definition is:

$$\text{Pre}(\text{op}, C) \ \& \ \text{op} \ x \ y \ ==> \ \text{Code}(\text{op}, C)$$

- HISTORY contains a predicate that defines the admissible sequences of execution for the operations of the class's objects. The predicate can be written in linear temporal logic or in RTL.

A.2 Invocation of Operations

In class C an operation declared as

$$\text{op} : X \rightarrow Y$$

and defined as

$$\text{Pre}(\text{op}, C) \ \& \ \text{op} \ x \ y \ ==> \ \text{Code}(\text{op}, C)$$

can be invoked as $\text{objectC.op}(\text{ap})$ where objectC is an object of class C and ap a list of actual parameters. The alternative notation $\text{objectC.op}[\text{ap}/\text{fp}]$ can be used, where ap is the set of actual parameters that substitute the formal parameters fp . It is also possible to highlight the output parameters y that result from the invocation of the operation by writing $y \leftarrow \text{objectC.op}(\text{ap})$.

An implicit operation New_C is available for each class C . The default name of the object created by New_C is $C!$, but the object can be suitably named by using the expression $\text{New}_C[\text{objectName!} / C!]$.

In Z++ the set of attributes changed by an operation is implicitly specified via the decoration of attributes, an attribute att that does not appear decorated as att' in the operation's definition being considered unchanged by the operation.

Operations of classes that have their name prefixed by * are denoted spontaneous (or internal) actions, invoked implicitly during the lifetime of the object. Also, an `init` operation can be included in a Z++ class to perform the initialisation of the object. In order to have `init` executed at the creation of class instances, the symbol * must precede the name of the operation, making it an internal action.

Within classes, the usual Z technique of splitting an operation in normal and error behaviour can be applied:

```

CLASS C
  OWNS
    ...
  OPERATIONS
    Op_OK      : X → Y
    Op_Error   : X → Y
    Op         : X → Y
  ACTIONS
    Op_OK      x? y! ==> Pre_OK ∧ Def_OK
    Op_Error   x? y! ==> Pre_Error ∧ Def_Error
    Op         x? y! ==> Op_OK ∨ Op_Error
END CLASS

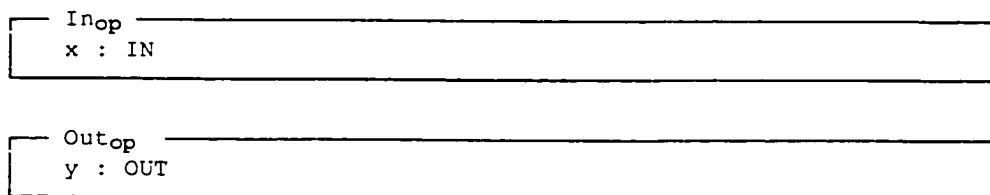
```

A.3. Notes on Semantics

A class declaration `c` as in A.1 defines implicitly the following Z schemas for, respectively, the class state:

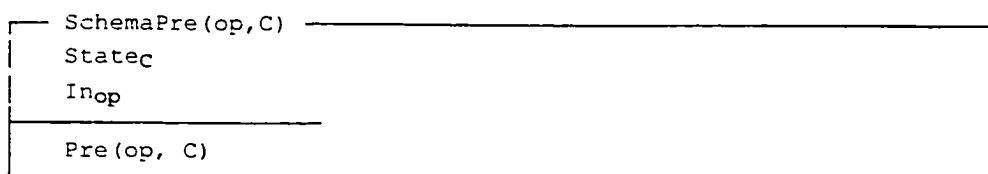


and for each operation op :

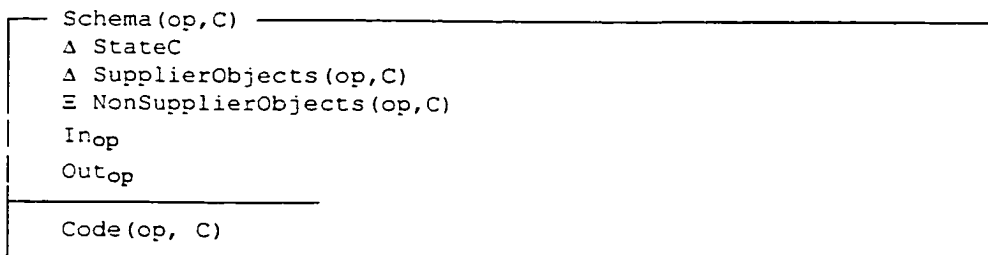


where c defines the state of C , Inv_C is the invariant of the class and, as in typical Z style declarations, the elements of parameter sequences x and y are matched position by position with their corresponding types contained in the sequences IN and, respectively, OUT (the parameters are listed in pairs $var:varType$).

The precondition $Pre(op, C)$ involves the class state and the input parameters:



and the operation itself is described by:



where $SupplierObjects(op, C)$ are schemas involved (directly or recursively) in the definitions of operations invoked from within op and $NonSupplierObjects(op, C)$ are the schemas for the other class instances in the specification.

For each class c the given set $[@C]$ represents the set of all possible objects of the class and the set \underline{c} denotes the existing objects of c . These are described in the following schema, which also includes a dereference map $*c$ from object identities to object values:

Objects_c $*c : @C \rightarrow \text{State}_c$ $\underline{c} : P(@C)$ <hr style="width: 50%; margin-left: 0;"/> $\underline{c} = \text{dom} (*c)$

The nil object reference $\text{nil}_c : @C$, which can never be an element of \underline{c} , can be used in Z++.

A.4 Extending and Restructuring the Specification

The specification can be extended by applying operations on classes, as follows:

```

ClassExpression ::= ClassIdentifier |
                  ClassExpression \ (FeatureList) |
                  ClassExpression [RenameList] |
                  GenericClassExpression [ParameterList] |
                  ClassExpression ± ClassExpression |
                  ClassExpression ^ ClassExpression |
                  ClassExpression  $\underline{x}$  ClassExpression |
                  ClassExpression  $\underline{\cap}$  ClassExpression |
                  ClassExpression  $\underline{*}$  Ident ClassExpression |

```

where `ClassIdentifier` is the class name assigned in its CLASS clause; `FeatureList` a list of attributes and operations of the class, `RenameList` a list of substitutions `new/old` involving constants, attributes and operations, and `ParameterList` a list of types that represent the generic parameters of the class (by definition, a generic class is a class with a non-empty list of parameters.) Succinct descriptions for some of the above operations:

- The hiding operation \setminus has the effect of making the attributes and operations of the class unavailable to the class' descendents;
- Renaming allows the changing of names for reusability purposes, but does not affect the semantics of the class features;
- The operation \wedge on classes creates a class that contains the disjoint union of class attributes and the same-named operations, with their definitions conjoined;
- The intersection operator \sqcap produces the syntactic intersection of the definitions of the two classes, only the features with identical descriptions being retained.

A.5 Translation to Standard Z

Z++ specifications can be translated into regular Z specifications using a “flattening” procedure that allows the use of available analysis tools for Z. However, as pointed out by Lano and Haughton, temporal constraints are not treated, but their handling is feasible by using explicit trace variables in class state schemas [Lano94e]. The specific details of translation to Z follow from the semantics of Z++ outlined in A.3.

Appendix B Java Implementation of the AFCD

B.1 Contents of the Program Listing

The listing included in this Appendix contains the code of a Java program that implements the AFCD described in Section 6.3 of the thesis. The listing has several components, presented in the order indicated in Table B.I.

Table B.I Contents of the FCD Program (continued on the next page)

No.	Component	Description
1	classdiagram.dtd	Document type definition that establishes the structure of the input provided to the FCD program. This input is a representation of a UML class diagram.
2	FDCManager.java	The highest level class of the program. Coordinates: (a) the parsing of the input description of the class diagram and the loading of the input's components into Java objects; (b) the verification of the well-formedness of the input class diagram; (c) the translation to Z++ of the class diagram.
3	CDParser.java	Parser and loader that processes the *.xml input file and populates the classes of the program with the elements of the input. By verifying the structure of the input data, it enforces some of the rules for well-formedness of the class diagram.
4	CDSyntaxChecker.java	Groups the more complex checks for well-formedness of class diagrams presented in Section 6.3.1.
5	CDTranslator.java	Coordinates the entire formalisation in Z++ of a UML class diagram, according to the principles of translation described in Section 6.3.2.
6	ClassDiagram.java	Class that models the UML class diagram provided as input to FCD.
7	UMLClass.java	Models regular classes from the UML space. It is also the superclass of the UMLParaBindClass.java class.
8	UMLParaBindClass.java	Models both parameterised and instantiating UML classes. For AFCD's purposes further specialisation of the class was not necessary.
9	UMLAttribute.java	Models attributes of classes from the UML space.
10	UMLOperation.java	Models operations of classes from the UML space.

Table B.1 Contents of the FCD Program (continued from the previous page)

No.	Component	Description
11	UMLParameter.java	Models parameters of UML operations.
12	Relationship.java	Models binary relationships between classes.
13	RelationshipEnd.java	Models the ends of relationships.
14	Multiplicity.java	Models the multiplicity attached to the ends of relationships.
15	Range.java	Models ranges of values. Used as components of multiplicity constraints.
16	ZPPCSpec.java	Models the Z++ specification that results from formalising a UML class diagram. It is the correspondent of the UML class diagram in the Z++ space.
17	ZPPClass.java	Provides representation of Z++ classes. Contains both the clause contents of a Z++ class ("external representation") and information that makes up an "internal representation" that facilitates translation from and to UML.
18	ZPPAttribute.java	Placeholder for information describing an attribute from the Z++ space.
19	ZPPOperation.java	Placeholder for information describing a Z++ operation.
20	ZPPOpSignature.java	Placeholder for information describing a Z++ operation's signature.
21	ZPPOpDefinition.java	Placeholder for information describing a Z++ operation's definition.
22	StatementList.java	A class for grouping Z++ statements.
23	Statement.java	Models Z++ statements. Each statements consists of one or more lines.
24	IdList.java	Models lists of identifiers.
25	Logger.java	Utility class handling all messages to the user and the formatting of the Z++ specification.
26	FCDConstants.java	Interface that groups the constants used in the program.

B.2 The Program Listing

The listing of the Java program that implements the AFCD is presented below.

```

<?xml version='1.0' encoding='us-ascii'?>

<!-- Document Data Type (DTD) for class diagram; specifies the structure of the AFCD input
The symbol * means "zero or more" while the symbol + means "one or more" -->
5
<!-- Class diagram declaration-->
<!ELEMENT classdiagram (class*, relationship*)>
<!ATTLIST classdiagram title CDATA #REQUIRED>

10 <!-- Class declaration -->
<!ELEMENT class (att*, op*) >
<!ATTLIST class
      name CDATA #REQUIRED
      ctype (reg | para | bind) "reg">

15 <!-- Attribute declaration -->
<!ELEMENT att (#PCDATA)>
<!ATTLIST att
      name CDATA #REQUIRED
20      type CDATA "null"
      initval CDATA "null"
      vistype (public | protected | private) "public"
      property (changeable | frozen) "changeable">

25 <!-- Operation declaration -->
<!ELEMENT op (param*) >
<!ATTLIST op
      name CDATA #REQUIRED
      vistype (public | protected | private) "public"
30      rettype CDATA "null"
      property (none | query) "none">

<!-- Parameter declaration -->
<!ELEMENT param (#PCDATA) >
35 <!ATTLIST param
      name CDATA #REQUIRED
      ptype CDATA #REQUIRED
      dir (in | out | inout) "in">

40 <!-- Relationship declaration -->
<!ELEMENT relationship (relationshipend*) >
<!ATTLIST relationship name CDATA "null">

<!-- Relationshipend declaration -->
45 <!ELEMENT relationshipend (multiplicity) >
<!ATTLIST relationshipend
      kind (assoc | aggreg | comp | super | generic | none) #IMPLIED
      classname CDATA #REQUIRED>

50 <!-- Multiplicity declaration -->
<!ELEMENT multiplicity (range+) >

<!-- Range declaration -->
<!ELEMENT range (#PCDATA) >
55 <!ATTLIST range
      begin CDATA "1"
      end CDATA "1">

```

Builder - Filename = D:/Workyard/Besthad/ars/2cd/FCDManager.java
 Printed on June 14, 2001 at 8:25 AM by People People

Page 1 of 1

```

// 2 FCDManager

package fcd;
5
import java.io.File;

/**
10 * Coordinates the three major functions of the AFCD:
* (a) parsing of the input description of the class diagram
* (b) verification of the well-formedness of the class diagram
* (c) translation to Z++ of the components of the class diagram
*/
public class FCDManager {
15 private ClassDiagram cd;
private ZPPSpec zspec;

// All main work components start from here
public void doWork(File file) {
20 try {

cd = new ClassDiagram();
CDParser parser = new CDParser(cd); // parsing & loading
parser.parse(file);

25 CDSyntaxChecker checker = new CDSyntaxChecker(cd);
if (!checker.checkCDSyntax()) // verification
Logger.log("Checking of CD syntax stopped.");
else {
30 Logger.log("Checking of CD syntax successfully completed.");
CDTranslator trans = new CDTranslator(cd);
trans.CDtranslate(); // translation
}

35 } catch (Exception e) {
e.printStackTrace();
Logger.log("Parsing of CD input stopped.");
}
}

40 public static void main(String[] args) { // entry point

File file = new File("classdiagram.xml"); // default input file

45 try {
if (args.length == 1)
file = new File(args[0]);
FCDManager mgr = new FCDManager(); // a manager to
mgr.doWork(file); // coordinate the work
50 } catch (Exception x) {
x.printStackTrace();
System.exit(1);
}
}

55 }

```

Builder - Filename = D:\Work\src\src\src\CDParser.java

Printed on June 14, 2001 at 12:12 PM by Sergio Dossola

Page 1 of 3

```

// 3 CDParser

package fcd;

5
import java.util.*;
import java.io.File;
import java.io.IOException;
import javax.xml.parsers.*;
10 import org.xml.sax.*;
import org.w3c.dom.*;

/** Parses the input and populates the classes modelling the UML class diagram */
public class CDParser {
15 private ClassDiagram cd;
private Map mappings = new HashMap();

public CDParser(ClassDiagram cd) {
20 this.cd = cd;
}

public void parse(File file) throws Exception {
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    factory.setValidating(true);
25 try {
        DocumentBuilder builder = factory.newDocumentBuilder();
        setErrorHandler(builder); // error handling code to deal with DTD validation
        Document document = builder.parse(file);
        Logger.log("The input class diagram " + document.getDocumentElement().getNodeName()
30 + " successfully read.");
        createElement(document); // create cd and its elements from the DOM tree
        Logger.log("The class diagram " + document.getDocumentElement().getNodeName()
+ " successfully created.");
    } catch (SAXException sxe) {
35 Exception x = sxe;
if (sxe.getException() != null)
x = sxe.getException();
Logger.log(x.getMessage());
throw x;
40 } catch (ParserConfigurationException pce) {
pce.printStackTrace();
} catch (IOException ioe) {
ioe.printStackTrace();
} catch (Exception x) {
45 throw x;
}
}

/** Creates an element corresponding to the DOM node and recursively processes its children */
50 private void createElement(Node node) throws Exception {
switch (node.getNodeType()) { // determines action based on node type
case Node.DOCUMENT_NODE:
    Document doc = (Document) node;
    Node next = doc.getDocumentElement();
55 mappings.put(next, cd);
createElement(next);
break;
case Node.ELEMENT_NODE:
    String name = node.getNodeName();
60 if (name.equals("classdiagram")) {
        String title = ((Element) node).getAttribute("title");
        cd.setName(title);
    } else if (name.equals("class")) {
        createClass(node);
65 } else if (name.equals("att")) {

```

```

        createAttribute(node);
    } else if (name.equals("op")) {
        createOperation(node);
    } else if (name.equals("param")) {
70     createParameter(node);
    } else if (name.equals("relationship")) {
        createRelationship(node);
    } else if (name.equals("relationshipend")) {
75     createRelationshipEnd(node);
    } else if (name.equals("multiplicity")) {
        createMultiplicity(node);
    } else if (name.equals("range")) {
        createRange(node);
    }
80     NodeList children = node.getChildNodes(); // recursive processing of children
    if (children != null)
        for (int i=0; i < children.getLength(); i++)
            createElement(children.item(i));
85     break;
    default :
        break;
}

90 /** Creates and initialises class diagram elements */
private void createClass(Node node) {
    Node parentNode = node.getParentNode();
    ClassDiagram container = (ClassDiagram) mappings.get(parentNode);
    String ctype = ((Element) node).getAttribute("ctype");
95     UMLClass cl = null;
    String token = ((Element) node).getAttribute("name");
    if (ctype.equals("req")) {
        cl = container.createClass();
        cl.setName(token);
100    } else {
        cl = container.createUMLParaBindClass();
        ((UMLParaBindClass) cl).setType(ctype);
        ((UMLParaBindClass) cl).setNameAndParameters(token);
    }
105    mappings.put(node, cl);
}

private void createAttribute(Node node) {
    Node parentNode = node.getParentNode();
110    UMLClass container = (UMLClass) mappings.get(parentNode);
    UMLAttribute att = container.createAttribute();
    att.setName(((Element) node).getAttribute("name"));
    att.setType(((Element) node).getAttribute("type"));
    att.setVisType(((Element) node).getAttribute("vistype"));
115    att.setInitValue(((Element) node).getAttribute("initval"));
    att.setProperty(((Element) node).getAttribute("property"));
}

private void createOperation(Node node) {
120    Node parentNode = node.getParentNode();
    UMLClass container = (UMLClass) mappings.get(parentNode);
    UMLOperation op = container.createOperation();
    op.setName(((Element) node).getAttribute("name"));
    op.setVisType(((Element) node).getAttribute("vistype"));
125    op.setRetType(((Element) node).getAttribute("rettype"));
    op.setProperty(((Element) node).getAttribute("property"));
    mappings.put(node, op);
}

130 private void createParameter(Node node) {

```


Builder - Filename = D:\Work\src\method\src\src\CDParser.java

Printed on June 14, 2001 at 12:12 PM by Sergio Sosa

Page 3 of 3

```

Node parentNode = node.getParentNode();
UMLOperation container = (UMLOperation) mappings.get(parentNode);
Parameter param = container.createParameter();
param.setName(((Element) node).getAttribute("name"));
135 param.setType(((Element) node).getAttribute("ptype"));
param.setDir(((Element) node).getAttribute("dir"));
}

private void createRelationship(Node node) {
140 Node parentNode = node.getParentNode();
ClassDiagram container = (ClassDiagram) mappings.get(parentNode);
Relationship rel = container.createRelationship();
rel.setName(((Element) node).getAttribute("name"));
mappings.put(node, rel);
145 }

private void createRelationshipEnd(Node node) throws Exception{
Node parentNode = node.getParentNode();
Relationship container = (Relationship) mappings.get(parentNode);
150 RelationshipEnd relEnd = new RelationshipEnd();
if ((container.getEnd1() == null)
    container.setEnd1(relEnd);
else if ((container.getEnd2() == null)
    container.setEnd2(relEnd);
155 else {
    Logger.log("INPUT ERROR: More than two ends for relationship " + container.getName() + ".");
    Exception x = new Exception();
    throw x;
}
relEnd.setKind(((Element) node).getAttribute("kind"));
160 relEnd.setClassName(((Element) node).getAttribute("classname"));
mappings.put(node, relEnd);
}

private void createMultiplicity(Node node) {
Node parentNode = node.getParentNode();
RelationshipEnd container = (RelationshipEnd) mappings.get(parentNode);
Multiplicity mult = container.createMultiplicity();
mappings.put(node, mult);
170 }

private void createRange(Node node) {
Node parentNode = node.getParentNode();
Multiplicity container = (Multiplicity) mappings.get(parentNode);
175 Range range = container.createRange();
range.setBegin(((Element) node).getAttribute("begin"));
range.setEnd(((Element) node).getAttribute("end"));
}

private void setErrorHandler(DocumentBuilder builder) {
180 builder.setErrorHandler(
    new org.xml.sax.ErrorHandler() {
        // ignore fatal errors
        public void fatalError(SAXParseException exception) throws SAXException {
        }
        public void error(SAXParseException e) throws SAXParseException {
185 throw e; // treat validation errors as fatal
        }
        public void warning(SAXParseException err) throws SAXParseException { // print warnings
            Logger.log("*** Warning " + " , line " + err.getLineNumber()
190 + " , uri " + err.getSystemId());
            Logger.log(" " + err.getMessage());
        }
    }
);
}
195 }

```

```

// 4. CDSyntaxChecker

package fcd;

5
import java.util.*;

/** Handles all checks of well-formedness */
public class CDSyntaxChecker implements FCDConstants {
10
    private ClassDiagram cd;
    private Collection classes = new ArrayList();
    private Collection relationships = new ArrayList();

    public CDSyntaxChecker(ClassDiagram cd) {
15
        this.cd = cd;
        this.classes = cd.getClasses();
        this.relationships = cd.getRelationships();
    }

    20
    /** Highest level of organising the checks */
    public boolean checkCDSyntax() {
        boolean ret = false;
        if (!checkRelationships()) return ret;
        if (!checkAccrossCD()) return ret;
25
        return checkClasses();
    }

    /** Contains a series of tests at relationship level */
    private boolean checkRelationships() {
30
        boolean ret = false;
        Logger.separator();
        if (!checkRelationshipEnds()) return ret;           // check ends of relationships are
        Logger.separator();                               // properly defined
        if (!checkWellFormedMultiplicities()) return ret; // check multiplicities are properly
35
        Logger.separator();                               // defined
        if (!checkAssociationsHaveNames()) return ret;    // check names exist for associations
        Logger.separator();
        if (!checkCompositionsMultOne()) return ret;      // check proper mult. of compositions
        Logger.separator();
40
        if (!checkRelationshipsMultOne(SUPER, GENERALISATION)) return ret; //same for generalisations
        Logger.separator();
        return checkRelationshipsMultOne(GENERIC, INSTANTIATION); // and instantiations
    }

    45
    /** Contains a series of tests at class diagram level */
    private boolean checkAccrossCD() {
        boolean ret = false;
        Logger.separator();
        if (!checkEndRelClassesExist()) return ret;       // check classes in rel. exist in CD
50
        Logger.separator();
        if (!checkClassNamesUnique()) return ret;        // check names of classes
        Logger.separator();
        if (!checkDistinctAssociationsNames()) return ret; // associations between same two
        Logger.separator();                               // classes must have distinct names
55
        if (!checkDuplicateRelationships()) return ret;  // check multiple rel. between same
        Logger.separator();                               // two classes
        if (!checkNoAncestorToSelf()) return ret;       // a class cannot be ancestor to itself
        Logger.separator();
        if (!checkInstantiationClasses()) return ret;    // check proper def. of instantiations
60
        Logger.separator();
        return checkMatchingBindings();                  // and proper matching of parameters
        // at instantiation
    }

    65
    /** Contains a series of tests at class level */

```

```

private boolean checkClasses() {
    boolean ret = false ;
    Logger.separator();
    if (!checkAttributeNamesUnique()) return ret;           // check names of attributes
70  Logger.separator();
    if (!checkOperationNamesUnique()) return ret;         // of operations
    Logger.separator();
    return checkOpParamNamesUnique();                     // and of parameters of operations
}

75
/** Verifies that the ends of relationships are properly formed */
public boolean checkRelationshipEnds() {
    boolean ret = true ;
    for (Iterator i = relationships.iterator(); i.hasNext(); ) {
80     Relationship rel = (Relationship) i.next();
        if (!rel.checkEnds())
            ret = false ;
    }
    Logger.logCheckResult("Relationship ends", ret);
85  return ret;
}

/** Verifies format of multiplicity constraints */
public boolean checkWellFormedMultiplicities() {
90  boolean ret = true ;
    for (Iterator i = relationships.iterator(); i.hasNext(); ) {
        Relationship rel = (Relationship) i.next();
        if (!rel.getEnd1().checkMultiplicity())           // test both ends
            ret = false ;
95  if (!rel.getEnd2().checkMultiplicity())
            ret = false ;
    }
    Logger.logCheckResult("Well-formed multiplicities", ret);
100 return ret;
}

/** Verifies names are provided for association relationships */
public boolean checkAssociationsHaveNames() {
105 boolean ret = true ;
    for (Iterator i = relationships.iterator(); i.hasNext(); ) {
        Relationship rel = (Relationship) i.next();
        if (!rel.checkAssociationHasName())
            ret = false ;
    }
110 Logger.logCheckResult("Associations have names", ret);
    return ret;
}

115 /** Verifies that multiplicity of the whole part of composition is one */
public boolean checkCompositionsMultOne() {
    boolean ret = true ;
    for (Iterator i = relationships.iterator(); i.hasNext(); ) {
120     Relationship rel = (Relationship) i.next();
        if (!rel.checkCompositionMultOne())
            ret = false ;
    }
    Logger.logCheckResult("Multiplicity of whole part of composition", ret);
125 return ret;
}

/** Verifies that specific kinds of relationships have multiplicity one */
public boolean checkRelationshipsMultOne(String endKind, String relKind) {
130     boolean ret = true ;

```

```

    for (Iterator i = relationships.iterator(); i.hasNext(); ) {
        Relationship rel = (Relationship) i.next();
        if (!rel.checkRelationshipMultOne(endKind, relKind))
            ret = false ;
135     }

    Logger.logCheckResult("Multiplicity of " + relKind + " ends", ret);
    return ret;
}

140 /** Verifies that the two classes involved in a relationship belong to the class diagram */
private boolean checkEndRelClassesExist() {
    boolean ret = true ;
    String clsName;

145     for (Iterator i = relationships.iterator(); i.hasNext(); ) {
        Relationship rel = (Relationship) i.next();
        clsName = rel.getEnd1().getClassName();
        if (!classFound(clsName)) {
150             Logger.log("CD SYNTAX ERROR: Class involved in relationship not found (" +
                clsName + ")");
            ret = false ;
        }
        clsName = rel.getEnd2().getClassName();
155         if (!classFound(clsName)) {
            Logger.log("CD SYNTAX ERROR: Class involved in relationship not found (" +
                clsName + ")");
            ret = false ;
        }
    }
160     Logger.logCheckResult("Exist classes at relationship ends", ret);
    return ret;
}

165 /** Verifies constraints on class names */
public boolean checkClassNamesUnique() {
    boolean ret = true ;

    int size = classes.size();
170     for (int i = 0; i < size; i++) {
        UMLClass cls = (UMLClass) ((ArrayList) classes).get(i);
        ret &= checkClassNameUnique(cls.getName(), i);
    }

175     Logger.logCheckResult("Class names unique", ret);
    return ret;
}

/** Verifies names of classes are unique within the class diagram */
180 private boolean checkClassNameUnique(String className, int index) {
    boolean ret = true ;
    int size = classes.size();
    for (int i = index+1; i < size; i++) {
        UMLClass cls = (UMLClass) ((ArrayList) classes).get(i);
185         if ((cls.getName()).equals(className)) {
            Logger.log("CD SYNTAX ERROR: Duplicate name of class detected (" + className + ")");
            ret = false ;
            break ;
190         }
    }
    return ret;
}

195 /** Verifies that associations between the same two classes have distinct names */

```

Builder - filename = D:\Work\src\method\src\End\CDSyntaxChecker.java

Printed on June 16, 2001 at 12:52 PM by Sergio Dancalia

Page 4 of 7

```

public boolean checkDistinctAssociationsNames() {
    boolean ret = true;
    int size = relationships.size();
    for (int i = 0; i < size; i++) {
200     Relationship rel1 = (Relationship) ((ArrayList) relationships).get(i);
        if (rel1.isAssociation()) {
            String cls1 = (rel1.getEnd1()).getClassName();
            String cls2 = (rel1.getEnd2()).getClassName();
            for (int j = i+1; j < size; j++) {
205                 Relationship rel2 = (Relationship) ((ArrayList) relationships).get(j);
                    if (rel2.isAssociation()) {
                        if ((rel2.hasClassEnds(cls1, cls2)) || (rel2.hasClassEnds(cls2, cls1)))
                            if ((rel1.getName()).equals(rel2.getName())) {
210                                Logger.log("CD SYNTAX ERROR: Duplicate name of association detected (" +
                                    rel1.getName() + ")");
                                    ret = false;
                            }
                    }
            }
        }
215    }
    Logger.logCheckResult("Distinct association names", ret);
    return ret;
}

220 /* Verifies that only aggregates/comps and assoc's may be duplicated between the same two classes */
public boolean checkDuplicateRelationships() {
    boolean ret = true;
    String clsName1;
225    String clsName2;
    String relKind;

    int size = relationships.size();
    for (int i = 0; i < size; i++) {
230     Relationship rel1 = (Relationship) ((ArrayList) relationships).get(i);
        relKind = rel1.getRelationshipKind();
        clsName1 = (rel1.getEnd1()).getClassName();
        clsName2 = (rel1.getEnd2()).getClassName();
        for (int j = i+1; j < size; j++) {
235             Relationship rel2 = (Relationship) ((ArrayList) relationships).get(j);
                if (rel2.hasClassEnds(clsName1, clsName2)) {
                    if ((rel2.getRelationshipKind()).equals(relKind)) {
                        if ((relKind.equals(GENERALISATION)) || (relKind.equals(INSTANTIATION))) {
240                             Logger.log("CD SYNTAX ERROR: Duplicate " + relKind + " relationships detected "
                                + "between classes (" + clsName1 + ", " + clsName2 + ")");
                                ret = false;
                                break;
                            }
                    }
                } else {
245                     Logger.log ("CD SYNTAX ERROR: Distinct relationships detected between classes (" +
                        clsName1 + ", " + clsName2 + ")");
                        ret = false;
                        break;
                    }
                }
250            }
        }
    }

    Logger.logCheckResult("Valid duplicate relationships", ret);
255    return ret;
}

/** Verifies that a class is not its own ancestor */
private boolean checkNoAncestorToSelf() {
260     boolean ret = true;

```

Builder - Filename = D:\Work\gen\testbed\src\cd4\CDSyntaxChecker.java

Printed on June 14, 2001 at 12:52 PM by Gergis Saecula

Page 5 of 7.

```

Collection classes = cd.getClasses();
for (Iterator i = classes.iterator(); i.hasNext(); ) { // look at all classes
    UMLClass cls = (UMLClass) i.next();
    //if (FCDManager.DEBUG) Logger.log("**** checkAncestorToSelf( " + cls.getName() + ")**** ");
265     if (!checkCycleToSelf(cls.getName())) ret = false; // and verify there are no
    // cycles in the inheritance
    // graph containing the class

    Logger.logCheckResult("No ancestor to self", ret);
    return ret;
270 }

/** Checks no ancestor to self for one class (or, equivalently, no successor to self) */
private boolean checkCycleToSelf(String cls) {
    boolean ret = true;
275     Collection one = (ArrayList) successors(cls); // gather all successors
    // (per generation)

    while (true) {
        if (one.isEmpty()) { // stop when the list is empty
            if (FCDManager.DEBUG) Logger.log("No cycles found for class " + cls);
280             return ret;
        }
        if (one.contains(cls)) { // or the root class itself is in the list
            Logger.log("CD SYNTAX ERROR: Generalisation cycle detected for class " + cls);
285             return !ret;
        }
        Collection two = new ArrayList();
        for (Iterator i = one.iterator(); i.hasNext(); ) {
            String temp = (String) i.next();
            two.addAll((ArrayList) successors(temp)); // update the generation of ancestors
290         }
        if (FCDManager.DEBUG) Logger.log(two.toString());
        one = new ArrayList(two); // go and test the next generation
    }
}
295

/** Gathers all direct successors of a given class (the "first generation") */
private Collection successors(String cls) {
    Collection one = new ArrayList();
    for (Iterator i = relationships.iterator(); i.hasNext(); ) { // check all relationships
300         Relationship rel = (Relationship) i.next();
        if (rel.getEnd1().getClassName().equals(cls)) // if generalisation check
            if (rel.getEnd1().getKind().equals(SUPER)) // if given class is superclass
                one.add(rel.getEnd2().getClassName()); // if yes, gather the successors
    else
305         if (rel.getEnd2().getClassName().equals(cls)) // same for the other rel. end
            if (rel.getEnd2().getKind().equals(SUPER))
                one.add(rel.getEnd1().getClassName());
    }
    return one;
310 }

/** Verifies that suitable classes are involved in instantiation relationships */
public boolean checkInstantiationClasses() {
    boolean ret = true;
    for (Iterator i = relationships.iterator(); i.hasNext(); ) { // look at all relationships
315         Relationship rel = (Relationship) i.next();
        String clsName1 = rel.getEnd1().getClassName();
        String clsName2 = rel.getEnd2().getClassName();
        if ((rel.getEnd1().getKind().equals(GENERIC)) // if instantiation
            if (!foundClassOfType(clsName1, PARA)) // check end classes
            !foundClassOfType(clsName2, BIND)) { // (para, bind)
320             Logger.log ("CD SYNTAX ERROR: Invalid classes involved in instantiation (" +
                clsName1 + ", " + clsName2 + ")");
            ret = false;
        }
    }
325     if ((rel.getEnd2().getKind().equals(GENERIC)) // or end classes (bind, para)

```

```

        if (!foundClassOfType(clsName1, BIND) ||
            !foundClassOfType(clsName2, PARA)) {
            Logger.log("CD SYNTAX ERROR: Invalid classes involved in instantiation (" +
                clsName1 + ", " + clsName2 + ")");
330         ret = false ;
        }
    }
    Logger.logCheckResult("Valid instantiation end classes", ret);
    return ret;
335 }

/** Verifies the correspondence between the number of params at instantiation */
public boolean checkMatchingBindings() {
    boolean ret = true ;
340     for (Iterator i = relationships.iterator(); i.hasNext(); ) {
        Relationship rel = (Relationship) i.next();
        if (rel.isRelationshipKind(GENERIC)) {
            UMLClass cls1 = cd.getUMLClass(rel.getEnd1()).getClassName();
            UMLClass cls2 = cd.getUMLClass(rel.getEnd2()).getClassName();
345             int size1 = 0;
             int size2 = 0;
             if (cls1 instanceof UMLParaBindClass &&
                 (((UMLParaBindClass)cls1).getCType()).equals(BIND)) {
                 size1 = (((UMLParaBindClass) cls1).getClassParameters()).size();
                 size2 = (((UMLParaBindClass) cls2).getClassParameters()).size();
350             } else {
                 size1 = (((UMLParaBindClass) cls1).getClassParameters()).size();
                 size2 = (((UMLParaBindClass) cls2).getClassParameters()).size();
            }
355             if (size1 != size2) {
                Logger.log("CD SYNTAX ERROR: Invalid parameter matching at instantiation (" +
                    cls1.getName() + ", " + cls2.getName() + ")");
                ret = false ;
            }
        }
    }
360     Logger.logCheckResult("Matching bindings (as number of parameters)", ret);
    return ret;
}

365 /** Verifies that names of attributes are unique within the class */
private boolean checkAttributeNamesUnique() {
    boolean ret = true ;
370     for (Iterator i = classes.iterator(); i.hasNext(); ) {
        UMLClass cls = (UMLClass) i.next();
        if (!cls.checkAttributeNamesUnique())
            ret = false ;
    }
375     Logger.logCheckResult("Attribute names unique", ret);
    return ret;
}

/** Verifies that names of operations are unique within the class */
380 public boolean checkOperationNamesUnique() {
    boolean ret = true ;

    for (Iterator i = classes.iterator(); i.hasNext(); ) {
        UMLClass cls = (UMLClass) i.next();
385         if (!cls.checkOperationNamesUnique())
            ret = false ;
    }
    Logger.logCheckResult("Attribute Names Unique", ret);
    return ret;
390 }

```

```

    /** Verifies that names of parameters are unique within an operation's list of parameters */
    public boolean checkOpParamNamesUnique() {
        boolean ret = true ;
395
        for (Iterator i = classes.iterator(); i.hasNext(); ) {
            UMLClass cls = (UMLClass) i.next();
            if (!cls.checkOpParamNamesUnique())
                ret = false ;
400
        }
        Logger.logCheckResult("Operation Parameter Names Unique", ret);
        return ret;
    }

405
    /** Determines if a given class belongs to the class diagram */
    private boolean classFound(String className) {
        Collection classNames = new ArrayList();

        for (Iterator i = classes.iterator(); i.hasNext(); ) {
410
            UMLClass cls = (UMLClass) i.next();
            classNames.add(cls.getName());
        }
        return classNames.contains(className);
    }

415
    /** Determines if a class of given name and type exists in the class diagram */
    private boolean foundClassOfType(String className, String ctype) {
        boolean ret = false ;
        Collection classes = cd.getClasses();
420
        for (Iterator i = classes.iterator(); i.hasNext(); ){
            UMLClass cls = (UMLClass) i.next();
            if (cls.getName().equals(className)
                && ((ctype.equals(PARA) || ctype.equals(PARA)) && (cls instanceof UMLParaBindClass)) ||
                (ctype.equals(REG) && !(cls instanceof UMLParaBindClass))) {
425
                    ret = true ;
                    break ;
                }
            }
        return ret;
430
    }

```



```

// 5. CDTranslator

package fcd;
5
import java.io.*;
import java.util.*;
import java.awt.*;

10 /** Manages the formalisation in Z++ of a UML class diagram */
public class CDTranslator implements FCDConstants {
    private ClassDiagram cd; // input class diagram
    private ZPPSpec zspec; // output Z++ specification

15 public CDTranslator(ClassDiagram cd) {
    this.cd = cd;
}

/** Translates a class diagram to Z++ */
20 public void CDtranslate() {
    zspec = new ZPPSpec(); // create the new Z++ spec

    translateClasses(); // process classes
    translateRelationships(); // process relationships
25 resolveVisibility(); // hide private features of classes
    zspec.printZPPSpecification();
}

/** Translates UML classes */
30 private void translateClasses() {
    for (Iterator i = (cd.getClasses()).iterator(); i.hasNext(); ) {
        UMLClass cls = (UMLClass) i.next();
        if (!(getCType(cls)).equals(BIND)) // process regular and generic
            translateClass(cls); // classes; ignore binding classes
35    }
}

/** Translates relationships */
private void translateRelationships() {
40 for (Iterator i = (cd.getRelationships()).iterator(); i.hasNext(); ) {
    Relationship rel = (Relationship) i.next();
    if (rel.isAggregation() || rel.isComposition()) // process aggregations & compositions
        translateAggregation(rel);
    if (rel.isAssociation()) // process associations
45 translateAssociation(rel);
}
// (generalisations and instantiations
// are processed during the
// translation of classes)

/** Hides private features of classes */
50 private void resolveVisibility() {
    for (Iterator i = (zspec.getClasses()).iterator(); i.hasNext(); ) {
        ZPPClass zcls = (ZPPClass) i.next();
        if (zcls.getHiddenFeatures() != null) { // if list of hidden features is not
            String cls = zcls.getName(); // empty, rename the original class
            Statement stmt = new Statement();
55 stmt.addLine(HIDDEN + cls + EQUIV + cls + HIDE + "["
                + (zcls.getHiddenFeatures()).listIds() + "]*"); // construct hiding operation on class
            zspec.appendHidingOps(stmt); // and add the operation to Z++ spec
60    }
}

/** Translates individual UML class to Z++ */
private void translateClass(UMLClass cls) {
65 String name = cls.getName();

```

```

    # ((getCType(cls)).equals(PARA))
        name = ((UMLParaBindClass) cls).getReducedName(); // use reduced name of generic classes
    ZPPClass zcls = zspec.appendClass(name); // create Z++ class with same name
70 # ((getCType(cls)).equals(PARA))
        zcls.setCParams(processCParams((UMLParaBindClass) cls)); // transfer class params to Z++
        zcls.setExtends(processParents(cls.getName())); // process parents of class
        translateAttributes(cls, zcls); // formalise attributes
        translateOperations(cls, zcls); // formalise operations
75 placeZPPAttributes(zcls); // place attributes in Z++ clauses
        placeZPPOperations(zcls); // place attributes in Z++ clauses
    }

    /** Creates the list of formal class parameters */
80 private IdList processCParams(UMLParaBindClass cls) {
        IdList idl = new IdList();

        for (Iterator i = (cls.getClassParameters()).iterator(); i.hasNext(); ) {
            String cparam = (String) i.next();
85 idl.append(cparam);
        }
        return idl;
    }

90 /** Gathers all direct superclasses of a class */
    private IdList processParents(String cls) {
        IdList idl = new IdList();
        Collection relationships = cd.getRelationships();

95 for (Iterator i = relationships.iterator(); i.hasNext(); ) {
            Relationship rel = (Relationship) i.next(); // check all relationships
            # (rel.getEnd1().getClassName().equals(cls) ) // if generalisation see
            # (rel.getEnd2().getKind().equals(SUPER)) // if the given class is superclass
                idl.append((rel.getEnd2().getClassName())); // if yes, gather its successors
100 else
            # (rel.getEnd2().getClassName().equals(cls)) // do the same for the other rel. end
            # ( rel.getEnd1().getKind().equals(SUPER))
                idl.append((rel.getEnd1().getClassName()));
        }
105 return idl;
    }

    /** Translates all the attributes of a class */
    private void translateAttributes (UMLClass cls, ZPPClass zcls) {
110 for (Iterator i = (cls.getAttributes()).iterator(); i.hasNext(); ) {
            UMLAttribute att = (UMLAttribute) i.next();
            zcls.appendAttribute(translateAttribute(att, zcls)); // process each attribute and add
            // to Z++ class
115 }
    }

    /** Translates an individual attribute */
    private ZPPAttribute translateAttribute (UMLAttribute att, ZPPClass zcls) {
120 ZPPAttribute zatt = new ZPPAttribute(att.getName()); // create att & get name from UML att

        zatt.setVisType(att.getVisType()); // get also visibility
        # (att.getInitValue() != null) // and initial value
            zatt.setInitValue(att.getInitValue());
        # (att.getProperty().equals(CHANGABLE)) // determine place of Z++ attribute
125 zatt.setClause(OWNS);
        else
            zatt.setClause(FUNCTIONS);
        # (att.getVisType().equals(PUBLIC)) // make provisions for visibility
            zcls.appendPublics(zatt.getName());
130 else # (att.getVisType().equals(PRIVATE))

```

```

        zcls.appendHiddenFeatures(zatt.getName());
        zatt.setType(processType(att.getType(), zcls)); // and determine type of Z++ attribute
        return zatt;
    }
135
    /** Translates all the operations of a class */
    private void translateOperations (UMLClass cls, ZPPClass zcls) {
        for (Iterator i = (cls.getOperations()).iterator(); i.hasNext(); ) { // check all operations
            UMLOperation op = (UMLOperation) i.next();
140            ZPPOperation zop = translateOperation(op, zcls); // process each operation and add
                zcls.appendOperation(zop); // to Z++ class
        }
    }

145
    /** Translates an individual operation of a class */
    private ZPPOperation translateOperation (UMLOperation op, ZPPClass zcls) {
        ZPPOperation zop = new ZPPOperation(op.getName()); // create new op & get name from UML op

150        zop.setVisType(op.getVisType()); // get also visibility
            # (op.getVisType().equals(PUBLIC)) // make provisions for visibility
                zcls.appendPublics(zop.getName());
            else # (op.getVisType().equals(PRIVATE))
                zcls.appendHiddenFeatures(zop.getName());
155        # (op.getProperty().equals(QUERY)) // determine place of op. signature
            zop.setClause(RETURNS);
        else
            zop.setClause(OPERATIONS);
        processOpParameters(op, zop, zcls); // process parameters of operation
160        processOpReturn(op.getRetType(), zop, zcls); // and the operation return
        return zop;
    }

    /** Process parameters of operation*/
165    private void processOpParameters(UMLOperation op, ZPPOperation zop, ZPPClass zcls) {
        for (Iterator i = (op.getParameters()).iterator(); i.hasNext(); ) { // check all params
            UMLParameter param = (UMLParameter) i.next();
            String pname = param.getName(); // get name and
            String dir = param.getDir(); // direction of param
170            String ztype = processType(param.getType(), zcls); // determine Z++ type
            # (dir.equals(IN)) { // if direction is "in"
                (zop.getOpSignature()).appendInputDomain(ztype); // append type to input
                (zop.getOpDefinition()).appendInputId(pname + QUESTION_MARK); // domain and decorated
                // name to input list
175            } else # (dir.equals(OUT)) {
                (zop.getOpSignature()).appendOutputDomain(ztype); // process "out" param
                (zop.getOpDefinition()).appendOutputId(pname + EXCLAM_MARK);
            } else {
180                (zop.getOpSignature()).appendInputDomain(ztype); // process "inout" param
                (zop.getOpDefinition()).appendInputId(pname + QUESTION_MARK);
                (zop.getOpSignature()).appendOutputDomain(ztype);
                (zop.getOpDefinition()).appendOutputId(pname + EXCLAM_MARK);
            }
185        }
    }

    /** Interprets the operation's return type */
    private void processOpReturn(String opret, ZPPOperation zop, ZPPClass zcls) {
190        # (opret != null) {
            String ztype = processType(opret, zcls); // determine Z++ type
            # (!opret.equals(BOOLEAN) && !opret.equals(VOID)) {
                (zop.getOpDefinition()).appendOutputId(RESULT); // update op. definition
                (zop.getOpSignature()).appendOutputDomain(ztype); // and op. signature
195        }
    }

```

```

    }

    /** Place operation signature and definition in appropriate clauses of Z++ class */
200 private void placeZPPOperations(ZPPClass zcls) {
    for (Iterator i = (zcls.getZPPOperations()).iterator(); i.hasNext(); ) {
        ZPPOperation zop = (ZPPOperation) i.next();
        if (zop.getClause() != null) {
            if ((zop.getClause()).equals(RETURNS))
205         zcls.appendReturns(zop.assembleSignature());
            else
                zcls.appendOperations(zop.assembleSignature());
                zcls.appendActions(zop.assembleDefinition());
        }
    }
210 }

    /** Place attribute info in appropriate clauses of Z++ class */
215 private void placeZPPAttributes(ZPPClass zcls) {
    Statement st = new Statement();
    for (Iterator i = (zcls.getZPPAttributes()).iterator(); i.hasNext(); ) {
        ZPPAttribute zatt = (ZPPAttribute) i.next();

        if ((zatt.getClause()).equals(OWNS)) { // OWNS or FUNCTIONS?
220         zcls.appendOwms(zatt.assembleZPPAttribute(false));
            if (zatt.getInitValue() != null) { // check if init value
                Statement stmt = zatt.assembleZPPAttAssignOwms(); // provided
                ZPPOperation init = null;
                if (zcls.isInitOpEmpty()) {
225                 init = zcls.getInitOp();
                    init.setClause(OPERATIONS); // include init operation
                } // in the class
                ((zcls.getInitOp()).getOpDefinition()).appendCode(stmt); // append initialisation
                // to init code
230         } else {
            zcls.appendFunctions(zatt.assembleZPPAttribute(true));
            if (zatt.getInitValue() != null) {
                if (st.size() == 0) st.addSeparator(); // construct axiomatic
                Statement stmt = zatt.assembleZPPAttAssignFunctions(); // definition for
235                 st.updateStatement(stmt); // constant values
            }
        }
    }

    if (st.size() > 0) {
240         zcls.appendFunctions(st);
    }
}

    /** Translates aggregations and compositions */
245 private void translateAggregation(Relationship rel){
    String whole = rel.getWholeName(); // names of the two classes
    String part = rel.getPartName();
    boolean mp = rel.whatPartMultiplicity(); // one = F, many = T
    ZPPAttribute watt = new ZPPAttribute(); // attribute to be added
250 // to whole class

    if (!mp)
        watt.setNameType(lower(part), part);
    else
255         watt.setNameType(lower(part) + "s", POWerset + part); // multiplicity many

    ZPPClass zcls = zspec.getClass(whole);
    zcls.appendAttribute(watt); // append att to whole class
    zcls.appendOwms(watt.assembleZPPAttribute(false)); // and info in OWNS
260 }

```

```

/** Translates associations */
private void translateAssociation(Relationship rel){

    String relName = rel.getName();
    String aName = (rel.getEnd1()).getClassName();           // names of the two classes
    String bName = (rel.getEnd2()).getClassName();
    String typeLine = "";

    boolean ma = ((rel.getEnd1()).getMultiplicity()).whatMultiplicity(); // one = F, many = T
    boolean mb = ((rel.getEnd2()).getMultiplicity()).whatMultiplicity();

    ZPPAttribute zatt = new ZPPAttribute();                // first attribute to be
    zatt.setNameType(INSTANCESOF + aName, POWERSET + aName); // added to
    ZPPClass zcls = zspec.appendClass(upper(relName) + DESCR); // the class created to
    zcls.appendOwns(zatt.assembleZPPAttribute(false));     // describe the association

    zatt = new ZPPAttribute();
    zatt.setNameType(INSTANCESOF + bName, POWERSET + bName); // second attribute
    zcls.appendOwns(zatt.assembleZPPAttribute(false));

    zatt = new ZPPAttribute();                            // third attribute
    if (ma && mb) typeLine = aName + REL_SIGN + bName;
    else if (!ma && mb) typeLine = bName + PFUNCTION + aName;
    else typeLine = aName + PFUNCTION + bName;
    zatt.setNameType(lower(relName) + INSTANCES, typeLine);
    zcls.appendOwns(zatt.assembleZPPAttribute(false));

    Statement stmt = new Statement();                    // constraint
    stmt.addLine(DOMAIN + lower(relName) + INSTANCES + EQUAL + INSTANCESOF + aName);
    stmt.addLine(RANGE + lower(relName) + INSTANCES + EQUAL + INSTANCESOF + bName);
    zcls.appendInvariant(stmt);

    ZPPClass system = zspec.getClass(SYSTEM);
    zatt = new ZPPAttribute();                            // object descriptor for
    zatt.setNameType(THE + upper(relName) + DESCR, upper(relName) + DESCR); // the association
    system.appendOwns(zatt.assembleZPPAttribute(false));
}

/** Translates UML type to a Z++ type*/
private String processType(String type, ZPPClass zcls) {
    if (type == null)                                     // type not provided,
    return null;                                         // do nothing

    if (type.indexOf("[") == -1)                          // process scalar type
    return processScalarType(type, zcls);

    String str = type.substring(type.indexOf("[") + 1, type.indexOf("]"));
    if ((str.trim()).length() == 0)                       // if array type, add seq
    return SEQ + "[" + processScalarType(type.substring(0, type.indexOf("[")), zcls) + "]";
    return type;                                         // if generic type, keep unchanged
}

/** Interprets UML types expressed in scalar form */
private String processScalarType(String type, ZPPClass zcls) {
    if (type == null)
    return null;

    if (type.equals(NAT) || type.equals(BYTE))            // compare against recognised basic types
    return NATURALS;
    if (type.equals(INT) || type.equals(INTEGER) || type.equals(LONG))
    return INTEGERS;
    if (type.equals(REAL) || type.equals(DOUBLE) || type.equals(FLOAT))
    return REALS;
    if (type.equals(BOOLEAN))
    return BOOL;
}

```

```

    if (type.equals(VOID))
        return VOID; // compare with the formal
    if (zcls.getCParams() != null) // parameters of the class, if any
        if ((zcls.getCParams()).existsId(type))
            return type;
330    if (cd.existsUMLRegClass(type)) // compare with existing class types
        return type;
    if (zspec.getGivenSets() != null) // compare with existing given sets
        if ((zspec.getGivenSets()).existsId(type.toUpperCase()))
335    return type.toUpperCase();
    zspec.appendGivenSet(type.toUpperCase()); // if nothing found, create a given set
    return type.toUpperCase();
}

340 /** Determines the type of a UML class */
private String getType(UMLClass cls) {
    if (cls instanceof UMLParaBindClass)
        if (((UMLParaBindClass) cls).getType().equals(PARA))
            return PARA;
345    else
        return BIND;
    else
        return REG;
}

350 /** Helper operation that capitalises the first letter of a string */
private String upper(String text) {
    String first = text.substring(0,1);
    return text = first.toUpperCase() + text.substring(1);
355 }

/** Helper operation that makes lowercase the first letter of a string */
private String lower(String text) {
    String first = text.substring(0,1);
360    return text = first.toLowerCase() + text.substring(1);
}
}

```

```

// 6. CLASS DIAGRAM

package fcd;
5
import java.util.*;

/**
 * Models the UML class diagram provided as input to FCD. Contains both checks for well-formedness
10 * and operations that implement parts of the translation to Z++
 */
public class ClassDiagram implements FCDConstants {

    private String name; // Contents of class diagram:
    private Collection classes = new ArrayList(); // classes
    private Collection relationships = new ArrayList(); // and relationships

    // Data access operations

    public void setName(String name) {
20         this.name = name;
    }

    public String getName() {
25         return name;
    }

    public Collection getRelationships () {
        return relationships;
    }

    public Collection getClasses() {
30         return classes;
    }

    public UMLClass getUMLClass(String className) {
35         UMLClass cls = null ;
        for (Iterator i = classes.iterator(); i.hasNext(); ) {
            cls = (UMLClass) i.next();
            if ((cls.getName()).equals(className))
                break ;
40         }
        return cls;
    }

    public UMLClass getUMLClass(int index) {
45         return (UMLClass) ((ArrayList) classes).get(index);
    }

    public boolean existsUMLRegClass(String className) {
50         boolean found = false ;
        UMLClass cls = null ;
        for (Iterator i = classes.iterator(); i.hasNext(); ) {
            cls = (UMLClass) i.next();
            if ( !(cls instanceof UMLParaBindClass) && ((cls.getName()).equals(className)) ) {
55                 found = true ;
                break ;
            }
        }
        return found;
    }

60 // Utilities needed by the parser for populating the class diagram with components

    public UMLClass createClass() {
        UMLClass cls = new UMLClass();
65         classes.add(cls);
    }

```

```
        return cls;
    }

    public UMLParaBindClass createUMLParaBindClass() {
70     UMLParaBindClass cls = new UMLParaBindClass();
        classes.add(cls);
        return cls;
    }

75     public Relationship createRelationship() {
        Relationship rel = new Relationship();
        relationships.add(rel);
        return rel;
    }

80     /** Prints contents of class diagram */
    public void printClassDiagram() {
        Logger.log("CD title = " + name);
        for (Iterator i = classes.iterator(); i.hasNext(); ) {
85         UMLClass cls = (UMLClass) i.next();
            cls.printUMLClass();
        }
        for (Iterator i = relationships.iterator(); i.hasNext(); ) {
90         Relationship rel = (Relationship) i.next();
            rel.printRelationship();
        }
    }

95
```


JBuilder - Filename = D:/Workyard/Workshop/uml/Code/UMLClass.java

Printed on June 14, 2001 at 1:00 PM by Sergio Dussault

Page 1 of 2

```

// 7. UMLClass
package fcd;

5 import java.util.*;

/** Models regular UML classes; superclass of UMLParaBind class */
public class UMLClass implements FCDConstants{
    protected String name;

10    protected Collection attributes = new ArrayList();
    protected Collection operations = new ArrayList();

    // Data access methods

15    public void setName(String name) {
        this.name = name;
    }

20    public String getName() {
        return name;
    }

    public Collection getAttributes() {
25        return attributes;
    }

    public Collection getOperations() {
30        return operations;
    }

    // Utility methods needed by the parser

    public UMLAttribute createAttribute() {
35        UMLAttribute att = new UMLAttribute();
        attributes.add(att);
        return att;
    }

    public UMLOperation createOperation() {
40        UMLOperation op = new UMLOperation();
        operations.add(op);
        return op;
    }

45    /** Verifies names of attributes within the class */
    public boolean checkAttributeNamesUnique() {
        boolean ret = true;
        Set s = new TreeSet();
50        for (Iterator i = attributes.iterator(); i.hasNext(); ) {
            UMLAttribute att = (UMLAttribute) i.next();
            String unq = att.getName();
            if (!s.add(unq)) {
                Logger.log("CD Syntax Error: Duplicate attribute name detected: "+ unq +
55                " for class " + getName());
                ret = false;
            }
        }
        for (Iterator i = operations.iterator(); i.hasNext(); ) {
60            UMLOperation op = (UMLOperation) i.next();
            String unq = op.getName();
            if (!s.add(unq)) {
                Logger.log("CD Syntax Error: Duplicate attribute/operation name detected: "+ unq +
65                " for class " + getName());
                ret = false;
            }
        }
    }
}

```

```

    }
    return ret;
  }
70  /** Verifies names of operations within the class */
    public boolean checkOperationNamesUnique() {
        boolean ret = true;
        Set s = new TreeSet();
        for (Iterator i = operations.iterator(); i.hasNext(); ) {
75      UMLOperation op = (UMLOperation) i.next();
          String unq = op.getName();
          if (!s.add(unq)) {
              Logger.log("CD Syntax Error: Duplicate operation name detected: "+ unq +
80                " for class " + getName());
              ret = false;
          }
        }
        for (Iterator i = attributes.iterator(); i.hasNext(); ) {
85      UMLAttribute att = (UMLAttribute) i.next();
          String unq = att.getName();
          if (!s.add(unq)) {
              Logger.log("CD Syntax Error: Duplicate operation/attribute name detected: "+ unq +
90                " for class " + getName());
              ret = false;
          }
        }
        return ret;
    }

95  /** Verifies names of operation parameters */
    public boolean checkOpParamNamesUnique() {
        boolean ret = true;
        for (Iterator i = operations.iterator(); i.hasNext(); ) {
            UMLOperation op = (UMLOperation) i.next();
100      if (!op.checkOpParamNamesUnique(name))
                ret = false;
        }
        return ret;
    }

105  /** Prints contents of the class */
    public void printUMLClass() {
        Logger.log("UMLClass name = " + name);
        for (Iterator i = attributes.iterator(); i.hasNext(); ) {
110      UMLAttribute att = (UMLAttribute) i.next();
            att.printUMLAttribute();
        }
        for (Iterator i = operations.iterator(); i.hasNext(); ) {
115      UMLOperation op = (UMLOperation) i.next();
            op.printOperation();
        }
        Logger.separator();
    }
}

```

```

// 8. UMLParaBind
package fcd;
5
import java.util.*;

/** Models UML parameterised and binding classes */
public class UMLParaBindClass extends UMLClass{
10
    private Collection classParameters = new ArrayList();

    private String reducedName;           // T[params] is the name of the class
                                           // and T is its reduced name
15    private String ctype;               // distinction para/bind specified here

    // Data access methods

    public String getName() {
20        return name;
    }

    public String getReducedName() {
25        return reducedName;
    }

    public void setCType(String ctype) {
        this .ctype = ctype;
    }
30

    public String getCType() {
        return ctype;
    }

    public Collection getClassParameters() {
35        return classParameters;
    }

    public void setNameAndParameters(String name) {
40        this .name = name;
        reducedName = name.substring(0, name.indexOf('('));
        assembleParameters();
    }

    /** Assemble the parameters of the class for external representation */
    private void assembleParameters() {
        String param = null ;
        String params = name.substring(name.indexOf('(') + 1, name.indexOf(')'));
        while (true) {
50            if (params.indexOf(',') == -1) {
                param = params;
                classParameters.add(param);
                break;
            } else {
55                param = params.substring(0, params.indexOf(','));
                params = params.substring(params.indexOf(',')+1);
                classParameters.add(param);
            }
        }
60    }

    /** Prints contents of the class */
    public void printUMLClass() {
        Logger.log("Class reduced name = " + reducedName + " type = " + ctype);
65        Logger.logLine("Class parameters =");

```

Builder - Filename = D:/Workarea/Build/arc/Esd/UMLParameterClass.java
Printed on June 14, 2001 at 1:16 PM by Sergio Davala

Page 2 of 2

```
    for (Iterator i = classParameters.iterator(); i.hasNext(); ) {  
        Logger.logLine(" " + (String) i.next());  
    }  
    Logger.log("");  
    super.printUMLClass();  
70 }  
}
```

```
// 9. UMLAttribute

package fcd;

5
/* Models attributes of classes from the UML space */
public class UMLAttribute {

    private String name;                // attribute structure
10    private String type;
    private String visType;
    private String initValue;
    private String property;

15    // Data access methods

    public void setName(String name) {
        this.name = name;
    }

20    public String getName() {
        return name;
    }

25    public void setType(String type) {
        if (!type.equals("null"))
            this.type = type;
    }

30    public String getType() {
        return type;
    }

    public void setVisType(String visType) {
35        this.visType = visType;
    }

    public String getVisType() {
        return visType;
40    }

    public void setProperty(String property) {
        this.property = property;
    }

45    public String getProperty() {
        return property;
    }

    public void setInitValue(String initValue) {
50        if (!initValue.equals("null"))
            this.initValue = initValue;
    }

55    public String getInitValue() {
        return initValue;
    }

    /** Prints contents of attribute */
60    public void printUMLAttribute() {
        StringBuffer buf = new StringBuffer("Attribute name: " + name);
        if (type != null) {
            buf.append("\n           type: ");
            buf.append(type);
65    }
}
```

Spiciler - File name = D:\work\src\testbed\src\cod\initAttribute.java
Printed on June 14, 2001 at 1:11 PM by Sergio Escobar

Page: 2 of 2

```
        buf.append("\n          visType: ?");  
        buf.append(visType);  
        buf.append("\n          property: ?");  
        buf.append(property);  
70    if (initValue != null) {  
            buf.append("\n          initValue: ?");  
            buf.append(initValue);  
        }  
        String out = buf.toString();  
75    Logger.log(out);  
    }  
}
```

```
// 10. UMLOperation

package fcd;

5
import java.util.*;

/** Models operation of classes from the UML space */
public class UMLOperation {
10
    private String name; // operation structure
    private String visType;
    private String retType;
    private String property;
15
    private Collection parameters = new ArrayList();

    // Data access methods

    public void setName(String name) {
20
        this.name = name;
    }

    public String getName() {
        return name;
25
    }

    public void setVisType(String visType) {
        this.visType = visType;
    }
30

    public String getVisType() {
        return visType;
    }

    public void setRetType(String retType) {
35
        if (!retType.equals("null"))
            this.retType = retType;
    }

    public String getRetType() {
40
        return retType;
    }

    public void setProperty(String property) {
45
        this.property = property;
    }

    public String getProperty() {
50
        return property;
    }

    public Collection getParameters() {
        return parameters;
    }
55

    // Utility method needed during parsing

    public Parameter createParameter() {
        Parameter param = new Parameter();
60
        parameters.add(param);
        return param;
    }

    /** Verifies that names of operation parameters are unique */
65
    public boolean checkOpParamNamesUnique(String className) {
```

Builder - Filename = D:\Work\src\testbed\src\test\CDOperation.java
Printed on June 14, 2001 at 1:20 PM by Sergio Rosales

Page 2 of 2

```
boolean ret = true ;
Set s = new TreeSet();

    for (Iterator i = parameters.iterator(); i.hasNext(); ) {
70     Parameter att = (Parameter) i.next();
        String unq = att.getName();
        if (!s.add(unq)) {
            Logger.log("CD Syntax Error: Duplicate parameter name detected: "+ unq +
25             " for operation: "+ name + " in class " + className);
            ret = false ;
        }
    }
    return ret;
}

80 /** Prints contents of operation */
public void printOperation() {
    StringBuffer buf = new StringBuffer("Operation name: " + name);
    buf.append("\n          visType: ");
85     buf.append(visType);
    if (retType != null) {
        buf.append("\n          retType: ");
        buf.append(retType);
    }
90     buf.append("\n          property: ");
    buf.append(property);
    String out = buf.toString();
    Logger.log(out);

95     for (Iterator i = parameters.iterator(); i.hasNext(); ) {
        Parameter param = (Parameter) i.next();
        param.printParameter();
    }
100 }
```



```
// 11. UMLParameter

package fcd;

5 public class UMLParameter {

    private String name;
    private String type;
10 private String dir;

    // Data access methods

    public void setName(String name) {
15         this.name = name;
    }

    public String getName() {
20         return name;
    }

    public void setType(String type) {
        this.type = type;
    }
25

    public String getType() {
        return type;
    }

30 public void setDir(String dir) {
        this.dir = dir;
    }

    public String getDir() {
35         return dir;
    }

    /** Prints contents of parameter */
    public void printParameter() {
40         StringBuffer buf = new StringBuffer("Parameter name: " + name);
        buf.append("\n         type: ");
        buf.append(type);
        buf.append("\n         dir: ");
        buf.append(dir);
45         String out = buf.toString();
        Logger.log(out);
    }
}
```

```
// 12. Relationship

package fcd;

5
/** Models binary relationship between classes */
public class Relationship implements FCDConstants {

    private String name;
10 private RelationshipEnd end1;
    private RelationshipEnd end2;

    // Data access methods

15 public void setName(String name) {
    if (!name.equals("null"))
        this.name = name;
    }

20 public String getName() {
    return name;
    }

    public void setEnd1(RelationshipEnd end) {
25     this.end1 = end;
    }

    public RelationshipEnd getEnd1() {
30     return end1;
    }

    public void setEnd2(RelationshipEnd end) {
        this.end2 = end;
    }

35 public RelationshipEnd getEnd2() {
    return end2;
    }

40 public String getWholeName() {
    if (isAggregation()) {
        if ((end1.getKind()).equals(AGGREG))
            return end1.getClassName();
        else
45         return end2.getClassName();
    }
    return null ;
    }

50 public String getPartName() {
    if (isAggregation()) {
        if ((end1.getKind()).equals(AGGREG))
            return end2.getClassName();
        else
55         return end1.getClassName();
    }
    return null ;
    }

60
/** Determines the kind of the relationship */
public String getRelationshipKind() {
    if (isAssociation())
        return ASSOCIATION;
65     else if (isAggregation())
```

```

        return AGGREGATION;
    else if (isComposition())
        return COMPOSITION;
    else if (isGeneralisation())
70     return GENERALISATION;
    else
        return INSTANTIATION;
    }

75  /** Checks if two given classes are the ones involved in the relationship */
    public boolean hasClassEnds(String classNameA, String classNameB) {
        return ((end1.getClassName().equals(classNameA) &&
                (end2.getClassName().equals(classNameB) ||
80         (end2.getClassName().equals(classNameA) &&
                (end1.getClassName().equals(classNameB)));
    }

    /** Determines the multiplicity of the whole end of composition (one or many) */
85  public boolean whatWholeMultiplicity() {
        if ((end1.getKind()).equals(AGGREG))
            return (end1.getMultiplicity()).whatMultiplicity();
        else
            return (end2.getMultiplicity()).whatMultiplicity();
    }

90  /** Determines the multiplicity of an aggregation end */
    public boolean whatPartMultiplicity() {
        if ((end1.getKind()).equals(AGGREG))
95         return (end2.getMultiplicity()).whatMultiplicity();
        else
            return (end1.getMultiplicity()).whatMultiplicity();
    }

    // Operations to determine if the relationship is of a given kind */
100 public boolean isAssociation() {
        return (end1.getKind()).equals(ASSOC) && (end2.getKind()).equals(ASSOC);
    }

105 public boolean isAggregation() {
        return ((end1.getKind()).equals(AGGREG) && (end2.getKind()).equals(NONE)) ||
                ((end1.getKind()).equals(NONE) && (end2.getKind()).equals(AGGREG));
    }

110 public boolean isComposition() {
        return ((end1.getKind()).equals(COMP) && (end2.getKind()).equals(NONE)) ||
                ((end1.getKind()).equals(NONE) && (end2.getKind()).equals(COMP));
    }

115 public boolean isInstantiation() {
        return ((end1.getKind()).equals(GENERIC) && (end2.getKind()).equals(NONE)) ||
                ((end1.getKind()).equals(NONE) && (end2.getKind()).equals(GENERIC));
    }

120 public boolean isGeneralisation() {
        return ((end1.getKind()).equals(SUPER) && (end2.getKind()).equals(NONE)) ||
                ((end1.getKind()).equals(NONE) && (end2.getKind()).equals(SUPER));
    }

125 /** Determines if the relationship is of a given kind */
    public boolean isRelationshipKind(String endKind) {
        return ((end1.getKind()).equals(endKind) && (end2.getKind()).equals(NONE)) ||
                ((end1.getKind()).equals(NONE) && (end2.getKind()).equals(endKind));
    }

130

```

```

/** Verifies that the two ends of the relationship are correctly formed */
public boolean checkEnds() {
    boolean ret = true ;

135     if (((end1.getKind()).equals(ASSOC) && !(end2.getKind()).equals(ASSOC)) || // association
        (!(end1.getKind()).equals(ASSOC) && (end2.getKind()).equals(ASSOC)) ) {
        Logger.log("CD SYNTAX ERROR: Incorrect association ends (" +
            end1.getKind() + ", " + end2.getKind() + ")");
        ret = false ;
140     }

    if (((end1.getKind()).equals(AGGREG) && !(end2.getKind()).equals(NONE)) || // aggregation
        (!(end1.getKind()).equals(NONE) && (end2.getKind()).equals(AGGREG)) ) {
145     Logger.log("CD SYNTAX ERROR: Incorrect aggregation ends (" +
        end1.getKind() + ", " + end2.getKind() + ")");
        ret = false ;
    }

150     if (((end1.getKind()).equals(COMP) && !(end2.getKind()).equals(NONE)) || // composition
        (!(end1.getKind()).equals(NONE) && (end2.getKind()).equals(COMP)) ) {
        Logger.log("CD SYNTAX ERROR: Incorrect composition ends (" +
            end1.getKind() + ", " + end2.getKind() + ")");
        ret = false ;
155     }

    if (((end1.getKind()).equals(SUPER) && !(end2.getKind()).equals(NONE)) || // generalisation
        (!(end1.getKind()).equals(NONE) && (end2.getKind()).equals(SUPER)) ) {
160     Logger.log("CD SYNTAX ERROR: Incorrect generalisation ends (" +
        end1.getKind() + ", " + end2.getKind() + ")");
        ret = false ;
    }

165     if (((end1.getKind()).equals(GENERIC) && !(end2.getKind()).equals(NONE)) || // instantiation
        (!(end1.getKind()).equals(NONE) && (end2.getKind()).equals(GENERIC)) ) {
        Logger.log("CD SYNTAX ERROR: Incorrect instantiation ends (" +
            end1.getKind() + ", " + end2.getKind() + ")");
        ret = false ;
    }

170     return ret;
}

/** Verifies that a name is given to an association */
175 public boolean checkAssociationHasName() {
    boolean ret = true ;
    if (((end1.getKind()).equals(ASSOC) && (name == null)) ) {
        Logger.log("CD SYNTAX ERROR: Association without name detected");
        ret = false ;
180     }

    return ret;
}

/** Verifies that the whole part of composition has multiplicity one */
185 public boolean checkCompositionMultOne() {
    boolean ret = true ;
    if (((end1.getKind()).equals(COMP) && !(end1.getMultiplicity().isMultiplicityOne())
        || ((end2.getKind()).equals(COMP) && !(end2.getMultiplicity().isMultiplicityOne())) ) {
190     Logger.log("CD SYNTAX ERROR: Multiplicity of whole part of composition not 1");
        ret = false ;
    }

    return ret;
195 }

```

```
/** Verifies that both ends of the relationship have multiplicity one */
public boolean checkRelationshipMultOne(String endKind, String relKind) {
    boolean ret = true;
200     if (
        (((end1.getKind().equals(endKind)) || (end2.getKind().equals(endKind)) ) &&
         !((end1.getMultiplicity().isMultiplicityOne() && end2.getMultiplicity().isMultiplicityOne(
        )))
        )
    ) {
205         Logger.log("CD SYNTAX ERROR: Multiplicity of " + relKind + " end not 1");
        ret = false;
    }

210     return ret;
}

/** Prints contents of relationship */
public void printRelationship() {
215     Logger.separator();
    if (name != null)
        Logger.log("Relationship name = " + name);
    Logger.log("Relationship end1 = ");
    end1.printRelationshipEnd();
220     Logger.log("Relationship end2 = ");
    end2.printRelationshipEnd();
}
}
```

```
// 13. RelationshipEnd

package fcd;
5
import java.util.*;

/** Models the ends of the relationships */
public class RelationshipEnd {
10
    private String kind;
    private String className;
    private Multiplicity multiplicity;

15    // Data access methods

    public void setKind(String kind) {
        this .kind = kind;
    }

20    public String getKind() {
        return kind;
    }

    public void setClassName(String className) {
25        this .className = className;
    }

    public String getClassName() {
30        return className;
    }

    public Multiplicity getMultiplicity() {
35        return multiplicity;
    }

    /** Utility method needed during parsing */
    public Multiplicity createMultiplicity() {
40        multiplicity = new Multiplicity();
        return multiplicity;
    }

    /** Verifies the validity of the end's multiplicity */
    public boolean checkMultiplicity() {
45        return multiplicity.isWellFormedMultiplicity();
    }

    /** Prints contents of relationship end */
    public void printRelationshipEnd() {
50        StringBuffer buf = new StringBuffer("Relationship end: ");
        buf.append("\n          kind: ");
        buf.append(kind);
        buf.append("\n          class name: ");
        buf.append(className);
55        String out = buf.toString();
        Logger.log(out);
        multiplicity.printMultiplicity();
    }
}
60
```

```

// 14. Multiplicity

package fcd;
5 import java.util.*;

/** Models the multiplicity constraint attached to a relationship end */
public class Multiplicity {
  private Collection ranges = new ArrayList();
10

  /** Utility method needed during parsing */
  public Range createRange() {
    Range range = new Range();
    ranges.add(range);
15    return range;
  }

  /** Determines if the multiplicity is one or many */
  public boolean whatMultiplicity() { // false = one, true = many
20    if (isMultiplicityMany())
      return true;
    else
      return false;
  }
25

  /** Checks that the multiplicity has the form : [a(1)..b(1), a(2)..b(2), a(K)..b(K)]
   * where K > 0, a(i) >= 0, b(0) > 0, a(i) <= b(i), 0 <= i <= K,
   * and b(i) < a(i+1), 0 <= i <= K-1 */
  public boolean isWellFormedMultiplicity() {
30    boolean ret = true;
    int size = ranges.size();
    int bsaved = 0;
    int i = 0;

35    while (ret && (i < size)) { // check all ranges as long
                                // as long as no error

      int a=0;
      int b=0;

40      Range range = (Range) ((ArrayList) ranges) .get(i);
      String begin = range.getBegin();
      String end = range.getEnd();

      try {
45        a = Integer.parseInt(begin); // ai must be numbers
        if (a >= 0) { // greater or equal to zero
          if (i > 0) {
            if (a <= bsaved) { // a(i+1) > b(i) ?
              Logger.log("CD SYNTAX ERROR: Multiplicity interranged improperly formed "+
50                "(end " + bsaved + " and next begin " + a + ")");
              ret = false;
            }
          }
        }
        try {
55          b = Integer.parseInt(end); // b(i) must be numbers except
          if (b >= 1) { // for b(K) which may be *
            if (b >= a) { // a(i) <= b(i) ?
              bsaved = b;
              i++; // so far, so good; check next
            } else { // a(i) > b(i)
60              Logger.log("CD SYNTAX ERROR: Multiplicity range improperly formed "+
                "(end " + a + " and begin " + b + ")");
              ret = false;
            }
          }
        }
        else { // all b(i) must be > 0
65

```

```

        Logger.log("CD SYNTAX ERROR: End of multiplicity range less "+
            " than 1 (" + b + ")");
        ret = false ;
    }
    } catch (Exception x) {
70         // (!((i == size-1) && (end.equals(""))) { // only b(K) may be ""
        Logger.log("CD SYNTAX ERROR: End of multiplicity range not a number (" +
            end + ")");
        ret = false ;
75     }
        i++;
    }
    } else { // a(i) < 0
        Logger.log("CD SYNTAX ERROR: Negative number in beginning of multiplicity range (" +
80             + begin + ")");
        ret = false ;
    }
    } catch (Exception x) { // a(i) not a number
        Logger.log("CD SYNTAX ERROR: Beginning of multiplicity range not a number (" +
85             begin + ")");
        ret = false ;
    }
}
return ret;
90 }

/** Checks if multiplicity is "one" -- given as range (1 .. 1) */
public boolean isMultiplicityOne(){
    return (ranges.size() == 1) && ((Range) ((ArrayList) ranges) .get(0)) .isOne();
95 }

/** Checks if multiplicity is "many" */
public boolean isMultiplicityMany(){
    return (ranges.size() > 1) || ((Range) ((ArrayList) ranges) .get(0)) .isMany();
100 }

/** Prints contents of multiplicity */
public void printMultiplicity() {
    Logger.log("Ranges: " + ranges.size());
105     for (Iterator i = ranges.iterator(); i.hasNext(); ) {
        Range range = (Range) i.next();
        range.printRange();
    }
}
110 }

```



```

// 15. Range

package fcd;
5
/** Models ranges of values; used for expressing multiplicity constraints */
public class Range {
  private String begin;
  private String end;
10
  // Data access methods
  public void setBegin(String begin) {
    this .begin = begin;
  }
  public String getBegin() {
15    return begin;
  }
  public void setEnd(String end) {
    this .end = end;
20  }
  public String getEnd() {
    return end;
  }
25
  /** Checks if range is "one", i.e., (1 .. 1) */
  public boolean isOne() {
    boolean ret = true ;
    try {
      int b = Integer.parseInt(begin);
30      int e = Integer.parseInt(end);
      if (b != 1 || e != 1)
        ret = false ;
    } catch (Exception e) {
      ret = false ;
35    }
    return ret;
  }
  /** Checks if range is "zero or one", i.e. (0..1) or (1..1) */
  public boolean isZeroOrOne() {
40    return !isMany();
  }
  /** Checks if range indicates a "many" multiplicity; i.e., neither (0 .. 1) nor (1 .. 1) */
  public boolean isMany() {
45    boolean ret = true ;
    try {
      int b = Integer.parseInt(begin);
      int e = Integer.parseInt(end);
      if ((b == 0 || b == 1) && (e == 1))
50        ret = false ;
    } catch (Exception e) {
      ret = false ;
    }
    return ret;
55  }
  /** Prints contents of range */
  public void printRange() {
    StringBuffer buf = new StringBuffer("Range begin = ");
    buf.append(begin);
60    buf.append(" .. end = ");
    buf.append(end);
    String out = buf.toString();
    Logger.log(out);
  }
65 }

```

```

// 16. ZPPSpec

package fcd;
5
import java.util.*;

/** Models a Z++ specification */
public class ZPPSpec implements FCDConstants {
10  private String name;
    private IdList givenSets; // given sets
    private StatementList globalDeclarations; // global declarations
    private Collection classes; // all other classes
    private StatementList hidingOps; // hiding operations on
15                                     // classes

    public ZPPSpec() {
        this (null);
    }

20  public ZPPSpec(String name) {
        this .name = name;
        appendClass(SYSTEM);
    }

25  // Data access methods

    public void setName(String name) {
        this .name = name;
    }

30  public String getName() {
        return name;
    }

    public IdList getGivenSets() {
35  return givenSets;
    }

    public Collection getClasses() {
40  return classes;
    }

    public ZPPClass getClass(String className) {
        ZPPClass zcls = null;
        for (Iterator i = classes.iterator(); i.hasNext(); ) {
45  zcls = (ZPPClass) i.next();
            if ((zcls.getName()).equals(className))
                break;
        }
        return zcls;
50  }

    // Append operations for the parts of the specification used in automated formalisation

    public ZPPClass appendClass(String name) {
55  ZPPClass cls = new ZPPClass(name);
        if (classes == null)
            classes = new ArrayList();
        classes.add(cls);
        return cls;
60  }

    public void appendGivenSet(String givenSet) {
        if (givenSets == null)
            givenSets = new IdList();
65  givenSets.append(givenSet);
    }

```

```
    }  
  
    public void appendHidingOps(Statement stmt) {  
        # (hidingOps == null )  
70         hidingOps = new StatementList();  
           hidingOps.append(stmt);  
    }  
  
    /** Prints contents of ZPP specification */  
75    public void printZPPSpecification() !  
        # (givenSets != null )  
           Logger.log("[ " + givenSets.listIds() + " ]");           // given sets  
        for (Iterator i = classes.iterator(); i.hasNext(); ) {  
            ZPPClass cls = (ZPPClass) i.next();  
80            cls.printZPPClass();           // classes  
        }  
        Logger.log("");  
        # (hidingOps != null ) {  
85            ArrayList statements = hidingOps.getStatements();  
               int size = statements.size();  
               for (int i = 0; i < size; i++ ) {  
                   Statement stmt = (Statement) statements.get(i);  
                   Logger.log(stmt.listLines(0));           // hiding operations  
               }  
90         }  
    }  
}
```

```

// 17. ZPPClass

package fcd;
5
import java.util.*;

/** Models a Z++ class */
10 public class ZPPClass implements FCDConstants {
    // 'external' representation
    private String name; // class name
    private IdList cparams; // class parameters
    private IdList zextends; // EXTEND clause
    private IdList publics; // PUBLICS
15 private StatementList types; // TYPES
    private StatementList functions; // FUNCTIONS
    private StatementList owns; // OWNS
    private StatementList operations; // OPERATIONS
    private StatementList returns; // RETURNS
20 private StatementList actions; // ACTIONS
    private StatementList invariant; // INVARIANT
    private StatementList history; // HISTORY

    // 'internal' representation
25 private Collection zppAttributes; // attributes, operations
    private Collection zppOperations; // and the list of hidden features

    // Constructors
    public ZPPClass() {
30     this (null );
    }
    public ZPPClass(String name) {
        if (name != null ) setName(name);
        ZPPOperation zop = new ZPPOperation("init");
35     zop.setStar(true );
        appendOperation(zop);
    }

    // Data access methods
40 public void setName(String name) {
        this .name = name;
    }
    public String getName() {
        return name;
45     }

    public void setCParams(IdList idl) {
        if (cparams == null )
            cparams = new IdList();
50     cparams.setItems(idl);
    }

    public IdList getCParams() {
        return cparams;
55     }

    public void setExtends(IdList idl) {
        if (zextends == null )
            zextends = new IdList();
60     zextends.setItems(idl);
    }

    public Collection getZPPAttributes() {
        return zppAttributes;
65     }
}

```

```
public Collection getZPPOperations() {
    return zppOperations;
}
70
public IdList getHiddenFeatures() {
    return hiddenFeatures;
}

75 // Append operations for those clauses used in the formalisation process
public void appendPublics(String feature) {
    if ( publics == null )
        publics = new IdList();
    publics.append(feature);
80 }

public void appendFunctions(Statement stmt) {
    if ( functions == null )
        functions = new StatementList();
85     functions.append(stmt);
}

public void appendOwns(Statement stmt) {
    if ( owns == null )
90     owns = new StatementList();
    owns.append(stmt);
}

public void appendReturns(Statement stmt) {
25     if ( returns == null )
        returns = new StatementList();
    returns.append(stmt);
}

100 public void appendOperations(Statement stmt) {
    if ( operations == null )
        operations = new StatementList();
    operations.append(stmt);
}
105

public void appendActions(Statement stmt) {
    if ( actions == null )
        actions = new StatementList();
    actions.append(stmt);
110 }

public void appendHiddenFeatures(String feature) {
    if ( hiddenFeatures == null )
115     hiddenFeatures = new IdList();

    hiddenFeatures.append(feature);
}

public void appendAttribute(ZPPAttribute zatt) {
120     if ( zppAttributes == null )
        zppAttributes = new ArrayList();
    zppAttributes.add(zatt);
}

125 public void appendOperation(ZPPOperation zop) {
    if ( zppOperations == null )
        zppOperations = new ArrayList();
    zppOperations.add(zop);
}
130
```

```

    public void appendInvariant(Statement stmt) {
        # (invariant == null)
        invariant = new StatementList();
        invariant.append(stmt);
135    }

    public ZPPOperation getInitOp() {
        ZPPOperation zop = null;
        for (Iterator i = zppOperations.iterator(); i.hasNext(); ) {
140            zop = (ZPPOperation) i.next();
            # ((zop.getName()).equals("init"))
                break;
            }
        return zop;
145    }

    public boolean isInitOpEmpty() {
        # ((getInitOp()).getClause() == null)
        return true;
150    return false;
    }

    /** Prints contents of class */
    public void printZPPClass() {
155        Logger.separator();
        Logger.logLine("CLASS " + name + " "); // name
        # (cparams != null)
            Logger.logLine("[ " + cparams.listIds() + " ] "); // formal parameters, if any

160        # (zextends != null && zextends.getItems() != null) // EXTENDS
            Logger.log(EXTENDS + " " + zextends.listIds() + " ");
        else
            Logger.log("");

165        Logger.logZPPListClause(PUBLICICS, publicics);
        Logger.logZPPStmtClause(TYPES, types, 3);
        Logger.logZPPStmtClause(FUNCTIONS, functions, 3);
        Logger.logZPPStmtClause(OWNS, owns, 0);
        Logger.logZPPStmtClause(RETURNS, returns, 0);
170        Logger.logZPPStmtClause(OPERATIONS, operations, 0);
        Logger.logZPPStmtClause(INVARIANT, invariant, 0);
        Logger.logZPPStmtClause(ACTIONS, actions, 0);
        Logger.logZPPStmtClause(HISTORY, history, 0);
        Logger.log("END CLASS");
175    }
}

```

```

// 18. ZPPAttribute

package fcod;

5
/** Models a Z++ attribute */
public class ZPPAttribute implements FCDConstants {
    private String name; // info needed for formalisation
    private String type; // ("internal representation")
10    private String visType = PROTECTED;
    private String initValue;
    private String clause = OWNS;

    // Constructors
15
    public ZPPAttribute() {
        this (null , null );
    }

20    public ZPPAttribute(String name) {
        this (name, null );
    }

    public ZPPAttribute(String name, String type) {
25        if (name != null ) setName(name);
        if (type != null ) setType(type);
    }

    // Data access methods
30
    public void setName(String name) {
        this .name = name;
    }

35    public String getName() {
        return name;
    }

    public void setType(String type) {
40    this .type = type;
    }

    public void setNameType(String name, String type) {
45    this .name = name;
    this .type = type;
    }

    public String getType() {
50    return type;
    }

    public void setVisType(String visType) {
        this .visType = visType;
    }
55    public String getVisType() {
        return visType;
    }

    public void setInitValue(String initValue) {
60    this .initValue = initValue;
    }
    public String getInitValue() {
        return initValue;
    }
    }
65

```

```
    public void setClause(String clause) {
        this .clause = clause;
    }

70    public String getClause() {
        return clause;
    }

    /** Assembles the external representation of the attribute definition */
75    public Statement assembleZPPAttribute(boolean frozen) {
        Statement stmt = new Statement();
        if (type == null) type = "";
        String line = name + COLON + type;                // prepare for OWNS clause
        if (frozen) line = BAR + line;
80        else line += SEMICOLON;                        // or for FUNCTIONS
        stmt.addLine(line);
        return stmt;
    }

85    /** Assembles the external representation in OWNS of the attribute initialisation */
    public Statement assembleZPPAttAssignOwns() {
        Statement stmt = new Statement();
        String line = name + EQUAL + initValue;
        stmt.addLine(line);
90        return stmt;
    }

    /** Assembles the external representation in FUNCTIONS of the attribute initialisation */
95    public Statement assembleZPPAttAssignFunctions() {
        Statement stmt = new Statement();
        stmt.addLine(BAR + name + EQUAL + initValue);
        return stmt;
    }

100 }
}
```



```
// 19. ZPPOperation

package fcd;
5
import java.util.*;

/** Models a Z++ operation */
10 public class ZPPOperation implements FCDConstants{
    private String name;
    private String visType;
    private boolean star; // false by default, true if internal operation
    private String clause;
    private ZPPOpSignature opSignature = new ZPPOpSignature();
15 private ZPPOpDefinition opDefinition = new ZPPOpDefinition();

    // Constructors
    public ZPPOperation() {
20     this (null );
    }

    public ZPPOperation(String name) {
        if (name != null ) setName(name);
    }
25

    // Data access methods
    public void setName(String name) {
        this .name = name;
    }
30
    public String getName() {
        return name;
    }

    public void setStar(boolean star) {
35     this .star = star;
    }

    public boolean getStar() {
        return star;
40     }

    public void setVisType(String visType) {
        this .visType = visType;
    }
45

    public String getVisType() {
        return visType;
    }

    public void setClause(String clause) {
50     this .clause = clause;
    }

    public String getClause() {
55     return clause;
    }

    public void setSignature() {
        opSignature.addInputId();
60     }

    public void setDefinition() {
        opSignature.addInputId();
65     }
}
```

```
public ZPPopSignature getOpSignature() {
    return opSignature;
}

70 public ZPPopDefinition getOpDefinition() {
    return opDefinition;
}

/** Assembles signature for presentation in OPERATIONS or RETURNS clause */
75 public Statement assembleSignature() {
    Statement stmt = new Statement();
    String line = name + COLON + opSignature.getSignSpec() + SEMICOLON;
    stmt.addLine(line);
    return stmt;
80 }

/** Assembles definition for presentation in ACTION clause */
public Statement assembleDefinition() {
    Statement stmt = new Statement();
85 String line = name + " " + opDefinition.getDefSpec() + OPDEF;
    stmt.addLine(line);
    StatementList stmtList = opDefinition.getCode();

    if (opDefinition.getCode() != null) {
90     for (Iterator i = (stmtList.getStatements()).iterator(); i.hasNext(); ) {
        Statement st = (Statement) i.next();
        for (Iterator it = (st.getLines()).iterator(); it.hasNext(); ) {
            String str = (String) it.next();
            stmt.addLine(str+ SEMICOLON);
95         }
        }
    }
    return stmt;
100 }
```

```
// 20. ZPPOpSignature

package fcd;

5 /** Models the signature of a Z++ operation*/
public class ZPPOpSignature implements FCDConstants{

    private IdList inputDomains = new IdList();
10 private IdList outputDomains= new IdList();

    // Data access methods
    public void appendInputDomain(String dom) {
15         inputDomains.append(dom);
    }

    public void appendOutputDomain(String dom) {
        outputDomains.append(dom);
    }

20 public IdList getInputDomains() {
    return inputDomains;
}

25 public IdList getOutputDomains() {
    return outputDomains;
}

    /** Assembles the external representation of the operation's signature */
30 public String getSignSpec() {
    return inputDomains.listIds() + ARROW_RIGHT + outputDomains.listIds();
}
}
```

```

// 21. ZPPOpDefinition

package fcd;

5
/** Models the definition of a Z++ operation */
public class ZPPOpDefinition implements FCDCConstants{
  private Statement precondition;
  private IdList inputList = new IdList();
10  private IdList outputList= new IdList();
  private StatementList code;

  // Data access methods
  public void setPrecondition(Statement precondition) {
15    this .precondition = precondition;
  }

  public Statement getPrecondition() {
20    return precondition;
  }

  public void appendCode(Statement stmt) {
    if (code == null)
      code = new StatementList();
25    code.append(stmt);
  }

  public StatementList getCode() {
30    return code;
  }

  public void appendInputId(String id) {
    inputList.append(id);
  }
35

  public IdList getInputList() {
    return inputList;
  }

40  public void appendOutputId(String id) {
    outputList.append(id);
  }

  public IdList getOutputList() {
45    return outputList;
  }

  /** Assemble the external representation of the operation's definition */
  public String getDefSpec() {
50    return inputList.listIds() + " " + outputList.listIds();
  }

  /** Prints contents of operation */
  public void printZPPOpDefinition() {
55    StringBuffer buf = new StringBuffer("Operation signature");
    buf.append("\nInput list: " + inputList.listIds());
    buf.append("\nOutput list: " + outputList.listIds());
    String out = buf.toString();
    Logger.log(out);
60  }
}

```

```
// 22. StatementList

package fcd;
5
import java.util.*;

/** Models a list of Z++ statements */
10 public class StatementList {
    private Collection statements = new ArrayList();

    /** Appends statement to the list */
15 public void append(Statement stmt) {
        statements.add(stmt);
    }

    /** Returns the contents of the list */
20 public ArrayList getStatements() {
        return (ArrayList) statements;
    }

    /** Determines number of statements in the list */
25 public int size() {
        return statements.size();
    }
}
}
```

```

// 23. Statement

package fcd;
5
import java.util.*;

/* Models a Z++ statement of one or more lines */
public class Statement implements FCDConstants{
10  private Collection lines = new ArrayList();
    private String kind;

    // Data access methods

15  public void setKind(String kind) {
        this .kind = kind;
    }

    public String getKind() {
20      return kind;
    }

    public Collection getLines() {
25      return lines;
    }

    public int size() {
        return lines.size();
    }

30  public void addLine(String line) {
        lines.add(line);
    }

35  public void updateStatement(Statement stmt) {
        for (Iterator i = (stmt.getLines()).iterator(); i.hasNext(); )
            lines.add((String) i.next());
    }

40  /** Adds a text separator to the statement's lines */
    public void addSeparator() {
        StringBuffer buf = new StringBuffer("-");
        for (int i = 0; i < 5; i++) buf.append(buf);
        lines.add( BAR + buf.toString());
45  }

    /** Outputs lines in the form: line [\nline]* */
    public String listLines(int indent) {
        StringBuffer buf = new StringBuffer();
50      int size = lines.size();
        buf.append((String) ((ArrayList) lines).get(0));
        StringBuffer ind = new StringBuffer();
        for (int i = 0; i < indent; i++)
            ind.append(" ");
55      for (int i = 1; i < size; i++)
            buf.append("\n"+ ind + (String) ((ArrayList) lines).get(i) );

        return buf.toString();
60  }
}

```

```
// 24. IdList

package fcd;
5
import java.util.*;

/** Models a list of identifiers */
public class IdList implements FCDCConstants {
10
    private Collection items;

    public IdList() {
    }

15 // Data access methods

    public void setItems(IdList idl) {
        items = idl.getItems();
    }

20
    public Collection getItems() {
        return items;
    }

25
    public void append(String id) {
        if (items == null)
            items = new ArrayList();
        items.add(id);
    }

30
    /** Checks if a particular item belongs to the list */
    public boolean existsId(String id) {
        return items.contains(id);
    }

35
    /** Outputs items in the form: item [,item]* */
    public String listIds() {
        if (items == null)
            return EMPTY;

40
        StringBuffer buf = new StringBuffer();
        int size = items.size();
        buf.append((String) ((ArrayList) items).get(0));
        for (int i = 1; i < size; i++)
45
            buf.append(" " + (String) ((ArrayList) items).get(i) );
        return buf.toString();
    }
}
```

```

// 25. Logger

package fcd;
5 import java.io.*;
import java.util.*;

/** Utility class for handling messages to the user */
10 public class Logger implements FCDConstants {

    private static FileWriter writer;
    private static final String nl = System.getProperty("line.separator");

    /** Displays general message */
15 public static void log (String string) {
    if (writer != null) {
        try {
            writer.write(string);
            writer.write(nl);
            writer.flush();
20        } catch (Exception e) {
            System.err.println("Unable to write");
        }
    } else
25     System.out.println(string);
}

public static void logLine (String string) {
    if (writer != null) {
30     try {
        writer.write(string);
        writer.flush();
    } catch (Exception e) {
        System.err.println("Unable to write");
35     }
    } else
        System.out.print(string);
}

40 /** Displays result of test */
public static void logCheckResult(String text, boolean result) {
    if (result)
        log ("CHECK " + text + " PASSED");
    else
45     log ("CHECK " + text + " FAILED");
}

/** Prints a text separator */
public static void separator() {
50     StringBuffer buf = new StringBuffer("-");
    for (int i = 0; i < SEP_WIDTH; i++) buf.append("-");
    log(buf.toString());
}

55 /** Prints statements of Z++ clauses with given indentation */
public static void logZPPStmtClause(String clauseName, StatementList clause, int indent) {
    log(clauseName);
    if (clause != null) {
        ArrayList statements = clause.getStatements();
60
        int size = statements.size();
        for (int i = 0; i < size; i++) {
            Statement stmt = (Statement) statements.get(i);
            Logger.log(INDENT + stmt.listLines(indent));
65     }
}

```


JBuilder - Filename = D:\Work\src\jbuilder\src\src\Logger.java

Printed on June 16, 2001 at 2:12 PM by Sergio Rosales

Page 2 of 2

```
    }  
    }  
  
    /** Prints lists of identifiers included in clauses EXTENDS and PUBLICS*/  
70 public static void logZPFListClause(String clauseName, IdList clause) {  
    log(clauseName);  
    if (clause != null )  
        Logger.log(INDENT + clause.listIds());  
    }  
75  
  
    /** Initialises the output file */  
    public static void init (String fileName) {  
    try {  
80     writer = new FileWriter(fileName, false ); // false means start from the beginning  
    } catch (Exception e) {  
        System.out.println ("Unable to create the output file");  
    }  
    }  
85 }
```

```

// 26. FCD Constants

package fcd;
5
/* Collection of constants used by the AFCD */
public interface FCDConstants {
    public static final String REG      = "reg";           // kinds of classes
    public static final String PARA     = "para";
10    public static final String BIND    = "bind";

    public static final String ASSOC    = "assoc";        // relationships ends
    public static final String COMP     = "comp";
    public static final String GENERIC  = "generic";
15    public static final String AGGREG  = "aggreg";
    public static final String SUPER   = "super";
    public static final String NONE    = "none";

    public static final String ASSOCIATION = "association"; // relationships
20    public static final String AGGREGATION = "aggregation";
    public static final String COMPOSITION = "composition";
    public static final String GENERALISATION = "generalisation";
    public static final String INSTANTIATION = "instantiation";

25    public static final String PUBLIC   = "public";      // visibility
    public static final String PROTECTED = "protected";
    public static final String PRIVATE  = "private";

    public static final String CHANGEABLE = "changeable"; // property and
30    public static final String FROZEN   = "frozen";     // direction
    public static final String QUERY     = "query";
    public static final String IN        = "in";
    public static final String OUT       = "out";
    public static final String INOUT     = "inout";

35    public static final String EXTENDS  = "EXTENDS";    // clauses of Z++ class
    public static final String PUBLICS   = "PUBLICS";
    public static final String TYPES     = "TYPES";
    public static final String FUNCTIONS = "FUNCTIONS";
40    public static final String OWNS    = "OWNS";
    public static final String RETURNS   = "RETURNS";
    public static final String OPERATIONS = "OPERATIONS";
    public static final String INVARIANT = "INVARIANT";
    public static final String ACTIONS   = "ACTIONS";
45    public static final String HISTORY  = "HISTORY";

    public final static String NAT       = "unsigned int"; // data types
    public final static String BYTE     = "byte";
    public final static String INT      = "int";
50    public final static String INTEGER = "integer";
    public final static String LONG     = "long";
    public final static String REAL     = "real";
    public final static String FLOAT    = "float";
    public final static String DOUBLE   = "double";
55    public final static String VOID    = "void";
    public final static String BOOLEAN  = "boolean";
    public final static String SEQ      = "seq";

    public final static String ARROW_RIGHT = " --> "; // special symbols
60    public final static String REL_SIGN  = " <--> "; // for Z statements
    public final static String PFUNCTION = " -|-> ";
    public final static String DEF       = " ::= ";
    public final static String OPDEF     = " ==> ";
    public final static String NATURALS  = " |N";
65    public final static String INTEGERS  = " |Z";

```

JBuilder - Filename = D:/Workyard/Testbed/src/foa/PCDConstants.java

Printed on June 14, 2001 at 2:45 PM by Sergiu Dascales

Page 2 of 2

```

70 public final static String REALS      = "|R";
public final static String BOOL        = "|B";
public final static String FINITESET   = "|F";
public final static String POWERSET    = "|P";
public final static String EQUIV       = " ~ ";
public final static String QUESTION_MARK = "?";
public final static String EXCLAM_MARK = "!";
public final static String EMPTY       = "";
public final static String COLON       = ":";
75 public final static String SEMICOLON = ";";
public final static String BAR         = "|";
public final static String EQUAL       = "=";
public final static String HIDE        = "\\ ";
public final static String DOMAIN      = "dom ";
80 public final static String RANGE     = "ran ";

public final static String RESULT = "result!"; // special output variable
public final static String HIDDEN = "H";      // hidden class symbol

85 public final static String CNT       = "container"; // names of attributes
public final static String DESCR       = "Descriptor"; // and classes
public final static String INSTANCESOF = "instancesof";
public final static String INSTANCES  = "Instances";
public final static String SYSTEM      = "System";
90 public final static String THE       = "the";

public final static String INDENT = " "; // constants for output
public final static int SEP_WIDTH = 80; // format

95

```

Appendix C Harmony's User Interface

This Appendix provides details about Harmony's GUI and the functionality accessible through it. Descriptions for screenshots are presented in tabular form, each relevant element captured in a screenshot being succinctly explained by a line in the table that accompanies the screenshot.

C.1 The Harmony Window

The entire functionality of Harmony is accessible through the environment's window, shown in Fig. C.1. The main components of Harmony's GUI are visible in this figure: the menu bar, the main tool bar, the Project Pane, the UML Space, the Z++ Space, the message console, and the status bar. Several other components are also indicated.

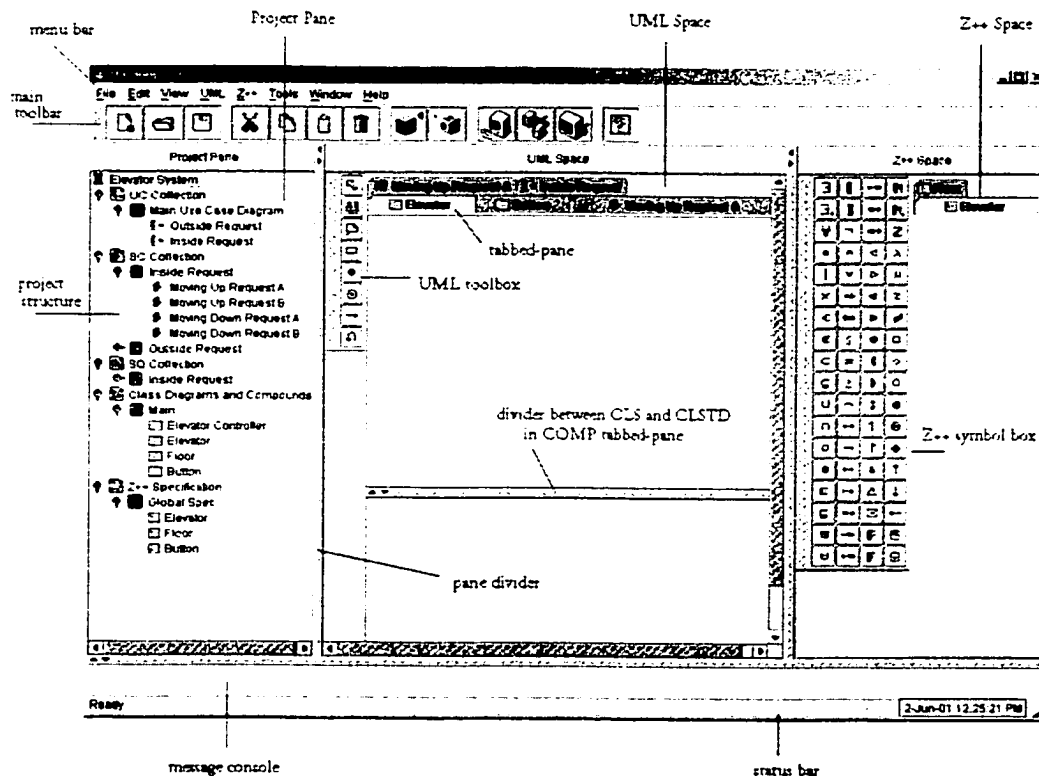


Fig. C.1 The Harmony Window

C.2 The Menu Bar

The menu bar of Harmony is shown in Fig. C.2. It is placed at the top of the Harmony window and represents the environment's main menu. Table C.I briefly describes the menus available on the menu bar. Each of these menus is further detailed later in the Appendix.

File Edit View UML Z++ Tools Window Help

Fig. C.2 Harmony's Menu Bar

Table C.I Menus of the Menu Bar

Menu	Description
File	File oriented operations including New, Open, Save project, as well as Print, Import Z++ Specification, and Export Z++ Specification
Edit	Text and graphical editing related options
View	Facilities for setting viewing options for the contents of Harmony's panes
UML	Operations related to the UML Space
Z++	Operations related to the Z++ Space
Tools	Various customisation options and facilities for add-ins
Window	Management of windows
Help	User help containing general and detailed information about Harmony

C.3 The Main Toolbar













Harmony has a main toolbar on which shortcut buttons for the most frequently used operations are placed (Fig. C.3). This floating toolbar is initially placed at the top of the Harmony window, just below the main menu, and its display can be turned on or off from the View menu.



Fig. C.3 Harmony's Main Toolbar

The correspondence between the buttons and their equivalent menu options is indicated in Table C.II.

Table C.II Shortcut Buttons and Their Equivalent Menu Options

Button	Menu Equivalent	Description
	File New	Open the New Model Element Selector (Fig. C.5) for creating a new project or a new model element
	File Open	Open an existing project
	File Save	Save the current project
	Edit Cut	Cut text or graphical object and save it in the clipboard
	Edit Copy	Copy text or graphical object to the clipboard
	Edit Paste	Paste text or graphical object from the clipboard to the location of the cursor
	Edit Delete	Delete selected text, graphical object, model element, or group of elements
	Z++ Translate to UML	Invoke the automated formalisation process
	UML Translate to Z++	Invoke the automated deformatisation process
	View Tandem Off	Turn off the tandem mode of operation
	View Tandem On	Turn on the tandem mode of operation
	Z++ Analyse	Check the syntax of Z++ specifications

C.4 The File Menu

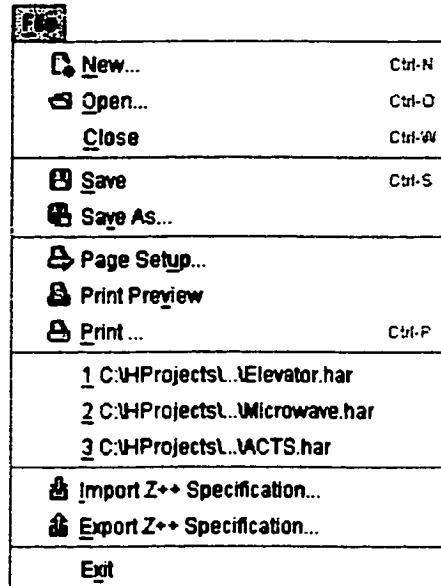


Fig. C.4 Harmony's File Menu

Table C.III File Menu Items

Item	Description
New	Display New Model Element Selector to create new project or model element
Open	Open existing project
Close	Close current project
Save	Save current project
Save As	Save current project under a different name
Page Setup	Specify settings for the printed page
Print Preview	Preview the information to be printed
Print	Print selected contents from Project Pane, UML Space, or Z++ Space
Last Projects	List of the most recent projects developed with Harmony
Import Z++ Specification	Import Z++ specifications from external files into the Z++ Space
Export Z++ Specification	Export Z++ specifications from Z++ Space to an external file
Exit	Exit the Harmony environment

C.5 The New Model Element Selector

When the New option is selected from the File Menu the user has the possibility to either create a new project or to add a new model element or group of model elements to the current project. Fig. C.5 shows the list of available options when the File | New function is invoked. Besides the icons for artefacts and groups of artefacts shown in Fig. C.5, in Harmony there are six additional symbols associated with collections of artefacts, as indicated in Table C.IV. These symbols are used in the Project Pane and the collections they represent are generated by default when each new project is created.

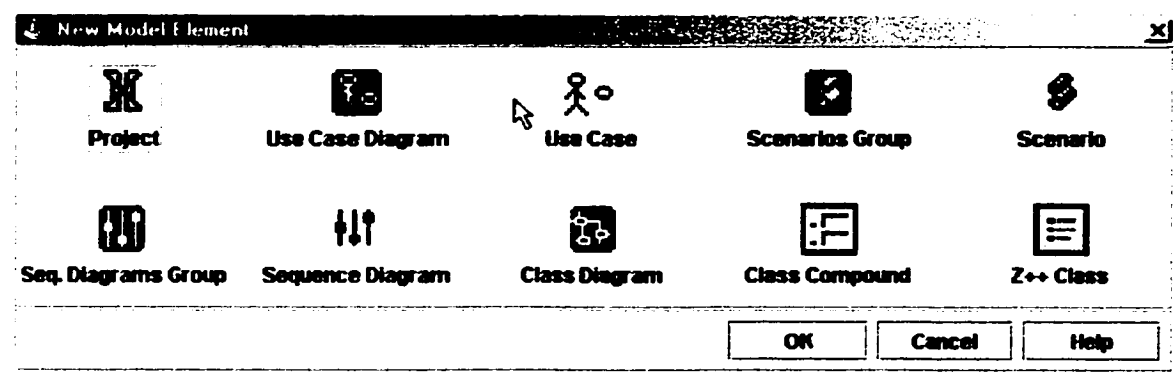








Fig. C.5 Harmony's New Model Element Selector

Table C.IV Other Icons for Artefacts

Item	Description
	The UC Collection; consists of all the use case diagrams and use cases pertaining to a project
	The SC Collection; consists of all the scenarios groups and scenarios pertaining to a project
	The SQD Collection; all the seq. diagrams groups and sequence diagrams pertaining to a project
	All the UML class diagrams and class compounds pertaining to a project
	The Z++ specification of the project
	The Global Spec, a facility for displaying in various formats the contents of the Z++ specification

C.6 The Edit Menu










Edit		
	U ndo	Ctrl-Z
	R edo	Ctrl-Y
	C ut	Ctrl-X
	C opy	Ctrl-C
	P aste	Ctrl-V
	D elete	Delete
Select		
	S elect A ll	Ctrl-A
Comment		
U ncomment		
	S earch...	Ctrl-F
	R eplace...	Ctrl-R
	S earch A gain	Ctrl-G

Fig. C.6 Harmony's Edit Menu

Table C.V Edit Menu Items

Item	Description
Undo	Undo the last editing operation
Redo	Redo the last editing operation
Cut	Cut selected text or graphical object and save it into the clipboard
Copy	Copy selected text or graphical object to the clipboard
Paste	Paste text or graphical object from the clipboard to the position of the cursor
Delete	Delete selected text or graphic object
Select	Select a segment of text or an area of a graphic
Select All	Select the entire contents of the active pane
Comment	Automatically comment selected text in the Z++ Space
Uncomment	Remove comment symbols from the selected text in the Z++ Space
Search	Search for a target text
Replace	Replace target text with new text
Search Again	Repeat search for last indicated target text

C.7 The View Menu

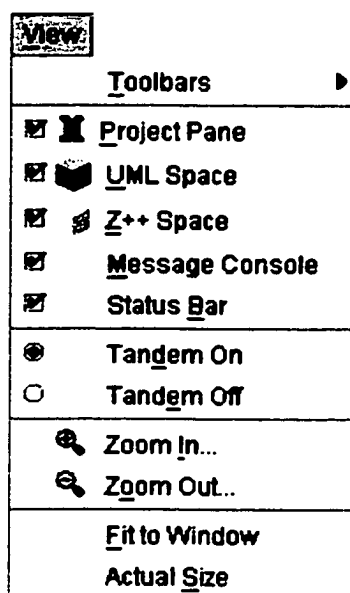


Fig. C.7 Harmony's View Menu

Table C.VI View Menu Items

Item	Description
Toolbars	Turn on or off the display of an environment's toolbar (main, UML, or Z++)
Project Pane	Checkbox for showing or hiding the Project Pane
UML Space	Checkbox for showing or hiding the UML Space
Z++ Space	Checkbox for showing or hiding the Z++ Space
Message Console	Checkbox for showing or hiding the Message Console
Status Bar	Checkbox for showing or hiding the Project Pane
Tandem On	Enable the tandem mode of operation in the UML and Z++ Spaces
Tandem Off	Disable the tandem mode of operation in the UML and Z++ Spaces
Zoom In	Zoom in on the active tabbed-pane of either the UML or the Z++ Space
Zoom Out	Zoom out on the active tabbed-pane of either the UML or the Z++ Space
Fit to Window	Show in the visible area of the active tabbed-pane the entire contents of the pane
Actual Size	Show the contents of the active tabbed-pane in actual printing size

C.8 The UML Menu

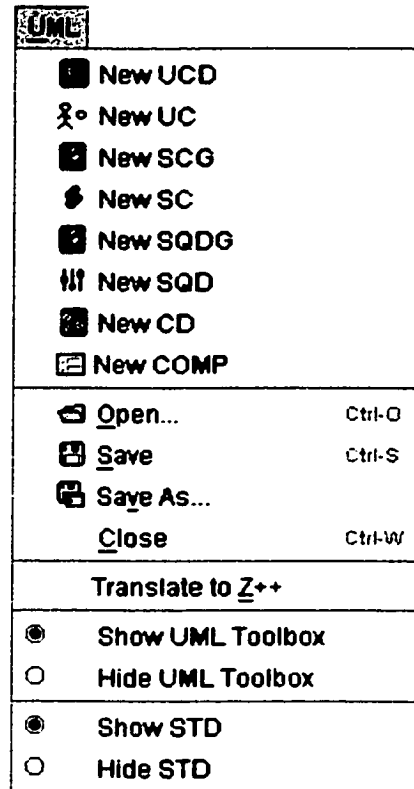


Fig. C.8 Harmony's UML Menu

Table C.VII UML Menu Items

Item	Description
New Element	Add a new UML model element to the current project
Open	Open an existing UML element into a new tabbed-pane of the UML Space
Save	Save the model element displayed in the top tabbed-pane of the UML Space
Save As	Save under a new name the model element displayed in the topmost tabbed-pane of the UML Space
Close	Close the top tabbed-pane of the UML space,
Translate to Z++	Translate to Z++ the element shown in the topmost UML tabbed-pane
Show/Hide UML Toolbox	Show or hide the UML toolbox associated with the topmost tabbed-pane
Show/Hide STD	Show or hide the STD part of a COMP construct

C.9 The Z++ Menu

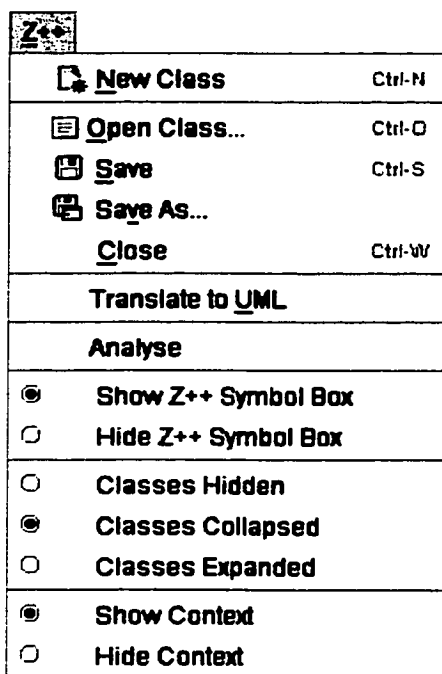


Fig. C.9 Harmony's Z++ Menu

Table C.VIII Z++ Menu Items

Item	Description
New Class	Create a new Z++ class
Open Class	Open exiting Z++ class into a new tabbed-pane in the Z++ Space
Save	Save the Z++ class shown in the topmost tabbed-pane of the Z++ Space
Save As	Save under a new name the Z++ class shown in the topmost tabbed-pane
Close	Close the Z++ class shown in the topmost tabbed-pane of the Z++ Space
Translate to UML	Invoke deformatisation on Z++ class or Z++ specification
Analyse	Execute syntax and consistency checking of the Z++ specification
Show/Hide Z++ Symbol Box	Show or hide the Z++ Symbol Box in the Z++ Space
Classes Hidden/Collapsed/Expanded	Select format of Global Spec: classes hidden, classes shown as names only, or classes fully detailed
Show/Hide Context	Show or hide above the Z++ class the global statements of Z++ specification

C.10 The Tools Menu

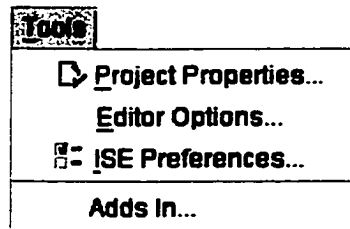


Fig. C.10 Harmony's Tools Menu

Table C.IX Tools Menu Items

Item	Description
Project Properties	Customise project properties
Editor Options	Customise text and graphical editing options
ISE Preferences	Specify general preferences for the use of Harmony
Adds In	Add connections to external tools

C.11 The Window Menu

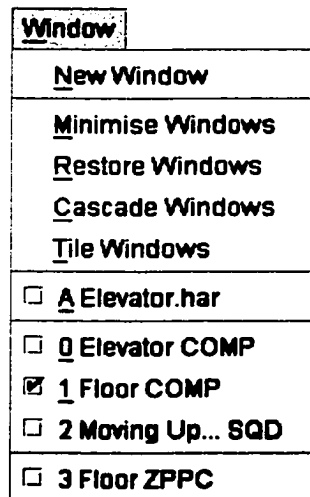


Fig. C.11 Harmony's Window Menu

Table C.X Windows Menu Items

Item	Description
New Window	Create new Harmony Window
Minimise Window	Minimise the current active window
Restore Windows	Restore all Harmony windows to their original position and size
Cascade Windows	Display windows left to right and top-down overlapping fashion
Tile Windows	Display windows in tiled format (non-overlapping)
Project Pane Contents	Contains list of opened projects; the one in the active window is checked
UML Space Contents	List opened UML model elements; the one in the active window is checked
Z++ Space Contents	List opened Z++ classes; the one in the active window is checked

C.12 The Help Menu

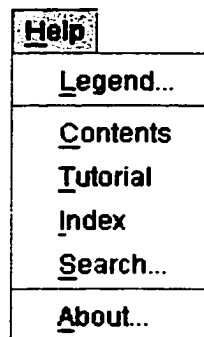
**Fig. C.12** Harmony's Help Menu

Table C.XI Help Menu Items

Item	Description
Legend	Show icons associated with artefacts or group of artefacts (see also Fig. C.15)
Contents	Display help contents
Tutorial	Invoke a tutorial on using Harmony
Index	Display keyword index of help
Search	Search for a particular keyword in the Harmony help manual
About	Show the About notice

C.13 UML Toolboxes

There are five UML toolboxes available in Harmony, each corresponding to a type of artefact produced during the modelling process. They are shown in Fig. C.12 and because several symbols appear in more than one toolbox a single table that gathers the descriptions of all UML symbols is presented (Table C.XII). This table contains 28 symbols, four of them not belonging to the UML standard. Marked with a star in the table, they are added for aiding the informal development of scenarios.

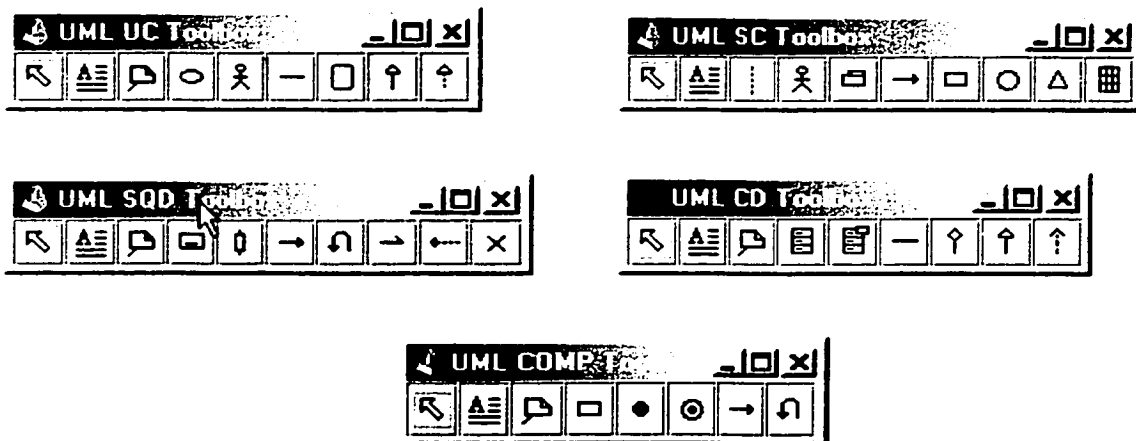
**Fig. C.13** Harmony's UML Toolboxes

Table C.XII UML Toolbox Items

Symbol	Description	Symbol	Description
	item selector		triangle*
	Text		table*
	Annotation		Object or class in sequence diagrams
	use case		activation bar
	Actor		message or transition to self
	Association		asynchronous message
	system boundary		Return message
	Generalisation		destroy object
	Uses		Class
	Timeline		parameterised class
	system or subsystem		aggregation
	Message		dependency
	state (also rectangle* in the SC toolbox)		start state
	circle*		end state

C.14 Z++ Symbol Box

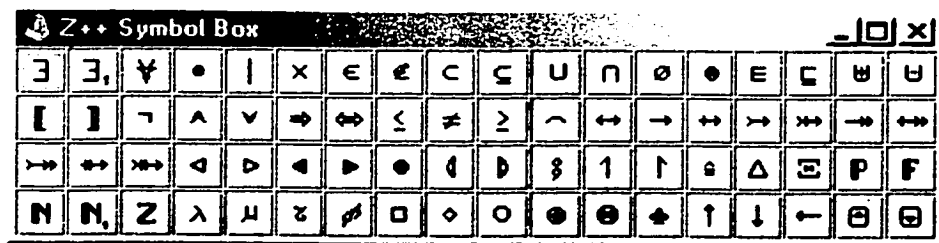


Fig. C.14 Harmony's Z++ Symbol Box

Table C.XIII Items of the Z++ Symbol Box

Symbol	Description	Symbol	Description
\exists	existential quantifier	\rightarrow	bijection
$\exists_!$	unique existential quantifier	\leftrightarrow	finite partial function
\forall	universal quantifier	$\rightarrow!$	finite partial injection
\bullet	separator in Z expressions	\triangleleft	domain restriction
	separator in Z expressions	\triangleright	range restriction
\times	Cartesian product	\triangleleft	domain subtraction
\in	Membership	\triangleright	range subtraction
\notin	non-membership	\bullet	overriding
\subset	proper subset relation	\downarrow	relation image (left marker)
\subseteq	subset relation	\uparrow	relation image (right marker)
\cup	set union	\S	relations or schemas composition
\cap	set intersection	\uparrow	extraction
\emptyset	empty set	\uparrow	filtering
\otimes	bag scaling	\triangleq	definition
\in	bag membership	Δ	delta convention
\sqsubseteq	sub-bag membership	Ξ	xi convention
\oplus	bag union	\mathcal{P}	power set
\ominus	bag difference	\mathcal{F}	finite sets
[bag display (left marker)	\mathbb{N}	the natural numbers
]	bag display (right marker)	$\mathbb{N}_!$	the strictly positive natural numbers
\neg	Negation	\mathbb{Z}	the integer numbers
\wedge	conjunction	λ	lambda convention
\vee	disjunction	μ	mu convention
\Rightarrow	implication	τ	tau
\Leftrightarrow	equivalence	ϕ	phi
\leq	less or equal	\square	always
\neq	inequality	\diamond	eventually
\geq	greater or equal	\circ	next operation initiation time
()	concatenation	\oplus	holds at
\rightarrow	binary relationship	\oplus	marker for term values ("value at")
\rightarrow	total function	\oplus	time marker for event occurrences
\rightarrow	partial function	\uparrow	begin action (operation)
$\rightarrow!$	total injection	\downarrow	end action (operation)
$\rightarrow!$	partial injection	\leftarrow	operation invocation and return
\rightarrow	total surjection	\supset	superscript
\rightarrow	partial surjection	$\substack{\supset}$	subscript

C.15 The Legend Pane

From the Help Menu the user has the possibility of displaying a legend pane which shows the icons associated with artefacts and groups of artefacts, as indicated in Fig. C. 15. In this figure, only one of the five tabbed-panes of the Legend is shown, the other icons being already presented in some of the previous Harmony snapshots.

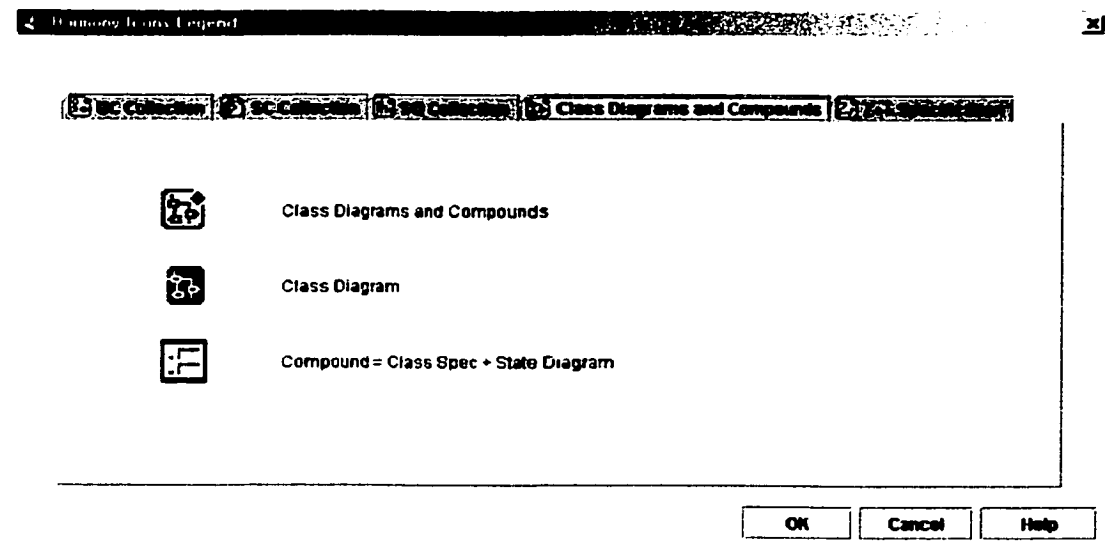


Fig. C.15 Harmony's Legend Pane: COMP and CD Symbols

C.16 Messages

In Harmony, there are three types of messages. The first type of messages are displayed in the message console and are used typically to indicate the success or failure of a specific action. Most of the messages pertaining to the activities of formalisation, deformalisation, and analysing Z++ specifications are displayed in the message console. Warning messages represent the second category of Harmony messages. They indicate that a specific type of action is not possible in a given context are labelled with a large exclamation mark. Request for confirmation messages, the third category of Harmony messages, are introduced as a safety check before performing potentially damaging operations such as overriding

specifications or deleting model elements. The three types of Harmony messages are shown in Fig. C. 16.

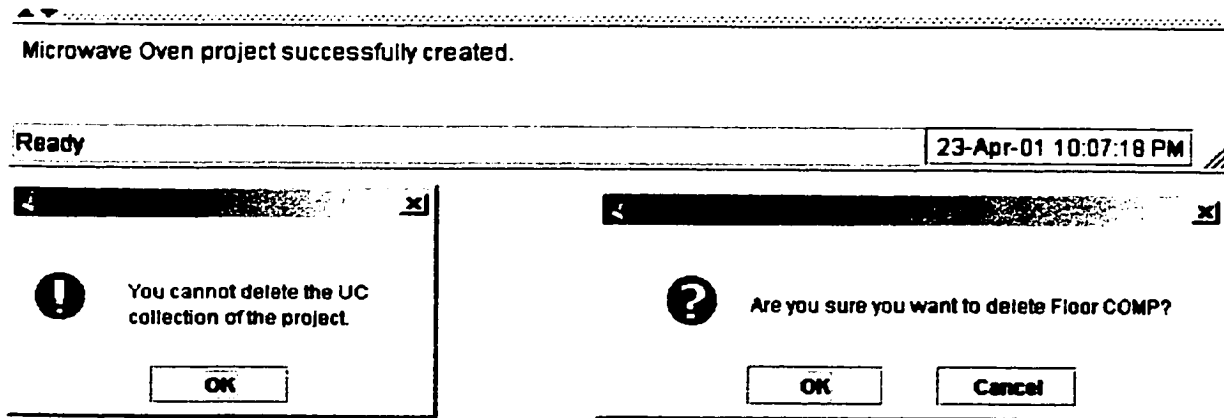


Fig. C.16 Examples of Harmony Messages

C.16 The About Pane

One last thing about Harmony is its About message, displayed through the Help | About option. As indicated in Fig. C. 17, it shows the environment's logo, its version, its authors, and the affiliation of its authors. The Harmony's logo, the metronome, is intended to suggest both our pursuit of harmony (in integrating specification notations) and the attention we pay to the passage of time (our focus on TCS). They are, we believe, the two most distinguishing aspects of our approach.

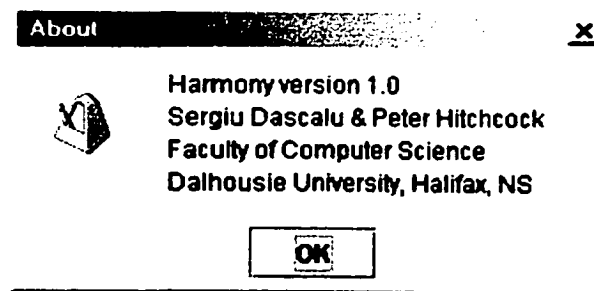


Fig. C.17 Harmony's About Message