A Paradigm Shift in Software Reengineering

by

28

Ciby Mathew

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
MASTER OF SCIENCE
in
Computer Science

Winnipeg, Manitoba, Canada, 1995

ISBN 0-612-13350-8

Name ______________________

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

COMPUTER SCIENCE — SUBJECT TERM

0 9 8 4 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

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

EDUCATION

General	0515
Administration	0514
Adult and Continuing	0516
Agricultural	0517
Art	0273
Bilingual and Multicultural	0282
Business	0688
Community College	0275
Curriculum and Instruction	0727
Early Childhood	0518
Elementary	0524
Finance	0277
Guidance and Counseling	0519
Health	0680
Higher	0745
History of	0520
Home Economics	0278
Industrial	0521
Language and Literature	0279
Mathematics	0280
Music	0522
Philosophy of	0998
Physical	0523
Psychology	0525
Reading	0535
Religious	0527
Sciences	0714
Secondary	0533
Social Sciences	0534
Sociology of	0340
Special	0529
Teacher Training	0530
Technology	0710
Tests and Measurements	0288
Vocational	0747

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

American Studies	0323
Anthropology	
Archaeology	0324
Cultural	0326
Physical	0327
Business Administration	
General	0310
Accounting	0272
Banking	0770
Management	0454
Marketing	0338
Canadian Studies	0385
Economics	
General	0501
Agricultural	0503
Commerce-Business	0505
Finance	0508
History	0509
Labor	0510
Theory	0511
Folklore	0358
Geography	0366
Gerontology	0351
History	
General	0578
Ancient	0579
Medieval	0581
Modern	0582
Black	0328
African	0331
Asia, Australia and Oceania	0332
Canadian	0334
European	0335
Latin American	0336
Middle Eastern	0333
United States	0337
History of Science	0585
Law	0398
Political Science	
General	0615
International Law and Relations	0616
Public Administration	0617
Recreation	0814
Social Work	0452
Sociology	
General	0626
Criminology and Penology	0627
Demography	0938
Ethnic and Racial Studies	0631
Individual and Family Studies	0628
Industrial and Labor Relations	0629
Public and Social Welfare	0630
Social Structure and Development	0700
Theory and Methods	0344
Transportation	0709
Urban and Regional Planning	0999
Women's Studies	0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture	
General	0473
Agronomy	0285
Animal Culture and Nutrition	0475
Animal Pathology	0476
Food Science and Technology	0359
Forestry and Wildlife	0478
Plant Culture	0479
Plant Pathology	0480
Plant Physiology	0817
Range Management	0777
Wood Technology	0746
Biology	
General	0306
Anatomy	0287
Biostatistics	0308
Botany	0309
Cell	0379
Ecology	0329
Entomology	0353
Genetics	0369
Limnology	0793
Microbiology	0410
Molecular	0307
Neuroscience	0317
Oceanography	0416
Physiology	0433
Radiation	0821
Veterinary Science	0778
Zoology	0472
Biophysics	
General	0786
Medical	0760

EARTH SCIENCES

Biogeochemistry	0425
Geochemistry	0996
Geodesy	0370
Geology	0372
Geophysics	0373
Hydrology	0388
Mineralogy	0411
Paleobotany	0345
Paleoecology	0426
Paleontology	0418
Paleozoology	0985
Palynology	0427
Physical Geography	0368
Physical Oceanography	0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences	0768
Health Sciences	
General	0566
Audiology	0300
Chemotherapy	0992
Dentistry	0567
Education	0350
Hospital Management	0769
Human Development	0758
Immunology	0982
Medicine and Surgery	0564
Mental Health	0347
Nursing	0569
Nutrition	0570
Obstetrics and Gynecology	0380
Occupational Health and Therapy	0354
Ophthalmology	0381
Pathology	0571
Pharmacology	0419
Pharmacy	0572
Physical Therapy	0382
Public Health	0573
Radiology	0574
Recreation	0575
Speech Pathology	0460
Toxicology	0383
Home Economics	0386

PHYSICAL SCIENCES

Pure Sciences

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

Applied Sciences

Applied Mechanics	0346
Computer Science	0984
Engineering	
General	0537
Aerospace	0538
Agricultural	0539
Automotive	0540
Biomedical	0541
Chemical	0542
Civil	0543
Electronics and Electrical	0544
Heat and Thermodynamics	0348
Hydraulic	0545
Industrial	0546
Marine	0547
Materials Science	0794
Mechanical	0548
Metallurgy	0743
Mining	0551
Nuclear	0552
Packaging	0549
Petroleum	0765
Sanitary and Municipal	0554
System Science	0790
Geotechnology	0428
Operations Research	0796
Plastics Technology	0795
Textile Technology	0994

PSYCHOLOGY

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

Nom ______________________________

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

______________________________ ☐☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS

Architecture	0729
Beaux-arts	0357
Bibliothéconomie	0399
Cinéma	0900
Communication verbale	0459
Communications	0708
Danse	0378
Histoire de l'art	0377
Journalisme	0391
Musique	0413
Sciences de l'information	0723
Théâtre	0465

ÉDUCATION

Généralités	515
Administration	0514
Art	0273
Collèges communautaires	0275
Commerce	0688
Économie domestique	0278
Éducation permanente	0516
Éducation préscolaire	0518
Éducation sanitaire	0680
Enseignement agricole	0517
Enseignement bilingue et multiculturel	0282
Enseignement industriel	0521
Enseignement primaire.	0524
Enseignement professionnel	0747
Enseignement religieux	0527
Enseignement secondaire	0533
Enseignement spécial	0529
Enseignement supérieur	0745
Évaluation	0288
Finances	0277
Formation des enseignants	0530
Histoire de l'éducation	0520
Langues et littérature	0279
Lecture	0535
Mathématiques	0280
Musique	0522
Orientation et consultation	0519
Philosophie de l'éducation	0998
Physique	0523
Programmes d'études et enseignement	0727
Psychologie	0525
Sciences	0714
Sciences sociales	0534
Sociologie de l'éducation	0340
Technologie	0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE

Langues	
Généralités	0679
Anciennes	0289
Linguistique	0290
Modernes	0291
Littérature	
Généralités	0401
Anciennes	0294
Comparée	0295
Mediévale	0297
Moderne	0298
Africaine	0316
Américaine	0591
Anglaise	0593
Asiatique	0305
Canadienne (Anglaise)	0352
Canadienne (Française)	0355
Germanique	0311
Latino-américaine	0312
Moyen-orientale	0315
Romane	0313
Slave et est-européenne	0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE

Philosophie	0422
Religion	
Généralités	0318
Clergé	0319
Études bibliques	0321
Histoire des religions	0320
Philosophie de la religion	0322
Théologie	0469

SCIENCES SOCIALES

Anthropologie	
Archéologie	0324
Culturelle	0326
Physique	0327
Droit	0398
Économie	
Généralités	0501
Commerce-Affaires	0505
Économie agricole	0503
Économie du travail	0510
Finances	0508
Histoire	0509
Théorie	0511
Études américaines	0323
Études canadiennes	0385
Études féministes	0453
Folklore	0358
Géographie	0366
Gérontologie	0351
Gestion des affaires	
Généralités	0310
Administration	0454
Banques	0770
Comptabilité	0272
Marketing	0338
Histoire	
Histoire générale	0578
Ancienne	0579
Médiévale	0581
Moderne	0582
Histoire des noirs	0328
Africaine	0331
Canadienne	0334
États-Unis	0337
Européenne	0335
Moyen-orientale	0333
Latino-américaine	0336
Asie, Australie et Océanie	0332
Histoire des sciences	0585
Loisirs	0814
Planification urbaine et régionale	0999
Science politique	
Généralités	0615
Administration publique	0617
Droit et relations internationales	0616
Sociologie	
Généralités	0626
Aide et bien-àtre social	0630
Criminologie et établissements pénitentiaires	0627
Démographie	0938
Études de l' individu et de la famille	0628
Études des relations interethniques et des relations raciales	0631
Structure et développement social	0700
Théorie et méthodes.	0344
Travail et relations industrielles	0629
Transports	0709
Travail social	0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES

Agriculture	
Généralités	0473
Agronomie.	0285
Alimentation et technologie alimentaire	0359
Culture	0479
Élevage et alimentation	0475
Exploitation des péturages	0777
Pathologie animale	0476
Pathologie végétale	0480
Physiologie végétale	0817
Sylviculture et faune	0478
Technologie du bois	0746
Biologie	
Généralités	0306
Anatomie	0287
Biologie (Statistiques)	0308
Biologie moléculaire	0307
Botanique	0309
Cellule	0379
Écologie	0329
Entomologie	0353
Génétique	0369
Limnologie	0793
Microbiologie	0410
Neurologie	0317
Océanographie	0416
Physiologie	0433
Radiation	0821
Science vétérinaire	0778
Zoologie	0472
Biophysique	
Généralités	0786
Medicale	0760

SCIENCES DE LA TERRE

Biogéochimie	0425
Géochimie	0996
Géodésie	0370
Géographie physique	0368
Géologie	0372
Géophysique	0373
Hydrologie	0388
Minéralogie	0411
Océanographie physique	0415
Paléobotanique	0345
Paléoécologie	0426
Paléontologie	0418
Paléozoologie	0985
Palynologie	0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT

Économie domestique	0386
Sciences de l'environnement	0768
Sciences de la santé	
Généralités	0566
Administration des hipitaux	0769
Alimentation et nutrition	0570
Audiologie	0300
Chimiothérapie	0992
Dentisterie	0567
Développement humain	0758
Enseignement	0350
Immunologie	0982
Loisirs	0575
Médecine du travail et thérapie	0354
Médecine et chirurgie	0564
Obstétrique et gynécologie	0380
Ophtalmologie	0381
Orthophonie	0460
Pathologie	0571
Pharmacie	0572
Pharmacologie	0419
Physiothérapie	0382
Radiologie	0574
Santé mentale	0347
Santé publique	0573
Soins infirmiers	0569
Toxicologie	0383

SCIENCES PHYSIQUES

Sciences Pures

Chimie	
Genéralités	0485
Biochimie	487
Chimie agricole	0749
Chimie analytique	0486
Chimie minérale	0488
Chimie nucléaire	0738
Chimie organique	0490
Chimie pharmaceutique	0491
Physique	0494
PolymÇres	0495
Radiation	0754
Mathématiques	0405
Physique	
Généralités	0605
Acoustique	0986
Astronomie et astrophysique	0606
Electronique et électricité	0607
Fluides et plasma	0759
Météorologie	0608
Optique	0752
Particules (Physique nucléaire)	0798
Physique atomique	0748
Physique de l'état solide	0611
Physique moléculaire	0609
Physique nucléoire	0610
Radiation	0756
Statistiques	0463

Sciences Appliqués Et Technologie

Informatique	0984
Ingénierie	
Généralités	0537
Agricole	0539
Automobile	0540
Biomédicale	0541
Chaleur et ther modynamique	0348
Conditionnement (Emballage)	0549
Génie aérospatial	0538
Génie chimique	0542
Génie civil	0543
Génie électronique et électrique	0544
Génie industriel	0546
Génie mécanique	0548
Génie nucléaire	0552
Ingénierie des systömes	0790
Mécanique navale	0547
Métallurgie	0743
Science des matériaux	0794
Technique du pétrole	0765
Technique minière	0551
Techniques sanitaires et municipales	0554
Technologie hydraulique	0545
Mécanique appliquée	0346
Géotechnologie	0428
Matières plastiques (Technologie)	0795
Recherche opérationnelle	0796
Textiles et tissus (Technologie)	0794

PSYCHOLOGIE

Généralités	0621
Personnalité	0625
Psychobiologie	0349
Psychologie clinique	0622
Psychologie du comportement	0384
Psychologie du développement	0620
Psychologie expérimentale	0623
Psychologie industrielle	0624
Psychologie physiologique	0989
Psychologie sociale	0451
Psychométrie	0632

A PARADIGM SHIFT IN SOFTWARE REENGINEERING

BY

CIBY MATHEW

A Thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Abstract

Object-orientation has rapidly attracted attention in industry and software research for its promise in promoting reuse and in reducing maintenance costs. Formal methods are being accepted in industries because of their potentials in developing reliable and verifiable systems. Object-oriented formal methods have aggregated advantages of both object-orientation and formal methods and hence are the focus of current software technology. This thesis is a contribution to transform a functional specification in Z into an object-oriented specification in Object-Z. The paradigm shift is a part of a re-engineering project to reverse a legacy code into a functional abstraction which may then be re-engineered into the object-oriented paradigm. The methodology has been tested for a sufficiently large case study, and is scalable to industry-size projects.

Acknowledgements

My foremost gratitude goes to my father and mother who made this thesis possible, and to my siblings and their families who backed my effort every step of the way.

I would like to express my sincere gratitude to Dr.John Van Rees (Computer Science) who chaired my thesis defense, Dr.Dan Salomon (Internal Examiner – Computer Science), and Dr.R. Padmanabhan (External Examiner – Applied Mathematics and Astronomy) who took the time and effort to read through this work in spite of very hectic schedules.

I would like to express my deepest gratitude to my program advisor, Dr. Kasi Periyasamy, who simply never compromised. I appreciate Dr. Kasi's dedication and constructive criticism in shaping this thesis.

Thank you all again.

Contents

1 Introduction **1**

2 Eight Syntactic Categories in Z **11**

2.1 Basic Type 12

2.2 Abbreviation Definition 12

2.2.1 Simple and Composite Types 13

2.3 Axiomatic Description 14

2.4 Global Constraint 15

2.5 Schema Definition 16

2.6 Generic Schema 17

2.7 Generic Constant 18

2.8 Free Type Definition 18

3 An Overview of Object-Z Syntax **20**

3.1 Reference Semantics 23

3.2 Reference Semantics and Promotion 24

3.2.1 Promotion in Z 24

3.2.2 Promotion in Object-Z 27

3.3 Operators in Object-Z 28

3.3.1 The Dereferencing Operator 28

3.3.2 The Nesting Operator 29

3.3.3 The Concurrency Operator 29

3.3.4 The Distributed Concurrency Operator 30
3.3.5 The Parallel Operator 30

4 The Paradigm Shift **32**
4.1 Basic Type Definition 32
4.2 Abbreviation Definition 33
4.2.1 Primitive, Cartesian Product, and Set Type 33
4.2.2 Basic and Schema Type 34
4.3 Axiomatic Description 36
4.4 Global Constraint 38
4.5 Schema Definition 40
4.5.1 Schema Type 40
4.5.2 Schemas formed by Schema Calculus 41
4.5.3 Operation Schema 43
4.6 Generic Schema 56
4.7 Generic Constant 58
4.8 Free Type Definition 60

5 The Shared-Diary System **62**

6 The Z Specification **64**
6.1 Structural Properties 64
6.2 Behavioral Properties 84

7 The Object-Z Specification **110**

8 Conclusion **155**

9 Future Work **162**

A The Re-analysis Phase **164**

Chapter 1

Introduction

Current trend in software engineering has shifted from developing new software to maintaining existing software[32]. The reason for this fundamental shift is due to the fact that software maintenance was not paid enough attention in the past[20]. One of the important requirements for software maintenance is program understanding, i.e., the maintenance personnel must be able to understand the functionalities of the software and the internal relationships among the various components of the software. Legacy software (those written several years ago and possibly running on obsolete hardware) are hard to maintain because (i) legacy systems are usually patched when fixing bugs; (ii) often there isn't sufficient documentation for such systems; (iii) existing documentation for legacy systems might be inconsistent with their current functionalities because of the patches; and (iv) most of the legacy systems rely extensively on the hardware while the latter are obsolete and are therefore no longer maintainable.

One possible solution to recover the functionalities of a legacy system is to reverse the code of the legacy system and reimplement it on a new platform. A code can be reversed into an abstract model which is easier to modify than code; the latter contains too much detail that is likely to obscure the essential characteristic of the software. For instance, *sortedness* may be an essential property of a list of numbers, but the code for implementing an efficient sorting algorithm may not reveal the property explicitly. The abstraction also makes it easier to correct errors in the legacy system, to modify or enhance its functionality, and to benefit from current technology such as object-orientation, distributed computing, etc. This process of reversing software, modifying it, and reimplementing it on a new platform is called *reengineering*.

In general, a reengineering process consists of three major phases - reverse engineering, modification, and forward engineering. Formally defined, reverse engineering is the process of recapturing information that was used to build a system[14]; the resulting abstraction may be described in a formal language, through a diagrammatic notation, or by an informal description. The second phase introduces modification to the abstraction and a suitable medium is, once again, chosen to convey the new information. In the forward engineering phase, the modified abstraction is reimplemented into an executable code. The reengineered code should be a dual of the legacy system in the sense that whatever was accomplished by the legacy system should also be possible with the reengineered system.

This thesis is a contribution to the second phase of the reengineering process. Specifically, we apply a paradigm shift to the procedural abstraction, obtained during reverse engineering, to get an object-oriented abstraction. We claim that it is reasonable to expect the result of reversing a legacy system to be procedural because the object-oriented paradigm was likely not available at the time the legacy system was developed. This approach is distinguished from other reengineering techniques because it employs formal methods to capture the information of the two abstractions, and second, it benefits from object-orientation.

Formal Methods are mathematical descriptions and are based on well defined formalism. They generally use a specification language which has a well defined syntax and semantics. When applied to software systems, formal methods describe the structure and behavior of the software being specified, at an abstract level, thus creating abstract models of the software. Formal specifications have been used in engineering to develop mathematical models and methods that provide an insight into and understanding of structures and processes in those disciplines. Formal specifications can be adapted effectively to software maintenance engineering[4]; the benefits include:

- an approach to program understanding through an unambiguous and concise description of software structure and behavior - this is the goal of reverse-engineering;
- a verification platform to compare software functionality to a mathematical specification of its behavior;
- a medium to ensure that modifications to a software are consistent and compatible with existing functionalities;
- simplified (and accelerated) software maintenance by maintaining the specifications and development record, and not the code itself.

Formal methods are often classified into two categories:

- *model-based* languages such as Z and VDM[8], which are usually based on set theory and logic. The system is typically described in terms of abstract data types modeled by sets, relations, functions and sequences. Such languages are also called *state-based* languages;

- *property-based* languages such as OBJ and CLEAR specify the properties of the software under consideration. These languages do not define the structure or the abstract models of the software.

Despite their advantages over diagrammatic notations and documentation in natural languages, it is often contended that formal methods demand skill and considerable background knowledge to be used effectively. This is possibly one of the reasons that software industries remain hesitant to adopt the formal approach to requirements analysis and system specification[5]. However, the various myths that hinder the wide acceptance of formal methods are rapidly being dispelled in light of its successful application to problems as diverse as the INMOS[15] chip and the subway projects in France and India.

Object-Orientation adopts a view of a system as a collection of objects which interact by passing messages. Each object is distinguishable and has a state. A class represents an abstraction of a collection of objects which share identical structure and behaviour. Every object is an instance of a class. A class is designed to encapsulate (hide) the state of its objects, and instead to provide a set of services (interface) by which other objects may retrieve state or invoke a method to update state. The various classes in an application may be organized in a hierarchical fashion, where a class at a lower level in a hierarchy inherits the features (state and behavior) of a class at a higher level. In addition, a particular class called *client* may use the services of another class called *server*, if the server is accessed as an object through the state of the client.

In comparison to the procedural approach, object-orientation offers certain benefits[26];

- *comprehensibility*: each object can be reasoned about in isolation, and understanding a system in its entirety is reduced to understanding the interactions between its objects;
- *reuse*: smaller components are more likely to be reused in a system than larger ones. In addition, libraries of reusable classes facilitate reuse across applications;
- *extensibility*: an object-oriented system is often easier to extend or modify. The effect of a change to the implementation of a class, on the entire system, is easy to trace as opposed to the procedural paradigm where the effect of a change is harder to analyze;
- *documentation*: there is a one-to-one mapping from analysis to design to implementation because the outcome of one phase is a direct input to the next[25]. In contrast, the procedural paradigm suffers a disjoint mapping across the three levels; for instance, a procedural analysis may capture the requirements but it is up to a designer to choose the appropriate constructs to implement the analysis. Therefore, it is considerably easier to maintain consistency of the analysis, design, and implementation records, in object-orientation [32].

In this thesis, we use the Z formal notation to represent the procedural abstraction, and Object-Z to represent the object-oriented abstraction.

The Z Specification Language originated at the Oxford University Computing Laboratory, UK, and has evolved over the last decade into a conceptually clear and mathematically well-grounded formal specification language. Spivey[29] has been instrumental in documenting the language and his effort is widely treated as a reference. Currently, Z is being standardized.

The Z specification language is based on set theory and logic. An application may be specified by partitioning it into small and manageable pieces called *schemas*. Each schema may be reasoned about in isolation, and again in the context of an entire specification[23]. The interaction between the various schemas may be modeled through *operation schemas* which describe the behavior of the system. In addition, the Z syntax allows for constant and type definitions, and the use of generic constructs, etc. While Z is popular as a model-based language, it is also possible to write a property-based specification in Z.

We chose Z because it has well-established semantics, and a strong base of user support and hence current interest. In addition, Z is more widely used than any other procedural specification language eg. VDM, Larch etc. In terms of tool support, a variety of type checkers and theorem provers have been developed for Z; we used the FUZZ type checker to type check the specifications in this thesis. In addition, to type checking, FUZZ provides LaTeX macros for pretty printing of the specifications.

In recognition of the potential benefits of formal methods and object-orientation to software engineering, a recent research interest has been to fuse the two concepts, resulting in a variety of object-oriented formal specification languages which include Object-Z, VDM++, Fresco, etc. Having chosen Z to convey the procedural abstraction, we were confined to the Z-based object-oriented languages, namely Object-Z, ZEST, Z++, MooZ, etc., from which we chose Object-Z.

The Object-Z Specification Language [10] originated at the University of Queensland, Australia, is an object-oriented extension to the Z specification language. The language evolved in response to certain problems that were experienced in specifying applications in Z; these included:

- the smaller granularity of schemas in Z, which capture only a portion of a state or operation. The schema is inadequate to model larger structures such as data types or entire systems, which represent a collection of schemas and other components such as global definitions;
- the inconvenience of promoting operations on a component state to the scope of a containing state. While Z provides constructs to facilitate promotion, the process may obscure the more significant aspects of the specification;
- the lack of notations to describe temporal constraints or time-dependent behavior.

Fundamentally, Object-Z is an extension to Z; the primary difference between the two languages is the introduction of the *class* construct to Object-Z. In terms of object-orientation, Object-Z facilitates:

- the specification of a system in terms of classes in place of schemas of Z, and *objects* which are instantiated from classes;
- extending the definition of a class through:

  - *inheritance*, where the features of one class (subclass) are enhanced by inheriting the features of another class (superclass);
  - *aggregation*, where one class (supplier) is instantiated within another (client), and the client uses the features of the supplier.

- interaction between objects through *messages*; an object requests a service from another by sending a request to the latter;
- a log of the temporal behavior of an object, that is maintained through a *history invariant*;
- an environment to incorporate aspects of object-orientation such as *polymorphism* and *object-identity*;
- encapsulating the internal state of an object. A class may control access to state by imposing an interface between the state and clients. Typically, the interface consists of a set of operations which control how the state may be modified or retrieved.

According to Lano[19], Object-Z is possibly the most mature object specification language given the number of applications in the language, and its international takeup. In addition to its popularity, we preferred Object-Z because there was adequate literature and tool support. At present there is no type checker for Object-Z. We used an enriched version [18] of the LaTeX macros in FUZZ for pretty printing of the specifications.

The Paradigm Shift is a methodology to transform a specification written in Z to one in Object-Z. We achieve the transformation by proposing object-oriented equivalents for eight syntactic categories that are established in the Z reference manual by Spivey[29]. The transformation rules have been justified by informally proving that the application domain information is preserved over the transformation. The process is demonstrated through the case study of a shared-diary system.

The rest of this thesis is structured as follows: Chapters 2 and 3 contain an overview of eight syntactic categories in Z, and the Object-Z formal specification language, respectively. In Chapter 4, we present the methodology to transform each of the eight syntactic categories of Z to an equivalent in Object-Z. Chapter 5 contains a brief outline of the case study; the Z and Object-Z formal specifications are presented in Chapters 6 and 7. The thesis concludes in Chapter 8 with (i) a brief summary of our effort, and (ii) a study of the various implications of using Z and Object-Z to achieve the transformation. We suggest future extensions to the thesis in Chapter 9. In the appendix, we outline a re-analysis phase that we have applied to the derived specification in order to enhance object-orientation. It is important to note that the presence of a re-analysis phase justifies the fact that the paradigm shift cannot be completely automated.

Background Work A number of methodologies have been proposed for object-oriented analysis and design[6, 24]. Some success has been achieved in integrating structural analysis and design techniques with object-orientation[33]. Sellers and Edwards[25] have shown that an object-oriented design and procedural design can be derived from each other. Bailin[3] has proposed a methodology for deriving an object-oriented design from procedural analysis techniques. Some strategies for gradually reengineering into object-orientation are described in [16, 9]. Analysis and partial automation are reported in [27, 12] and highly automated techniques for recovering abstract data types and object instances are discussed in [35, 22].

Most of the work that has been done to transform a procedural paradigm into an object-oriented paradigm is either based on diagrammatic notation and informal models, or is closely tied to finer code-level detail. Consequently, the resulting abstractions may bear some ambiguity or inconsistency in the case of diagrammatic and informal techniques, while those abstractions that are derived from code tend to be influenced by language-dependent designs.

Our approach differs from those given above because we apply the paradigm shift at an abstract level which is not biased by any programming language in particular; in addition, we propose to use formal methods which avoid the ambiguity of diagrammatic techniques and informal notation. To the best of our knowledge, the only other effort on these lines is a study in deriving an object-oriented design from a VDM specification[1].

Chapter 2

Eight Syntactic Categories in Z

A Z specification consists of interleaved paragraphs of formal and informal text. In the interests of this research, eight different syntactic categories in Z are identified[29]:

- basic type definitions;
- abbreviation definitions;
- axiomatic descriptions;
- global constraints;
- schema definitions;
- generic schemas;
- generic constants; and
- free type definitions.

An informal description and an example of each syntactic category is given below. Note that the scope of every declaration in Z starts at the point of declaration in the specification and extends until the end of the specification.

2.1 Basic Type

A basic type or given set in Z, is a type declaration consisting of a name alone. The characteristics of the basic type are not of interest to the current specification, and it is assumed that the basic type will be elaborated when the specification is refined. As an example, the declaration

$$[\mathit{COMMITTEE}]$$

introduces a basic type called *COMMITTEE*.

2.2 Abbreviation Definition

An abbreviation definition does not create a new type; rather, it composes existing types and assigns a new name to the composition. The semantics of the abbreviated definition requires that in the absence of constraints on the definition, the carrier set[1] of the abbreviation is equal to the carrier set of the expression on the right-hand side. When constraints are applied to the abbreviated definition, the elements of its carrier set are restricted to those elements of the carrier set of the right-hand side expression, that satisfy the constraints.

[1]Set of all values a variable of the type may assume.

In our terminology, the types of the expressions that may occur on the right-hand side of an abbreviation can be categorized as *simple* and *composite*. The following section examines the two types of expressions.

2.2.1 Simple and Composite Types

The simple types are the basic types or given sets, and primitive types. Basic types were discussed above. The primitive types provided by Z include the sets of *integers* ($\mathbb{Z}$), *natural numbers* ($\mathbb{N}$) and *natural numbers excluding zero* ($\mathbb{N}_1$). Composite types may be built from these simple types, and recursively from composite types themselves, by using any of three type constructors: *set*, *cartesian product*, and *schema*. Examples for simple and composite abbreviated types follow:

Abbreviated Definitions of Simple Types

The definition

$$A == \mathbb{N}$$

introduces an abbreviated definition A. In the absence of further constraints on this abbreviation, the carrier set of A is the set of natural numbers ($\mathbb{N}$) as defined in Z. If a constraint such as

$$\forall a : A \bullet a > 0$$

is defined, the carrier set of A reduces to those natural numbers greater than zero, namely, $\mathbb{N}_1$.

The type on the right hand side of an abbreviated definition may be a basic type as in

$[STRING]$

and given an abbreviation

$B == STRING$

the carrier set of B is the carrier set of $STRING$ and is not elaborated in the current specification.

Abbreviated Definitions of Composite Types

Consider the abbreviated definition of a set

$C == \mathbb{P}\, A$

and let A be the abbreviated definition of the type natural number. C represents a set of all possible sets of positive numbers.

The definition

$D == C \times \mathbb{Z}$

introduces an abbreviation for a cartesian product. The type D is now a set of all possible tuples of C and $\mathbb{Z}$.

2.3 Axiomatic Description

An axiomatic description consists of a declaration which introduces a constant, and an optional predicate which asserts a property on the constant and possibly some relationships to other constants. As an example, we consider an abbreviated definition such as

$$Time == ((hour \times mint) \times scnd)$$

and the axiomatic description

$$\begin{array}{|l}
Hour : Time \rightarrow hour \\
Minute : Time \rightarrow mint \\
Second : Time \rightarrow scnd \\
\hline
\forall t : Time \bullet \\
\quad Hour(t) = first(first(t)) \\
\quad \land\ Minute(t) = second(first(t)) \\
\quad \land\ Second(t) = second(t)
\end{array}$$

which introduces three global constants (projection functions from *Time* to its components) called *Hour*, *Minute*, and *Second*. The predicate of the axiomatic description defines the relationship between *Time* and its components.

2.4 Global Constraint

A global constraint is a predicate that constrains the value of a global declaration. In one situation, the constraint could well be the deferred predicate part of an axiomatic description that introduces the variables being constrained. As an example, consider the constraint

$$\forall t : Time \bullet Minute(t) = Second(t) = 0$$

which requires that for every instance of *Time*, the minute and second components must be zero.

2.5 Schema Definition

A schema introduces a type, and consists of a signature part and an optional predicate. When omitted, the predicate of the schema defaults to *true*. Schemas makes it possible to partition the structure and property of a specification into manageable pieces. Ideally, these fragments correspond to a conceptual unit of specification, describing some aspect of the system which has a certain independence from other aspects, though at some level all aspects will be interrelated[23].

As an example, consider the schema declaration

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Duration}} \\ \hline
\mathit{StartTime} : \mathit{Time} \\
\mathit{EndTime} : \mathit{Time} \\ \hline
\mathit{StartTime} < \mathit{EndTime} \\ \hline
\end{array}$$

which introduces a schema called *Duration*. A duration begins at *StartTime* which must occur before *EndTime*.

Operation Schema

An operation schema, a special kind of schema, describes the behavior of a structure in the specification. An operation schema may include one or more unprimed and primed schema definitions in its signature. An operation schema is said to act upon the *state space* that is instantiated from a schema definition. We present a detailed look at operation schemas in section 4.5.3.

2.6 Generic Schema

Generic schemas are similar in structure to schema definitions. A generic schema introduces a schema name and one or more generic parameters which have to be instantiated. To elaborate, a generic schema is instantiated when its formal parameters are substituted by actual parameters. An actual parameter that instantiates a formal parameter of a generic schema must be of *set* type. The predicate of the generic schema is enhanced by the predicates of the types of the actual parameters that replace the formal generic parameters. To illustrate, consider the declaration

$$
\begin{array}{|l}
\hline
\mathit{Association}\,[X, Y] \\
\mathit{dinstances} : \mathbb{P}\, X \\
\mathit{rinstances} : \mathbb{P}\, Y \\
\mathit{assoc} : X \leftrightarrow Y \\
\hline
\operatorname{dom} \mathit{assoc} = \mathit{dinstances} \\
\operatorname{ran} \mathit{assoc} = \mathit{rinstances} \\
\forall x : X \mid x \in \operatorname{dom} \mathit{assoc} \bullet \#(\mathit{assoc}\ x) = 1 \\
\hline
\end{array}
$$

which introduces a generic schema called *Association* that models the association between two generic types X and Y. The variable *assoc* expresses a relation between X and Y. The predicate of this generic schema requires that every element in the domain be mapped to exactly one element in the range. (We trade the expressive power of functions for a simplified explanation). As an example of instantiating generic schemas, consider the declaration

$$\mathit{DurDate} == \mathit{Association}[\mathit{Duration}, \mathit{Date}]$$

which requires that every *Duration* is associated with exactly one *Date*.

2.7 Generic Constant

A generic constant is similar in intent to a constant introduced by an axiomatic description. As in the case of generic schemas, a generic constant declaration requires one or more generic parameters to be instantiated by actual values. The generic constant also has an optional predicate part which defines the relationship among the generic parameters and also their relationships to other global constants. As an example, consider the generic constant called *CurrentTime*

$$
\begin{array}{|l}
\hline
[X] \\
\hline
\mathit{CurrentTime} : \emptyset[X] \twoheadrightarrow \mathit{Time} \\
\hline
\end{array}
$$

which takes a generic parameter called X and returns the current time of day. (As illustrated, the argument X itself is not of interest here, because it is perpetually substituted for the null value ($\emptyset$) of type X).

2.8 Free Type Definition

A free type introduces an enumerated type definition into a specification. The carrier set of a free type is partitioned by a collection of singleton sets each of which contains a constant, and the ranges of a set of injective functions called constructors each of which maps a well-formed expression onto the free type.

As an example, consider the declarations

$$
\begin{array}{rcl}
\mathit{SlotStatus} & ::= & \mathit{Available} \\
 & | & \mathit{Busy} \\
 & | & \mathit{Invalid}
\end{array}
$$

$$\begin{array}{lll} Annotation & ::= & persannot\langle\langle PersonalAnnotation\rangle\rangle \\ & | & commannot\langle\langle CommitteeAnnotation\rangle\rangle \end{array}$$

The declaration of *SlotStatus* defines a type whose values are the three constants *Available, Busy*, and *Invalid.* In the second example, the carrier set of *Annotation* is the union of the ranges of the two constructor functions, *persannot* and *commannot*, which accept parameters of type *PersonalAnnotation* and *CommitteeAnnotation*, respectively.

Chapter 3

An Overview of Object-Z Syntax

Object-Z is an object-oriented extension[1] to Z; the major syntactic distinction between the two languages is the introduction of the notion of *class* to Object-Z. In addition, Object-Z provides syntax to describe operations that are non-deterministic (choice) or which execute simultaneously (concurrent), and notations to describe the temporal behavior (history invariant) of objects in a specification.

The class definition is a named box with optional generic parameters. There are eight constituents in a class.

[1]Object-Z enriches the syntax of Z with object-oriented constructs.

ClassName [*Generic Parameters*]

visibility list
inherited classes
type definitions
constant definitions
state schema
initial state schema
operation schemas

---

history invariants

The type and constant definitions in a class are declared in the same syntax as in Z.

The *state schema* of a class is a nameless box. It consists of two parts, namely, a signature and a (optional) predicate. As in Z, the signature of a state schema introduces a set of variable declarations, called *state variables*, and their associated types. The predicate part of the state schema, called the *state predicate*, introduces relationships among the values of the state variables and in turn, their relationships to other declarations that are in scope to the class. To elaborate, such declarations may include global declarations, constant definitions of the class itself, and state variables and constant definitions which are made visible in other classes. (*Visibility* is explained further on in a discussion of the visibility list of a class). The state variables and constant definitions in a class are collectively referred to as *attributes*. The constant definitions can be regarded as a sub-environment of the class; they provide information that is necessary to constrain the values of the state variables, other constant definitions, and the predicates of operation schemas.

The scope[2] of the attributes and that of the class invariant extend to the initial

[2]Scope provides the semantics for encapsulation.

state schema and to every operation schema within the class. In addition, the state variables and state predicate are included (also implicitly) in unprimed and primed form in every operation schema. Hence, the class invariant is asserted, implicitly, in every possible initial state and over every operation.

The initial state schema is identified as a box labeled by the keyword *INIT*. This schema is treated as one of the class operations. The initial state schema describes a set of possible initial states for an object instantiated from the class. Note that the existence of the initial state schema does not imply that an instance of the class is initialized automatically; initialization has to be invoked explicitly.

Operation schemas are labeled boxes with the label being the name of the operation. Unlike in Z, operation schemas do not list the unprimed and primed declarations for the state being acted on. Recall that the state variables and predicate are available in unprimed and primed form to each operation schema. The state variables that are to be modified by the operation are *delta-listed* in the signature of the operation. It is implicit in Object-Z that any state variable that is not delta-listed, is guaranteed to remain unchanged over the operation. Delta-listing is one of the major semantic differences between Z and Object-Z.

Attributes and operations of a class are collectively referred to as the *features* of the class.

The visibility list is a list of features that are exported by the class. If a visibility list is omitted then all the features are visible by default. If a visibility list is defined then those class features that are not included in the list cannot be accessed outside the scope of the class. Inheritance, however, overrides the visibility settings of the parent class, within the inheriting class; where an inheritance relationship exists, all the features of the parent class are made visible to the child class. Consequently, the child class is free to decide which of its features are exportable, or should be hidden.

The *inherited classes* section lists the names of classes that are inherited by a class. Multiple inheritance is tolerated in Object-Z but the specifier is responsible for resolving conflicts in names and constraints.

The *history invariant* is a trace of the state of an object instantiated from the class; the trace may be constrained by the temporal logic operators that are available in Object-Z, which include: $\Box$ which specifies that the state always obeys some constraint; $\bigcirc$ which constrains the next state the object may assume, $\Diamond$ which constrains the object to eventually assume some state. In practice, most specification languages do not explicitly mention time[10]; an object's behavior is implicit in the operations that take the system from one state to another. However, the temporal property of a history invariant is necessary to specify concurrent behavior of objects.

3.1 Reference Semantics

Object identity is that property of an object which distinguishes it from all other objects. Object identity is implicitly incorporated into Object-Z through reference semantics[2]. Reference semantics is distinguished from value-based semantics; in the latter, objects are denoted by the value of their state. Value-based semantics are often problematic because they do not allow to distinguish between objects of the same class, which have identical states, or between objects of different classes. Reference semantics ensure that a state transition in an object changes only the state of the object and preserves the reference as a constant value.

In Object-Z, an object is instantiated by declaring a reference to the object. For example, the declaration

$c : C$

instantiates a reference to an object called c, which is instantiated from the class C.

The declaration

$c, d : C$

does not guarantee that c and d refer to different objects; if it is necessary to assert that the objects are distinct, then it has to be explicitly stated as

$c \neq d$

3.2 Reference Semantics and Promotion

The use of reference semantics helps to reduce the detail inherent in promoting an operation to an object.

3.2.1 Promotion in Z

Promotion is a technique by which an operation on a local state is extended (*promoted*) to the one on a global state. The major advantage of using promotion is a simplified specification which would otherwise be lengthy and would contain repetitive structures.

A typical promotion is carried out by selecting a state schema from the aggregation, through some criteria, and replacing the unprimed state of that entity with a primed state. The relationships among the unprimed and primed schema variables are described by the predicates of the operation schema to be promoted. A typical style for promotion is:

$$OpAgg == \exists \Delta Unit \bullet Promote \wedge OpUnit$$

Here, *OpAgg* is an operation that is promoted to an aggregation of state schemas. *Unit* is a single instance of a state schema. The operator Δ introduces the unprimed (state before the operation) and primed (state after the operation) states of *Unit* into the scope of the promotion. *Promote* is an operation schema that contains, in its signature, the unprimed and primed states of the state schema *Unit* and the state schema of the aggregation. The signature of *Promote* may also include the criteria to identify a state schema from the aggregation. *OpUnit* is an operation schema that acts on *Unit*; the predicate of *OpUnit* describes the actual operation on *Unit*. Promotion realizes a change to the state space of the aggregation through a change to the state space of the component *Unit*.

We demonstrate the technique of promotion through an example. Assume the following Z specification[17]:

Student

$id : \mathbb{N}_1$
$numcourses : \mathbb{N}$
$fees : \mathbb{N}$

StudentBody

$sset : \mathbb{P}\ Student$

$\forall s_1, s_2 : Student \mid \{s_1, s_2\} \subseteq sset \bullet$
$\quad s_1.id = s_2.id \Leftrightarrow s_1 = s_2$

CalcFees

$\Delta Student$
$nc? : \mathbb{N}$
$f? : \mathbb{N}$

$numcourses \geq nc?$
$fees' = f?$
$numcourses' = numcourses$
$id' = id$

PromStudBody

$\Delta StudentBody$
$\Delta Student$
$id? : \mathbb{N}_1$

$\theta Student \in sset$
$id? = (\theta Student).id$
$sset' = sset \setminus \{\theta Student\} \cup \{\theta Student'\}$

A student is described by a unique identifier (*id*), the number of courses (*numcourses*) in which the student is enroled, and the tuition fees (*fees*) to be paid by him/her. *Student Body* is a set of students. Operation *CalcFees* determines the fee a student has to pay based on the number of courses the student is enroled in. *PromStudBody* identifies a student on the basis of *id*, and replaces the unprimed state schema with a primed state. The promotion of *CalcFees* from *Student* to *StudentBody* may be defined through the schema calculus:

$$GCalcFees == \exists \Delta Student \bullet PromStudBody \wedge CalcFees$$

Thus the effect of a local operation *CalcFees* is promoted to the global state *StudentBody*.

3.2.2 Promotion in Object-Z

Reference semantics in Object-Z helps to alleviate some of the detail inherent in promoting operations. When an operation causes a change of state in an object, the change is implicit and the reference to the object remains constant. Consequently, when the operation is promoted to an aggregation of objects, it is no longer necessary to indicate the change to the selected object through unprimed and primed state schemas. We now present the effect of reference semantics on the example given above.

Under our transformation methodology, the state and operation schemas in the Z specification are transformed into the following classes:

Student

$id : \mathbb{N}_1$
$numcourses : \mathbb{N}$
$fees : \mathbb{N}$

CalcFees

$\Delta(fees)$
$nc? : \mathbb{N}$
$f? : \mathbb{N}$

$numcourses \geq nc?$
$fees' = f?$

StudentBody

$$sset : \mathbb{P}\, Student$$

$$\forall s_1, s_2 : Student \mid \{s_1, s_2\} \subseteq sset \bullet$$
$$s_1.id = s_2.id \Leftrightarrow s_1 = s_2$$

Select

$$s? : Student$$

$$s? \in sset$$

$$GCalcFees \mathrel{\widehat=} Select \bullet s?.CalcFees$$

We introduced an operation called *Select* which selects a student object (*s?*) from the aggregation (*sset*). The operation *GCalcFees* calculates the fees for a student by selecting the student object which is then requested (*CalcFees*) to perform the computation. Note that reference semantics have reduced the detail of promotion by replacing the explicit state change by an implicit one.

3.3 Operators in Object-Z

Object-Z introduces a number of operators that are specifically designed for use with object references. This section will detail the syntax and usage of these operators along with examples, some of which are inspired by [30].

3.3.1 The Dereferencing Operator

An object may access one of its class features through the dereferencing operator (dot). An attribute or operation may be accessed as $c.Att$ or $c.Op$ respectively.

3.3.2 The Nesting Operator

The nesting operator ($\bullet$) is a binary operator that is used to extend the signature of the schema expression on the left-hand side into the context of the schema expression on the right-hand side. Indeed, this is analogous to the ($\bullet$, read as "such that") notation used in schema expressions in Z.

As an example for the use of the nesting operator, we refer to the operation *GCalcFees* of the preceding section; in that example, the nesting operator was applied in order to extend the signature of the operation *Select* and make the object *s?* visible.

3.3.3 The Concurrency Operator

The concurrency operator ($\wedge$) is used to compose an operation to send requests to several objects, simultaneously. As an example, we refer back to the specification of section 3.2.2; we can extend the behavior of the specification by adding an operation to select any pair of students from the student body and to increment their fees. For this purpose, we may introduce an operation called *Select2*

StudentBody..contd.

Select2

$s_1? : Student$
$s_2? : Student$

$\{s_1?, s_2?\} \subseteq sset$

$PairCalcFees \mathrel{\widehat=} Select2 \bullet s_1?.CalcFees \wedge s_2?.CalcFees$

The operation *PairCalcFees* is defined as a conjunction of two requests (one to each student object) which are invoked simultaneously.

3.3.4 The Distributed Concurrency Operator

The distributed concurrency operator ($\bigwedge$) applies an operation, simultaneously, to every object belonging to some aggregate. As an example, we may introduce a new operation to the class *StudentBody* of section 3.2.2, called *IncrAll*

StudentBody..contd.

$IncrAll \mathrel{\widehat=} \bigwedge s : Student \mid s \in sset \bullet s.CalcFees$

which specifies that the fees for every student should be calculated simultaneously.

3.3.5 The Parallel Operator

The parallel operator ($\|$) is used to achieve inter-object communication. The operator equates and hides inputs and outputs having the same basename[3]. In effect, the signatures and predicates of the two operators are conjoined. It should be emphasized that the conjunction of two operations through the parallel operator is not associative; in contrast, the conjunction of the predicates of the two operations is associative.

As an example of its usage, we examine the role of the parallel operator in a system that queues jobs for printers. We present an abstract class called *PRNJOB*

PRNJOB

that represents some data to be printed. Further, it is assumed that all jobs destined for a printer are entered into a printer queue, *PrnQueue*, that is specific to a printer.

[3]The variable name with no decorations, *?*, *!*, etc.

PrnQueue

$items : \mathrm{seq}\, PRNJOB$

RemLast

$\Delta(items)$
$item! : PRNJOB$

$items = items' \frown \langle item! \rangle$

EntQueue

$\Delta(items)$
$item? : PRNJOB$

$items' = item \frown \langle item? \rangle$

The class *PrnQueue* has an operation called *RemLast* to remove the last entrant to the queue, and a second operation called *EntQueue* to append a print job to the queue. Now, we consider a scenario of two printers and hence two print queues, which is modeled through a class

TwoPrnQueues

$q_1, q_2 : PrnQueue$

$Reschedule \mathrel{\widehat=} q_1.Remlast \parallel q_2.EntQueue$

The role of the parallel operator is evident from the operation *Reschedule*: the operator extracts a print job from the end of one queue and appends it to the end of the other; this operation could be part of an optimization routine.

Chapter 4

The Paradigm Shift

In this chapter, we discuss the paradigm shift through a set of transformation rules. These rules are classified into several groups which address the transformation of the eight syntactic structures in Z, to their equivalent structures in Object-Z.

4.1 Basic Type Definition

When a basic type definition is transformed into Object-Z, a class is created and named after the basic type. This class is abstract because it has no state schema. Consider the example of the basic type

$[USER]$

which when transformed into Object-Z, becomes the abstract class *USER*

```
┌─ USER ──────────────────────────────
│ ┌───────────────────────────────
│ └───────────────────────────────
└─────────────────────────────────────
```

The transformation requires that any declaration of the basic type be transformed into an object reference to an instance of the corresponding abstract class.

It may be argued that an abstract class cannot be instantiated because it has no state. We do not contend this view, rather, we consider this class to be incomplete and we expect that the missing features will be filled in when the object specification is refined. However, to remain semantically consistent with the basic type definition of Z we propose that references to objects of the class may be declared and differentiated on the basis of their object identities.

4.2 Abbreviation Definition

We note in section 2.2 that the different type expressions that may appear on the right hand side of an abbreviated definition are the basic or schema types, which are user-defined, and the set, cartesian product, and primitive types, which are predefined in the mathematical tool-kit of Z.

Under our methodology, the user-defined types are transformed into classes, whereas the predefined types are retained. In the following sections, we propose a separate methodology to transform each category.

4.2.1 Primitive, Cartesian Product, and Set Type

These three types may be transformed as follows:

- create a class of the same name as the abbreviation;

- declare a state variable of the type of expression on the right-hand side of the abbreviation; and
- append any global constraints on the abbreviation to the state predicate.

4.2.2 Basic and Schema Type

The following rules are applied to transform the abbreviation definitions of a basic or schema type:

- create a class of the same name as the abbreviation;
- inherit the class created from the basic or schema type into the class created in the previous step; and
- append any global constraints on the abbreviation to the state predicate.

In both cases, the transformation requires that any variable declaration of the type of the abbreviation be transformed into an object reference to an instance of the corresponding class.

Examples

We now apply these rules to present the class definitions for each of the examples in section 2.2.

A

$a : \mathbb{N}$

$a > 0$

D

$d : C \times \mathbb{Z}$

C

$c : \mathbb{P}\, A$

B

STRING

Our methodology requires that a definition in Z be transformed into Object-Z before a second definition that refers to the first, may be transformed; this is in keeping with the define-before-use styles of Z and Object-Z. Therefore, we assume that the (abstract) class *STRING* has already been created, before being inherited by class *B*.

The transformation of an abbreviated definition of a predefined type has two implications:

- the scope of the type information of the abbreviation is reduced to the scope of a class; this implies that a dependence on the type of the abbreviation should be replaced by an access through an object reference into the state of an instance of the corresponding class;
- the transformation requires the introduction of an attribute in the class, of the type of the abbreviation; we may informally justify that the attribute preserves the carrier set of the abbreviation because the state of any instance of the corresponding class will assume a value from the carrier set of the abbreviation.

The transformation of an abbreviation definition of a user-defined type results in an inheritance relationship which we propose is semantically valid because:

- if the abbreviation merely renames the user-defined type then the class derived from the abbreviation inherits, redundantly, from the class derived from the user-defined type;
- if the carrier set of the abbreviation is constrained, then an instance of the subclass (derived from the abbreviation) is an instance of the superclass (derived from the user-defined type) which respects the constraint.

4.3 Axiomatic Description

When transformed into Object-Z, an axiomatic description retains its syntax, but no longer exists in a global context. Instead, it is added to, as a constant, into the classes which reference it.

When multiple schemas refer to an axiomatic description, the global constants and their predicates in reference, are transformed to each class formed from the schemas. This redundancy may be eliminated from classes related through inheritance. When every descendent of a class references a global constant, the axiomatic description may be moved to the first ancestor class that is in common to every subclass. In the event of a choice of multiply-qualifying ancestors, the Object-Z specifier is free to decide on an appropriate class. Since every declaration is local to a class, the scope of the transformed constant terminates at the end of the class definition. However, when the constant is defined in a superclass, its scope is extended appropriately.

As an example[29], consider the axiomatic description that introduces a global constant called *max*

$$\begin{array}{|l} max : \mathbb{N} \end{array}$$

and a global constant called *limit* whose value is constrained to range from zero to *max*

$$\begin{array}{|l} limit : \mathbb{N} \\ \hline limit \leq max \end{array}$$

We may use these definitions to constrain the number of items in a list (of some predefined type T)

$$\begin{array}{|l} \textit{List} \\ \hline items : \mathbb{P}\, T \\ \hline \#items < limit \\ \hline \end{array}$$

When transformed into Object-Z, the class *List* has the constants *max* and *limit* added to its constant declarations.

$$\begin{array}{|l} \textit{List} \\ \hline \begin{array}{|l} max : \mathbb{N} \\ limit : \mathbb{N} \\ \hline limit \leq max \end{array} \\ \\ \begin{array}{|l} \hline items : \mathbb{P}\, T \\ \hline \#items < limit \\ \hline \end{array} \\ \hline \end{array}$$

We may informally justify transforming an axiomatic description into a class:

- the constant is necessarily a part of the class because at least one feature refers to it;

- we eliminate the global declaration and instead encapsulate the constant;
- the constant is redundantly copied to every class which refers to it. An alternative, as in Eiffel, would be to transform every constant into a single class. However, this is undesirable because it introduces coupling between the 'constant' class and other classes which refer to the constant definitions; second, the 'constant' class may grow out of proportion and therefore fail to match the needs of another application.

However, the transformation introduces an overhead: since the original definition was global to every class, it becomes necessary to preserve the equality of the copies that are created over the transformation; this may be accomplished, for instance, in a first subclass in common to the classes which contain the constant, or in a root class which houses the entire application.

4.4 Global Constraint

A global constraint may restrict the value of a constant, or the carrier set of a type expression. By the rules of scope in Z, any reference to a constant, or declaration of a type expression, that occurs after the definition but before the constraint, is considered to be free of the constraint and is transformed, unconstrained, according to our methodology for the appropriate syntactic category.

On the other hand, a reference to a constant, or a declaration of a type expression, which is restricted by a global constraint is transformed as follows:

- if the constraint applies to a constant, then the constraint is appended to the local declarations section of every class into which the constant is transformed;

- if the constraint applies to a type expression which cannot be converted to a class, then the constraint is appended to the constant definitions section, or state predicate, in every class which contains a constant or state variable of the type expression, respectively;
- if the constraint applies to a type which can be converted into a class C_1, then:
  - if the specification introduces at least one reference to an object of C_1, ahead of the constraint, then the constraint is transformed into the state predicate of a new subclass C_2 of C_1; this restricts the carrier set of C_2 to those instances of C_1 which respect the constraint;
  - if no reference to an object of C_1 in introduced ahead of the constraint, then the constraint applies to the entire carrier set of C_1, and is transformed to the state predicate of C_1.

As an example for handling a constraint on a constant value, we refer to the example of section 4.3 where the value of *limit* is restricted by a global constraint $limit \leq max$; when transformed to Object-Z, the constraint is appended alongwith the constants *max* and *limit* to the local declarations sections of the class *List*.

As an example for handling a global constraint on a type which can be transformed into a class, we refer to the case study for the definition of a duration fragment

$$DurFrag == Duration$$

and the global constraint on the carrier set of duration fragments

$$\forall\, df : DurFrag \bullet df.EndTime = succ\ df.StartTime$$

When the types are transformed, duration fragments inherit from durations but the carrier set of duration fragments is restricted to only those durations which are an hour long. Formally, it is equivalent to

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{DurFrag}} \\
\textit{Duration} \\
\begin{array}{|l}
\hline
\textit{EndTime} = \textit{succ}\ \textit{StartTime} \\
\hline
\end{array} \\
\hline
\end{array}$$

4.5 Schema Definition

Schemas in Z serve as state space constructors as well as operation specifiers. These two types of schemas have different implications in the transformation and are consequently dealt with separately.

4.5.1 Schema Type

We apply the following rules to transform a schema type:

- the schema name is transformed into the name of a class;
- every variable in the signature of the schema becomes a state variable of the class;
- the schema predicate is transformed into a state predicate;

Informally, the transformation is justified because the carrier set of the schema definition is captured as the carrier set of the corresponding class.

As an example, consider the definition of a schema type:

```
┌─ Duration ──────────────────────────────
│ StartTime : Time
│ EndTime : Time
├──────────────
│ StartTime < EndTime
└─────────────────────────────────────────
```

When transformed, the schema *Duration*, is transformed into a class:

```
┌─ Duration ──────────────────────────────
│ ┌───────────────────────────────────────
│ │ StartTime : Time
│ │ EndTime : Time
│ ├─────────────
│ │ StartTime < EndTime
│ └───────────────────────────────────────
└─────────────────────────────────────────
```

where *StartTime* and *EndTime* are state variables and the schema predicate that constrains the schema variables now constrains the state variables.

4.5.2 Schemas formed by Schema Calculus

A schema may also be composed through the schema calculus operators ($\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$, etc.). We transform these schemas as follows:

- the schema name is transformed into a class name;
- every variable in the signature of a component schema becomes a state variable of the class;
- the schema predicates of the component schemas are transformed into the state predicate of the class; the predicates are related according to the logical connectives that relate the schemas from which they are extracted.

As an example, consider the following specification of a library[23]:

A global constant called *maxloans* defines the maximum number of loans to a patron

$$maxloans : \mathbb{N}$$

LibDB is a schema that records the stock of books in a library and the registered readers[1].

$$\begin{array}{|l}
\hline
\quad LibDB \\
\hline
stock : Copy \pfun Book \\
readers : \mathbb{F}\, Reader \\
\hline
\end{array}$$

LibLoans is concerned with information about the current loans

$$\begin{array}{|l}
\hline
\quad LibLoans \\
\hline
issued : Copy \pfun Reader \\
shelved : \mathbb{F}\, Copy \\
\hline
\forall r : Reader \bullet \#(issued \rhd \{r\}) \leq maxloans \\
shelved \cap \operatorname{dom} issued = \emptyset \\
\hline
\end{array}$$

We describe the *Library* through the conjunction operator in the schema calculus

$$Library == LibDB \wedge LibLoans$$

and present a global constraint on any *Library*

$$\begin{array}{l}
\forall lib : Library \bullet \operatorname{dom} lib.stock = lib.shelved \cup \operatorname{dom} lib.issued \\
\qquad\qquad \wedge \operatorname{ran} lib.issued \subseteq lib.readers
\end{array}$$

We can transform these definitions into Object-Z by our methodology for handling schema calculus[2]:

[1]We will assume that the types *Copy*, *Book*, and *Reader*, are defined.

[2]We assume that the classes are derived for the schema types *Copy*, *Book*, and *Reader*.

Library

$maxloans : \mathbb{N}$

$LibDB$

$LibLoans$

$stock : Copy \pfun Book$
$readers : \mathbb{F}\ Reader$
$issued : Copy \pfun Reader$
$shelved : \mathbb{F}\ Copy$

$\forall r : Reader \bullet \#(issued \rhd \{r\}) \leq maxloans$
$shelved \cap \mathrm{dom}\ issued = \emptyset$
$\mathrm{dom}\ stock = shelved \cup \mathrm{dom}\ issued$
$\mathrm{ran}\ issued \subseteq readers$

4.5.3 Operation Schema

To our knowledge, the Z base standard does not prescribe a fixed format for what exactly should appear in the signature of an operation schema. Therefore, if the problem warrants it, it is possible to omit schema types from the signature, or to have an operation schema that takes no inputs or delivers no outputs. The prevailing philosophy is the ease of expression and clarity of specification. However, we will follow a *general* schema format that is implied in the Z base standard [36], and the ISO standards document on Open Distributed Processing (ODP) [21]. At the very minimum, we anticipate that an operation schema will contain the unprimed *and* primed declarations of at least one state space, in its signature; however, it is indeed possible to omit input and output variable declarations from the signature of the operation schema.

In the next section, we present an example of the *general* operation schema; we assume this format in our discussion of the methodology for transforming operation schemas into operations in classes, in Object-Z.

A General Operation Schema

```
── Op ──────────────────────────────
│ State₁, State₁'
│ i? : IType
│ o! : OType
├────────────
│     predicate
└──────────────────────────────────
```

Op is an operation schema that acts on a state called $State_1$. The unprimed and primed declarations of $State_1$ are brought into scope in *Op* through schema inclusion which in turn introduces their respective schema variables into the scope of *Op*. The signature of *Op* contains[3] input (i) and output (o), variable declarations. By convention, the input variable name is decorated with a '?', and the output variable name is decorated with an '!'. The predicate of the operation schema asserts relationships between primed and unprimed variables as well as the input and output variables. Typically, it exhibits that:

- the values of some state variables are preserved;
- the values of some state variables change due to this operation;
- the values of the output variables are derived from the state variables and input variables.

Notations in the Transformation

[3]Assuming that the types *IType* and *OType* are defined in the specification.

For ease of understanding, we introduce additional notations in the transformation methodology (not to Z or Object-Z) which are inspired by Alagar and Kasi [1].

If a construct in the Z specification is to be transformed into a construct in the Object-Z specification, then we refer to the construct before transformation as the *pre-image* of the construct in the Object-Z specification; correspondingly, the construct after transformation is called the *image* of the construct in the Z specification. In order to ensure clarity and brevity, we distinguish the pre-image by a bar above the symbol, and its image by the original symbol itself. For example, an operation schema in Z may be named $\overline{S}$ and its image in Object-Z will be named S.

Operation on a Single Schema

In this section, we outline the steps to transform an operation schema that acts on a single state space. The transformation assumes a predicate $\overline{P}$ is expressed in conjunctive normal form $\overline{P} = \overline{P_1} \wedge \overline{P_2} \wedge \cdots \wedge \overline{P_n}$ where every $\overline{P_i}$ is called a *sub-predicate.*

The rules for the transformation follow:

- identify the class S corresponding to the state space $\overline{S}$ in the signature of the operation schema $\overline{O}$;
- create an operation O in S;
- ignore every subpredicate of the form $\overline{v_k\prime} = \overline{v_k}$ of the predicate $\overline{P_0}$ in $\overline{O}$, that exists solely to preserve the value of a schema variable $\overline{v_k}$. A justification for this step is due to the semantic difference between the Δ operators in Object-Z

and Z, as mentioned earlier. The conjunction of the remaining subpredicates of $\overline{P_0}$ is referred to as $\overline{P}$. Transform P to the predicate of the class operation O;

- identify the primed schema variables $\overline{v_k{}'}$ in the predicate $\overline{P}$ of $\overline{O}$; delta-list every attribute v_k (derived from $\overline{v_k}$) of S in the signature of O;
- transform the input and output declarations in the signature of $\overline{O}$ to the signature of O; and
- transform every global constant that is referenced in $\overline{P}$, and is not defined in S, into a local constant of S.

An Example

Assume that a global constant called *min* is defined as

$min : \mathbb{N}$

and a state space defined as

```
┌─ State₁ ──────────────────────────
│ a : ℕ
├──────────
│ a > min
└───────────────────────────────────
```

that contains a single schema variable (a) of the type natural number, with a constraint on the value of a. We create an operation schema called Op_1 that acts on an instance (state space) of $State_1$; the operation schema accepts an input and delivers an output, both of type natural number.

Op_1

$State_1, State_1'$
$i? : \mathbb{N}$
$o! : \mathbb{N}$

$min \leq i? \leq a \Rightarrow o! = i?$
$i? < min \Rightarrow o! = min$
$a < i? \Rightarrow o! = a$
$a' = a$

The predicate of Op_1 asserts that the output (o) is constrained to lie between the global constant min and the schema variable a. The postcondition of the operation

$$a' = a$$

exists solely to preserve the value of a, over the operation.

When transformed into Object-Z, the schema is converted into a class

$State_1$

$min : \mathbb{N}$

$a : \mathbb{N}$

$a > min$

Op_1

$i? : \mathbb{N}$
$o! : \mathbb{N}$

$min \leq i? \leq a \Rightarrow o! = i?$
$i? < min \Rightarrow o! = min$
$a < i? \Rightarrow o! = a$

The operation schema Op_1 is transformed into a class operation of the same name. The global constant min becomes a local constant in this class. The state variable

a is not delta-listed because it is preserved over the operation schema; we also eliminate the predicate that preserves its value. The input and output declarations are transformed to the signature of the operation, and the predicate of the operation schema becomes the predicate of the operation.

Operation on Multiple Schema Types

Let us now consider an operation schema that acts on multiple state spaces. The following rules are applied in this case:

- *Term Rewriting:* rewrite the predicate $\overline{P_0}$ of the operation schema $\overline{O}$, as follows:
  - for every primed schema variable $\overline{v_k\prime}$, determine the expression e_k which derives $\overline{v_k\prime}$;
  - determine every expression that references $\overline{v_k\prime}$ in $\overline{P_0}$ and replace $\overline{v_k\prime}$ with e_k;
  - eliminate every predicate $\overline{v_k\prime} = \overline{v_k}$, which exists solely to preserve a schema variable;
  - we refer to the *rewritten* predicate as $\overline{P}$;
- transform $\overline{P}$ into *conjunctive normal form*: $\overline{P} = \overline{P_1} \wedge \overline{P_2} \wedge \cdots \wedge \overline{P_n}$
- for each $\overline{P_i}$, create a suboperation O_i in a class S_j; S_j is the image of the state space $\overline{S_j}$ in the signature of $\overline{O}$;
- transform $\overline{P_i}$ into the predicate P_i of O_i. We recommend that $\overline{P_i}$ may be transformed into O_i if one or more of the following holds:

  - $\overline{P_i}$ asserts a *boolean condition* in $\overline{O}$ and the class S_j has at least one attribute x whose pre-image $\overline{x}$ is referenced in $\overline{P_i}$;
  - $\overline{P_i}$ derives the *primed* state of a schema variable $\overline{x}$ whose image x is an attribute of the class S_j; or
  - $\overline{P_i}$ derives an *output* and the class S_j has at least one variable x whose pre-image $\overline{x}$ is referenced in $\overline{P_i}$;

- the constants and variables that are referenced in P_i may be introduced into the context of O_i, as follows:
  - *primed schema variables*: the image of the schema variable $\overline{v_k\prime}$ is an attribute of the class S_j and the attribute v_k is delta-listed in the signature of O_i. On the other hand, if the primed schema variable $\overline{v_k\prime}$ is derived by another subpredicate $\overline{P_j}$ then $\overline{v_k}$ is transformed into the signature of O_i (see following section);
  - *unprimed schema variables*: if the image of the schema variable $\overline{v_k}$ is not an attribute of the class S_j, then $\overline{v_k}$ is transformed into the signature of O_i (see following section);
  - *inputs and outputs*: each input and output that is referenced in $\overline{P_i}$ is transformed into the signature of O_i (see following section);
  - *global constants*: every global constant that is referenced in an operation O_i is transformed into a local constant of the class to which O_i belongs;

- every invocation of the operation schema $\overline{O}$ may be transformed into a conjunct of suboperations which may be expressed in the form $O = [or_1.]\mathrm{O}_1 \wedge [or_2.]\mathrm{O}_2 \wedge \cdots \wedge [or_n.]\mathrm{O}_n$ where every or_i represents an optional object reference. We recognize the following situations for determining the expression of

O:

- identify every operation schema $\overline{O_c}$ that invokes the operation schema $\overline{O}$;
    - *Inherited Operations*: if O_c belongs to a class S_c that inherits the class S_i to which O_i is transformed, then O_i is inherited into S_c and the object reference or_i is implicit [4] and hence, is eliminated;
    - *Messages*: if O_c belongs to a class S_c that has a component class S_i to which O_i is transformed, then O_i is invoked through an object-reference to S_i[5]. The pre-image of the object reference $\overline{or_i}$ is either:
        - a schema variable of $\overline{S_i}$; or
        - introduced locally in a predicate that causes the state transition through $\overline{O_i}$;

      the object-reference or_i will, respectively, be:
        - an attribute of the class S_c; or
        - introduced locally into the expression that invokes O_i.

Establishing Type of Interface Variables

In the preceding section, we describe how a subpredicate is transformed into a suboperation of a class. In this context, we propose that every variable that is referenced in the suboperation, but is not in scope to the class of the suboperation, be transformed into an input of the suboperation. In this section, we establish the type of a variable that is transformed from the scope of an operation schema, into an interface (input or output) variable in the signature of the operation in a class.

[4]The equivalent of "this" in C++ or "self" in Smalltalk.

[5]A message is sent to the object.

If a variable $\overline{v_x}$ that is referenced in a subpredicate $\overline{P_i}$ is to be transformed into an interface variable i_x of an operation O_i then

- if $\overline{v_x}$ is a variable of a state space $\overline{S_c}$; or
- if $\overline{v_x}$ is a variable that is accessed through an input or output identifier to a state space $\overline{S_c}$; or
- if $\overline{v_x}$ is an identifier of an instantiation of a basic type $\overline{S_c}$

then i_x is declared as an object-reference to an instantiation of the class S_c, in the signature of the operation O_i. In the cases where i_x is an object-reference to the image of a schema, the variable v_x is accessed through the object-reference as: $i_x.v_x$. The object-reference i_x to an instantiation of the image of a basic type is used, as is.

On the other hand, if $\overline{v_x}$ is an interface (input or output) variable of a primitive type, then i_x will be declared to be of the same primitive type.

There are two cases to be considered while transforming operation schemas operating on multiple state spaces, namely, mutually exclusive transitions, and dependent transitions.

Mutually-Exclusive Transitions

In a mutually-exclusive transition, the state transitions of any two state schemas within an operation schema, are independent of each other. Consequently, we can partition the predicate $\overline{P}$ of the operation schema $\overline{O}$ into subpredicates $\overline{P_i}$ such that every $\overline{P_i}$ is asserted without reference to another (distinct) $\overline{P_j}$. The corresponding suboperations O_i are invoked as a conjunct; every suboperation completes with no 'knowledge' of the state space that another operation acts upon, and no access to the postcondition of any other operation.

As an example, we introduce two schema definitions:

$$\begin{array}{|l}
\multicolumn{1}{l}{S_1} \\ \hline
x : \mathbb{N} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{S_2} \\ \hline
y : \mathbb{N} \\ \hline
\end{array}$$

An operation schema $\overline{Op_m}$ squares the value of $\overline{x}$ and increments $\overline{y}$

$$\begin{array}{|l}
\multicolumn{1}{l}{Op_m} \\ \hline
S_1;\ S_1' \\
S_2;\ S_2' \\ \hline
x' = x * x \\
y' = y + 1 \\ \hline
\end{array}$$

The unprimed and primed state spaces $\overline{S_1}$ and $\overline{S_2}$ are brought into the scope of $\overline{Op_m}$ through schema inclusion. We qualify $\overline{Op_m}$ as an operation that completes through the mutually-exclusive transitions of the two state spaces of $\overline{S_1}$ and $\overline{S_2}$ because the predicate of $\overline{Op_m}$ does not assert a constraint between the variables of the two schemas.

When these definitions are transformed, we realize the classes S_1 and S_2, corresponding to $\overline{S_1}$ and $\overline{S_2}$; the operation schema $\overline{Op_m}$ is transformed into two operations OpS_x and OpS_y.

S_1

$x : \mathbb{N}$

OpS_x

$\Delta(x)$

$x' = x * x$

S_2

$y : \mathbb{N}$

OpS_y

$\Delta(y)$

$y' = y + 1$

We now examine how an operation schema that invokes $\overline{Op_m}$, may be transformed into Object-Z. Assume the definition of a schema $\overline{S_c}$

S_c

S_1

$s_2 : S_2$

We bring $\overline{S_1}$ into scope in $\overline{S_c}$ through schema inclusion and we also declare a schema variable of type $\overline{S_2}$.

Consider the operation schema $\overline{Op_M}$ that instantiates the state space that $\overline{Op_m}$ acts upon.

Op_M

ΔS_c

$o! : \mathbb{N}$

$\exists\, Op_m \mid \theta S_2 = s_2 \bullet o! = x' + (s_2').y$

We transform these definitions into Object-Z by creating a class S_c, and by transforming the operation schema $\overline{Op_m}$ into a class operation

```
┌─ $S_c$ ─────────────────────────────
│ $S_1$
│ ┌──────────────────────────────
│ │ $s_2 : S_2$
│ └──────────────────────────────
│ ┌─ $Op_M$ ─────────────────────
│ │ $o! : \mathbb{N}$
│ ├──────────
│ │ $\exists\ OpS_x \wedge s_2.OpS_y \bullet o! = x + s_2.y$
│ └──────────────────────────────
└─────────────────────────────────────
```

The conjunct $OpS_x \wedge s_2.OpS_y$, causes implicit state transitions in the two objects that are instantiated from classes S_1 and S_2; the effect is analogous to the state transitions of the two state schemas $\overline{S_1}$ and $\overline{S_2}$ over the operation schema $\overline{Op_M}$ and therefore Op_M is a *dual* of the operation $\overline{Op_M}$.

Dependent Transition

In a dependent transition, the state transitions of at least one state space in an operation schema, is dependent on the state of at least one other state space in the same operation schema.

As an example, we refer back to the definitions of the schemas $\overline{S_1}$ and $\overline{S_2}$ of the preceding section. If we change the definition of $\overline{Op_m}$ to

```
┌─ $Op_m$ ─────────────────────────────
│ $S_1;\ S_1'$
│ $S_2;\ S_2'$
├──────────
│ $x' = y$
│ $y' = x' + x$
└─────────────────────────────────────
```

we notice that each state transition of $\overline{S_1}$ and $\overline{S_2}$ is dependent on the other. We preserve the definitions of schema $\overline{S_1}$ and the operation schema $\overline{Op_M}$ which instantiates the state space that $\overline{Op_m}$ acts upon. Under the transformation methodology, we can rewrite the predicate of $\overline{Op_m}$ as:

$$x' = y$$
$$y' = y + x$$

by substituting the term x' in the predicate $y' = x' + x$. The predicate may be transformed into two operations OpS_x and OpS_y

S_1

$x : \mathbb{N}$

OpS_x

$\Delta(x)$
$s_2? : S_2$

$x' = s_2?.y$

S_2

$y : \mathbb{N}$

OpS_y

$\Delta(y)$
$s_1? : S_1$

$y' = y + s_1?.x$

each of which depends on an attribute of the other class. The predicate of $\overline{Op_M}$ (given in the preceding section) may be transformed into the predicate of Op_M

S_c

S_1

$s_2 : S_2$

Op_M

$\Delta(x)$

$o! : \mathbb{N}$

$\exists\, s_1 : S_1 \mid s_1.x = x \bullet$

$\quad \exists\, OpS_x[s_2/s_2?] \wedge s_2.OpS_y[s_1/s_1?] \bullet o! = x' + s_2.y \wedge x' = s_1.x$

This operation Op_M differs from the one we gave earlier on: the dependent transition requires that each state be made available to the other.[6].

The effect of Op_M is analogous to the state transitions of the state spaces of $\overline{S_1}$ and $\overline{S_2}$ over the operation schema $\overline{Op_M}$ and therefore Op_M is a dual of the operation schema $\overline{Op_M}$.

4.6 Generic Schema

A generic schema introduces a schema name and one or more generic parameters to the specification. Our methodology for transforming a generic schema into Object-Z is similar to that for a schema type:

- the generic schema name is transformed into a generic class name;
- the generic parameters of the schema become generic parameters of the class;

[6]In this example, we ignore the circular references to each class in the operation of the other. In an actual specification, it may be necessary to define at least one input object reference as a polymorphic parameter; this is indeed possible in Object-Z.

- every variable in the signature of the generic schema, becomes an state variable of the generic class;
- the schema predicate of the generic schema, is transformed into the state predicate of the generic class.

As an example, consider a generic schema called *AssocArray*, an associative array that relates two generic parameters, *Names* and *X*, through an injective function

$$\begin{array}{|l}\hline \textit{AssocArray}\,[\textit{Names}, X] \\ \textit{items} : \textit{Names} \rightarrowtail X \\ \hline \end{array}$$

Next, we define a basic type *USERS*, and a constant that specifies a limit on the number of users

$$[\textit{USERS}]$$
$$\textit{maxusers} : \mathbb{N}$$

The schema *AssocArray* may be instantiated by a definition such as

$$\textit{ValidUsers} \mathrel{\widehat{=}} \textit{AssocArray}[\textit{USERS}, \mathbb{N}]$$

where the generic parameters, *Names* and *X*, are instantiated by *USERS* and $\mathbb{N}$, respectively.[7]

This definition could be used in a schema *CompUsers*

$$\begin{array}{|l}\hline \textit{CompUsers} \\ \textit{vu} : \textit{ValidUsers} \\ \hline \#(\textit{vu}.\textit{items}) \leq \textit{maxusers} \\ \hline \end{array}$$

[7]Every user is given a distinct identification.

When transformed into Object-Z, a generic class called *AssocArray* is created.

$$\begin{array}{|l}
\hline
\textit{AssocArray}[\textit{Names}, X] \\
\begin{array}{|l}
\hline
\textit{items} : \textit{Names} \rightarrowtail X \\
\hline
\end{array} \\
\hline
\end{array}$$

We transform the definition of *ValidUsers* as is, to the class [8] *CompUsers*

$$\begin{array}{|l}
\hline
\textit{CompUsers} \\
\textit{maxusers} : \mathbb{N} \\
\textit{ValidUsers} \mathrel{\widehat{=}} \textit{AssocArray}[\textit{USERS}, \mathbb{N}] \\
\begin{array}{|l}
\hline
\textit{vu} : \textit{ValidUsers} \\
\hline
\#(\textit{vu}.\textit{items}) \leq \textit{maxusers} \\
\hline
\end{array} \\
\hline
\end{array}$$

4.7 Generic Constant

When transformed into Object-Z, a generic constant retains its syntax but no longer exists in a global context; instead, it is appended to, as a constant, into every class which references it. The type information that is required to instantiate the generic parameters of the constant will be in scope in each class.

As in the case of axiomatic descriptions, the transformation can result in redundant definitions of the generic constant; the redundancy may be eliminated in cases where every descendent of a class references the constant by moving the definition to the superclass common to all the descendents. In the case of multiply-qualifying ancestors, a specifier is free to decide on an appropriate class to host the definition.

[8]The basic type *USERS* is transformed into a class.

We may informally justify the transformation:

- the constant is necessarily a part of the class because at least one feature of the class refers to it;
- we eliminate the global declaration and instead encapsulate the constant.

As an example, consider the following definition for *singleton*[23], which relates an element, to a set containing just that element.

$[X]$

$singleton : X \rightarrow \{X\}$

Given that *PERSON* is a basic type

$[PERSON]$

we can create a schema called *School* that contains a set of people, one of whom is a coordinator.

School

$Secretary : PERSON$
$FacultyMembers : \mathbb{P}\ PERSON$
$WorkForce : \mathbb{P}\ PERSON$

$\langle singleton(Secretary), FacultyMembers\rangle$ partitions $WorkForce$

When these definitions are transformed into Object-Z, we define the following classes:

PERSON

School

$[X]$

$singleton : X \rightarrow \{X\}$

$Secretary : PERSON$
$FacultyMembers : \mathbb{P}\ PERSON$
$WorkForce : \mathbb{P}\ PERSON$

$\langle singleton(Secretary), FacultyMembers \rangle$ partitions $WorkForce$

4.8 Free Type Definition

A free type definition in Z is of the form:

$$F ::= c_1 \mid c_2 \mid \cdots \mid c_m \mid f_1 \langle\langle E_1 \rangle\rangle \mid f_2 \langle\langle E_2 \rangle\rangle \mid \cdots \mid f_n \langle\langle E_n \rangle\rangle$$

where every c_i is a constant and every f_j is a total injection from a well-formed type E_j to the type F. The constants and the ranges of the injective functions are all distinct and partition the carrier set of the free type [29]. We transform the free type definition according to the following rules:

- transform each well-formed expression E_j into a well-formed expression OE_j in Object-Z.
- replace every E_j in F with OE_j and rename F to F_0;
- create a class F;
- transform the free type definition F_0 into a type definition of the class F[9];

[9]Note that the scope of F_0 is now restricted to the scope of F, and is no longer available in a global context.

- create a state variable *att* of type F_0, in the class F;
- for each constant c_i, create an operation ISc_i; the predicate of ISc_i will assert the condition $att = c_i$;
- for each constructor f_j, create an operation Opf_j which asserts that a given object e_j of the type OE_j, maps through the constructor f_j onto an instance of F_0 which agrees with the value of *att*;
- given that f_j is an injective function, we also take into consideration the inverse of f_j: for each f_j, we create an operation Opf_j^{-1} which asserts that the inverse of the function f_j maps *att* onto an output object e_j of the type OE_j.

We illustrate these concepts with the actual transformation applied to the free type F. In this example, we have reduced the free type enumeration to a single constant c_1 and a single injective function f_1.

F

$F_0 ::= c_1 \mid f_1 \langle\langle OE_1 \rangle\rangle$

$att : F_0$

ISc_1

$att = c_1$

Opf_1

$e_1? : OE_1$

$att = f_1(e_1?)$

Opf_1^{-1}

$e_1! : OE_1$

$e_1! = f_1^{-1}(att)$

Chapter 5

The Shared-Diary System

In this chapter, we present the case study of a shared-diary system. This case study will first be specified using Z (in the next chapter) and then will be transformed according to our methodology, into Object-Z. For the sake of clarity and brevity, we attempt only as much detail in the case study as is necessary to demonstrate aspects of the methodology and the re-analysis process.

The shared-diary system is a tool to be used in an interactive work environment such as a department of a university. This work environment is populated by the faculty members of the department, and a secretary who coordinates the interaction among the faculty members. Each faculty member has a personal diary in which he/she can record personal annotations. There are various committees to run the business of the department; each such committee is formed by a group of faculty members. The secretary maintains a group diary in which she records committee meetings. The following services are provided by the system:

- reading and writing to diaries;

- finding records of events;
- finding free slots to schedule committee meetings;
- finding when a faculty member or committee is busy.

In order to schedule a committee meeting, the secretary finds a nearest free slot for all faculty members in that committee. A faculty member may be busy if he is scheduled for a meeting or has made an annotation in his personal diary, for a specific period in time. The secretary may query the group diary to determine the schedule for the faculty member, or may read the status of the faculty member from his personal diary. Reading status in a diary is distinguished from reading the entries in a diary: status reveals only whether the owner of the diary is busy at some period in time without revealing the contents of the diary. Each diary owner may record or remove annotations only from his or her diary. The faculty members cannot read each others' diaries, but they are allowed to read the entries in the group diary. The system allows read access to an entire diary, but a user may only write (add or modify) to a diary if the entry corresponds to some future time. In order to keep the specification short, we assume:

- a system administrator who controls access to the system and issues a fresh diary to each new user;
- the diaries are only valid for a year. We do not maintain a repository of old diaries, nor are we concerned with events in future years;
- error checking and its specification are not attempted in the current version.

Chapter 6

The Z Specification

6.1 Structural Properties

In this section we present the structure of the shared-diary system. We have adopted the conventional style of writing Z specifications, where each formal paragraph is followed by an informal explanation.

$[STRING, USER, COMMITTEE]$

We introduce three basic types:

- *STRING* models the text of an annotation that a user may enter in a diary;
- *USER* introduces a type to describe the users of the shared-diary system;
- *COMMITTEE* introduces a type to describe the committees in the shared-diary system

We have chosen *STRING, USER* and *COMMITTEE* as basic types because the actual details of the three types are not of interest in describing this application.

$$Time == 0 \,..\, 23$$

We assume that personal appointments and committee meetings fall on the hour boundaries. Consequently, *Time* is based on hours only.

$$CurrentTime : Time$$

CurrentTime is an instance of time that models the current time of day. We assume that the current time of day may be obtained through a system call. Note that the concept of current time is important because we need to distinguish between events[1] that occur in the past or in the future.

$$\begin{array}{|l}
StartOfDay, EndOfDay : Time \\
\hline
StartOfDay = 0 \\
EndOfDay = 23
\end{array}$$

We introduce two constants, namely, *StartOfDay* to model the beginning of any day (midnight), and *EndOfDay* to model the end of the day (11 pm).

$$\begin{array}{l}
day == 1 \,..\, 31 \\
month == 1 \,..\, 12 \\
year == 1995 \,..\, 2005
\end{array}$$

We model day, month, and year as enumerated integers. We presume that the shared-diary system will be operational over a ten-year period beginning 1995.

[1]An event may denote a personal appointment or a committee meeting.

$$Date == ((day \times month) \times year)$$

Having defined day, month, and year, we can now compose the three types to introduce a type called *Date*. We choose a cartesian product to model *Date* so that we may use the projection functions, *first* and *second*, of the mathematical tool-kit of Z.

$$\begin{array}{|l}
Day : Date \rightarrow day \\
Month : Date \rightarrow month \\
Year : Date \rightarrow year \\
\hline
\forall\, d : Date \bullet \\
\quad Day(d) = first(first(d)) \wedge Month(d) = second(first(d)) \wedge Year(d) = second(d)
\end{array}$$

We introduce three projection functions, *Day*, *Month*, and *Year*, to extract the three components of a date.

$$\begin{array}{|l}
DaysInMonth : month \times year \rightarrow day \\
\hline
\forall\, d : day;\ m : month;\ y : year \bullet DaysInMonth(m, y) = d \Leftrightarrow \\
\quad (m = 1 \vee m = 3 \vee m = 5 \vee m = 7 \vee m = 8 \vee m = 10 \vee m = 12 \Leftrightarrow d = 31) \vee \\
\quad (m = 4 \vee m = 6 \vee m = 9 \vee m = 11 \Leftrightarrow d = 30) \vee \\
\quad (m = 2 \wedge y \bmod 100 \neq 0 \wedge y \bmod 4 = 0 \Leftrightarrow d = 29) \vee \\
\quad (m = 2 \wedge y \bmod 100 = 0 \wedge y \bmod 4 \neq 0 \Leftrightarrow d = 28)
\end{array}$$

The function *DaysInMonth* returns the number of days in a given month, in a given year.

$$\begin{array}{l}
\forall\, d : Date \bullet \\
\quad Day(d) \leq DaysInMonth(Month(d), Year(d))
\end{array}$$

Given a date, we constrain the number of days in the date to range from the first day of the month up to the number of days in the month.

$$\mid\ \mathit{CurrentDate} : \mathit{Date}$$

As in the case of *CurrentTime*, we need the current date to relate some activity in the system as belonging to the past or future. *CurrentDate*, an instance of *Date*, is only important as a concept; we assume that the value is determinable through a system call.

$$\begin{array}{|l}
\mathit{FirstDayOfYear}, \mathit{LastDayOfYear} : \mathit{Date} \\
\hline
\mathit{first}\ \mathit{FirstDayOfYear} = (1, 1) \\
\mathit{first}\ \mathit{LastDayOfYear} = (31, 12)
\end{array}$$

Having introduced the type *Date*, we now define two constants, namely, *FirstDay-OfYear* to represent January 1 of a year, and *LastDayOfYear* to represent December 31. We note that these values remain constant over the lifetime of a specification. For simplicity, we modeled the diary for one calendar year and we do not consider appointments that fall beyond the current year's boundary.

$$
\begin{array}{|l}
\_ < \_ : Date \leftrightarrow Date \\
\_ \leq \_ : Date \leftrightarrow Date \\
\_ \geq \_ : Date \leftrightarrow Date \\
\_ > \_ : Date \leftrightarrow Date \\
\hline
\forall d_1, d_2 : Date \bullet \\
\qquad d_1 < d_2 \Leftrightarrow \\
\qquad\qquad Year(d_1) < Year(d_2) \\
\qquad\qquad\qquad \vee \\
\qquad\qquad (Year(d_1) = Year(d_2) \wedge (Month(d_1) < Month(d_2) \\
\qquad\qquad\qquad \vee \\
\qquad\qquad (Month(d_1) = Month(d_2) \wedge Day(d_1) < Day(d_2)))) \\
\quad \wedge \\
\qquad d_1 \leq d_2 \Leftrightarrow d_1 < d_2 \vee d_1 = d_2 \\
\quad \wedge \\
\qquad d_1 \geq d_2 \Leftrightarrow \neg (d_1 < d_2) \\
\quad \wedge \\
\qquad d_1 > d_2 \Leftrightarrow \neg (d_1 \leq d_2)
\end{array}
$$

Given any two dates, we would like to determine whether one falls before or after another. We introduce four relational operators $(<, \leq, \geq, >)$ to compare dates.

$$
\begin{array}{|l}
\_ \mathsf{oneDless} \_ : Date \leftrightarrow Date \\
\hline
\forall d_1, d_2 : Date \bullet \\
\quad d_1 \ \mathsf{oneDless} \ d_2 \Leftrightarrow \\
\quad (Year(d_2) = succ(Year(d_1)) \\
\qquad \wedge \ first\ (d_1) = first\ LastDayOfYear \ \wedge \\
\qquad first\ (d_2) = first\ FirstDayOfYear) \\
\quad \vee \\
\qquad Year(d_1) = Year(d_2) \\
\qquad \wedge \ (Month(d_1) = Month(d_2) \wedge Day(d_2) = succ(Day(d_1))) \\
\qquad\qquad \vee \\
\qquad (Month(d_2) = succ(Month(d_1)) \wedge Day(d_2) = 1 \\
\qquad \wedge \ Day(d_1) = DaysInMonth(Month(d_1), Year(d_1)))
\end{array}
$$

We introduce one more comparison operator for dates, to determine whether one date is a predecessor of another. For instance, we would like to establish that a page in a diary is dated one day prior to the date on the next page. The axiom *oneDless* asserts that a date (d_1) occurs exactly one day before another (d_2) if:

- d_1 falls on the last day of a year, and d_2 falls on the first day of the following year; or
- d_1 and d_2 are in the same year and month, and d_1 is exactly one day before d_2; or
- d_1 and d_2 fall on the same year, and d_1 is the last day of a month, and d_2 is the first day of the following month.

```
── Duration ──────────────────────────────
│ StartTime, EndTime : Time
│ StartDate : Date
├──────────────
│ StartTime < EndTime
└──────────────────────────────────────────
```

Next, we define a duration. We disregard events that span multiple days because it is sufficient to describe a property over a single day and to expect that refinement will introduce repetition. However, we consider events that span multiple hours. The reasons behind this decision will become evident in the descriptions of operations which schedule committee meetings, check for conflicts, etc.

The invariant condition of *Duration*, denotes that a duration must span at least an hour.

$$DurFrag == Duration$$
$$\forall df : DurFrag \bullet df.EndTime = succ\ df.StartTime$$

Having defined the concept of duration, we would like to express the smallest duration possible, namely *DurFrag*. A duration fragment is a duration having a span of exactly one hour. We may justify, informally, that a duration fragment is indeed the smallest duration possible because we do not consider a time interval that is less than an hour.

$$
\begin{array}{|l}
\_ > \_ : Duration \leftrightarrow Duration \\
\hline
\forall\, d_1, d_2 : Duration \bullet \\
\qquad d_1 > d_2 \Leftrightarrow \\
\qquad\qquad d_1.StartDate > d_2.StartDate \\
\qquad\qquad \vee \\
\qquad\qquad d_1.StartDate = d_2.StartDate \wedge d_1.StartTime > d_2.StartTime
\end{array}
$$

As in the case of type *Date*, we would like to decide if one duration occurs after another. A duration (d_1) follows another (d_2) if:

- d_1 occurs on a later date than d_2; or
- the two durations fall on the same date, but d_1 begins at a later time than d_2.

$$
\begin{array}{|l}
\_ \mathsf{inside} \_ : Duration \leftrightarrow Duration \\
\hline
\forall\, d_1, d_2 : Duration \bullet \\
\qquad d_1 \ \mathsf{inside}\ d_2 \Leftrightarrow \\
\qquad\qquad d_1.StartDate = d_2.StartDate \\
\qquad\qquad \wedge \\
\qquad\qquad\qquad (d_1.StartTime > d_2.StartTime \wedge d_1.EndTime \leq d_2.EndTime \\
\qquad\qquad\qquad \vee \\
\qquad\qquad\qquad d_1.StartTime = d_2.StartTime \wedge d_1.EndTime < d_2.EndTime)
\end{array}
$$

A duration falls inside another if:

- the two durations fall on the same date;
- the time bound (start time to end time) of one duration completely overlaps the time bound of the other;
- the two durations differ in length by at least one hour.

$$
\begin{array}{|l}
\_\,\mathsf{within}\,\_ : \mathit{Duration} \leftrightarrow \mathit{Duration} \\
\hline
\forall d_1, d_2 : \mathit{Duration} \bullet \\
\quad d_1 \;\mathsf{within}\; d_2 \Leftrightarrow \\
\quad\quad d_1 \;\mathsf{inside}\; d_2 \lor d_1 = d_2
\end{array}
$$

A duration is within another if:

- one duration is inside another; or
- both durations begin and end at the same time, and fall on the same date.

We note that the conditions for one duration to lie within another, are weaker than the conditions for one duration to be inside another. By these rules, two different duration fragments will always fail the inside test, but one may lie within another. Also, a duration can never lie inside a duration fragment.

$$
\begin{array}{|l}
\mathsf{isolated}\_ : \mathbb{P}(\mathbb{P}\,\mathit{Duration}) \\
\hline
\forall \mathit{dset} : \mathbb{P}\,\mathit{Duration} \bullet \mathsf{isolated}(\mathit{dset}) \Leftrightarrow \\
\quad (\forall d_1, d_2 : \mathit{Duration} \mid \{d_1, d_2\} \subseteq \mathit{dset} \bullet \\
\quad\quad d_1.\mathit{StartDate} \neq d_2.\mathit{StartDate} \\
\quad\quad \lor \\
\quad\quad d_1.\mathit{StartDate} = d_2.\mathit{StartDate} \land \\
\quad\quad\quad (d_1.\mathit{EndTime} < d_2.\mathit{StartTime} \lor d_2.\mathit{EndTime} < d_1.\mathit{StartTime}))
\end{array}
$$

A set of durations is isolated if:

- no two durations in the set are adjacent. Two durations are adjacent if they fall on the same day and one begins when the other ends;
- no two durations may overlap, even in part. Two durations overlap if they have at least one hour in common.

We note that isolated will be *true* if every duration in *dset* falls on a different date.

$$
\begin{array}{|l}
\mathsf{IsFutureDuration\_} : \mathbb{P}\ Duration \\
\hline
\forall d : Duration \bullet \mathsf{IsFutureDuration}(d) \Leftrightarrow \\
\quad d > (\mu\ Duration \mid \\
\qquad StartDate = CurrentDate \\
\qquad \wedge\ StartTime = CurrentTime \\
\qquad \wedge\ EndTime = succ\ StartTime)
\end{array}
$$

A situation will occur later in the case study, where we want to assert that a duration begins at some time in the future. A duration relates to a future instant in time, if:

- it falls on the current date and yet begins after the current time; or
- it falls on a date after the current date.

The concept of a future duration is required to prevent a user from modifying any portion that is backdated in a diary.

The set operators such as $\cup$ and $\cap$ in Z, operate on discrete elements and are inadequate for manipulating sets of durations. Consider for example, two overlapping durations d_1 and d_2; let $d_1 < d_2$. The result of the union of d_1 and d_2 is a duration d where the start time of d is that of d_1 and the end time of d is that of d_2. We therefore redefine the set operators to be applicable only to sets of durations and therefore to sets of duration fragments.

$$
\begin{array}{|l}
\_ \setminus_1 \_ : \mathbb{P}\, Duration \times \mathbb{P}\, Duration \rightarrow \mathbb{P}\, Duration \\
\hline
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, Duration \bullet \\
\quad dset_1 \setminus_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{ d : Duration \mid \\
\quad\quad\quad \forall\, df : DurFrag \mid df \ \mathsf{within}\ d \bullet \\
\quad\quad\quad\quad (\exists\, d_1 : Duration \mid d_1 \in dset_1 \bullet df \ \mathsf{within}\ d_1) \\
\quad\quad\quad \land \\
\quad\quad\quad\quad \neg\, (\exists\, d_2 : Duration \mid d_2 \in dset_2 \bullet df \ \mathsf{within}\ d_2)\} \\
\quad\quad \land\ \mathsf{isolated}(dset_3)
\end{array}
$$

We may visualize the set difference $(\setminus_1)$[2] operation on two sets of durations, as follows:

- break up every duration in either set, into the corresponding duration fragments. Here, a duration fragment emanates from a duration if it lies within the duration;
- extract only those duration fragments of the first set that are not in common to any duration fragment of the second set;
- assert that the resulting set of durations is isolated. This is so only if we reconstitute the duration fragments into the largest durations possible such that no two durations either overlap or are adjacent. We acknowledge that this step is unnecessary, but we would like to continue the discussion in terms of durations.

[2] We subscript our set difference operator to distinguish it from the set difference operator of Z.

$$
\begin{array}{|l}
\_ \cup_1 \_ : \mathbb{P}\, \mathit{Duration} \times \mathbb{P}\, \mathit{Duration} \rightarrow \mathbb{P}\, \mathit{Duration} \\
\hline
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, \mathit{Duration} \bullet \\
\quad dset_1 \cup_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{ d : \mathit{Duration} \mid \\
\quad\quad\quad \forall\, df : \mathit{DurFrag} \mid df \text{ within } d \bullet \\
\quad\quad\quad\quad (\exists\, d_1 : \mathit{Duration} \mid d_1 \in dset_1 \lor d_1 \in dset_2 \bullet df \text{ within } d_1)\} \\
\quad\quad \land\ \mathsf{isolated}(dset_3)
\end{array}
$$

Similarly, we define union of two sets of durations as follows:

- break up every duration in either set, into the corresponding duration fragments;
- extract every duration fragment from both sets, into a third set;
- assert that the resulting set is isolated. As explained in the context of the $\setminus_1$ operator, this will be *true* only if the resulting set consists of durations that are neither adjacent, nor overlapping.

$$
\begin{array}{|l}
\_ \cap_1 \_ : \mathbb{P}\, \mathit{Duration} \times \mathbb{P}\, \mathit{Duration} \rightarrow \mathbb{P}\, \mathit{Duration} \\
\hline
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, \mathit{Duration} \bullet \\
\quad dset_1 \cap_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{ d : \mathit{Duration} \mid \\
\quad\quad\quad \forall\, df : \mathit{DurFrag} \mid df \text{ within } d \bullet \\
\quad\quad\quad\quad (\exists\, d_1, d_2 : \mathit{Duration} \mid d_1 \in dset_1 \land d_2 \in dset_2 \bullet \\
\quad\quad\quad\quad\quad df \text{ within } d_1 \land df \text{ within } d_2)\} \\
\quad\quad \land\ \mathsf{isolated}(dset_3)
\end{array}
$$

The definition for intersection of two sets of durations follows:

- break up the durations of each set into the corresponding fragments;
- extract only those duration fragments that are in common to both sets;

- assert that the resulting set is isolated.

$$PersonalAnnotation == STRING$$
$$CommitteeAnnotation == COMMITTEE$$

An annotation is simply an entry in a diary. We can differentiate between annotations that are personal annotations, and committee annotations. A personal annotation is a *textual* reminder of some event, and is considered to be private information to its owner. A committee annotation is a reminder of a committee meeting; we are concerned with keeping the specification short and clear, and therefore we simply equate a committee annotation to a committee. We may interpret a committee annotation as follows:

- a committee annotation refers to a specific committee;
- details such as committee members, venue, etc., are considered to be irrelevant to the description. We presume that these issues are handled by the secretary who schedules the committee meetings.

$$\begin{aligned} Annotation \ ::= \ & \mathsf{PersAnnot}\langle\langle PersonalAnnotation\rangle\rangle \\ \mid \ & \mathsf{CommAnnot}\langle\langle CommitteeAnnotation\rangle\rangle \end{aligned}$$

Annotation introduces a general representation for both personal annotation and committee annotation. We find this definition to be convenient because we can decide how to add or remove an annotation from a diary without having to tailor the operations to handle specific representations. By the semantics for free types, the injective functions PersAnnot and CommAnnot map every personal annotation and committee annotation, respectively, to a unique annotation.

$$
\begin{array}{|l}
\mathsf{IsCommAnnot\_} : \mathbb{P}\ \mathit{Annotation} \\
\mathsf{IsPersAnnot\_} : \mathbb{P}\ \mathit{Annotation} \\
\hline
\forall a : \mathit{Annotation} \bullet \\
\quad \mathsf{IsCommAnnot}(a) \Leftrightarrow \\
\quad\quad (\exists\, ca : \mathit{CommitteeAnnotation} \bullet a = \mathsf{CommAnnot}(ca)) \\
\land \\
\quad \mathsf{IsPersAnnot}(a) \Leftrightarrow \\
\quad\quad (\exists\, pa : \mathit{PersonalAnnotation} \bullet a = \mathsf{PersAnnot}(pa))
\end{array}
$$

We would also like to distinguish an annotation as being a personal annotation or committee annotation.

$$\mathit{SlotStatus} ::= \mathit{Available} \mid \mathit{Busy} \mid \mathit{Invalid}$$

We are now in a position to describe how the annotations are stored in the diary. We will first introduce a free type called *SlotStatus* which describes the status of a slot (entry) in one page of a diary. The status is:

- *Available* if it contains no annotations;
- *Busy* if it contains at least one annotation;
- *Invalid* if it is not accessible.

$$
\begin{array}{|l}
\textit{Entry} \\
\hline
\mathit{ListOfAnnotations} : \mathbb{P}\ \mathit{Annotation} \\
\mathit{SlotTime} : \mathit{Time} \\
\mathit{Status} : \mathit{SlotStatus} \\
\hline
\mathit{Status} \neq \mathit{Invalid} \Rightarrow \\
\quad\quad (\mathit{ListOfAnnotations} = \varnothing \Leftrightarrow \mathit{Status} = \mathit{Available} \\
\quad \lor \quad \mathit{ListOfAnnotations} \neq \varnothing \Leftrightarrow \mathit{Status} = \mathit{Busy}) \\
\mathit{Status} = \mathit{Invalid} \Rightarrow \\
\quad\quad \mathit{ListOfAnnotations} = \varnothing \\
\hline
\end{array}
$$

A slot is formally called an *Entry*. An entry contains a set of annotations, a time stamp called *SlotTime*, and a status indicating whether the entry is available or busy or invalid. The following are the properties of an entry:

- if an entry is valid, then an annotation may be added to or removed from it.
- if an entry is invalid then it cannot be accessed and consequently, cannot contain an annotation.

Slot time indicates the hour of a day. We may combine the slot time and slot status to determine whether the owner of the diary is available or busy at that hour of the day.

$$\begin{array}{l} EntrySeq == iseq(Entry) \\ \forall\, es : EntrySeq \bullet \\ \quad (\forall\, i : 1 \,..\, \#es - 1 \bullet (es(i+1)).SlotTime = succ\ (es(i)).SlotTime) \end{array}$$

We have defined a sequence of entries in which every entry falls one hour before its successor.

$$\begin{array}{|l} StartComm, EndComm : Time \\ \hline StartComm < EndComm \end{array}$$

It is reasonable to assume that committee meetings will be held only during working hours. We introduce two constants of time, namely, *StartComm* and *EndComm* to denote these boundaries. Since a committee meeting has to run for at least an hour (the unit of time in our model), there is an explicit constraint that *StartComm* should be strictly less than *EndComm*.

$$
\begin{array}{|l}
\hline
\textit{Page} \\
\textit{ListOfPageEntries} : \textit{EntrySeq} \\
\textit{SlotDate} : \textit{Date} \\
\hline
\#\textit{ListOfPageEntries} = 24 \\
(\textit{head}\ \textit{ListOfPageEntries}).\textit{SlotTime} = \textit{StartOfDay} \\
\forall\, e : \textit{Entry} \mid e \in \operatorname{ran} \textit{ListOfPageEntries} \bullet \\
\quad (\forall\, a : \textit{Annotation} \mid a \in e.\textit{ListOfAnnotations} \bullet \\
\quad\quad \mathsf{IsCommAnnot}(a) \Rightarrow \textit{StartComm} \leq e.\textit{SlotTime} < \textit{EndComm}) \\
\hline
\end{array}
$$

Informally stated, a page corresponds to one sheet in a diary. A page contains a sequence of entries, and a date to which the page corresponds. The invariant of *Page* follows:

- a page contains exactly 24 entries;
- the timestamp on the first entry corresponds to the first hour of a day, namely, midnight;
- if an entry has a committee annotation, then the time stamp of that entry should be between *StartComm* (included) and *EndComm* (excluded).

We may informally justify that since a page contains a sequence of 24 entries, and the first entry relates to the break of day, a page contains one entry for each hour of the day.

$$
\begin{array}{l}
\textit{PageSeq} == \textit{iseq}(\textit{Page}) \\
\forall\, \textit{ps} : \textit{PageSeq} \bullet \\
\quad (\forall\, i : 1 \mathinner{..} \#\textit{ps} - 1 \bullet (\textit{ps}(i)).\textit{SlotDate}\ \mathsf{oneDless}\ (\textit{ps}(i+1)).\textit{SlotDate})
\end{array}
$$

PageSeq is a sequence of pages that is totally ordered on the basis of *SlotDate*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SDSUsers}} \\
\hline
\textit{Secretary} : \textit{USER} \\
\textit{ListOfFacultyMembers} : \mathbb{P}\ \textit{USER} \\
\hline
\end{array}$$

We represent the users of the shared-diary system by a schema called *SDSUsers* which includes a secretary, and a set of faculty members.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Diary}} \\
\hline
\textit{ListOfDiaryPages} : \textit{PageSeq} \\
\hline
\textit{Year}(\textit{CurrentDate}) \bmod 100 \neq 0 \land \textit{Year}(\textit{CurrentDate}) \bmod 4 = 0 \Leftrightarrow \\
\qquad \#\textit{ListOfDiaryPages} = 366 \\
\textit{Year}(\textit{CurrentDate}) \bmod 4 \neq 0 \Leftrightarrow \#\textit{ListOfDiaryPages} = 365 \\
(\textit{head}\ \textit{ListOfDiaryPages}).\textit{SlotDate} = \textit{FirstDayOfYear} \\
\hline
\end{array}$$

At this point, we have introduced sufficient detail into the specification in order to explain a diary. As with most desktop-planners, a diary is simply a sequence of pages. We determine that a diary may have 366 pages if it is current in a leap-year, and have 365 pages otherwise. The first page in a diary corresponds to the first day in a year, namely, January 1. As with *EntrySeq* we can informally justify that a diary contains exactly one page for each day in a year.

$$\textit{AccessRight} ::= \textit{ReadTotal} \mid \textit{ReadPartial} \mid \textit{Write}$$

Every user may have one or more of the following access rights to a diary:

- *Write* which specifies that a user may modify the contents of the diary;
- *ReadTotal* which means a user can read the entire contents of the diary, but cannot modify it;

- *ReadPartial* which restricts read access so that a user may only read the status information.

$$
\begin{array}{|l}
\hline
\textit{SDSDiary} \\
\hline
\textit{Diary} \\
\textit{SDSUsers} \\
\textit{DiaryOwner} : \textit{USER} \\
\textit{DiaryAccess} : \textit{USER} \rightarrow\!\!\!\!\!+\; \mathbb{P}\,\textit{AccessRight} \\
\hline
\operatorname{dom} \textit{DiaryAccess} \subseteq \{\textit{Secretary}\} \cup \textit{ListOfFacultyMembers} \\
\textit{DiaryOwner} \in \operatorname{dom} \textit{DiaryAccess} \\
\hline
\end{array}
$$

We can tailor the definition of a diary to suit the needs in the shared-diary system. A *SDSDiary* inherits the properties of a diary through schema inclusion. Additionally, it includes:

- users of the system and their access rights to the diary;
- a user who owns the diary.

The invariants of *SDSDiary* are:

- access rights have to be specified for every user;
- the diary owner is necessarily one of the users in the shared-diary system.

$$
\begin{array}{|l}
\mathsf{IsGroupDiary\_} : \mathbb{P}\, SDSDiary \\
\hline
\forall d : SDSDiary \bullet \\
\quad \mathsf{IsGroupDiary}(d) \Leftrightarrow \\
\qquad d.DiaryOwner = d.Secretary \\
\qquad \wedge \\
\qquad (\forall p : Page \mid p \in \operatorname{ran} d.ListOfDiaryPages \bullet \\
\qquad\quad (\forall e : Entry \mid e \in \operatorname{ran} p.ListOfPageEntries \bullet \\
\qquad\qquad (\forall a : Annotation \mid a \in e.ListOfAnnotations \bullet \\
\qquad\qquad\quad \mathsf{IsCommAnnot}(a)))) \\
\qquad \wedge \\
\qquad d.DiaryAccess = \{d.DiaryOwner \mapsto \{ReadTotal, ReadPartial, Write\}\} \cup \\
\qquad\quad \{fm : USER \mid fm \in d.ListOfFacultyMembers \bullet fm \mapsto \{ReadTotal\}\}
\end{array}
$$

We would like to distinguish a diary that belongs to the secretary as a group diary because the access privileges to a group diary are different from those to a personal diary; this is asserted by IsGroupDiary. A *SDSDiary* is a group diary, if and only if:

- the secretary owns the diary;
- every annotation in the diary is a committee annotation;
- the secretary has read and write access to the diary and the faculty members may only read from the diary.

$$
\begin{array}{|l}
\mathsf{IsPersDiary\_} : \mathbb{P}\, SDSDiary \\
\hline
\forall d : SDSDiary \bullet \\
\quad \mathsf{IsPersDiary}(d) \Leftrightarrow \\
\qquad d.DiaryOwner \in d.ListOfFacultyMembers \\
\qquad \wedge \\
\qquad (\forall p : Page \mid p \in \operatorname{ran} d.ListOfDiaryPages \bullet \\
\qquad\quad (\forall e : Entry \mid e \in \operatorname{ran} p.ListOfPageEntries \bullet \\
\qquad\qquad (\forall a : Annotation \mid a \in e.ListOfAnnotations \bullet \\
\qquad\qquad\quad \mathsf{IsPersAnnot}(a)))) \\
\qquad \wedge \\
\qquad d.DiaryAccess = \{d.DiaryOwner \mapsto \{ReadTotal, ReadPartial, Write\}, \\
\qquad\quad d.Secretary \mapsto \{ReadPartial\}\}
\end{array}
$$

Similar to IsGroupDiary, IsPersDiary asserts the conditions for an *SDSDiary* to be a personal diary; these conditions are:

- a faculty member owns the diary;
- every annotation in the diary is a personal annotation;
- the diary owner alone has read and write access to the diary;
- the secretary has partial read access to the diary;
- other faculty members have no access to the diary.

$$
\begin{array}{|l}
\mathit{ValidSDS} : \mathbb{N}_1 \\
\hline
\mathit{ValidSDS} = 2
\end{array}
$$

For a shared diary system to be active, we need at least two users; one of them must be the secretary, and we deduce that the other has to be a faculty member. For simplicity of specification, we require that neither the secretary, nor a last remaining faculty member may be removed from the system.

$$
\begin{array}{|l}
\hline
\textit{SDSMaps} \\
\mathit{SDSUsers} \\
\mathit{Owns} : \mathit{USER} \rightarrowtail\!\!\!\!\rightarrow \mathit{SDSDiary} \\
\hline
\#(\mathrm{dom}\ \mathit{Owns}) \geq \mathit{ValidSDS} \\
\langle \{\mathit{Secretary}\}, \mathit{ListOfFacultyMembers} \rangle\ \mathrm{partitions}\ \mathrm{dom}\ \mathit{Owns} \\
\forall u : \mathit{USER} \mid u \in \mathrm{dom}\ \mathit{Owns} \bullet (\mathit{Owns}\ u).\mathit{DiaryOwner} = u \\
\mathsf{IsGroupDiary}(\mathit{Owns}\ \mathit{Secretary}) \\
\forall \mathit{fm} : \mathit{USER} \mid \mathit{fm} \in \mathit{ListOfFacultyMembers} \bullet \mathsf{IsPersDiary}(\mathit{Owns}\ \mathit{fm}) \\
\hline
\end{array}
$$

SDSMaps maps each user to a distinct diary. The invariant of *SDSMaps* asserts the following:

- there must be at least two users in the system;
- every user in the system must have a diary;
- the secretary owns the group diary;
- each faculty member owns a personal diary.

$$\begin{array}{|l}
\mathit{ValidComm} : \mathbb{N}_1 \\
\hline
\mathit{ValidComm} = 2
\end{array}$$

It is reasonable to assume that every committee should have at least two members.

$$\begin{array}{|l}
\textit{SDSComm} \\
\hline
\mathit{SDSUsers} \\
\mathit{HasMembers} : \mathit{COMMITTEE} \pfun \mathbb{P}\ \mathit{USER} \\
\mathit{Active} : \mathbb{F}\ \mathit{COMMITTEE} \\
\mathit{InActive} : \mathbb{F}\ \mathit{COMMITTEE} \\
\hline
\langle \mathit{Active}, \mathit{InActive} \rangle \text{ partitions } \mathrm{dom}\ \mathit{HasMembers} \\
\forall c : \mathit{COMMITTEE} \mid c \in \mathrm{dom}\ \mathit{HasMembers} \bullet \\
\qquad \mathit{HasMembers}\ c \subseteq \mathit{ListOfFacultyMembers} \\
\quad \wedge \\
\qquad \#(\mathit{HasMembers}\ c) \geq \mathit{ValidComm} \Rightarrow c \in \mathit{Active} \\
\quad \wedge \\
\qquad \#(\mathit{HasMembers}\ c) < \mathit{ValidComm} \Rightarrow c \in \mathit{InActive} \\
\hline
\end{array}$$

SDSComm maps (indicated by *HasMembers*) each committee in the system to the set of members on that committee. Committees may be classified as being:

- *Active* if the committee has at least two members;
- *InActive* otherwise.

```
┌─ SDS ─────────────────────────────
│ SDSMaps
│ SDSComm
└───────────────────────────────────
```

The entire shared diary system consisting of users, committees, and diaries, is represented by a schema called *SDS*.

6.2 Behavioral Properties

In this section we describe the various operations of the shared diary system.

```
┌─ OpLoAEntry ──────────────────────
│ ΔEntry
├──────────────
│ SlotTime' = SlotTime
└───────────────────────────────────
```

While describing the behavior of the shared diary system, we wish to follow Hall's[13] style. Accordingly, we first define a partial operation[3] on a state space, which lists those state variables that are preserved by this operation. For example, any operation on entry should not change the slot time associated with that entry. This can be defined through the operation *OpLoAEntry*. Later, when we define additional operations on an entry, we simple include[4] *OpLoAEntry* in those operations. In fact, Hall's style does not add anything new to the Z specification, but simplifies

[3]By convention, the notation ΔState introduces the unprimed and primed schema variables of *State* into the signature of an operation schema.

[4]Schema inclusion may take one of two forms: a schema may be included in the signature of another schema therefore conjoining the two signatures and the two predicates, respectively; a schema may be included in the predicate of another schema if the signature of the included schema is a subset of the signature of the including schema. In the latter case, only the predicates of the two schemas are conjoined.

writing. Moreover, one can easily see what is preserved by any operation over a particular state space.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddAnnotToEnt}} \\
\hline
\textit{OpLoAEntry} \\
a? : \textit{Annotation} \\
\hline
\textit{Status} \neq \textit{Invalid} \\
a? \notin \textit{ListOfAnnotations} \\
\textit{ListOfAnnotations}' = \textit{ListOfAnnotations} \cup \{a?\} \\
\hline
\end{array}$$

We now define an operation to add an annotation to an entry. The constraints of *AddAnnotToEnt* assert that the entry should have a valid status and the new annotation should not already be present in the entry.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ChngStatus}} \\
\hline
\textit{OpLoAEntry} \\
\hline
\textit{Status} = \textit{Available} \land \textit{ListOfAnnotations}' \neq \varnothing \Rightarrow \textit{Status}' = \textit{Busy} \\
\textit{Status} = \textit{Busy} \land \textit{ListOfAnnotations}' = \varnothing \Rightarrow \textit{Status}' = \textit{Available} \\
\hline
\end{array}$$

An entry may change status when it is modified. If the entry was previously available, and now has an annotation, then it changes status to busy. On the contrary, if the entry was busy and now contains no annotation, then it becomes available.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{KeepStatus}} \\
\hline
\textit{OpLoAEntry} \\
\hline
(\textit{Status} = \textit{Available} \land \textit{ListOfAnnotations}' = \varnothing \\
\qquad \lor \\
\textit{Status} = \textit{Busy} \land \textit{ListOfAnnotations}' \neq \varnothing) \Rightarrow \\
\qquad\qquad \textit{Status}' = \textit{Status} \\
\hline
\end{array}$$

It is also likely that even after modifying an entry, the status of the entry remains unchanged. On one hand, an entry may contain two or more annotations and simply removing an annotation does not affect its status. On the other hand, an entry may not have any annotations, and so it retains its state.

$$AddAnnotationToEntry \mathrel{\widehat{=}} AddAnnotToEnt \land (ChngStatus \lor KeepStatus)$$

We may now integrate the three primitive operations on an entry, to complete the requirements of adding an annotation to an entry.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemAnnotFromEnt}} \\
\hline
OpLoAEntry \\
a? : Annotation \\
\hline
Status \neq Invalid \\
a? \in ListOfAnnotations \\
ListOfAnnotations' = ListOfAnnotations \setminus \{a?\} \\
\hline
\end{array}$$

In order to remove an annotation from an entry, the former must be already present in the entry.

$$RemAnnotationFromEntry \mathrel{\widehat{=}} RemAnnotFromEnt \land (ChngStatus \lor KeepStatus)$$

The operation *RemAnnotationFromEntry* specifies the complete operation of removing an annotation from an entry.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ReadAnnotationInEntry}} \\
\hline
\Xi Entry \\
aset! : \mathbb{P}\ Annotation \\
\hline
Status \neq Invalid \Rightarrow aset! = ListOfAnnotations \\
Status = Invalid \Rightarrow aset! = \varnothing \\
\hline
\end{array}$$

In addition to adding and deleting annotations, we would like to read the contents of an entry. The operation *ReadAnnotationInEntry* [5] simply returns the set of annotations from an entry.

$$\begin{array}{|l}
\textit{ReadStatusInEntry} \\
\hline
\Xi \textit{Entry} \\
s! : \textit{SlotStatus} \\
\hline
s! = \textit{Status} \\
\hline
\end{array}$$

ReadStatusInEntry returns the status of an entry.

$$\begin{array}{|l}
\textit{OpLoPEPage} \\
\hline
\Delta \textit{Page} \\
\hline
\textit{SlotDate}' = \textit{SlotDate} \\
\hline
\end{array}$$

The operation *OpLoPEPage* specifies a partial operation on a page, where the date remains constant.

$$\begin{array}{|l}
\textit{OpPromoteEntPage} \\
\hline
\textit{OpLoPEPage} \\
\textit{OpLoAEntry} \\
df? : \textit{DurFrag} \\
\hline
df?.\textit{StartDate} = \textit{SlotDate} \\
\theta \textit{Entry} = (\mu\, e : \textit{Entry} \mid e \in \operatorname{ran} \textit{ListOfPageEntries} \wedge \\
\qquad\qquad e.\textit{SlotTime} = df?.\textit{StartTime}) \\
\qquad\qquad\qquad\qquad \wedge \\
\textit{ListOfPageEntries}' = \textit{ListOfPageEntries} \oplus \\
\qquad \{\textit{ListOfPageEntries}^{-1} \theta \textit{Entry} \mapsto \theta \textit{Entry}'\} \\
\hline
\end{array}$$

[5]By convention, the notation ΞState introduces the unprimed and primed schema variables of *State* into the signature of an operation schema. The convention requires that the state space of *State* is not modified over the operation, namely, θState' = θState.

Note that the state space of an entry is in fact a part of the state space of a page. Therefore, when an operation changes the state space of an entry, it has to be uplifted to the page that contains the entry. One way to accomplish this is to create a new operation on the page and restate all the conditions, mentioned in the operation on entry. However, this approach leads to an unacceptably high rate of repetition. Z offers an elegant solution to this problem through a technique called *promotion*. Using promotion, one can write a template of an uplifting operation which can be conjoined with the operation on entry, to specify the operation on page. The operation *OpPromoteEntPage* described above is the promoting operation which uplifts every operation on entry to a corresponding operation on page. The linkage between a particular entry and the page which contains the entry is established by the common names in the three operations - operation on entry, operation on page, and the promotion operation.

OpPromoteEntPage asserts that a particular entry is selected whose slot time coincides with the selection (input, in this case duration fragment) parameter. *θEntry* refers to the template of the entry being selected and *θEntry'* refers to its updated version by the operation on entry. *OpPromoteEntPage* describes, further, how *θEntry* and *θEntry'* can be used to update the page.

We may need to specify an additional operation to select each entry in a page when we wish to iterate over all the entries in a given page. This is specified next.

$$
\begin{array}{|l}
\textit{SelectEntsInDur} \\
\hline
\Delta \textit{Page} \\
\Delta \textit{Entry} \\
d? : \textit{Duration} \\
\hline
\forall\, df : \textit{DurFrag} \mid df \ \mathsf{within}\ d? \bullet \textit{OpPromoteEntPage}[df / df?] \\
\hline
\end{array}
$$

SelectEntsInDur specifies the iteration over all the entries in a page, which have a timestamp between the start time and end time of the input duration.

$$
\begin{aligned}
&\mathit{AddAnnotationToPage} \mathrel{\widehat=} \exists\, \Delta \mathit{Entry} \bullet \mathit{SelectEntsInDur} \land \mathit{AddAnnotationToEntry}\\
&\mathit{RemAnnotationFromPage} \mathrel{\widehat=} \exists\, \Delta \mathit{Entry} \bullet \mathit{SelectEntsInDur} \land \mathit{RemAnnotationFromEntry}\\
&\mathit{ReadAnnotationInPage} \mathrel{\widehat=} \exists\, \Delta \mathit{Entry} \bullet \mathit{SelectEntsInDur} \land \mathit{ReadAnnotationInEntry}\\
&\mathit{ReadStatusInPage} \mathrel{\widehat=} \exists\, \Delta \mathit{Entry} \bullet \mathit{SelectEntsInDur} \land \mathit{ReadStatusInEntry}
\end{aligned}
$$

We may now conjoin these promotion and selection operations with each operation on an entry, to introduce new operations which affect an entire page. These operations are given above.

```
── FindAnnotationInPage ─────────────────────
│ ΞPage
│ ΞEntry
│ a? : Annotation
│ dset! : ℙ Duration
├──────────────
│ dset! = {e : Entry |
│       e ∈ ran ListOfPageEntries ∧ a? ∈ e.ListOfAnnotations •
│           (μ Duration |
│               StartTime = e.SlotTime
│               ∧ EndTime = succ e.SlotTime
│               ∧ StartDate = SlotDate)}
│     ∧
│ isolated(dset!)
└─────────────────────────────────────────────
```

We draw a distinction between operations that read and write to entries, and operations that find items in entries: in the former case, we deal the with the contents of entries, and in the latter case, we are interested in when they occur. Typically, given an input duration we wish to:

- verify that we are searching through the right page (date on page equals start date of duration);

- search every entry in the page, with a slot time between the time bounds of the duration;
- construct a set of durations, each of which has a start time equal to the slot time of an entry which contains the annotation;
- assert that the resulting set of durations is isolated: no two durations may overlap, or be adjacent.

We note that this operation is appropriate at the page level because in addition to the time of an entry, we also need the slot date of the page in which the entry occurs; this composite information is used to construct the resulting set of durations.

FindStatusInPage

$\Xi Page$
$\Xi Entry$
$s? : SlotStatus$
$dset! : \mathbb{P}\ Duration$

$$\begin{aligned}
&dset! = \{e : Entry \mid \\
&\quad e \in \operatorname{ran} ListOfPageEntries \land s? = e.Status \bullet \\
&\qquad (\mu\ Duration \mid \\
&\qquad\quad StartTime = e.SlotTime \\
&\qquad\quad \land\ EndTime = succ\ e.SlotTime \\
&\qquad\quad \land\ StartDate = SlotDate)\} \\
&\quad \land \\
&\mathsf{isolated}(dset!)
\end{aligned}$$

We specify an operation to find which entries in a page, with slot times between the time bounds of an input duration, relate to a particular status. The specification of this operation is almost identical to that for finding an annotation in a page, the only difference being the item of interest.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{OpLoDPSDSDiary}} \\
\Delta \mathit{SDSDiary} \\
\hline
\theta \mathit{SDSUsers}' = \theta \mathit{SDSUsers} \\
\mathit{DiaryOwner}' = \mathit{DiaryOwner} \\
\mathit{DiaryAccess}' = \mathit{DiaryAccess} \\
\hline
\end{array}
$$

Once again, we take advantage of Hall's style and specify an operation which asserts those properties that are preserved by every operation on a diary.

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{OpPromotePageDiary}} \\
\mathit{OpLoDPSDSDiary} \\
\mathit{OpLoPEPage} \\
d? : \mathit{Duration} \\
\hline
\theta \mathit{Page} = (\mu\, i : \mathbb{N}_1 \mid i \in \operatorname{dom} \mathit{ListOfDiaryPages} \\
\qquad\qquad \wedge\ (\mathit{ListOfDiaryPages}(i)).\mathit{SlotDate} = d?.\mathit{StartDate} \bullet \\
\qquad\qquad\qquad \mathit{ListOfDiaryPages}(i)) \\
\mathit{ListOfDiaryPages}' = \mathit{ListOfDiaryPages} \oplus \\
\qquad\qquad \{\mathit{ListOfDiaryPages}^{-1} \theta \mathit{Page} \mapsto \theta \mathit{Page}'\} \\
\hline
\end{array}
$$

Next, we specify a promotion which uplifts an operation on a page, to a diary.

$$
\begin{array}{l}
\mathit{AddAnnotationToDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{AddAnnotationToPage} \\
\mathit{RemAnnotationFromDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{RemAnnotationFromPa} \\
\mathit{ReadAnnotationInDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{ReadAnnotationInPage} \\
\mathit{ReadStatusInDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{ReadStatusInPage} \\
\mathit{FindAnnotationInDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{FindAnnotationInPage} \\
\mathit{FindStatusInDiary} \mathrel{\widehat=} \exists\, \Delta \mathit{Page} \bullet \mathit{OpPromotePageDiary} \wedge \mathit{FindStatusInPage}
\end{array}
$$

We may now compose the promotion from page to diary, and the operations on pages.

$OpChkDiaryAccess$

$\Delta SDSMaps$
$\Delta SDSDiary$
$orig? : USER$
$owner? : USER$
$acc? : AccessRight$
$d? : Duration$

$\{orig?, owner?\} \subseteq \mathrm{dom}\ Owns$
$\theta SDSDiary = Owns\ owner?$
$orig? \in \mathrm{dom}(\theta SDSDiary).DiaryAccess \land acc? \in (\theta SDSDiary).DiaryAccess\ orig?$
$acc? = Write \Rightarrow \mathsf{IsFutureDuration}(d?)$

As a necessary precondition, we have to verify that a user may only access a diary that he or she is entitled to. We restrict access to the various diaries, through the operation *OpChkDiaryAccess*; the inputs to the operation schema include:

- *orig*: the user who wants to access the diary;
- *owner*: the diary owner; we note that the originator could also own the diary;
- *acc*: the requested access, namely, read or write;
- *d*: the duration which is used to identify the page and corresponding entries that the originator is interested in.

OpChkDiaryAccess asserts the following:

- verify that the user belongs in the system;
- verify the access rights of the originator with that stored in *SDSMaps*;
- ensure that a request to modify a diary will only affect entries that relate to some time in the future.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OpPromoteDiarySDSMaps}} \\
\Delta SDSMaps \\
\Delta SDSDiary \\
owner? : USER \\
\hline
\theta SDSUsers' = \theta SDSUsers \\
owner? \in \operatorname{dom} Owns \\
\theta SDSDiary = Owns\ owner? \\
Owns' = Owns \oplus \{owner? \mapsto \theta SDSDiary'\} \\
\hline
\end{array}$$

We may promote the various operations on diaries to *SDSMaps* which specifies that each user owns a diary. The operation schema *OpPromoteDiarySDSMaps* promotes the change to a diary, to *SDSMaps*, by replacing the previous state of the diary with the modified state. In addition, the operation asserts that a change to a diary neither affects the users in the system, nor who owns the diary.

$$\begin{aligned}
&AddAnnotationSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[Write/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land AddAnnotationToDiary \\
&RemAnnotationSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[Write/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land RemAnnotationFromDiary \\
&ReadAnnotationSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[ReadTotal/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land ReadAnnotationInDiary \\
&ReadStatusSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[ReadPartial/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land ReadStatusInDiary \\
&FindAnnotationSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[ReadTotal/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land FindAnnotationInDiary \\
&FindStatusSDSMaps \mathrel{\widehat=} \exists \Delta SDSDiary \bullet \\
&\quad OpChkDiaryAccess[ReadPartial/acc?] \land OpPromoteDiarySDSMaps \\
&\qquad \land FindStatusInDiary
\end{aligned}$$

We may now integrate the two operations which:

- check a user's right to access a diary;
- promote the effect of modifying a diary to *SDSMaps*

and conjoin the composite operation with each of the 6 operations on diaries to introduce 6 new operations, at the *SDSMaps* level. At this point, we have sufficient information to identify each operation with a specific access right, for instance, a user needs write access to add an annotation to a diary.

```
┌─ OpSDSMapsSDS ──────────────────────────────
│ ΔSDS
├──────────────
│ HasMembers′ = HasMembers
│ Active′ = Active
│ InActive′ = InActive
└─────────────────────────────────────────────
```

We recall that the schema *SDS* integrates the entire specification. We have reached the level at which users interact with the system and therefore we need to promote the various operations on the diaries from the level of *SDSMaps* to *SDS*. In the operation *OpSDSMapsSDS*, we ensure that an operation on a diary does not affect any committees in the system.

$$
\begin{aligned}
&AddAnnotationSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land AddAnnotationSDSMaps \\
&RemAnnotationSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land RemAnnotationSDSMaps \\
&ReadAnnotationSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land ReadAnnotationSDSMaps \\
&ReadStatusSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land ReadStatusSDSMaps \\
&FindAnnotationSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land FindAnnotationSDSMaps \\
&FindStatusSDS \mathrel{\widehat=} \exists\, \Delta SDSMaps \bullet \\
&\quad OpSDSMapsSDS \land FindStatusSDSMaps
\end{aligned}
$$

It has certainly cost us much effort to promote the 6 operations on a diary to the level of the shared-diary system, where the users may actually use them. However, we maintain that the promotion is necessary because we want to provide these services to users, in addition to other operations, all of which are invoked at the *SDS* level.

$$
\begin{array}{|l}
\hline
\quad FindCommBusySDS \\
\Delta SDS \\
c? : COMMITTEE \\
d? : Duration \\
dset! : \mathbb{P}\, Duration \\
\hline
c? \in \operatorname{dom} HasMembers \\
\exists\, a : Annotation \mid a = \mathsf{CommAnnot}(c?) \bullet \\
\quad dset! = (\mu\, FindAnnotationSDS[Secretary/owner?, a/a?] \bullet dset!) \\
\hline
\end{array}
$$

We may now compose the primitive operations that we have specified, into complex, but regular services of the system. The operation *FindCommBusySDS* returns a set of durations during which a given committee is busy.

$$
\begin{array}{|l}
\hline
\textit{FindFMBusySDS} \\
\Delta SDS \\
fm? : USER \\
d? : Duration \\
dset! : \mathbb{P}\, Duration \\
\hline
fm? \in ListOfFacultyMembers \\
dset! = \{ d : Duration \mid \\
\quad \exists\, c : COMMITTEE \mid c \in Active \land fm? \in HasMembers\ c \bullet \\
\quad\quad d \in (\mu\, FindCommBusySDS[c/c?] \bullet dset!) \} \\
\quad\quad\quad \cup_1 \\
\quad (\mu\, FindStatusSDS[Busy/s?, fm?/owner?] \bullet dset!) \\
\hline
\end{array}
$$

Another useful operation is to determine when a faculty member is busy within a given duration. A faculty member is busy when any active committee that he/she is a member of is busy, and when he/she has scheduled appointments in his/her personal diary. The resulting set of durations at which the faculty member is busy, is simply the union of these two situations.

$$
\begin{array}{|l}
\hline
\textit{FindSchdCnflctsSDS} \\
\Delta SDS \\
fm? : USER \\
c? : COMMITTEE \\
dset! : \mathbb{P}\, Duration \\
\hline
fm? \in ListOfFacultyMembers \\
c? \in \operatorname{dom} HasMembers \\
fm? \notin HasMembers\ c? \\
dset! = \{ d : Duration \mid \\
\quad\quad \exists\, d_1 : Duration \mid \mathsf{IsFutureDuration}(d_1) \bullet \\
\quad\quad\quad d \in (\mu\, FindFMBusySDS[d_1/d?] \bullet dset!) \} \\
\quad\quad\quad\quad \cap_1 \\
\quad \{ d : Duration \mid \\
\quad\quad \exists\, d_1 : Duration \mid \mathsf{IsFutureDuration}(d_1) \bullet \\
\quad\quad\quad d \in (\mu\, FindCommBusySDS[d_1/d?] \bullet dset!) \} \\
\hline
\end{array}
$$

A faculty member may only be included on an active committee if there is no conflict between his/her appointments, namely committee meetings and personal business, and the appointments of the committee that he/she is to become a member of. The operation *FindSchdCnflctsSDS* specifies the durations at which conflicts may occur when scheduling a meeting for a committee *c?*, for a member *fm?*.

The operation asserts that:

- the person under consideration is indeed a faculty member;
- the faculty member is to be included in a committee which belongs to the system;
- the faculty member is not already on the committee.

The two sets of durations which make up the situations at which a faculty member is busy, are determined as follows:

- determine all the possible durations from the current date and time, to the end of the year (these are *future* durations);
- invoke the operations with the future durations to determine when the faculty member and committee are busy in the future;
- the conflict is that set of durations which are in common to the two set comprehensions.

$$\begin{array}{|l}
\hline
\textit{FindFreeSlotForCommSDS} \\
\Delta SDS \\
d? : Duration \\
c? : COMMITTEE \\
dset! : \mathbb{P}\ Duration \\
\hline
c? \in \mathrm{dom}\ HasMembers \\
dset! = \{ d : Duration \mid \\
\quad \forall fm : USER \mid fm \in HasMembers\ c? \bullet \\
\qquad (\exists d_1 : Duration \mid \\
\qquad\quad d_1 \in (\{d?\} \setminus_1 (\mu\ FindFMBusySDS[fm/fm?] \bullet dset!)) \bullet \\
\qquad\qquad d \ \mathsf{within}\ d_1)\} \\
\quad \wedge \\
\mathsf{isolated}(dset!) \\
\hline
\end{array}$$

The secretary may need to know when a committee is free, in order to schedule them for a meeting. We may determine this to be the set of durations during which every committee member is free. Correspondingly, the set of durations for which a faculty member is free is determined as the complement of the durations for which the faculty member is busy, within the input duration.

$$\begin{array}{|l}
\hline
\textit{ChooseFreeSlotSDS} \\
\Delta SDS \\
d? : Duration \\
rqd? : Duration \\
d! : Duration \\
\hline
rqd? \ \mathsf{within}\ d? \\
(\exists d : Duration \mid d \in (\mu\ FindFreeSlotForCommSDS \bullet dset!) \bullet \\
\quad rqd? \ \mathsf{within}\ d) \Rightarrow d! = rqd? \\
\hline
\end{array}$$

We refer back to our operation to determine when a committee is free in a given duration, and realize that it is likely that a committee may only be free for parts of a given duration. In any case, we identify exactly when we would like to schedule the meeting, through the input duration *rqd?*. If the set of durations in which a committee is free has a free slot that can accommodate the requested duration, then the operation returns the requested duration.

```
── SchdCommMeetSDS ──────────────────────
  $\Delta SDS$
  $c? : COMMITTEE$
  $orig? : USER$
──────────
  $c? \in Active$
  $(\mathrm{let}\ a == \mathsf{CommAnnot}(c?) \bullet$
      $(\mathrm{let}\ d == (\mu\ ChooseFreeSlotSDS \bullet d!) \bullet$
      $AddAnnotationSDS[d/d?, a/a?, Secretary/owner?]))$
─────────────────────────────────────────
```

The secretary can schedule a committee meeting by:

- determining a duration for when the committee is free;
- entering a committee annotation (relating to the committee) into every entry that relates to the duration, into the group diary.

```
── RemFutureAnnotsCommSDS ───────────────
  $\Delta SDS$
  $c? : COMMITTEE$
  $orig? : USER$
──────────
  $c? \in Active$
  $\exists\ dset : \mathbb{P}\ Duration \mid dset = \{d_1 : Duration \mid$
      $(\exists\ d_2 : Duration \mid \mathsf{IsFutureDuration}(d_2) \bullet$
          $d_1 \in (\mu\ FindCommBusySDS[d_2/d?] \bullet dset!))\}$
      $\wedge$
      $\mathsf{isolated}(dset) \bullet$
      $(\exists\ a : Annotation \mid a = \mathsf{CommAnnot}(c?) \bullet$
          $(\forall\ d : Duration \mid d \in dset \bullet$
              $RemAnnotationSDS[d/d?, a/a?, Secretary/owner?]))$
─────────────────────────────────────────
```

We need an operation to remove all annotations in the future that relate to a particular committee, if that committee is removed from the system. Of course, we are only concerned with removing the committee annotations from the group diary. We define an operation *RemFutureAnnotsCommSDS* which:

- accepts:
  - a committee which identifies the annotations to be removed;
  - a user who originates the request.
- determines the durations on which the annotation that relates to the committee, occurs in the group diary. Here, we are only concerned with those annotations that are scheduled for future dates, so we search through the group diary from the current time and date, up to the end of the year;
- we may now remove the committee annotations which fall on each duration in the resulting set, from the group diary.

$$\begin{array}{|l}\hline \textit{OpCommSDSComm} \\ \Delta \textit{SDSComm} \\ \hline \theta \textit{SDSUsers}' = \theta \textit{SDSUsers} \\ \hline \end{array}$$

OpCommSDSComm asserts that an operation on the committees in the system, does not affect the users.

$$\begin{array}{|l}\hline \textit{AddCommSDSComm} \\ \textit{OpCommSDSComm} \\ c? : \textit{COMMITTEE} \\ \hline c? \notin \operatorname{dom} \textit{HasMembers} \\ \textit{HasMembers}' = \textit{HasMembers} \oplus \{c? \mapsto \emptyset[\textit{USER}]\} \\ \textit{InActive}' = \textit{InActive} \cup \{c?\} \\ \textit{Active}' = \textit{Active} \\ \hline \end{array}$$

We define an operation to introduce a new committee into the system. As a pre-condition, no committee can be duplicated. When a committee is created, it has no users, and therefore is inactive.

$$
\begin{array}{|l}
\hline
\textit{RemActCommSDSComm} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{Active} \\
\textit{Active}' = \textit{Active} \setminus \{c?\} \\
\textit{InActive}' = \textit{InActive} \\
\textit{HasMembers}' = \{c?\} \ntriangleleft \textit{HasMembers} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{RemInActCommSDSComm} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{InActive} \\
\textit{InActive}' = \textit{InActive} \setminus \{c?\} \\
\textit{Active}' = \textit{Active} \\
\textit{HasMembers}' = \{c?\} \ntriangleleft \textit{HasMembers} \\
\hline
\end{array}
$$

When a committee is removed from the system, we need to distinguish it as being either active or inactive. In particular, if an active committee is removed from the system, we would also require to remove any scheduling for that committee, from the group diary. In both the operations given above:

- we ensure that the committee belongs to the system;
- we remove the committee along with its users from the database of committees and the active or inactive sets.

$$
\begin{array}{|l}
\hline
\textit{OpCommSDS} \\
\Delta \textit{SDS} \\
\hline
\theta \textit{SDSMaps}' = \theta \textit{SDSMaps} \\
\hline
\end{array}
$$

The operations on committees were specified at the *SDSComm* level. In order to be consistent with the other operations, we need to promote the committee-related operations to the *SDS* level at which users interact with the system. *OpCommSDS* ensures that an operation on a committee does not affect the users and diaries in the system.

$$AddCommSDS \mathrel{\widehat=} \exists \Delta SDSComm \bullet OpCommSDS \land AddCommSDSComm$$

We conjoin *OpCommSDS* and an operation to add a committee to the database of committees, to create an operation at the *SDS* level.

$$\begin{array}{l} RemCommSDS \mathrel{\widehat=} \exists \Delta SDSComm \bullet \Delta SDS \land \\ \quad ((RemActCommSDSComm \fatsemi RemFutureAnnotsCommSDS) \\ \qquad \lor \\ \quad (OpCommSDS \land RemInActCommSDSComm)) \end{array}$$

We may define an operation to remove a committee from the database of committees. As in the case of *AddCommSDS*, this operation is introduced at the *SDS* level. Note the following:

- when an active committee is removed from the system, we also need to remove any scheduling information for that committee, from the future entries in the group diary;
- we simply remove a committee if it is inactive.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddFMCommSDSComm}} \\
\hline
OpCommSDSComm \\
c? : COMMITTEE \\
fm? : USER \\
\hline
fm? \in ListOfFacultyMembers \\
c? \in \operatorname{dom} HasMembers \\
fm? \notin HasMembers\ c? \\
HasMembers' = HasMembers \oplus \{c? \mapsto HasMembers\ c? \cup \{fm?\}\} \\
\hline
\end{array}
$$

In addition to adding and removing committees, we need operations to add and remove the committee members. *AddFMCommSDSComm* is an operation to add a faculty member to a committee. As preconditions, we require that:

- both the faculty member and committee belong to the system;
- the faculty member is not already a member of the concerned committee.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemFMCommSDSComm}} \\
\hline
OpCommSDSComm \\
fm? : USER \\
c? : COMMITTEE \\
\hline
fm? \in ListOfFacultyMembers \\
c? \in \operatorname{dom} HasMembers \\
fm? \in HasMembers\ c? \\
HasMembers' = HasMembers \oplus \{c? \mapsto HasMembers\ c? \setminus \{fm?\}\} \\
\hline
\end{array}
$$

We may also remove a faculty member from a committee as long as:

- both the faculty member and committee belong to the system;
- the faculty member is a member of that committee.

$$\begin{array}{|l}
\hline
\textit{InActToInAct} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{InActive} \\
\#(\textit{HasMembers}\ c?) < \textit{ValidComm} \\
\textit{InActive}' = \textit{InActive} \\
\textit{Active}' = \textit{Active} \\
\hline
\end{array}$$

We recall that a committee is functional only if it has at least 2 members. If a committee is inactive and a member is removed from the committee, then the committee will remain inactive.

$$\begin{array}{|l}
\hline
\textit{ActToAct} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{Active} \\
\#(\textit{HasMembers}\ c?) > \textit{ValidComm} \\
\textit{InActive}' = \textit{InActive} \\
\textit{Active}' = \textit{Active} \\
\hline
\end{array}$$

Similarly, if a committee is active, and a faculty member is included in the committee, the committee continues to be active.

$$\textit{KeepAct} \mathrel{\widehat{=}} \textit{InActToInAct} \lor \textit{ActToAct}$$

We may compose the two previous operations into a convenient representation called *KeepAct*. If its predicate holds true, the operation asserts that a change to a committee's membership does not affect its activity.

$$\begin{array}{|l}
\hline
\textit{Activate} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{InActive} \\
\#(\textit{HasMembers}\ c?) = \textit{ValidComm} \\
\textit{InActive}' = \textit{InActive} \setminus \{c?\} \\
\textit{Active}' = \textit{Active} \cup \{c?\} \\
\hline
\end{array}$$

A committee may be activated if:

- it is inactive;
- by adding a new member, we raise its strength to the membership that is required of an active committee.

In this case, the committee must be transferred from the set of inactive committees to the set of active committees.

$$\begin{array}{|l}
\hline
\textit{DeActivate} \\
\textit{OpCommSDSComm} \\
c? : \textit{COMMITTEE} \\
\hline
c? \in \textit{Active} \\
\#(\textit{HasMembers}\ c?) < \textit{ValidComm} \\
\textit{InActive}' = \textit{InActive} \cup \{c?\} \land \textit{Active}' = \textit{Active} \setminus \{c?\} \\
\hline
\end{array}$$

A committee may be deactivated if by removing a committee member, its strength falls below the membership that is required of an active committee. In this case, we transfer the committee from the active set to the inactive set.

$$\begin{array}{l}
\textit{AddFMCommSDS} \mathrel{\widehat=} \exists\, \textit{OpCommSDS} \bullet \\
\quad (\textit{AddFMCommSDSComm} \mathbin{⨾} \\
\qquad (\textit{InActToInAct} \lor \textit{Activate} \lor (\textit{ActToAct} \Rightarrow \\
\qquad\quad (\exists\, \textit{dset}! : \mathbb{P}\, \textit{Duration} \mid \textit{dset}! = \varnothing \bullet \textit{FindSchdCnflctsSDS}))))
\end{array}$$

We may compose operations on committees, to provide a complete specification of what is required when a faculty member is added to a committee. The schema *AddFMCommSDS*:

- adds the faculty member to the committee;
- determines if the committee continues as inactive, or becomes active;
- if the committee is already active, then ensures that there is no conflict between the schedule of the new member, and the schedule for the committee he is added into.

$$\begin{array}{l}
RemFMCommSDS \mathrel{\widehat{=}} \exists\, \Delta SDS \bullet \\
\quad (RemFMCommSDSComm \mathbin{;} \\
\quad\quad (KeepAct \land \Xi SDSMaps) \\
\quad\quad \lor \\
\quad\quad (DeActivate \mathbin{;} RemFutureAnnotsCommSDS))
\end{array}$$

Similarly, we may provide a complete specification of what happens when a member is removed from a committee:

- first, remove the member from the committee;
- the operation terminates successfully if the committee continues to remain active or inactive;
- if the committee becomes inactive, we remove the future scheduling for that committee from the group diary.

$$\begin{array}{|l}
\hline
RemFMAllCommSDS \\
\Delta SDS \\
fm? : USER \\
orig? : USER \\
\hline
\forall c : COMMITTEE \mid \\
\quad c \in \operatorname{dom} HasMembers \land fm? \in HasMembers\ c \bullet RemFMCommSDS[c/c?] \\
\hline
\end{array}$$

The operation *RemFMAllCommSDS* ensures that when a faculty member is removed from the system, he is also removed from all committees of which he is a member.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{OpLoFMSDSUsers}} \\
\Delta \mathit{SDSUsers} \\
\hline
\mathit{Secretary}' = \mathit{Secretary} \\
\hline
\end{array}$$

We may now describe the various operations to add and remove users from the system. In particular, we need to ensure that the secretary is not affected by any of these operations.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AddFMSDSUsers}} \\
\mathit{OpLoFMSDSUsers} \\
\mathit{fm?} : \mathit{USER} \\
\hline
\mathit{fm?} \notin \mathit{ListOfFacultyMembers} \\
\mathit{ListOfFacultyMembers}' = \mathit{ListOfFacultyMembers} \cup \{\mathit{fm?}\} \\
\hline
\end{array}$$

AddFMSDSUsers asserts that no faculty member is represented more than once.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemFMSDSUsers}} \\
\mathit{OpLoFMSDSUsers} \\
\mathit{fm?} : \mathit{USER} \\
\hline
\mathit{fm?} \in \mathit{ListOfFacultyMembers} \\
\#(\mathit{ListOfFacultyMembers} \cup \{\mathit{Secretary}\}) > \mathit{ValidSDS} \\
\mathit{ListOfFacultyMembers}' = \mathit{ListOfFacultyMembers} \setminus \{\mathit{fm?}\} \\
\hline
\end{array}$$

We may also remove a faculty member if:

- he/she already belongs in the system;
- the system remains operational (at least two users) after removing him/her.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AssignDiaryToFM}} \\
\Delta SDSMaps \\
fm? : USER \\
\hline
fm? \notin \operatorname{dom} Owns \\
\exists\, di : SDSDiary \bullet Owns' = Owns \oplus \{fm? \mapsto di\} \\
\hline
\end{array}$$

We need to describe operations on users at the *SDSMaps* level which collects the various users and diaries. Note that when a new faculty member is added, a personal diary is assigned to the faculty member and the database of diaries and faculty members is updated.

$$AddFMSDSMaps \mathrel{\widehat=} \exists\, \Delta SDSUsers \bullet AssignDiaryToFM \land AddFMSDSUsers$$

The operation *AddFMSDSMaps* promotes the operations which (1) add a faculty member to the set of users, and (2) grant him a diary, to the *SDSMaps* level.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RemFMAndDiary}} \\
\Delta SDSMaps \\
fm? : USER \\
\hline
fm? \in \operatorname{dom} Owns \\
Owns' = \{fm?\} \mathbin{\lhd} Owns \\
\hline
\end{array}$$

A faculty member may be removed only if he/she already belongs to the system; if so, the system removes the faculty member and his/her associated diary.

$$RemFMSDSMaps \mathrel{\widehat=} \exists\, \Delta SDSUsers \bullet RemFMAndDiary \land RemFMSDSUsers$$

The operation *RemFMSDSMaps* promotes the effect of removing (1) a faculty member from the set of users, and (2) the associated diary, to the *SDSMaps* level.

$$AddFMSDS \mathrel{\widehat{=}} \exists \Delta SDSMaps \bullet OpSDSMapsSDS \wedge AddFMSDSMaps$$

We promote the operation to add a faculty member, from the *SDSMaps* level, to a form which may be invoked by users at the *SDS* level.

$$\begin{array}{l} RemFMSDS \mathrel{\widehat{=}} \exists \Delta SDSMaps \bullet \Delta SDS \wedge \\ \quad (RemFMSDSMaps \mathbin{;} RemFMAllCommSDS) \end{array}$$

We also promote the operation to remove a faculty member, from the *SDSMaps* level, to *SDS*. However, we need to ensure that when a faculty member is removed from the system, he is also removed from all the committees that he is a member of.

Chapter 7

The Object-Z Specification

STRING

USER

COMMITTEE

We transform the three basic types, namely string, user, and committee, into abstract classes.

Time

$\upharpoonright (<, \leq, \geq, >, \mathsf{Is0}, \mathsf{Succ})$

$t : \mathbb{N}$

$0 \leq t \leq 23$

$<$

$t? : Time$

$t < t?.t$

$\leq$

$t? : Time$

$t \leq t?.t$

$\geq \mathrel{\widehat{=}} \neg <$

$> \mathrel{\widehat{=}} \neg \leq$

*Is*0

$t = 0$

Succ

$t? : Time$

$t = succ(t?.t)$

The type *Time* in Z is transformed into a class that has an attribute of the same type as that of *Time* in the Z specification. As explained in appendix A on enhancing the object-orientedness of the specification through encapsulation, we propose to hide the attributes of a class from its clients and so the attribute t in *Time* is not exported. We follow the same approach throughout the transformation.

The class *Time* provides a set of comparative operations $(<, \leq, \geq, >)$. These operations do not have an obvious counterpart in the Z specification because there we define time as a natural number and we rely on the relational operators that are provided in the mathematical tool-kit. However, in the Object-Z specification we hide the fact that time is modeled as a natural number and so we are obliged to provide these operators explicitly.

A second important consequence of remodeling the relational operators is that we capture the various *associations* between two instances of time. Here, the name of the operation suggests the role of the association between two instances; we assert the association by invoking the operation through an instance of the class, and transform the associated object into a parameter to the operation.[1]

We introduce two additional operations in class *Time*:

- Is0 to assert the earliest hour in a day, namely, midnight;
- Succ to find the next instance of time, for a given instance.

[1] In object-oriented terminology, we may distinguish this relationship as a *parametric association*.

day

$\upharpoonright (<, \leq, \mathsf{Is1}, \mathsf{Is28}, \mathsf{Is29}, \mathsf{Is30}, \mathsf{Is31}, \mathsf{Succ})$

$d : \mathbb{N}_1$

$1 \leq d \leq 31$

$<$

$d? : day$

$d < d?.d$

$\leq$

$d? : day$

$< \lor \mathbf{d} = \mathbf{d?.d}$

*Is*1

$d = 1$

*Is*28

$d = 28$

*Is*29

$d = 29$

*Is*30

$d = 30$

*Is*31

$d = 31$

Succ

$d? : day$

$d = succ(d?.d)$

Similar to time, we remodel the relational operators in class *day*. In the operation $\leq$ we come across one of the shortcomings in Object-Z, namely the lack of a notation for object identity. Therefore, in order to equate one day to another we are required to equate the corresponding attributes of the two objects. Even though we could define all of the relational operators, we ignore them here due to space limitations. However, in a real application it is preferable to define all the relational operators so that one can reuse them in other applications. The class also provides services which assert that a day falls on a particular day of the month. In appendix A, we note that if a schema variable in Z, is constrained by a constant value, we may transform the constraint into an operation, and bring the constant into the scope of the operation; this is the reason behind the operations $\mathsf{Is1} \mathrel{..} \mathsf{Is31}$. Even though these operations could be merged into a single operation that takes a natural number parameter we avoided doing so because it leads to type dependency between clients and the class.

month

$\upharpoonright (<, \mathsf{IsJanuary}, \mathsf{IsFebruary}, \mathsf{IsDecember}, \mathsf{Is30DayMonth}, \mathsf{Is31DayMonth}, \mathsf{Succ})$

$m : \mathbb{N}_1$

$1 \leq m \leq 12$

$<$

$m? : month$

$m < m?.m$

IsJanuary

$m = 1$

IsFebruary

$m = 2$

IsDecember

$m = 12$

Is30DayMonth

$m = 4 \lor m = 6 \lor m = 9 \lor m = 11$

Is31DayMonth

$m = 1 \lor m = 3 \lor m = 5 \lor m = 7 \lor m = 8 \lor m = 10 \lor m = 12$

Succ

$m? : month$

$m = succ(m?.m)$

year

$\upharpoonright (<, \mathsf{IsLeapYear}, \mathsf{IsNonLeapYear}, \mathsf{Succ})$

$y : \mathbb{N}_1$

$1995 \leq y \leq 2005$

$<$

$y? : year$

$y < y?.y$

IsLeapYear

$(y \bmod 100 \neq 0 \land y \bmod 4 = 0) \lor (y \bmod 400 = 0)$

$IsNonLeapYear \mathrel{\widehat=} \neg\ IsLeapYear$

Succ

$y? : year$

$y = succ(y?.y)$

Specifications for the classes *month* and *year* follow the same pattern as for *Time* and *day*, and are self-explanatory.

Date

$\upharpoonright (<, \leq, \geq, >, Day, Month, Year, oneDless)$

$FirstDayOfYear : Date$
$LastDayOfYear : Date$

$(FirstDayOfYear.Day \bullet d!).Is1$
$(FirstDayOfYear.Month \bullet m!).IsJanuary$
$(LastDayOfYear.Day \bullet d!).Is31$
$(LastDayOfYear.Month \bullet m!).IsDecember$

$dt : ((day \times month) \times year)$

$(Day \bullet d!). \leq [\mathsf{DaysInMonth} \bullet \mathsf{d!/d?}]$

<

$dt? : Date$

$(\mathbf{let}\ d_1 == Day \bullet d! \bullet$
$(\mathbf{let}\ m_1 == Month \bullet m! \bullet$
$(\mathbf{let}\ y_1 == Year \bullet y! \bullet$
$(\mathbf{let}\ d_2 == dt?.Day \bullet d! \bullet$
$(\mathbf{let}\ m_2 == dt?.Month \bullet m! \bullet$
$(\mathbf{let}\ y_2 == dt?.Year \bullet y! \bullet$
$\quad y_1. < [\mathsf{y_2/y?}]$
$\quad\quad \vee$
$\quad y_1 = y_2 \wedge m_1. < [\mathsf{m_2/m?}]$
$\quad\quad \vee$
$\quad y_1 = y_2 \wedge m_1 = m_2 \wedge d_1. < [\mathbf{d_2/d?}]))))))$

≤

$dt? : Date$

$< \ \vee$
$Day \bullet d! = dt?.Day \bullet d! \wedge$
$Month \bullet m! = dt?.Month \bullet m! \wedge$
$Year \bullet y! = dt?.Year \bullet y!$

$\geq \ \widehat{=} \neg <$
$> \ \widehat{=} \neg \leq$

Day

$d! : day$

$d! = first(first(dt))$

Date..contd.

Month

$m! : month$

$m! = second(first(dt))$

Year

$y! : year$

$y! = second(dt)$

oneDless

$dt? : Date$

```
(let $d_1 == Day \bullet d! \bullet$
(let $m_1 == Month \bullet m! \bullet$
(let $y_1 == Year \bullet y! \bullet$
(let $d_2 == dt?.Day \bullet d! \bullet$
(let $m_2 == dt?.Month \bullet m! \bullet$
(let $y_2 == dt?.Year \bullet y! \bullet$
    $y_2.Succ[y_1/y?] \wedge$
    $d_1 = LastDayOfYear.Day \bullet d! \wedge$
    $m_1 = LastDayOfYear.Month \bullet m! \wedge$
    $d_2 = FirstDayOfYear.Day \bullet d! \wedge$
    $m_2 = FirstDayOfYear.Month \bullet m!$
        $\vee$
    $y_1 = y_2 \wedge m_1 = m_2 \wedge d_2.Succ[d_1/d?] \vee$
    $y_1 = y_2 \wedge m_2.Succ[m_1/m?] \wedge d_2.Is1 \wedge d_1 = DaysInMonth \bullet d!))))))$
```

DaysInMonth

$d! : day$

```
(let $m == Month \bullet m! \bullet$
(let $y == Year \bullet y! \bullet$
    $(m.Is30DayMonth \Leftrightarrow d!.Is30) \vee$
    $(m.Is31DayMonth \Leftrightarrow d!.Is31) \vee$
    $(m.IsFebruary \wedge y.IsLeapYear \Leftrightarrow d!.Is29) \vee$
    $(m.IsFebruary \wedge y.IsNonLeapYear \Leftrightarrow d!.Is28)))$
```

We introduce the constants FirstDayOfYear and LastDayOfYear into the class *Date* because they are referred to by the operation oneDless of the class. The comparative operators on *Date* ($<, \leq, \geq, >$) are transformed from axiomatic description form in Z; we propose that the operation-form is suitable because we can encapsulate the internal details of the comparisons.

Duration

$\upharpoonright (>, within, IsFutureDuration, SameDate, EndsBeforeOnDate, FallsOnDate, BeginsAtTime)$

$CurrentTime : Time$
$CurrentDate : Date$

$StartTime : Time$
$EndTime : Time$
$StartDate : Date$

$StartTime. < [\mathsf{EndTime/t?}]$

>

$dr? : \downarrow Duration$

$StartDate. > [dr?.StartDate/dt?]$
$\quad \vee$
$StartDate = dr?.StartDate \wedge StartTime. > [\mathsf{dr?.StartTime/t?}]$

inside

$dr? : \downarrow Duration$

$StartDate = dr?.StartDate \wedge$
$\quad (StartTime. > [\mathsf{dr?.StartTime/t?}] \wedge \mathsf{EndTime.} \leq [\mathsf{dr?.EndTime/t?}]$
$\qquad \vee$
$\quad StartTime = dr?.StartTime \wedge EndTime. < [\mathsf{dr?.EndTime/t?}])$

within

$dr? : \downarrow Duration$

$inside \vee$
$\quad StartTime = dr?.StartTime \wedge$
$\quad EndTime = dr?.EndTime \wedge$
$\quad StartDate = dr?.StartDate$

Duration..contd.

$$\begin{array}{l} IsFutureDuration \mathrel{\widehat{=}} \; >[(\mu \downarrow Duration \mid \\ \qquad\qquad StartDate = CurrentDate \wedge \\ \qquad\qquad StartTime = CurrentTime \wedge \\ \qquad\qquad EndTime. > [\mathsf{StartTime/t?}])/\mathsf{dr?}] \end{array}$$

SameDate

$$dr? :\downarrow Duration$$

$$FallsOnDate[dr?.StartDate/dt?]$$

EndsBeforeOnDate

$$dr? :\downarrow Duration$$

$$SameDate \wedge EndTime. < [\mathsf{dr?.StartTime/t?}]$$

FallsOnDate

$$dt? : Date$$

$$StartDate = dt?$$

BeginsAtTime

$$t? : Time$$

$$StartTime = t?$$

Notice that the comparative operation $>$ accepts a polymorphic reference to a duration which means we may reuse the operation to compare a duration with another duration, or with any of its subclasses. We also transform the axiomatic descriptions of within, inside, and IsFutureDuration into operations which accept polymorphic references to duration. In addition, the class *Duration* contains four new operations:

- SameDate checks whether two durations fall on the same date;
- EndsBeforeOnDate asserts that for two durations on the same date, one ends before the other begins;

- FallsOnDate models an association between a duration and a date;
- BeginsAtTime verifies that the start time of a duration matches a particular value of time; it depicts an association between duration and time.

It is important to note that the above four operations were created to satisfy certain constraints that we will encounter later in this specification.

DurFrag

$\upharpoonright (>, within, IsFutureDuration, SameDate, EndsBeforeOnDate,$
$FallsOnDate, BeginsAtTime)$

$Duration$

$EndTime.Succ[StartTime/t?]$

A duration fragment is a specialization of the class duration. Through inheritance, the operations of a duration are made available to duration fragment with the connotation that the operations of duration which accept a polymorphic reference to a duration may be used to compare any combination of a duration and a duration fragment.

Annotation

We model *Annotation* as an abstract class.

In the Z specification, we introduce an annotation as a free type whose values are uniquely mapped to a personal annotation, or to a committee annotation. However, we propose that from an object-oriented perspective, it is worth to rework the actual transformation, into one of inheritance, for the following reasons:

- intuitively, an annotation is a generalization of a committee annotation or of a personal annotation;
- we may continue with the class *Annotation* that is derived through the transformation of the free type, but throughout the specification, we don't deal with annotations, but rather the classes of personal and committee annotations. Here, it becomes necessary for every instance of annotation to distinguish itself as a personal or committee annotation before being used — this is captured elegantly through polymorphism where an object knows its class.

```
┌─ PersonalAnnotation ─────────────────────────
│ Annotation
│ STRING
└──────────────────────────────────────────────
```

A personal annotation multiply inherits the properties of an annotation and that of a string.

```
┌─ CommitteeAnnotation ────────────────────────
│ ↾ (RefersTo)
│ Annotation
│
│ ┌────────────────────────────────────────
│ │ comm : COMMITTEE
│ └────────────────────────────────────────
│
│ ┌─ RefersTo ─────────────────────────────
│ │ c? : COMMITTEE
│ ├──────────
│ │ comm = c?
│ └────────────────────────────────────────
└──────────────────────────────────────────────
```

Similar to personal annotation, a committee annotation inherits from an annotation. However, we cannot claim an inheritance relationship between a committee annotation and a committee: intuitively, a committee annotation has (aggregates) a record of a committee.

In addition to redesigning the class, we also introduce an operation called RefersTo to verify that a committee annotation refers to a particular committee.

SlotStatus

$\upharpoonright (IsAvailable, IsBusy, IsInvalid)$

$SlotStatus_0 ::= Available \mid Busy \mid Invalid$

$s : SlotStatus_0$

IsAvailable

$s = Available$

IsBusy

$s = Busy$

IsInvalid

$s = Invalid$

The class *SlotStatus* is derived by transforming the corresponding free type definition from the Z specification.

Entry

$\upharpoonright (Contains, IsEarliest, IsInCommTime, oneTless, BeginsOnDur,$
$HasStatus, AddAnnotationToEntry, RemAnnotationFromEntry,$
$ReadAnnotationInEntry, ReadStatusInEntry)$

$StartOfDay : Time$
$StartComm : Time$
$EndComm : Time$

$StartOfDay.Is0$
$StartComm.< [\mathsf{EndComm/t?}]$

$ListOfAnnotations : \mathbb{P} \downarrow Annotation$
$SlotTime : Time$
$Status : SlotStatus$

$\neg\ Status.IsInvalid \Rightarrow$
$\quad (ListOfAnnotations = \varnothing \Leftrightarrow Status.IsAvailable$
$\qquad \vee$
$\quad ListOfAnnotations \neq \varnothing \Leftrightarrow Status.IsBusy)$
$Status.IsInvalid \Rightarrow ListOfAnnotations = \varnothing$

Contains

$a? :\downarrow Annotation$

$a? \in ListOfAnnotations$

IsEarliest

$SlotTime = StartOfDay$

IsInCommTime

$SlotTime. \geq [\mathsf{StartComm/t?}] \wedge \mathsf{SlotTime.} < [\mathsf{EndComm/t?}]$

oneTless

$e? : Entry$

$e?.SlotTime.Succ[SlotTime/t?]$

BeginsOnDur

$dr? :\downarrow Duration$

$dr?.BeginsAtTime[SlotTime/t?]$

Entry..contd.

HasStatus

$s? : SlotStatus$

$Status = s?$

AddAnnotToEnt

$\Delta(ListOfAnnotations)$
$a? :\downarrow Annotation$

$\neg\ Status.IsInvalid$
$a? \notin ListOfAnnotations$
$ListOfAnnotations' = ListOfAnnotations \cup \{a?\}$

ChngStatus

$\Delta(ListOfAnnotations, Status)$

$Status.IsAvailable \wedge ListOfAnnotations' \neq \varnothing \Rightarrow Status'.IsBusy$
$Status.IsBusy \wedge ListOfAnnotations' = \varnothing \Rightarrow Status'.IsAvailable$

KeepStatus

$\Delta(ListOfAnnotations, Status)$

$(Status.IsAvailable \wedge ListOfAnnotations' = \varnothing$
$\vee$
$Status.IsBusy \wedge ListOfAnnotations' \neq \varnothing) \Rightarrow$
$Status' = Status$

$AddAnnotationToEntry \mathrel{\widehat=} AddAnnotToEnt \wedge (ChngStatus \vee KeepStatus)$

RemAnnotFromEnt

$\Delta(ListOfAnnotations)$
$a? :\downarrow Annotation$

$\neg\ Status.IsInvalid$
$a? \in ListOfAnnotations$
$ListOfAnnotations' = ListOfAnnotations \setminus \{a?\}$

$RemAnnotationFromEntry \mathrel{\widehat=} RemAnnotFromEnt \wedge (ChngStatus \vee KeepStatus)$

ReadAnnotationInEntry

$aset! : \mathbb{P} \downarrow Annotation$

$\neg\ Status.IsInvalid \Rightarrow aset! = ListOfAnnotations$
$Status.IsInvalid \Rightarrow aset! = \varnothing$

Entry..contd.

ReadStatusInEntry

$s! : SlotStatus$

$s! = Status$

Within the class *Entry*, we begin to observe the effect of transforming a constraint on a variable, into an operation of the class of the corresponding attribute. For instance, we assert that of the two time boundaries for scheduling a committee meeting, one occurs before another:

$$StartComm. < [EndComm/t?]$$

This constraint is resolved by sending a request to the object *StartComm* which:

- searches for a method called $<$ in its class, namely *Time*;
- substitutes the value of *EndComm* for the formal parameter t. This is valid only because the objects *EndComm* and t refer to instances of the same class and hence are type-compatible;
- satisfies the original constraints of the Z specification, namely:

  $$StartComm < EndComm$$

  which may be visually expanded in the context of the class *Time* to:

  $$StartComm.t < EndComm.t$$

Some features of the class *Entry* deserve special mention:

- the list of annotations is now defined as a set of polymorphic references to annotation; this change was warranted by remodeling *Annotation* in Object-Z;

- the operation Contains models an association between an entry and an annotation;
- an instance of entry may invoke the operation IsEarliest to determine if it begins at the break of day; in the context of a page, such an entry would be the first in a sequence of entries;
- the operation IsInCommTime asserts that an entry falls inside the appropriate time for scheduling a committee meeting;
- oneTless asserts that the slot time of one entry is exactly an hour behind that of another entry;
- the operation BeginsOnDur models the association between an entry and a duration: an entry begins on a duration if its time stamp matches the start time of the duration;
- HasStatus simply checks if an entry has a particular status.

In addition, we transform the various operation schemas, AddAnnotationToEntry, ReadAnnotationInEntry, etc., according to our methodology, into operations of the class.

$ItemSeq[X]$

$\upharpoonright (Head, Contains, NumItems, Succ)$

$is : \operatorname{iseq}(X)$

$Head$

$i! : X$

$i! = head(is)$

$Contains$

$i? : X$

$i? \in \operatorname{ran} is$

$NumItems$

$n! : \mathbb{N}$

$n! = \# is$

$Succ$

$i? : X$

$i! : X$

$Contains$

$(\mathbf{let}\ posn == is^{-1}(i?) \bullet$

$\quad posn < \# is \Rightarrow i! = is(posn + 1))$

On examining the Z specification, it becomes evident that *EntrySeq* and *PageSeq* are defined identically as sequences of items, namely entries and pages, respectively. We suggest that the corresponding classes are indeed prime candidates for reuse; the class *ItemSeq* generalizes these reusable properties through the following features:

- the class *ItemSeq* is a sequence of generic items;
- the operation Head returns the first item in the sequence;
- Contains checks if an item is in the sequence;

- NumItems returns the cardinality of the sequence;
- if an input item is in the sequence, the operation Succ returns the next item in the sequence.

```
── EntrySeq ──────────────────────────────────
│ ↾ (Head, Contains, NumItems, Succ)
│ ItemSeq[Entry][es/is, e?/i?, e!/i!]
└─────────────────────────────────────────────
```

We may specialize the generic class *ItemSeq* by inheriting it into a class called *EntrySeq*, and instantiating it with the type *Entry*. The interface to the class *ItemSeq* is inherited to become the interface to the class EntrySeq; in addition, we rename certain variables of *ItemSeq* to remain meaningful with entry.

Page

$\upharpoonright (Contains, IsEarliest, oneDless, FallsOnDur,$
$AddAnnotationToPage, RemAnnotationFromPage, ReadAnnotationInPage,$
$ReadStatusInPage, FindAnnotationInPage, FindStatusInPage)$

$FirstDayOfYear : Date$

$(FirstDayOfYear.Day \bullet d!).Is1$
$(FirstDayOfYear.Month \bullet m!).IsJanuary$

$\mathsf{isolated\_} : \mathbb{P}(\mathbb{P} \downarrow Duration)$

$$\begin{aligned}
&\forall dset : \mathbb{P} \downarrow Duration \bullet \mathsf{isolated}(dset) \Leftrightarrow \\
&\quad (\forall d_1, d_2 : Duration \mid \{d_1, d_2\} \subseteq dset \bullet \\
&\qquad \neg\ d_1.SameDate[d_2/dr?] \\
&\qquad\qquad \vee \\
&\qquad d_1.EndsBeforeOnDate[d_2/dr?] \\
&\qquad\qquad \vee \\
&\qquad d_2.EndsBeforeOnDate[d_1/dr?])
\end{aligned}$$

$ListOfPageEntries : EntrySeq$
$SlotDate : Date$

$$\begin{aligned}
&(ListOfPageEntries.Head \bullet e!).IsEarliest \\
&\forall e_1, e_2 : Entry \mid ListOfPageEntries.Succ[e_1/e?] \bullet e! = e_2 \Leftrightarrow \\
&\quad e_1.oneTless[e_2/e?] \\
&\forall e : Entry \mid ListOfPageEntries.Contains[e/e?] \bullet \\
&\quad (\forall a :\downarrow Annotation \mid e.Contains[a/a?] \bullet \\
&\qquad (\exists ca : CommitteeAnnotation \bullet a = ca \Rightarrow e.IsInCommTime)) \\
&ListOfPageEntries.NumItems \bullet n! = 24
\end{aligned}$$

$Contains \mathrel{\widehat{=}} ListOfPageEntries.Contains$

IsEarliest

$SlotDate = FirstDayOfYear$

oneDless

$p? : Page$

$SlotDate.oneDless[p?.SlotDate/dt?]$

FallsOnDur

$dr? :\downarrow Duration$

$dr?.FallsOnDate[SlotDate/dt?]$

Page..contd.

OpPromoteEntPage

$df? : DurFrag$
$e! : Entry$

$FallsOnDur[df?/dr?]$
$e! = (\mu\, e : Entry \mid ListOfPageEntries.Contains[e/e?]$
$\quad \land\, e.BeginsOnDur[df?/dr?])$

SelectEntsInDur

$d? :\downarrow Duration$
$eset! : \mathbb{P}\, Entry$

$eset! = \{e : Entry \mid$
$\quad \exists\, df : DurFrag \mid df.within[d?/dr?] \bullet$
$\qquad OpPromoteEntPage[df/df?] \bullet e! = e\}$

$AddAnnotationToPage \mathrel{\widehat=} \bigwedge e : Entry \mid$
$\quad e \in SelectEntsInDur \bullet eset! \bullet e.AddAnnotationToEntry$

$RemAnnotationFromPage \mathrel{\widehat=} \bigwedge e : Entry \mid$
$\quad e \in SelectEntsInDur \bullet eset! \bullet e.RemAnnotationFromEntry$

$ReadAnnotationInPage \mathrel{\widehat=} \bigwedge e : Entry \mid$
$\quad e \in SelectEntsInDur \bullet eset! \bullet e.ReadAnnotationInEntry$

$ReadStatusInPage \mathrel{\widehat=} \bigwedge e : Entry \mid$
$\quad e \in SelectEntsInDur \bullet eset! \bullet e.ReadStatusInEntry$

FindAnnotationInPage

$a? :\downarrow Annotation$
$dset! : \mathbb{P} \downarrow Duration$

$dset! = \{e : Entry \mid ListOfPageEntries.Contains[e/e?]$
$\quad \land\, e.Contains \bullet$
$\quad (\mu\, df : DurFrag \mid e.BeginsOnDur[df/dr?]$
$\qquad \land\, FallsOnDur[df/dr?])\} \land$
$\mathsf{isolated}(dset!)$

Page..contd.

FindStatusInPage

$s? : SlotStatus$
$dset! : \mathbb{P} \downarrow Duration$

$$\begin{array}{l}
dset! = \{e : Entry \mid ListOfPageEntries.Contains[e/e?] \\
\quad \wedge\ e.HasStatus \\
\quad (\mu\, df : DurFrag \mid e.BeginsOnDur[df/dr?] \\
\qquad \wedge\ FallsOnDur[df/dr?])\} \wedge \\
\mathsf{isolated}(dset!)
\end{array}$$

The variable *ListOfPageEntries* is now a reference to an object of class *EntrySeq*. We can use the features of *EntrySeq* to assert that a page contains a sequence of exactly 24 entries, and each entry is one hour apart from its predecessor and successor.

We introduce the following operations in the class *Page*:

- Contains models the association between page and entry. This operation replaces a direct access such as:

  $$e \in \operatorname{ran} p.ListOfPageEntries$$

  (where e is an entry, and p is a page) with two levels of encapsulation:

  - the operation Contains in the class of *ListOfPageEntries*, hides the fact that the collection of entries is modeled as a sequence;
  - the operation Contains in the class *Page* prevents clients from directly accessing the attribute *ListOfPageEntries*.

  In effect, the association is transitive and may be interpreted as: a page contains an entry if the entry is in the set of page entries.

- IsEarliest verifies that a page is the first in a diary (if it is dated January 1);
- oneDless asserts that a page precedes another if the slot date of the former is exactly a day behind the slot date of the latter;
- FallsOnDur is interpreted as an association between a page and a duration where a page falls on a duration if its slot date matches the start date of the duration;
- OpPromoteEntPage no longer represents a promotion of an operation from an entry to a page; here, we retain the name to emphasize the role of this operation in the specification. The operation selects an entry from the set of page entries, as an output;
- SelectEntsInDur yields a set of entries which fall inside a given duration;
- we may now apply an operation on a set of successive entries as a distributed concurrent operation (AddAnnotationToPage ⋯ ReadStatusInPage);
- the associations that we emphasized between page and entry/duration, and entry and duration/annotation become evident in the operations to find annotation/status in a page:
  - select those entries in the set of page entries, which contain an annotation or match some status;
  - map each such entry to a duration which begins at the slot time of the entry, and falls on the slot date of the page.

PageSeq

$\upharpoonright (Head, Contains, NumItems, Succ)$
$ItemSeq[Page][ps/is, p?/i?, p!/i!]$

As with *EntrySeq*, the class *PageSeq* inherits the generic class *ItemSeq* and instantiates it with the type *Page*.

SDSUsers

$\upharpoonright (GetSecy, GetFaculty, AddFMSDSUsers, RemFMSDSUsers)$

$ValidSDS : \mathbb{N}_1$

$ValidSDS \geq 2$

$Secretary : USER$
$ListOfFacultyMembers : \mathbb{P}\ USER$

$Secretary \notin ListOfFacultyMembers$

GetSecy

$secy! : USER$

$secy! = Secretary$

GetFaculty

$fms! : \mathbb{P}\ USER$

$fms! = ListOfFacultyMembers$

AddFMSDSUsers

$\Delta(ListOfFacultyMembers)$
$fm? : USER$

$fm? \notin ListOfFacultyMembers$
$ListOfFacultyMembers' = ListOfFacultyMembers \cup \{fm?\}$

RemFMSDSUsers

$\Delta(ListOfFacultyMembers)$
$fm? : USER$

$fm? \in ListOfFacultyMembers$
$\#(ListOfFacultyMembers \cup \{Secretary\}) > ValidSDS$
$ListOfFacultyMembers' = ListOfFacultyMembers \setminus \{fm?\}$

The operations GetSecy and GetFaculty simply return the secretary and set of faculty members, respectively. These operations are necessary to provide strict encapsulation of state. The other operations of the class serve to add and remove a faculty member from the system.

$Diary$

$\upharpoonright (AddAnnotationToDiary, RemAnnotationFromDiary,$
$ReadAnnotationInDiary, ReadStatusInDiary, FindAnnotationInDiary,$
$FindStatusInDiary)$

$| \; CurrentDate : Date$

$ListOfDiaryPages : PageSeq$

$$\begin{array}{l}
(CurrentDate.Year \bullet y!).IsLeapYear \Leftrightarrow \\
\quad ListOfDiaryPages.NumItems \bullet n! = 366 \\
\qquad\qquad \wedge \\
(CurrentDate.Year \bullet y!).IsNonLeapYear \Leftrightarrow \\
\quad ListOfDiaryPages.NumItems \bullet n! = 365 \\
(ListOfDiaryPages.Head \bullet p!).IsEarliest \\
\forall p_1, p_2 : Page \mid ListOfDiaryPages.Succ[p_1/p?] \bullet p! = p_2 \Leftrightarrow \\
\quad p_1.oneDless[p_2/p?]
\end{array}$$

$OpPromotePageDiary$

$$\begin{array}{l}
dr? :\downarrow Duration \\
p! : Page \\
\hline
p! = (\mu\, p : Page \mid ListOfDiaryPages.Contains[p/p?] \wedge p.FallsOnDur)
\end{array}$$

$$\begin{array}{l}
AddAnnotationToDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.AddAnnotationToPage \\
RemAnnotationFromDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.RemAnnotationFromPage \\
ReadAnnotationInDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.ReadAnnotationInPage \\
ReadStatusInDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.ReadStatusInPage \\
FindAnnotationInDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.FindAnnotationInPage \\
FindStatusInDiary \mathrel{\widehat=} OpPromotePageDiary \bullet p!.FindStatusInPage
\end{array}$$

The state predicate of the class *Diary* illustrates an opportunity to reuse the interface to the class *ItemSeq*. For instance, we capture the constraint that two successive pages in a diary have slot dates that are one day apart, by sending a request to the object *ListOfDiaryPages* which invokes the operation Succ from the class *ItemSeq*.

The operation OpPromotePageDiary is an altered form of the original promotion; here, we select a page from the diary as an output of the operation.

We use the nesting operator ($\bullet$) to extend the signature of the selection operation (making the selected page visible) and define the various operations on a diary, as the corresponding operations on the selected page.

$$
\begin{array}{|l}
\textit{AccessRight} \\
\hline
\upharpoonright (\textit{IsReadTotal}, \textit{IsReadPartial}, \textit{IsWrite}) \\
\textit{AccessRight}_0 ::= \textit{ReadTotal} \mid \textit{ReadPartial} \mid \textit{Write} \\
\\
\quad \boxed{ar : \textit{AccessRight}_0} \\
\\
\quad \textit{IsReadTotal} \\
\quad ar = \textit{ReadTotal} \\
\\
\quad \textit{IsReadPartial} \\
\quad ar = \textit{ReadPartial} \\
\\
\quad \textit{IsWrite} \\
\quad ar = \textit{Write} \\
\hline
\end{array}
$$

The class *AccessRight* is derived by transforming the corresponding free type of the Z specification.

SDSDiary

$\upharpoonright (GetOwner, ChkAccess)$

$Diary$

$IsAccessedBy : SDSUsers$
$DiaryOwner : USER$
$DiaryAccess : USER \twoheadrightarrow \mathbb{P}\, AccessRight$

$\langle IsAccessedBy.GetFaculty \bullet fms!, \{IsAccessedBy.GetSecy \bullet secy!\}\rangle$
$\quad \text{partitions} \operatorname{dom} DiaryAccess$
$DiaryOwner \in \operatorname{dom} DiaryAccess$

GetOwner

$do! : USER$

$do! = DiaryOwner$

ChkAccess

$u? : USER$
$acc? : AccessRight$

$u? \in \operatorname{dom} DiaryAccess \land acc? \in DiaryAccess(u?)$

An *SDSDiary* inherits the properties of a diary and adds two other operations, namely, GetOwner, and ChkAccess.

In the initial transformation, the schema inclusion of *SDSUsers* in *SDSDiary* is transformed into an inheritance of the class *SDSUsers* into the class *SDSDiary*. Clearly, a diary is not a specialization of a set of users; however, it is valid to reason that *SDSDiary* is associated with *SDSUsers* in order to define the access rights. Our approach is to redefine the relationship between the two classes, as an instantiation in *SDSDiary*, such as:

$IsAccessedBy : SDSUsers$

where the object-reference is named appropriately to reflect the association. The information in *SDSUsers* may be accessed through the object-reference, in order to maintain the state predicate in *SDSDiary*.

GroupDiary

$\upharpoonright (GetOwner, ChkAccess)$

SDSDiary

$$\begin{array}{l}
DiaryOwner = IsAccessedBy.GetSecy \bullet secy! \\
\forall p : Page \mid ListOfDiaryPages.Contains[p/p?] \bullet \\
\quad (\forall e : Entry \mid p.Contains[e/e?] \bullet \\
\quad\quad (\forall a : \downarrow Annotation \mid e.Contains[a/a?] \bullet \\
\quad\quad\quad (\exists ca : CommitteeAnnotation \bullet a = ca))) \qquad \wedge \\
DiaryAccess = \{DiaryOwner \mapsto \{(\mu\, AccessRight \mid IsReadTotal), \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mu\, AccessRight \mid IsReadPartial), \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mu\, AccessRight \mid IsWrite)\}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \cup \\
\qquad\qquad \{fm : USER \mid fm \in IsAccessedBy.GetFaculty \bullet fms! \bullet \\
\qquad\qquad\qquad fm \mapsto \{(\mu\, AccessRight \mid IsReadTotal)\}\}
\end{array}$$

$PersonalDiary$

$\upharpoonright (GetOwner, ChkAccess)$

$SDSDiary$

$$
\begin{array}{l}
DiaryOwner \in IsAccessedBy.GetFaculty \bullet fms! \\
\forall p : Page \mid ListOfDiaryPages.Contains[p/p?] \bullet \\
\quad (\forall e : Entry \mid p.Contains[e/e?] \bullet \\
\qquad (\forall a : \downarrow Annotation \mid e.Contains[a/a?] \bullet \\
\qquad\quad (\exists pa : PersonalAnnotation \bullet a = pa))) \qquad \wedge \\
DiaryAccess = \{DiaryOwner \mapsto \{(\mu\, AccessRight \mid IsReadTotal), \\
\qquad\qquad\qquad\qquad\qquad (\mu\, AccessRight \mid IsReadPartial), \\
\qquad\qquad\qquad\qquad\qquad (\mu\, AccessRight \mid IsWrite)\}, \\
\qquad IsAccessedBy.GetSecy \bullet secy! \mapsto \{(\mu\, AccessRight \mid IsReadPartial)\}\} \\
\qquad\quad \cup \\
\qquad \{fm : USER \mid \\
\qquad\quad fm \in (\mathrm{dom}\, DiaryAccess \setminus \{IsAccessedBy.GetSecy \bullet \\
\qquad\qquad secy!, DiaryOwner\}) \bullet \\
\qquad\qquad\quad fm \mapsto \{\}\}
\end{array}
$$

As in the case of *Annotation*, we reason that:

- intuitively, group and personal diaries are specializations of *SDSDiary*;
- the axioms IsGroupDiary and IsPersDiary in the Z specification, exist only to distinguish a diary as a group or personal diary. We replace these axioms as follows:
  - create the classes *GroupDiary* and *PersonalDiary* as specializations of *SDSDiary*;
  - transform the predicate of the two axioms into the state predicate of the corresponding classes.

Indeed, this rework solves one of the difficulties in specifying the case study in Z:

- consider the mapping from each user to a diary in the schema *SDSMaps*:

  $$Owns : USER \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;+\!\!\!\!\!\!\rightarrow SDSDiary$$

  In order to include both the group and personal diaries in the range of *Owns*, we are forced to choose from:

  - distinguishing an *SDSDiary* as a group or personal diary through an axiomatic description; or
  - mapping both types through constructor functions, onto a free type. For instance, we may introduce *Owns* as a mapping from each user to a free type called *DiaryType*:

    $$\begin{array}{l} DiaryType ::= \mathsf{grdi}\langle\!\langle GroupDiary \rangle\!\rangle \mid \mathsf{prdi}\langle\!\langle PersonalDiary \rangle\!\rangle \\ Owns : USER \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;+\!\!\!\!\!\!\rightarrow DiaryType \end{array}$$

    and for instance, distinguish the secretary as the owner of the group diary:

    $$\begin{array}{l} \exists\, dt : DiaryType \mid dt \in \operatorname{ran} Owns \bullet \\ \quad \exists\, gd : GroupDiary \bullet \\ \quad\quad Owns(Secretary) = dt \land dt = \mathsf{grdi}(gd) \\ \quad\quad\quad \ldots \end{array}$$

    which is indeed cumbersome. However, Object-Z offers an adequate solution to this problem, through polymorphic references. If both group and personal diary inherit from *SDSDiary*, then we can model the attribute *Owns* as:

    $$Owns : USER \rightarrowtail\!\!\!\!\!\!\!\!\;\;\;+\!\!\!\!\!\!\rightarrow \downarrow SDSDiary$$

    where we may choose an appropriate subclass of *SDSDiary* to assign to each user. This is another motivation for introducing an inheritance hierarchy to model the relationship between *SDSDiary*, and group and personal diaries.

SDSMaps

$\upharpoonright (GetUsers, AddAnnotationSDSMaps, RemAnnotationSDSMaps,$
$ReadAnnotationSDSMaps, ReadStatusSDSMaps, FindAnnotationSDSMaps,$
$FindStatusSDSMaps, AddFMSDSMaps, RemFMSDSMaps)$

$ValidSDS : \mathbb{N}_1$

$ValidSDS = 2$

$users : SDSUsers$
$Owns : USER \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow\!\!\!\!\!\!\nrightarrow \downarrow SDSDiary$

$\#Owns \geq ValidSDS$
$\langle \{users.GetSecy \bullet secy!\}, users.GetFaculty \bullet fms!\rangle \text{ partitions dom } Owns$
$\forall\, u : USER \mid u \in \text{dom } Owns \bullet (Owns(u)).GetOwner \bullet do! = u$
$\exists\, gd : GroupDiary \bullet Owns(users.GetSecy \bullet secy!) = gd$
$\forall\, fm : USER \mid fm \in users.GetFaculty \bullet fms! \bullet$
$\quad (\exists\, pd : PersonalDiary \bullet Owns(fm) = pd)$

GetUsers

$uset! : SDSUsers$

$uset! = users$

OpChkDiaryAccess

$orig? : USER$
$owner? : USER$
$acc? : AccessRight$
$dr? : Duration$

$\{orig?, owner?\} \subseteq \text{dom } Owns$
$(\textbf{let } diary == Owns(owner?) \bullet$
$\quad diary.ChkAccess[orig?/u?])$
$acc?.IsWrite \Rightarrow dr?.IsFutureDuration$

OpPromoteDiarySDSMaps

$diary! :\downarrow SDSDiary$
$owner? : USER$

$owner? \in \text{dom } Owns$
$diary! = Owns(owner?)$

SDSMaps..contd.

$$
\begin{array}{l}
AddAnnotationSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsWrite)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.AddAnnotationToDiary \\
RemAnnotationSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsWrite)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.RemAnnotationFromDiary \\
ReadAnnotationSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsReadTotal)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.ReadAnnotationInDiary \\
ReadStatusSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsReadPartial)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.ReadStatusInDiary \\
FindAnnotationSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsReadTotal)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.FindAnnotationInDiary \\
FindStatusSDSMaps \mathrel{\widehat{=}} OpChkDiaryAccess[(\mu\, AccessRight \mid IsReadPartial)/acc?] \\
\qquad \land\ OpPromoteDiarySDSMaps \bullet diary!.FindStatusInDiary
\end{array}
$$

AssignDiaryToFM

$$
\begin{array}{|l}
\Delta(Owns) \\
fm? : USER \\
\hline
fm? \notin \operatorname{dom} Owns \\
\exists\, di : PersonalDiary \bullet Owns' = Owns \oplus \{fm? \mapsto di\} \\
\hline
\end{array}
$$

$$AddFMSDSMaps \mathrel{\widehat{=}} AssignDiaryToFM \land users.AddFMSDSUsers$$

RemFMAndDiary

$$
\begin{array}{|l}
\Delta(Owns) \\
fm? : USER \\
\hline
fm? \in \operatorname{dom} Owns \\
Owns' = \{fm?\} \lhd Owns \\
\hline
\end{array}
$$

$$RemFMSDSMaps \mathrel{\widehat{=}} RemFMAndDiary \land users.RemFMSDSUsers$$

Among the various features introduced in *SDSMaps*, the important ones that deserve special attention are:

- operations to modify/read/find entries in a diary, which are specified by conjoining the operations to select a diary and check a user's right to access the diary. The nesting operator extends the signature of the conjoint operation so that each operation is invoked by sending a request to the selected diary.

- operations to add/remove a faculty member, which are specified by modifying the database of users and diaries in *SDSMaps*, and by sending a request to the object *users* to update the set of users in the system.

$SDSComm$

$\upharpoonright (GetUsers, Contains, IsActive, IsInActive,$
$IsCommMember, IsNotCommMember, AddCommSDSComm,$
$RemActCommSDSComm, RemInActCommSDSComm,$
$AddFMCommSDSComm, RemFMCommSDSComm, InActToInAct,$
$ActToAct, KeepAct, Activate, DeActivate)$

$ValidComm : \mathbb{N}_1$

$ValidComm = 2$

$users : SDSUsers$
$HasMembers : COMMITTEE \twoheadrightarrow \mathbb{P}\ USER$
$Active : \mathbb{F}\ COMMITTEE$
$InActive : \mathbb{F}\ COMMITTEE$

$\langle Active, InActive \rangle$ partitions dom $HasMembers$

$$(\forall c : COMMITTEE \mid c \in \mathrm{dom}\ HasMembers \bullet$$
$$HasMembers\ c \subseteq users.GetFaculty \bullet fms!$$
$$\land$$
$$\#(HasMembers\ c) \geq ValidComm \Rightarrow c \in Active$$
$$\land$$
$$\#(HasMembers\ c) < ValidComm \Rightarrow c \in InActive)$$

$GetUsers$

$uset! : SDSUsers$

$uset! = users$

$Contains$

$c? : COMMITTEE$

$c? \in \mathrm{dom}\ HasMembers$

$IsActive$

$c? : COMMITTEE$

$c? \in Active$

$IsInActive \mathrel{\widehat=} Contains \land \neg\ IsActive$

SDSComm..contd.

IsOnComm

$fm? : USER$
$c? : COMMITTEE$

$fm? \in HasMembers(c?)$

$IsCommMember \mathrel{\widehat{=}} Contains \land IsOnComm$

$IsNotCommMember \mathrel{\widehat{=}} Contains \land \neg\ IsOnComm$

AddCommSDSComm

$\Delta(HasMembers, InActive)$
$c? : COMMITTEE$

$\neg\ Contains$
$HasMembers' = HasMembers \oplus \{c? \mapsto \varnothing[USER]\}$
$InActive' = InActive \cup \{c?\}$

RemActCommSDSComm

$\Delta(HasMembers, Active)$
$c? : COMMITTEE$

$IsActive$
$Active' = Active \setminus \{c?\}$
$HasMembers' = \{c?\} \lhd HasMembers$

RemInActCommSDSComm

$\Delta(HasMembers, InActive)$
$c? : COMMITTEE$

$IsInActive$
$InActive' = InActive \setminus \{c?\}$
$HasMembers' = \{c?\} \lhd HasMembers$

AddFMCommSDSComm

$\Delta(HasMembers)$
$c? : COMMITTEE$
$fm? : USER$

$fm? \in users.GetFaculty \bullet fms!$
$IsNotCommMember$
$HasMembers' = HasMembers \oplus \{c? \mapsto HasMembers(c?) \cup \{fm?\}\}$

SDSComm..contd.

RemFMCommSDSComm

$\Delta(HasMembers)$
$c? : COMMITTEE$
$fm? : USER$

$fm? \in users.GetFaculty \circledcirc fms!$
$IsCommMember$
$HasMembers' = HasMembers \oplus \{c? \mapsto HasMembers(c?) \setminus \{fm?\}\}$

InActToInAct

$c? : COMMITTEE$

$IsInActive$
$\#(HasMembers(c?)) < ValidComm$

ActToAct

$c? : COMMITTEE$

$IsActive$
$\#(HasMembers(c?)) > ValidComm$

$KeepAct \mathrel{\widehat=} InActToInAct \lor ActToAct$

Activate

$\Delta(Active, InActive)$
$c? : COMMITTEE$

$IsInActive$
$\#(HasMembers(c?)) = ValidComm$
$InActive' = InActive \setminus \{c?\}$
$Active' = Active \cup \{c?\}$

DeActivate

$\Delta(Active, InActive)$
$c? : COMMITTEE$

$IsActive$
$\#(HasMembers(c?)) < ValidComm$
$InActive' = InActive \cup \{c?\}$
$Active' = Active \setminus \{c?\}$

We introduce the following features in the class *SDSComm*:

- the operation GetUsers retrieves a reference to the object *users*;
- Contains verifies that a particular committee exists in the database of committees;
- IsActive checks if a particular committee is active;
- IsInActive verifies that a committee is inactive if it exists in the system but is not active;
- IsOnComm verifies that a user is a member of a particular committee;
- the operations IsCommMember and IsNotCommMember compose the sub-operations Contains and IsOnComm to verify that a user is a committee member;
- the remaining operations in the class are derived from those of the Z specification, which act on the schema *SDSMaps*.

```
SDS
↾ (AddAnnotationSDS, RemAnnotationSDS, ReadAnnotationSDS,
ReadStatusSDS, FindAnnotationSDS, FindStatusSDS, FindCommBusySDS,
FindFMBusySDS, FindFreeSlotForCommSDS, SchdCommMeetSDS,
AddCommSDS, RemCommSDS, AddFMCommSDS, RemFMCommSDS,
AddFMSDS, RemFMSDS)
```

$$\mathsf{isolated\_} : \mathbb{P}(\mathbb{P} \downarrow \mathit{Duration})$$

$$\begin{array}{l}
\forall\, dset : \mathbb{P} \downarrow \mathit{Duration} \bullet \mathsf{isolated}(dset) \Leftrightarrow \\
\quad (\forall\, d_1, d_2 : \mathit{Duration} \mid \{d_1, d_2\} \subseteq dset \bullet \\
\quad\quad \neg\; d_1.\mathit{SameDate}[d_2 / dr?] \\
\quad\quad\quad \vee \\
\quad\quad d_1.\mathit{EndsBeforeOnDate}[d_2 / dr?] \\
\quad\quad\quad \vee \\
\quad\quad d_2.\mathit{EndsBeforeOnDate}[d_1 / dr?])
\end{array}$$

SDS..contd.

$$\_ \cup_1 \_ : \mathbb{P}\, Duration \times \mathbb{P}\, Duration \rightarrow \mathbb{P}\, Duration$$

$$\begin{array}{l}
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, Duration \bullet \\
\quad dset_1 \cup_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{d : Duration \mid \\
\quad\quad\quad \forall\, df : DurFrag \mid df.within[d/dr?] \bullet \\
\quad\quad\quad\quad (\exists\, d_1 : Duration \mid d_1 \in dset_1 \vee d_1 \in dset_2 \bullet df.\mathsf{within}[d_1/dr?])\} \\
\quad\quad \wedge\ \mathsf{isolated}(dset_3)
\end{array}$$

$$\_ \cap_1 \_ : \mathbb{P}\, Duration \times \mathbb{P}\, Duration \rightarrow \mathbb{P}\, Duration$$

$$\begin{array}{l}
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, Duration \bullet \\
\quad dset_1 \cap_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{d : Duration \mid \\
\quad\quad\quad \forall\, df : DurFrag \mid df.\mathsf{within}[d/dr?] \bullet \\
\quad\quad\quad\quad (\exists\, d_1, d_2 : Duration \mid d_1 \in dset_1 \wedge d_2 \in dset_2 \bullet \\
\quad\quad\quad\quad\quad df.\mathsf{within}[d_1/dr?] \wedge df.\mathsf{within}[d_2/dr?])\} \\
\quad\quad \wedge\ \mathsf{isolated}(dset_3)
\end{array}$$

$$\_ \setminus_1 \_ : \mathbb{P}\, Duration \times \mathbb{P}\, Duration \rightarrow \mathbb{P}\, Duration$$

$$\begin{array}{l}
\forall\, dset_1, dset_2, dset_3 : \mathbb{P}\, Duration \bullet \\
\quad dset_1 \setminus_1 dset_2 = dset_3 \Leftrightarrow \\
\quad\quad dset_3 = \{d : Duration \mid \\
\quad\quad\quad \forall\, df : DurFrag \mid df.\mathsf{within}[d/dr?] \bullet \\
\quad\quad\quad\quad (\exists\, d_1 : Duration \mid d_1 \in dset_1 \bullet df.\mathsf{within}[d_1/dr?]) \\
\quad\quad\quad \wedge \\
\quad\quad\quad\quad \neg\, (\exists\, d_2 : Duration \mid d_2 \in dset_2 \bullet df.\mathsf{within}[d_2/dr?])\} \\
\quad\quad \wedge\ \mathsf{isolated}(dset_3)
\end{array}$$

SDS..contd.

$maps : SDSMaps$
$comm : SDSComm$

$maps.GetUsers \bullet uset! = comm.GetUsers \bullet uset!$

$AddAnnotationSDS \mathrel{\widehat=} maps.AddAnnotationSDSMaps$
$RemAnnotationSDS \mathrel{\widehat=} maps.RemAnnotationSDSMaps$
$ReadAnnotationSDS \mathrel{\widehat=} maps.ReadAnnotationSDSMaps$
$ReadStatusSDS \mathrel{\widehat=} maps.ReadStatusSDSMaps$
$FindAnnotationSDS \mathrel{\widehat=} maps.FindAnnotationSDSMaps$
$FindStatusSDS \mathrel{\widehat=} maps.FindStatusSDSMaps$

FindCommBusySDS

$c? : COMMITTEE$
$d? : Duration$
$dset! : \mathbb{P}\ Duration$

$comm.Contains$
$\exists\ ca : CommitteeAnnotation \mid ca.RefersTo \bullet$
$\quad FindAnnotationSDS[(maps.GetUsers \bullet uset!).GetSecy \bullet$
$\qquad secy!/owner?, ca/a?] \bullet dset! = dset!$

FindFMBusySDS

$fm? : USER$
$d? : Duration$
$dset! : \mathbb{P}\ Duration$

$fm? \in (maps.GetUsers \bullet uset!).GetFaculty \bullet fms!$
$dset! = \{d : Duration \mid$
$\quad \exists\ c : COMMITTEE \mid comm.IsActive[c/c?]$
$\qquad \wedge\ comm.IsCommMember[c/c?] \bullet$
$\qquad\quad d \in FindCommBusySDS[c/c?] \bullet dset!\}$
$\qquad\qquad \cup_1$
$\quad FindStatusSDS[(\mu\ SlotStatus \mid IsBusy)/s?, fm?/owner?] \bullet dset!$

SDS..contd.

FindSchdCnflcts

$fm? : USER$
$c? : COMMITTEE$
$dset! : \mathbb{P}\ Duration$

$$\begin{array}{l}
fm? \in (maps.GetUsers \circledcirc uset!).GetFaculty \circledcirc fms! \\
comm.IsNotCommMember \\
dset! = \{d : Duration \mid \\
\qquad \exists\, d_1 : Duration \mid d_1.IsFutureDuration \bullet \\
\qquad\quad d \in FindFMBusySDS[d_1/d?] \circledcirc dset!\} \\
\qquad\qquad \cap_1 \\
\quad \{d : Duration \mid \\
\qquad \exists\, d_1 : Duration \mid d_1.IsFutureDuration \bullet \\
\qquad\quad d \in FindCommBusySDS[d_1/d?] \circledcirc dset!\}
\end{array}$$

FindFreeSlotForCommSDS

$d? : Duration$
$c? : COMMITTEE$
$dset! : \mathbb{P}\ Duration$

$$\begin{array}{l}
dset! = \{d : Duration \mid \\
\quad \forall fm : USER \mid comm.IsCommMember[fm/fm?] \bullet \\
\qquad (\exists\, d_1 : Duration \mid \\
\qquad\quad d_1 \in (\{d?\} \setminus_1 FindFMBusySDS[fm/fm?] \circledcirc dset!) \bullet \\
\qquad\qquad d.\mathsf{within}[d_1/dr?])\} \wedge \mathsf{isolated}(dset!)
\end{array}$$

ChooseFreeSlotSDS

$d? : Duration$
$rqd? : Duration$
$d! : Duration$

$$\begin{array}{l}
rqd?.\mathsf{within}[d?/dr?] \\
(\exists\, d : Duration \mid d \in (FindFreeSlotForCommSDS \circledcirc dset!) \bullet \\
\quad rqd?.\mathsf{within}[d/dr?]) \Rightarrow d! = rqd?
\end{array}$$

SchdCommMeetSDS

$c? : COMMITTEE$
$orig? : USER$

$$\begin{array}{l}
comm.IsActive \\
\exists\, ca : CommitteeAnnotation \mid ca.RefersTo \bullet \\
\quad (\exists\, d : Duration \mid ChooseFreeSlotSDS \circledcirc d! = d \bullet \\
\qquad AddAnnotationSDS[d/d?, ca/a?, \\
\qquad\quad maps.(GetUsers \circledcirc uset!).GetSecy \circledcirc secy!/owner?])
\end{array}$$

SDS..contd.

RemFutureAnnotsCommSDS

$c? : COMMITTEE$
$orig? : USER$

$comm.IsActive$
$\exists\, dset : \mathbb{P}\, Duration \mid dset = \{d_1 : Duration \mid$
$\quad (\exists\, d_2 : Duration \mid d_2.IsFutureDuration \bullet$
$\qquad d_1 \in FindCommBusySDS[d_2/d?] \bullet dset!)\}$
$\land\ \mathsf{isolated}(dset!) \bullet$
$(\exists\, ca : CommitteeAnnotation \mid ca.RefersTo \bullet$
$\quad (\forall\, d : Duration \mid d \in dset \bullet$
$\qquad RemAnnotationSDS[d/d?, ca/a?,$
$\qquad\quad (maps.GetUsers \bullet uset!).GetSecy \bullet secy!/owner?]))$

$AddCommSDS \mathrel{\widehat=} comm.AddCommSDSComm$
$RemCommSDS \mathrel{\widehat=} (comm.RemActCommSDSComm \fatsemi RemFutureAnnotsCommSDS$
$\qquad \lor$
$\quad comm.RemInActCommSDSComm$

RemFMAllCommSDS

$fm? : USER$
$orig? : USER$

$\forall\, c : COMMITTEE \mid comm.IsCommMember[c/c?] \bullet RemFMCommSDS[c/c?]$

$AddFMCommSDS \mathrel{\widehat=} (comm.AddFMCommSDSComm \fatsemi$
$\quad (comm.InActToInAct$
$\qquad \lor$
$\quad comm.Activate$
$\qquad \lor$
$\quad (comm.ActToAct \Rightarrow FindSchdCnflcts \bullet dset! = \varnothing)))$
$RemFMCommSDS \mathrel{\widehat=} (comm.RemFMCommSDSComm \fatsemi$
$\quad (comm.KeepAct \lor (comm.DeActivate \fatsemi RemFutureAnnotsCommSDS)))$
$AddFMSDS \mathrel{\widehat=} maps.AddFMSDSMaps \land comm.AddFMSDSComm$
$RemFMSDS \mathrel{\widehat=} maps.RemFMSDSMaps \fatsemi RemFMAllCommSDS$

The class *SDS* is an aggregation of users, diaries, and committees in the system. We introduce the following features into the class:

- the axiomatic descriptions for isolated, $\cup_1$, $\cap_1$, and $\setminus_1$, are transformed into the class because these are used by various operations in the class. In chapter 8 we explain why these functions on durations are not transformed into operations on durations;
- a constraint which asserts that the references to *SDSUsers*, in *SDSMaps* and *SDSComm*, are references to the same object. This constraint ensures that when a user is added to or removed from the system, both *SDSMaps* and *SDSComm* experience the same change;
- the operations in *SDS* are transformed from those in the Z specification; the visibility list includes only those operations through which users may interact with the system. We note a distinction between the operations in Z and their counterparts in Object-Z:
  * In Z, a change to a part of a state space, is applied to the larger context through promotion;
  * in Object-Z, the change to an aggregate object is accomplished by sending a request to a component object. For instance, we trace the chain of requests which are triggered by a user who wishes to add an annotation to an entry:

    $AddAnnotationSDS$
    $\xrightarrow{maps} AddAnnotationSDSMaps$
    $\xrightarrow{diary!} AddAnnotationToDiary$
    $\xrightarrow{p!} AddAnnotationToPage$
    $\xrightarrow{e} AddAnnotationToEntry$

    Here, an arrow represents a transition from one request to the next, and the name above the arrow indicates the object which interprets the request.

If an entry is successfully modified, then the change propagates backward, through the chain of object references, to indicate the change of state to the shared-diary system. In essence, where promotion specifies an explicit change to the state space in the Z specification, objects communicate through messages to achieve the same effect, but implicitly.

Chapter 8

Conclusion

In this thesis we presented a methodology to transform a Z specification into Object-Z; it included rules to transform each of eight syntactic categories of Z as identified in[29] to an equivalent structure in Object-Z. The methodology has been successfully applied to a specification of a shared-diary system. The derived Object-Z specification was then subjected to a re-analysis phase to enhance the object-orientedness of the design.

The motivation for this thesis came from the difficulties encountered in the maintenance of legacy systems that are commonplace in the software industry today. We expect that such legacy systems originated well before the object-oriented paradigm came into existence, and evolved with very little attention to structured analysis and design, or readable documentation. We propose that it is well worth the effort to reengineer such legacy software in order to take advantage of the benefits of object-orientation, namely, lowered coupling, maintainability, and reuse.

A detailed comparison of Z and Object-Z is beyond the scope of this thesis. However, we do summarize their similarities and differences based on our experience in developing the transformation methodology and the case study to which the methodology was applied. These details are elaborated in the following sections.

Scope. The scope of a definition in Z extends from its introduction to the end of the specification; in contrast, the scope of a definition in Object-Z is limited to the class in which it is defined. Over the transformation, axiomatic descriptions and generic constants may appear redundantly across every class which references them. We may eliminate the redundancy by moving the definition to the nearest superclass in common to all the classes which reference the definition; if this is not possible, as in the case of definitions that are transformed into multiple inheritance subtrees which are not related, it will be necessary to equate the definitions in a first subclass that derives from the subtrees. This overhead pays for itself because we eliminate global definitions, which is one of the requirements of object-orientation.

Promotion. It is contended[10] that although Z provides a mechanism to promote an operation to a larger context, the process obscures the specification. Object-Z offers a cleaner solution through reference semantics: promotion is effected by selecting an object from an aggregate and sending a request to the object to invoke the operation; the object undergoes an implicit change and reference semantics ensure that the particular object-reference remains constant.

Delta-Listing A Δ-list is a notation used in operations in Object-Z to list only those attributes of an object, that are changed by the operation; unlisted attributes are implicitly preserved by the operation. This feature is semantically different from the Ξ and Δ notations of Z, which introduce the unprimed and primed forms of an entire state space, and require every state space variable to be accounted for, over the operation.

In our experience, Δ-lists in Object-Z help to shorten a Z specification by eliminating Halls' style operations which exist only as a convenience to preserve a portion of the state space over several operations. Whether the Δ-list concept eliminates some clutter and obscurity in a specification by preserving unlisted attributes implicitly, is a matter of personal taste.

Reusing Suboperations. In a Z specification, an operation schema may act on multiple state spaces; this form is indeed convenient to represent a change to two or more state spaces from within a larger context as, for example, is captured naturally by promotion. We propose that such operation schemas should be partitioned to isolate the effects on each state space, and each partition should be transformed into a suboperation of a corresponding class. The composite effect of the original operation schema is then achieved by replacing every invocation with a composition of object references to each of which a message is sent to invoke the corresponding suboperation; in this context, each suboperation is a necessary service of a class. We regard this transformation to be a fair trade-off: at the very least, we preserve the effect of the original schema and encapsulate the details of each suboperation; the responsibility for reusing the suboperation in new compositions, without violating the constraints of the corresponding classes, is delegated to a future specifier. This problem of finding the right fit by reusing existing features is still a current interest in object-oriented research.

The Re-analysis Phase The paradigm shift is a semi-automatic process because the initial transformation from Z to Object-Z can only be partially automated. The initial transformation should then be subjected to a re-analysis phase to:

- choose between the inheritance or aggregation relationship that is appropriate for a given situation;
- provide adequate encapsulation to ensure information hiding;
- determine an appropriate interface to each class;
- capture associations between classes through generic classes, or by parametric associations;
- identify and extract reusable features of each class;
- redesign generic properties through generic classes;
- exploit opportunities to use polymorphism.

This list is not exhaustive as the various opportunities to enhance object-orientation in a design, depend on the experiences of a specifier and the application domain. In perspective, the re-analysis phase can be regarded as a deferred analysis phase of an object-oriented development process which is started from scratch.

Encapsulation Object-Z is a hybrid object-oriented specification language because with the exception of the class construct, it relies heavily on the mathematical tool-kit of Z to provide the semantics for primitive types, cartesian products, sets, relations, functions, etc. MooZ, on the other hand, keeps to the object-oriented paradigm and provides a library of primitive classes, which eliminates the mathematical tool-kit of Z and provides a uniform notation. Lano[19] claims that the MooZ approach is truly object-oriented because the primitive classes capture the semantics of the mathematical tool-kit of Z, in an object-oriented style.

With the purely object-oriented approach of MooZ in mind, we will reexamine the types of the various objects that exist after the transformation. An object declaration may be introduced in one of three manners:

- as an attribute of a class;
- through an axiomatic description;
- as a parameter to an operation.

In each case, a constraint may be specified on the possible states that the object may assume; however, this constraint will conflict with our goal of encapsulating state to reduce coupling, if the constraint refers to the state of the object itself and the declaration is out of scope to the class of the object. To avoid this conflict, we proposed that the constraint may be transformed into an operation of a class; we assert the constraint on the object by requesting the object to select the operation; this request may be interpreted as a boolean operator which asserts the valid state(s) that an object may assume.

However, it is only possible to transform the constraint into a class, if the declaration introduces a true object. This assumption is valid in MooZ because every declaration in MooZ is an object reference. However, this is not always possible in Object-Z; if a declaration is of a type that belongs to the mathematical tool-kit of Z then that type is not a class and hence we cannot transform a constraint on that declaration to any class in particular. At best, we can provide some optimizations to encapsulate state, such as transforming the constraint on one parameter of a relation, into an operation of the class of that parameter; as an example, given an axiomatic description

$$\begin{array}{|l} \_ < \_ : Date \leftrightarrow Date \\ \hline \quad \ldots \end{array}$$

we would like to encapsulate the comparison which is performed on the state of two date objects, as an operation of the class *Date*. From the case study, it becomes evident that almost every operation that was introduced and which does not have a counterpart in the Z specification, represents a work-around to the hybrid nature of Object-Z in order to achieve encapsulation. Also, for the reason that the set type of Object-Z was chosen from the mathematical tool-kit, we were unable to transform the $\cup_1$, $\cap_1$, $\setminus_1$ axioms (class *SDS* in case study) on sets of durations, into redefinitions, in a *duration-set class*, of the set operations that are inherited from a *set-class*. One may argue that Object-Z does provide generic classes which we may tailor to replace the mathematical tool-kit; however, in order to encapsulate state, we would have to redefine the various operations on each construct in the mathematical tool-kit as an operation of a corresponding class, and this approach is unacceptable because we would end up defining the semantics of Object-Z in the Object-Z language itself.

Object-Identity and Self-Reference Object-Z lacks a self-referencing mechanism,[1] which makes it cumbersome to refer to the state of an object from within an operation of a class. The concept of object-identity is implicitly modeled in the semantics of Object-Z[2] and is the basis for distinguishing objects of the same class, which have identical state, and also for distinguishing objects of different classes. However, the designers of Object-Z have intentionally delayed providing a notation to refer to the identity of an object because it is liable to constrain the choices for interpreting concepts such as genericity, inheritance, and polymorphism.

We encounter the need of a notation for self-referencing in several parts of the case study; for instance, in the operation $\leq$ of the class *Date*, one date is asserted to be before another by invoking the $<$ operation, but it is not possible to equate the states of the two date objects from within the operation $\leq$ without equating their corresponding attributes.

[1] *this* in C++, or *self* in Smalltalk.

Chapter 9

Future Work

As an extension to this thesis, we propose that the methodology may be strengthened by providing a formal proof of the duality between the syntactic categories of Z, and the corresponding structures that we chose in Object-Z.

A second cause for concern is the semi-automatic nature of the transformation; while the actual transformation methodology may be automated, the success of the re-analysis phase relies on the skill and experience of an object-oriented designer. We suggest that it is beneficial to explore the methodology over multiple case studies in order to identify opportunities to automate, at least to some extent, the re-analysis phase. As an example, we point to a non-recursive free type definition whose enumerators are exclusively constructor functions; this type of definition may be conducive to be transformed into an inheritance hierarchy, instead of a class to represent the free type.

It is also desirable to compare an object-oriented solution that is analyzed and designed from scratch, to one that is obtained through the paradigm shift. On one hand, a paradigm shift seems attractive because an input procedural abstraction contains most of the data structures and operations that are required to implement a solution. On the other hand, it is not clear how much effort is required at the re-analysis phase.

Object-Z is a hybrid object-oriented specification language because it includes the semantics of the mathematical tool-kit of Z with object-oriented constructs such as class, object, inheritance, etc. This reliance on Z precludes the use of classes to represent the constructor types in Z which, in our experience, makes it difficult to propose a uniform transformation to certain constructs in a Z specification. For instance, a relation in Z may be transformed by using a generic relation class where the generic parameters are instantiated by the corresponding parameters of the relation. In turn, a constraint or operation on the relation may be transformed into the state predicate or operation, respectively, of the relation class. This approach will eliminate some of the work-around solutions that we developed in order to achieve encapsulation. What is clearly desirable is to evaluate the paradigm shift in a purely object-oriented specification language such as MooZ, to determine whether the methodology can be automated.

Appendix A

The Re-analysis Phase

In this section, we explain a re-analysis process to enhance the object-orientedness of the transformed procedural specification. Note that this phase is indeed subjective because it involves design decisions and knowledge of object-orientation therefore, it is practical to only suggest possibilities to improve an object specification which is derived from the transformation. In the following sections, we will revisit the object-oriented shared diary system to explain our reasons for redesigning the specification, and the gains that we anticipate from the effort. These ideas have been applied to the case study, and the object specification represents an enhancement over the result of applying the paradigm shift to the procedural specification. First, we will briefly examine what object orientation involves from a research standpoint.

There is still considerable debate over a precise definition of what exactly constitutes object-orientation. For instance, Peter Wegner [34] considers object-orientation to include objects, classes, inheritance, and encapsulation. Stefik and Bobrow[31] regard an object as an entity that performs computation and saves state; they also consider message-passing and inheritance as fundamental concepts of object-orientation. Snyder[28] argues that objects do not necessarily have state; an object is any entity that provides services to clients where a client could be a person or a program, and a service is any activity that is performed at the client's request. Snyder considers the key concepts of object-orientation to be: data abstraction, encapsulation, object-identity, polymorphism, and implementation inheritance.

We will adopt the path of widest acceptance and consider inheritance, encapsulation, and polymorphism, to be fundamental to object-orientation. In the following sections, we will examine how the object specification embodies these concepts and then explore possibilities to correct any inadequacies. We conclude this discussion by illustrating opportunities to prepare the specification for reuse.

Inheritance

Inheritance is a specialization mechanism which promotes reuse. There are various forms of inheritance such as implementation inheritance, inheritance by restriction or variance, etc.[7]. In this section, we focus on *implementation inheritance*[28], which supports the incremental construction of an object's implementation by extending or refining other object implementations. Implementation inheritance is generally accepted without reservation because it provides the same type of size and maintenance benefits of sharing a full implementation. Indeed, a full implementation will likely reduce the opportunity for reuse because the needs of the application may not match exactly what the implementation offers. Therefore, inheritance makes reuse possible when requirements are similar, but not identical. At the specification level, implementation inheritance may simplify proofs because, for instance, properties proved for a superclass object are true for its subclass objects as well.

Interchanging Inheritance and Aggregation

In Z, two schemas may be related either by schema inclusion, or through a variable of one schema which is of another schema type. Over the transformation, the former relationship is equated to inheritance, and the latter to aggregation. However, these assumptions may not always be acceptable; a Z specification is usually written in a procedural style and it is likely that any semblance to inheritance or aggregation in the specification, is accidental. In the following discussions, we examine one such situation in the case study, and we propose a possible solution to the problem. Note that distinguishing between inheritance and aggregation is a subjective matter that relies on the needs of an application, and the experience of the specifier. However, we may eliminate some of the uncertainty by using a principle called *op-inheritance* given in [1].

Op-Inheritance

A class B is said to op-inherit a class A, if for every operation OpA_i in A, there is a corresponding operation OpB_i in B, such that:

- the precondition of OpA_i implies the precondition of OpB_i; and
- the postcondition of OpB_i implies the postcondition of OpA_i.

The first rule requires that the precondition of OpA_i should be stronger than the precondition of OpB_i; correspondingly, the operation OpB_i can commence in every situation that OpA_i can commence. The second rule guarantees that OpA_i will terminate in every situation that OpB_i terminates in.

We can only suggest that op-inheritance is indicative of an inheritance relationship between two classes[1]. However, the uncertainty is further limited: the association between two schemas is restricted to schema inclusion and schema aggregation; therefore, the only corresponding associations between two classes that exist after the transformation have to be either inheritance or aggregation.

We have cited instances in the case study, for example *SDSUsers* in *SDSDiary*, where inheritance and aggregation are inadequately identified.

Encapsulation

Encapsulation is another key concept of object-orientation, that facilitates the grouping of structure and behavior into a single object. It also facilitates information hiding by providing an interface and an implementation; part of the latter is invisible outside the object. The interface is, ideally, a collection of operations that together define the behavior of the class[7]. The invisible part contains the data variables that are used to maintain the internal state of an object, and a collection of auxiliary operations which are hidden from clients and are typically invoked by a service to perform, for instance, an internal computation on state.

In Object-Z, encapsulation is facilitated through a *visibility list*. An empty visibility

[1]Classification based on interface conformance need not produce a strict hierarchy [28].

list designates every feature of the class as visible. The features that are included in the visibility list form the interface to the class, and all other features are hidden from clients of the class.

Enhancing the Object-Specification with Encapsulation

The transformation produces classes with features that are entirely visible; this is because there is no encapsulation or information hiding in Z specifications. In addition, the Z specification may have, for instance, a constraint in a schema which relates a variable of the schema with a variable of another schema; over the transformation, the constraint relates the attributes of the corresponding classes, which may violate encapsulation. In Snyder's[28] opinion, a client should not directly access or manipulate the data associated with an object. We extend this view to include auxiliary operations; an auxiliary operation is typically composed into the context of an interface operation, and is hidden from clients. As an example of an auxiliary operation, we consider an operation *DaysInMonth* of the class *Date* of the case study, which returns the number of days for a given month; this operation exists only to maintain the integrity of any date object, and is not an essential service to its clients.

We propose two requirements to introduce encapsulation into the various classes that are derived from the transformation:

- the attributes and auxiliary operations of every class should be hidden;
- the interface to a class is a set of operations which are invoked by clients; each operation is either:

- the image of an operation schema which acts on the pre-image of the class; or
- introduced as a new operation to the class. We propose that such an operation should be created to replace a constraint which is asserted outside the scope of the class, but which constrains the value of an attribute of the class.

The latter case may be created in the form of:

1. operations which assert a constraint between an attribute and a constant;
2. operations which retrieve an attribute;
3. operations which assert a constraint between an attribute and a variable, where the value of the variable is drawn from the scope of another class.

As an example for each category, we refer to the following operations in the case study:

- in category 1, the operation *Is0* of the class *Time*, which tests if a time object corresponds to the start of day - a constant value;
- in category 2, the operation *GetSecy* in the class *SDSUsers*, to retrieve the secretary object;
- in category 3, the operation *Contains* in the class *Entry*, to test if an annotation is contained in the entry.

Next, we present a formal account of how to identify and handle the three categories of operations. Note that we generalize the solutions to categories 2 and 3 because in both cases the attribute is constrained by a variable. In each case, the effect of encapsulating a constraint on an attribute, through an operation, introduces the overhead of a new operation, increases specification size, and may require the specifier to prove that the original constraint is respected.

Replacing Constraints with Operations. We will assume these notations in the following discussion:

- *att* is an attribute;
- *or* is an object reference;
- *Op* is an operation;
- *C* is a class;
- or.att is an access to *att* through *or*.

When required, we will use subscripts to distinguish between different occurrences of a notation.

Constrained by a Constant Assume that att_1 of C_1, which is accessed through or_1 in C_2, is constrained by a constant k; we interpret this constraint to mean an association between the classes C_1 and C_2. Our goal is to eliminate the direct access from att_1 through or_1, from the scope of C_2, and we achieve this as follows:

- transform k into C_1 if it is not already in scope;
- create an operation Op_1 in C_1;

- transform the constraint that relates att_1 to k, into the predicate of Op_1;
- replace the original constraint with a request to or_1 to select Op_1;
- if k was used only to constrain att_1 in C_2, then k is no longer required in C_2 and may be eliminated from C_2.

Note that Op_1 is a boolean operation which respects the original constraint in every situation.

Constrained by a Variable Assume that att_1 of C_1, which is accessed through or_1 in C_2, is constrained by a variable v_2 of C_2. We interpret the constraint to mean an association between C_1 and C_2. Of the two associated classes, we distinguish one as a *source* if it plays an active role in the association, and the other as a *sink* class for its passive role[2]. Here, our goal is to eliminate the direct reference to att_1 through or_1, from the scope of C_2.

If the class C_1 is a source class, then we create an operation Op_1 in C_1, with a given input of the type v_2. The original constraint between att_1 and v_2 is transformed into the predicate of Op_1. We may now replace the original constraint in C_2 by sending a request to or_1 to select Op_1 with an actual parameter v_2.

On the other hand, if we distinguish C_2 as the source class, we create an operation Op_1 in C_1 to retrieve the value of att_1. We now replace the original constraint with a request to or_1 to select Op_1 and therefore to retrieve att_1. Note that if it benefits

[2]For instance, *Company pays Employee*, where *Company* (source) plays an active role, and *Employee* (sink) is a passive entity; this association between *Company* and *Employee* is modeled through the operation *pay* introduced into the class *Company*.

the design, we may even capture the constraint in an operation Op_2 to emphasize the nature of the association. If we decide on Op_2, the formal parameter to Op_2 is an object-reference to C_1. The original constraint is transformed to the predicate of Op_2; we may now replace the original constraint by invoking Op_2 with or_1 as an actual parameter. The predicate of Op_2 sends a request to or_1 to select Op_1, which returns the value of att_1 to be constrained by v_2 in Op_2.

The request $or_2.Op_2$ assumes the role of a boolean operator: the predicate of Op_2 will respect the original constraint in every situation.

Indirect Associations In the preceding discussion, we assumed that an attribute was accessed directly through an object reference in an aggregate class. However, it is also necessary to consider those situations where an attribute may only be accessed through two or more object references, such as $or_1.or_i.att_i$ where or_i stands for an intermediate set of object references ($i \geq 2$). For instance, consider an ordering of classes which is transformed from a Z specification: a class C_1 inherits a class C_2, and C_2 contains an object reference or_3 to a class C_3, but over the re-analysis phase we decide that C_1 is ideally an aggregate class that contains a reference or_2 to an instance of C_2. This requires that an attribute att_3 of C_3 may only be accessed in C_1 through an expression $or_2.or_3.att_3$. Note that this access to att_3 is indirect because it traverses two object references. There are two issues to be considered:

- we cannot remodel the access directly from C_1 to C_3 (and therefore choose to ignore the indirection)[3];
- we need to encapsulate the attribute att_3 and instead to provide a service in C_3 to access the value of att_3;

To solve these issues, we refer to the concept of *delegation* as proposed by Rumbaugh[24]: "delegation consists of catching an operation on one object, and sending it to another object that is part of or related to the first object". With this definition in mind, we can adopt the methodology of the preceding section to rework the constraint into an operation which hides the access to the attribute. We then introduce an operation into the class of each intermediate object reference, which simply delegates the message to the next class in sequence. The message-chain originates in the class which hosts the original constraint, and terminates in the class which hides the attribute. As an example, we refer to the operation *Contains* in the class *Page* of the case study: a page contains an entry if the entry is in the list of page entries. The encapsulation is two-fold:

- the attribute *ListOfPageEntries* is hidden from clients;
- the operation *Contains* hides the implementation of the attribute, in this case, as a sequence.

A Sample Specification

We present a schema called *HoursElapsed* that models the duration in hours, between two events

[3]This follows the philosophy of Rumbaugh[24] that an indirect access should not be reworked through a transition.

HoursElapsed

$t : 0 \mathrel{..} 23$

and a schema called *TimeSlices* which is a sequence of durations, presumably in the order in which they occurred

TimeSlices

$tmslcs : \operatorname{seq} HoursElapsed$

and a third schema called *Monitor* which maintains an instance of the ordered collection of the durations, and a record of the longest duration

Monitor

$gaps : TimeSlices$
$biggap : HoursElapsed$

$biggap \in \operatorname{ran} gaps.tmslcs$
$\forall he : HoursElapsed \mid he \in \operatorname{ran} gaps.tmslcs \bullet$
$\quad biggap.t \geq he.t$

When these schemas are transformed into Object-Z we realize three classes where the schema variables are transformed into state variables, and the predicate that constrains each schema is transformed into an invariant of the corresponding class.

HoursElapsed

$t : 0 \mathrel{..} 23$

TimeSlices

$tmslcs : \operatorname{seq} HoursElapsed$

Monitor

$$\begin{array}{l}
gaps : TimeSlices \\
biggap : HoursElapsed \\
\hline
biggap \in \operatorname{ran} gaps.tmslcs \\
\forall he : HoursElapsed \mid he \in \operatorname{ran} gaps.tmslcs \bullet \\
\quad biggap.t \geq he.t
\end{array}$$

However, there are a few issues to be reconsidered:

- the Z specification exposes the internal representation of an abstract data type. In this example, the attributes *t* and *tmslcs* of *HoursElapsed* and *TimeSlices*, respectively, are directly referenced in *Monitor*;
- a change to the representation of the abstract data type is likely to propagate to those parts of the specification that refer to it. This may necessitate a good deal of reworking and correspondingly, developing proofs all over again;
- the Z specification leaves the relationships between the various types as implicit. In our opinion, associations between types should be exposed when appropriate in order to enhance the descriptiveness of specifications.

In response to these problems, we rework the object-specification. We expect to reduce the coupling between classes by designing an interface to each class, that is sufficient to fulfill the constraints that the class participates in, in the specification.

First, consider the enhanced description of the class *HoursElapsed*.

HoursElapsed

$\upharpoonright (\geq)$

$t : 0 \mathrel{..} 23$

$\geq$

$he? : HoursElapsed$

$t \geq he?.t$

We have introduced an operation $(\geq)$ to compare two durations. We propose that this operation deserves a role in this class because it operates on the internal state of instances of this class. The interface to this class is defined as the operation $(\geq)$, in its visibility list; consequently, the attribute t of the class is hidden from access by clients of the class.

Similarly, we change the class implementation of *TimeSlices*, by hiding its internal representation, and by introducing an interface.

TimeSlices

$\upharpoonright (Contains)$

$tmslcs : \mathrm{seq}\, HoursElapsed$

Contains

$he? : HoursElapsed$

$he? \in \mathrm{ran}\, tmslcs$

The method *Contains* verifies that a duration is indeed a member of a set of durations.

We now apply the effects of encapsulating both classes, within the class *Monitor*.

Monitor

$$\begin{array}{l} gaps : TimeSlices \\ biggap : HoursElapsed \\ \hline gaps.Contains[biggap/he?] \\ \forall he : HoursElapsed \mid gaps.Contains[he/he?] \bullet \\ \quad biggap. \geq [he/he?] \end{array}$$

We note the changes as follows:

- the class *Monitor* no longer directly accesses the attributes of classes *HoursElapsed* and *TimeSlices*, instead, the constraints are replaced by requests to the two classes;
- the method *Contains* plays a natural role: the class *TimeSlices* will likely be refined into a *container class* [24], and the method documents the *membership* association between an entry and a set of entries.
- we realize the benefit of encapsulation: as long as the signatures of the two operations ($\geq$, *Contains*) remains unchanged, we may change the internal representation of the two classes without affect their client - *Monitor*; for instance, we may reimplement the sequence of time slices as a set without affecting *Monitor*.

Polymorphism

Polymorphism is the ability for an entity to take on different forms in different contexts. In object-orientation, polymorphism is applied to entities such as objects and operations[7]. A *polymorphic object* is an entity such as a variable or the argument to an operation, that can hold values of differing types. *Polymorphic operations* are operations that have polymorphic arguments.

In object-orientation, polymorphism occurs as a natural result of implementation-inheritance and the mechanisms of message-passing. One of the strengths of object-orientation is that these devices can be combined in a variety of ways, yielding a number of techniques for sharing and reusing specifications.

Typically, any expression involving polymorphism should be applicable to all possible objects to which the expression may be assigned. This responsibility can be reduced if a given class is *signature-compatible* with the classes it inherits [26]:

- a class must have all the operations it inherits; and
- the signature of a redefined operation must match exactly the signature of the inherited operation.

With this background, we now turn to an example, from the case study, which illustrates how we apply polymorphism and achieve reuse of specification.

Enhancing the Object-Specification with Polymorphism

Consider a type called *Duration* which is used to record the activities in a diary.

Duration

$StartTime : Time$
$EndTime : Time$
$StartDate : Date$

$StartTime < EndTime$

The schema *Duration* records the time when an event begins and ends, and on which day it occurs.[4] We then define an abbreviated definition of *Duration*, called *DurFrag*

$$DurFrag == Duration$$

and restrict its carrier set to only those durations that last for one hour

$$\forall\, df : DurFrag \bullet df.EndTime = succ\ df.StartTime$$

As part of the requirements of the specification, we would like to compare two durations to determine which one occurs later. So, we define a relationship $>$ between durations

$$\begin{array}{|l}
\_ > \_ : Duration \leftrightarrow Duration \\
\hline
\forall\, d_1, d_2 : Duration \bullet d_1 > d_2 \Leftrightarrow \\
\qquad\qquad \ldots
\end{array}$$

which holds *true* if the duration d_1 corresponds to a time and date that begins after the duration d_2 begins.

We note that by the semantics for abbreviation definitions, the type of the abbreviation is identical to the type of the expression on its right-hand side. Consequently, any operation that is permissible for a *Duration* should also apply to a *DurFrag*, as long as the integrity constraints of the type *DurFrag* are preserved. Indeed, the axiom $>$ can be regarded as polymorphic because we can use it to compare any combination of *Duration* and *DurFrag*. An immediate benefit that we realize is that we do not have to duplicate the specifications to include the other three

[4]For brevity, we assume the types *Time* and *Date*.

combinations.[5] We would like to carry this benefit through to the Object-Z specification, because in addition to a reduced specification, we may also benefit from reduced proofs.

When the Z specification is transformed, we obtain a class in place of the schema type *Duration*

```
┌─ Duration ──────────────────────────────────
│ ┌──────────────────────────────────────────
│ │ StartTime : Time
│ │ EndTime : Time
│ │ StartDate : Date
│ ├─────────
│ │ StartTime < EndTime
│ └──────────────────────────────────────────
│
│ ┌─ > ──────────────────────────────────────
│ │ dr? :↓ Duration
│ ├─────────
│ │         ...
│ └──────────────────────────────────────────
└─────────────────────────────────────────────
```

where the axiom $>$ has been transformed into an operation of the class. The input parameter to this operation is a polymorphic object (indicated by $\downarrow$) that can assume the type of *Duration* or any of its subclasses, here *DurFrag*.

We transform *DurFrag* into a subclass of *Duration*

```
┌─ DurFrag ───────────────────────────────────
│ Duration
│ ┌──────────────────────────────────────────
│ │ EndTime = succ StartTime
│ └──────────────────────────────────────────
└─────────────────────────────────────────────
```

We may now invoke the operation $>$ through a message to an object reference to one of the two classes, and pass in an object-reference to either class, to compare

[5] *Duration* and *DurFrag*, *DurFrag* and *Duration*, and *DurFrag* and *DurFrag*.

with. The operation is polymorphic, and we are assured of valid behavior with any combination of the two types.

Reuse

Reusable specifications reduce the effort in developing proofs which consumes a considerable portion of a specifier's time. Reducing the amount of specification also simplifies understanding, which increases the likelihood that the specification is correct. Object-oriented specification languages greatly enhance the possibility of specification reuse[19].

We identify two kinds of reuse in specifications: reuse within a new specification, and reuse of existing specifications. As examples of reuse within specifications, we have already encountered:

- reuse through inheritance;
- reuse through polymorphism, namely, with generic classes, generic constants, and generic operations;
- transforming operations in Z into suboperations in Object-Z and reusing the suboperations through different compositions;
- reusing a single constant definition in a superclass, in place of identical definitions in all its subclasses; etc.

Reuse of existing specifications can be facilitated through carefully designed libraries of specifications. Ideal candidates for such libraries are specifications of container classes[24], such as lists, maps, trees, etc., which describe abstract properties and may be applied to vastly simplify a specification.[6]

In the following section, we present an example from the case study, for reuse within a specification. We will conclude our views on reuse by examining opportunities to rework the classes for use in future development.

An Example for Reuse Within a Specification

In the Z specification, we describe the entries in a page of a diary, through an injective sequence; the ordering is established on the basis of slot time, where the slot time of one entry is an hour behind that of its successor. We use an analogous definition for the pages in a diary; pages are ordered on the basis of slot dates, where the slot date of a page is a day behind the slot date of its successor. When the two abbreviated definitions of *EntrySeq* and *PageSeq*, are transformed, the corresponding classes appear to share nearly identical properties.

EntrySeq

$$es : iseq(Entry)$$

$$\forall i : 1 \mathrel{..} \#es - 1 \bullet (es(i+1)).SlotTime \ \mathsf{oneTless}(es(i)).SlotTime$$

PageSeq

$$ps : iseq(Page)$$

$$\forall i : 1 \mathrel{..} \#ps - 1 \bullet (ps(i+1)).SlotDate \ \mathsf{oneDless}(ps(i)).SlotDate$$

[6]We note however, that this form of reuse would be better suited to refinement when the specification is elaborated, and yet is abstract.

The structural properties of the two classes are identical in format, and it is reasonable to expect that the two classes will also share some behavior. Correspondingly, we may abstract the properties in common into a generic class called *ItemSeq*

ItemSeq[X]

$is : iseq(X)$

Head

$\cdots$

Contains

$\cdots$

NumItems

$\cdots$

Succ

$\cdots$

The classes *EntrySeq* and *PageSeq* may be redesigned to inherit and instantiate the class *ItemSeq* with the appropriate types, without losing domain information.

EntrySeq

$ItemSeq[Entry][es/is, \cdots]$

PageSeq

$ItemSeq[Page][ps/is, \cdots]$

Notice that the new definitions no longer carry the ordering information as a class invariant. It seems appropriate to define the basis for ordering two entries as an operation in the class *Entry*; we accomplish this constraint through the operation oneTless of the class *Entry* which appropriately delegates the responsibility for comparing the slot times of two entries, to the class *Time*. It also seems appropriate to expect the class which uses the sequence of entries to define the basis for ordering; we capture this constraint in the class *Page*. The classes *Page* and *Diary* are transformed, analogously, to capture the constraints on a sequence of pages.

The redesign facilitates the following situations for reuse within the specification:

- reuse of the generic class *ItemSeq*;
- reuse of the generic operations in *ItemSeq*, namely *Head*, *Contains* etc., which would otherwise be duplicated in the classes *EntrySeq* and *PageSeq*;
- the potential for reusing *ItemSeq*, for instance, when the specification is refined and new classes with matching properties are introduced.

Preparing Classes for Reuse

One of the goals of object-orientation is to prepare reusable designs which may help to reduce costs of maintenance and future development. However, the task of distinguishing reusable features of a class is indeed subjective, and depends on the application. Also, it is difficult to pinpoint a class as being perfectly reusable, because it is unlikely that a class will match, exactly, the needs of another application. However, one can generalize a given specification for possible reuse in a later application; this effort is important because it is unlikely that reuse is emphasized in the procedural paradigm. Consequently, the paradigm shift results in an object specification which, ideally, should be restructured for reusability. Given the subjectivity, it is indeed impractical to propose a formal treatment of identifying reusability; instead, we examine the case study, through an example, for the potential for preparing reusable classes. Note that we did not modify the object design for reusability, in order to conserve the effects of the paradigm shift.

An Example for Preparing a Reusable Class

The class *Duration*, in the case study, is actually one possible specialization of an abstract concept. In the case study, a duration records the actual times and date of an activity, whereas another application may require a duration to imply the intervals between two activities, perhaps in microseconds.

Assuming the potential for reuse, we may reconsider a duration to mark two instances of time, where an instant is an abstract concept that may be measured to a finer granularity, perhaps microseconds, or to a broader definition such as the time and date concepts we use in the case study. We may define an instant as an abstract class

```
┌─ INSTANT ──────────────────────────────
│ ┌──────────────────────────────────
│ └──────────────────────────────────
│
│ ┌─ < ──────────────────────────────
│ │ int :↓ INSTANT
│ └──────────────────────────────────
└────────────────────────────────────────
```

and include the signatures of those operations that exist in the object specification and which are suggestive of reuse. Here, we have included only a comparison operator (<) to illustrate our point, but there exist other operators in the case study, which may also be applicable to other implementations. The abstract operation (<) requires that every specialization has to be strictly signature compatible (see section A).

We may record an instant as occurring at a time and on a date, where both time and date are also abstract concepts with abstract operations that are inspired by the case study, and which are suggestive of reuse

$TIME$

$<$

$tm? : \downarrow TIME$

$DATE$

$<$

$dt? : \downarrow DATE$

$INSTANT_1$

$INSTANT$

$tm : \downarrow TIME$
$dt : \downarrow DATE$

The abstract concepts of time and date may be tailored into the representation that we have chosen for the case study.

$Time$

$TIME$

$t : \mathbb{N}$

$0 \leq t \leq 23$

$<$

$tm? : Time$

$t < tm?.t$

$Date$

$DATE$

$dt : ((day \times month) \times year)$

$\ldots$

$<$

$dt? : Date$

$dt < dt?.dt$

By the semantics for inheritance, the state of each of *Time* and *Date* is merged with the abstract state of *TIME* and *DATE* respectively; the matching operations are also merged: polymorphic reference in the signature of the superclass operation is substituted for an instance of the subclass.

The definitions of *Time* and *Date* may be matched with our requirement for an instant

$INSTANT_2$

$INSTANT_1$

$tm_1 : Time$
$dt_1 : Date$

$<$

$ins_2 : INSTANT_2$

$$dt_1 = ins_2.dt_1 \land tm_1.<[ins_2.tm_1/tm?]$$
$$\lor\ dt_1.<[ins_2.dt_1/dt?]$$

$INSTANT_2$ inherits the relatively abstract definition of $INSTANT_1$, and the attributes of $INSTANT_2$ override the corresponding polymorphic references in $INSTANT_1$ with the more concrete definitions of *Time* and *Date*. The abstract operation ($<$)

of *INSTANT* is tailored to compare two specialized instants. We may now use this definition to specify a duration, which is actually two instants

2INSTANTS

$$\begin{array}{|l}
start : INSTANT_2 \\
end : INSTANT_2 \\
\hline
start. < [end/ins_2]
\end{array}$$

which are ordered by requiring that one instant begin before another. In addition, we can capture the requirement of the case study that every duration should fall on a specific date, but may span multiple hours

Duration

$$\begin{array}{|l}
2INSTANTS \\
\hline
start.dt_1 = end.dt_1
\end{array}$$

In this particular example, we can justify the overhead of introducing 6 new classes because the design is conducive to reuse by other applications. The potential benefits of this exercise are:

- *lowered maintenance*: as long as the signature of an interface operation remains unchanged, the operation may be respecified to correct an error or for optimizations, without affecting clients of the class;
- *lowered future development costs*: a future developer may take advantage of the analysis effort by reusing the classes as they are, or by using a class definition as a template to be specialized to a particular application.

The example also conveys an important distinction between the procedural and object-oriented approaches: the procedural approach may capture requirements without much concern for future development; in contrast, object-orientation motivates a developer to prepare a design for reuse through facilities such as inheritance, message-sending, polymorphism, etc.

Bibliography

[1] V.S. Alagar and K. Periyasamy, "A Methodology to Transform Functional Specification into Object-Oriented Design", *Software Engineering Journal(UK)*, Vol 7, No. 4, July 1992, pp. 247–263.

[2] A. Griffiths and G. Rose, "A Semantic Foundation for Object Identity in Formal Specification", *Technical Report, TR-94-21*, Software Verification Research Center, University of Queensland, Australia, September 1994.

[3] S.C. Bailin, "An Object-Oriented Requirements Specification Method", *Communications of the ACM*, Vol 32, No. 5, May 1989, pp. 608–623.

[4] J. Bowen, P. Breuer and K. Lano, "A Compendium of Formal Techniques for Software Maintenance", *Software Engineering Journal*, Vol 8, No. 5, September 1993, pp. 253–262.

[5] J.P. Bowen and M.G. Hinchey, "Seven More Myths of Formal Methods", *Technical Report, PRG-TR-7-94*, Programming Research Group, Oxford University Computing Laboratory, 1994.

[6] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin Cummings Publishing Company, 1994.

[7] T. Budd, *Introduction to Object-Oriented Programming*, Addison-Wesley Publishing Company, 1991.

[8] J. Dawes, *The VDM-SL Reference Manual*, Pitman, 1990.

[9] W. Dietrich, L. Nackman and F. Gracer, "Saving a Legacy with Objects", *Proceedings of Object-Oriented programming Systems, Languages, and Applications*, 1989, pp. 77–88.

[10] D.J. Duke, *Object-Oriented Formal Specification*, Ph.D Thesis, Department of Computer Science, University of Queensland, Australia, 1992.

[11] R. Duke, P. King, G. Rose and G. Smith, "The Object-Z Specification Language: Version 1", *Technical Report TR-91-1*, Software Verification Research Centre, University of Queensland, Australia, 1991.

[12] H. Gall and R. Klosch, "Finding Objects in Procedural Programs: An Alternative Approach", *Working Conference on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

[13] J.A. Hall, "Using Z as a Specification Calculus for Object-Oriented Systems", *in Bjorner et.al.*, 1990, pp. 290–318.

[14] W.E. Howden and S. Pak, "Problem Domain, Structural, and Logical Abstractions in Reverse-Engineering", *IEEE Conference on Software Maintenance*, Orlando, Florida, November 1992, pp. 214–224.

[15] INMOS Limited, *Specification of Instruction Set / Specification of Floating Point Unit Instructions*, Prentice Hall, 1988.

[16] I. Jacobson and F. Lindstrom, "Reengineering of Old Systems to an Object-Oriented Architecture", *Proceedings of Object-Oriented programming Systems, Languages, and Applications*, 1991, pp. 340–350

[17] V.S. Alagar and K. Periyasamy, "Specification of Software Systems", Lecture Notes for the course *Software Specification Systems*, Department of Computer Science, Concordia University, Montreal, Canada, 1991-1995.

[18] P. King, "Printing Z and Object-Z LaTeXdocuments", Department of Computer Science, University of Queensland, Australia, 1990.

[19] K. Lano and H. Houghton, *Object-Oriented Specification Case Studies*, Addison-Wesley Publishing Company, 1993.

[20] K. Lano and H. Houghton, *Reverse Engineering and Software Maintenance*, McGraw-Hill International (UK) Ltd., 1994.

[21] *Basic Reference Model of Open Distributed Processing, Part 4, Architectural Semantics*, Committee Draft, ISO/IEC 10746-4, July 1995.

[22] P. Newcomb, "Reengineering Procedural into Object-Oriented Systems", *Working Conference on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

[23] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*, Prentice Hall International Series in Computer Science, 1991.

[24] J. Rumbaugh et al, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[25] B.H. Sellers and J.M. Edwards, "Object-Oriented Systems Life Cycle", *Communications of the ACM*, Vol 33, No. 9, September 1990, pp. 142–159.

[26] G.P. Smith, *An Object-Oriented Approach to Formal Specification*, Ph.D Thesis, Department of Computer Science, University of Queensland, Australia, 1992.

[27] H.M. Sneed and E. Nayary, "Extracting Object-Oriented Specifications from Procedurally Oriented Programs", *Working Conference on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press.

[28] A. Snyder, "The Essence of Objects: Concepts and Terms", *IEEE Software*, January 1993, pp. 31–42.

[29] J.M. Spivey, *The Z Notation: A Reference Manual* (Second Edition), Prentice-Hall International Series in Computer Science, 1992.

[30] S. Stepney, R. Barden and D. Cooper, *Object-Orientation in Z*, Workshops in Computing Series, Springer-Verlag, 1992.

[31] M. Stefik and D. Bobrow, "Object-oriented Programming: Themes and Variations", *AI Magazine*, Winter 1986, pp. 40–62.

[32] S.R. Tilley, H.A. Muller, M.J. Whitney and K. Wong, "Domain-Retargetable Reverse Engineering", *Proceedings of Conference on Software Maintenance*, September 1993.

[33] P.T. Ward, "How to Integrate Object-Orientation with Structured Analysis and Design", *IEEE Software*, March 1989, pp. 74–82.

[34] P. Wegner, "Learning the Language", *Byte*, March 1989, pp. 245–253.

[35] A.S. Yeh, D.R. Harris and H.B. Rubenstein, "Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language", *Working*

Conference on Reverse Engineering, Toronto, July 1995, IEEE Computer Society Press.

[36] *Z Notation, Version 1.2*, Committee Draft, ISO/JTC1/SC22/WG19, September 1995.