

A Behavioural Study of a Graphical User Interface for a CASE Tool

Dennis Lovie

A Thesis in
The Department of Computer Science

Presented in Partial Fulfilment of the Requirements
for the degree of Master of Science at
University of Manitoba
Winnipeg, Manitoba, Canada

December 1994

© Dennis Lovie, 1994



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION.

ISBN 0-315-99131-3

Canada

Name A Behavioral Study of a Graphical User Interface for a CASE Tool
 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

0984

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
 Paleocology 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



A BEHAVIOURAL STUDY OF A GRAPHICAL USER
INTERFACE FOR A CASE TOOL

BY

DENNIS LOVIE

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

© 1995

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA
to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to
microfilm this thesis and to lend or sell copies of the film, and LIBRARY
MICROFILMS to publish an abstract of this thesis.

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

Abstract

Computer-Aided Software Engineering (CASE) refers to the automation of various stages of the software development process. Consequently, CASE tools provide functionality to its users through sophisticated user interfaces such as a graphical user interface (GUI). A GUI for a CASE tool should integrate all components of the tool and enable the user to manipulate its contents in a simple and easy to user manner. This requires a behavioural study of GUIs for such CASE tools. In this thesis, an abstract GUI is developed for a CASE tool; the tool is intended for developing an object-oriented design of an application. The Z formal notation is used for this purpose. A modular approach is taken such that the primitive components of the GUI are specified first and then the GUI itself is specified by aggregating the primitive specifications. The development of specifications for a simple and a specialized diagrammatic tool using the GUI specifications is also illustrated. Finally, refinement of the specifications towards a prototype is discussed.

Keywords: CASE tool, GUI, formal specifications, Z, behavioural study.

Acknowledgements

There are a number of people without whom this work would not have been completed. First, thanks to many friends, colleagues and family who have expressed interest, feigned or otherwise, in my graduate work. Financial assistance and flexible working hours have been provided by Online Business Systems and The Great-West Life Assurance Company, and for this I am grateful. I am deeply indebted to my supervisor, Dr. Kasi Periyasamy, who has proven to be a pillar of patience, even when it meant sacrificing his evenings to meet with me. Finally, my most heartfelt thanks goes to my partner Nancy. She has put up with many years of my schooling, suffered through my frustrations and patiently accepted my lack of free time. Thank you for everything.

Contents

1	Introduction	1
2	The Z Specification Language	5
2.1	Z Type System	5
2.2	Schemas	6
2.3	Functions	8
2.4	Object-Oriented Design in Z	8
3	Pixels and Coordinate Frames	9
3.1	Pixels	9
3.1.1	Comparison of Pixels	11
3.1.2	Operations on Pixels	14
3.2	Regions Containing Pixels	15
3.2.1	Operations on Simple Regions	18
3.3	Virtual Screen Pixels and Regions	28
3.4	Summary	32
4	The Mouse and Its Behaviour	34
4.1	Time	34
4.2	The Mouse	36
5	Controls	43
5.1	Representational Abstraction of Controls	44
5.1.1	Primitive Controls	44
5.1.2	Composite Controls	45
5.1.3	Scrollable Region Control	50

5.1.4	Menus	59
5.1.5	Summary of Control Types	62
5.2	Visual Representation of Controls	63
5.2.1	Summary of Visual Characteristics	70
5.3	Behavioural Abstraction of Controls	71
5.4	Summary	86
6	Windows	87
6.1	The Window	87
6.2	Visual Characteristics of Windows	88
6.3	Operations On A Window	94
7	The Display	101
7.1	Structure	101
7.2	Operations on a Display	102
8	A Simple Diagramming Tool	107
8.1	Diagram Objects and a Simple Diagram	108
8.2	Semantics of a Simple Diagram	110
8.2.1	Diagram Object Pixel Mappings	110
8.2.2	Relationships Between Diagram Objects	114
8.3	Diagram Object Operations	116
8.4	Diagram Operations	121
8.4.1	A Simple Diagramming Tool	127
8.4.2	Operations of a Diagramming Tool	128
8.5	Summary	137
9	A Specialized Diagramming Tool	138
10	Refinement	141
10.1	Design and Implementation From Formal Specifications	141
10.2	Simple Prototype of a GUI	142
11	Conclusion	144

A Z Notations	148
B Preliminary Functions and Data Types	152

Chapter 1

Introduction

Computer-Aided Software Engineering (CASE) tools were introduced into the software development process to automate various development stages, thus improving the quality and reducing the development cost of software. CASE tools use well defined methodologies and standards and so many errors in system design can be identified early in the development process. These errors are typically difficult and expensive to correct after the design and implementation phases. Ideally, a CASE tool should provide integration between software development life cycle phases [3]. This can be accomplished only if the tool has a variety of functions, each corresponding to a different but complementary task in the system development process, centered around shared data, a common user interface, and the tool's knowledge of the software development process [15]. Some CASE tools also automate the creation of some or all of the system components, such as source code or database objects, based on the software engineer's design.

Previous generations of CASE tools were usually expensive, mainframe-based, and had a character-based interface. They used a command language for user interaction. Integration between components of such CASE tools was often loose or incomplete. With the advent of graphics-based work stations, many CASE tools in use today take advantage of a graphical user interface (GUI) to pictorially convey design information between the tool and the designer through the use of various diagrammatic techniques. Many diagrammatic notations have been designed for use in the software development process, including those to model data flows, data, and program structure [5, 6, 18] and more recently to model object-oriented design [1, 3].

Some diagrammatic techniques are complementary in that one type of diagram can be created by transforming a diagram of another type [13]. Designers can freely communicate with these CASE tools using diagrams. This ease of use allows designers to concentrate more on the application domain rather than having to remember the cryptic commands prevalent in text-based CASE tools.

This thesis is a contribution to the development of a GUI for a CASE tool; the CASE tool is intended for the design and verification of object-oriented systems. Most existing CASE tools support only the structured design methodology of software development. The main reason for this is that functional analysis and structured design techniques such as data flow diagrams, entity-relationship diagrams and structure charts have been successfully used in practice for several decades. A few CASE tools have been developed and are available commercially for object-oriented analysis and design, however, their supported methodology is built on top of a traditional structured approach which is functional. Such a hybrid approach often fails to provide rigorous object-oriented analysis and design support since: (i) they are designed to model systems where data and procedures are separate; and (ii) they do not easily support rapid prototyping, a major advantage claimed by object-oriented software practitioners. A critical comparison of several object-oriented analysis techniques has been given in [4] where it is mentioned that “tool support for these methods is still in its infancy ...”. For support of object-oriented methods in CASE tools, the ideal tool should support the transition between system requirements, object-oriented design and construction [9].

The CASE tool under development will allow the designer to visually model an application using principles of object-orientation and to verify the structural integrity of the design according to object-oriented design principles. The structural integrity may also be further verified based on the semantics of the target language for the system, for example C++ or Smalltalk. A code generation component will take the system design, verify it, and produce code in the selected object-oriented language. The immediate functionality of this code will depend on how much information has been supplied during the design stage. With this tool, the application may be modeled at any level of abstraction, promoting rapid prototyping.

The underlying GUI for this type of CASE tool must support a rich set of features since the tool supports the display and manipulation of diagrams as part of the application design process. A GUI itself is a complex and large software product, generally operating independent of any other software and consequently, can be designed independently. Due to the voluminous nature of the work, the implementation details of a complete GUI are not presented here. The major focus of this thesis lies in developing an abstraction of the GUI starting from scratch, by defining primitives such as pixels, window controls, display areas, windows and the interactions and compositions of all of these. Applicability of the abstract design to the CASE tool is then illustrated by specifying the components necessary for a generic diagramming tool and tailoring them towards a diagramming tool meant for object-oriented software design. A modular approach to this abstraction has been taken so that many GUI components could be reused in the design of other applications with little or no modification. The primary goal of these specifications, however, is to provide a tool kit of GUI functions for the CASE tool which is independent of implementation details. This thesis is supported by an implementation of the basic controls from the specifications.

Abstract system design is generally based on formal notations. In this thesis, the GUI is formally defined using a model-based formal specification technique, called *the Z notation* [16]. Advantages of using formal methods in the software development process have been extensively discussed in literature. Some of the prominent features of such an approach are:

- The system can be more precisely modeled.
- Inconsistencies and ambiguities in the model can be detected early in the design process and thus can be eliminated.
- System design can be analyzed and verified for consistency and correctness using the underlying mathematical formalism.

Z is a model-based specification language that allows data and the operations in a system to be specified as states and state changes respectively. It allows specifications to be structured into small chunks that can be combined to create larger and more complex specifications. Notational conventions and extensions to Z have been

proposed that can make Z specifications “object-oriented” [17]. As should be the case with all formal specifications, the formal specifications in this thesis are supported by informal descriptions to improve the readability of the document.

The organization of this thesis is as follows: Chapter 2 provides a short introduction to formal specification techniques using the Z specification language and discusses its support of object-oriented design. The specification of the GUI starts with an introduction to pixels and their properties both as individual and composite entities in Chapter 3. Chapter 4 discusses the behaviour of a mouse, through which a user interacts with the system. Window controls are discussed in Chapter 5. In Chapter 6, various window operations are specified. Actual mapping of the window contents to the physical screen is dealt with in Chapter 7. Specifications for a simple diagramming tool using primitives of the GUI outlined in the previous chapters are given in Chapter 8. Finally, in Chapter 9, the specifications of the GUI for the object-oriented CASE tool are illustrated. The thesis concludes with a discussion of the implementation details of these specifications. For convenience, a summary of the Z notation used in this thesis appears in Appendix A, and specifications for primitive data types and functions used throughout this thesis are contained in Appendix B.

Chapter 2

The Z Specification Language

The Z specification language is model-based language that allows a software designer to build an abstract model of a system. It was developed at the Oxford University Computing Laboratory [16]. Z is based on set theory and first order predicate logic, and hence it is easy to reason about specifications written in Z.

2.1 Z Type System

Every Z specification document begins with a definition of data types corresponding to objects that play a part in the specifications but whose details are not currently of interest. These data types are referred to as *basic types* of the specification and are written between square parentheses. An example follows:

[*CHAR*]

Z also has three predefined types: natural numbers (represented as $\mathbb{N}$), positive natural numbers ($\mathbb{N}_1$) and integers ($\mathbb{Z}$). More complex data types can be constructed using Z's *type constructors*. These include set types, cartesian product types and schema types, binary relations and functions, sequences and bags. For brevity, only schemas and functions are discussed in this chapter. For a more detailed treatment of other data types, readers are referred to [14, 16]. A summary of Z notation used in this thesis is given in Appendix A.

2.2 Schemas

Z specifications are typically partitioned into small chunks, called *schemas*. This partitioning improves understandability and clarity for the reader since each schema can be described independent of the rest of the specifications. Both static (representational abstraction) and dynamic (behavioural abstraction) aspects of a system can be described through schemas.

A schema is characterized by a unique name, a declaration part called its *signature*, and a predicate part called its *property* over the signature. When defining a schema type, a named state space is actually being created whose name acts as a global type name within the current specification. The declaration part defines the local typed variables that are used in the state space, and the predicate part describes the requirements about the values of the variables. These requirements are *invariant* in that they must be true irrespective of any state changes. Note that a schema definition merely defines a schema type, not an actual instance of the type. As an example, a “text line” type may be defined as:

<i>TextLine</i>
<i>text</i> : seq <i>CHAR</i>
<i>textlength</i> : $\mathbb{N}$
<i>textlength</i> = # <i>text</i>

This schema defines a state space called *TextLine* having two properties: *text* is a sequence of characters (which are themselves basic types) and *textlength* is a natural number corresponding to the number of characters in the line. The invariant asserts that *textlength* must remain equal to the cardinality of the sequence of characters in all states of the state space.

An operation in Z defines a state change. The schema notation is used both for state space definition and for defining operations. To distinguish between these two uses of schemas, certain conventions are followed when schemas are used for operations. Some of them are: i) input variables are decorated with a question mark; ii) output variables are decorated with an exclamation mark; iii) a prime is used to decorate a state variable to show that it is updated within the current operation; and iv) the state *S* before the operation and the state *S'* after the operation are included

in the declaration of the schema operation. The schema operation below accesses the state space *TextLine* defined previously and inserts additional characters at a given position:

<i>InsertStringInTextLine</i>	_____
<i>TextLine</i>	
<i>TextLine'</i>	
<i>position?</i> : $\mathbb{N}$	
<i>string?</i> : $\text{seq } \text{CHAR}$	
$\text{position?} \leq \text{textlength} + 1$	
$\text{text}' = \{i : 1 \dots \text{position?} - 1\} \upharpoonright \text{text} \hat{\ } \text{string?} \hat{\ } \{i : \text{position?} \dots \# \text{text}\} \upharpoonright \text{text}$	

The variables *position?* and *string?* denote the inputs for this schema. The invariant asserts that the position of insertion must be less than or equal to the length of the string plus one. It is also asserted that after the state change, *text* contains the contents of *string?* inserted into *text* starting in position *position?*. A number of additional predicates might also have been included in the schema, such as

$$\text{textlength}' = \text{textlength} + \# \text{string?}$$

however, it is redundant to include this since it can be easily proven using the stated assertions and the invariant of the state space. Thus, the predicate part of the schema usually contains only the minimum set of predicates necessary to state the invariant properties.

Z provides the facility to duplicate a schema structure using subscripts. For example, if *S* is a schema name, then *S*₁ represents another distinct schema in which the declarations of *S* are duplicated but subscripted as in *S*₁. This facilitates the management of two versions of the same schema within a specification. Examples and more details of this notation will be described later in other sections of this thesis.

Finally, schema operations can be combined to create more complex operations using *schema calculus*. This allows a top-down approach to the development of specifications. For example, one type of schema calculus, called *schema composition* allows the final state of one operation to be used as the initial state for the next operation in a series, thus creating a composite operation with two state transitions. Discussing

schema calculus in depth is beyond the scope of this thesis. For a formal treatment of schema calculus, see [14, 16].

2.3 Functions

Global functions are often described in Z through the use of *axiomatic descriptions*. These constructs typically introduce a global variable having a *function* type, and give restrictions and relationships between the domain and range of the function. For example, given a *TextLine* schema, we can specify a function to return its length as:

$$\left| \begin{array}{l} \text{TextLineLength} : \text{TextLine} \longrightarrow \mathbb{N} \\ \hline \forall tl : \text{TextLine} \bullet \\ \text{TextLineLength}(tl) = tl.\text{textlength} \end{array} \right.$$

Since a function is a special type of binary relation, all operations on binary relations can also be applied to functions. Notations for different types of functions are given in Appendix A.

2.4 Object-Oriented Design in Z

Efforts have been made to provide the Z specification language with the capability to model a system in an object-oriented style [17]. There have been two approaches to this; the first defines conventions that can be used with the standard Z language, and the second proposes extensions to the Z language to allow mechanisms for object-oriented design. Because the standard Z language is more theoretically sound and has good tool support, the specifications in this paper have been written using standard Z and Hall's style [7].

Chapter 3

Pixels and Coordinate Frames

A graphical user interface (GUI) uses *pixel mappings* or *bit maps* to display information on the screen. These bit maps contain information about the image to appear on the display and are generated both by the GUI itself to display its primitive components and by the application software to display application domain information. It is the responsibility of the GUI to maintain the mapping between the actual display and all bitmaps.

In this chapter, primitives of an abstract GUI, namely *pixels* and *coordinate frames* are specified. The specifications start with the abstract structures of pixels and coordinate frames and introduce various operations on them. The properties of pixels and coordinate frames are used throughout this thesis. Even though they are primitive and their properties are fundamental, their specifications are included here for the sake of completeness and compatibility with the rest of the specifications. Readers are referred to [2] for specifications of windowing systems which follow a similar approach.

3.1 Pixels

An image in a GUI-based system is represented by a collection of discrete picture elements called *pixels*. A set of pixels is used by the GUI sub-system to visually represent an image on the display. In practice, pixel mappings also characterize the dimension and resolution of an image.

Pixel addresses are specified using a rectangular coordinate system (X,Y) where X is the horizontal axis and Y is the vertical axis. The addresses can be defined using ordered pairs (x, y) where all coordinate values are non-negative, and the top-

leftmost position is designated as $(0,0)$. A formal representation of the coordinate frame follows:

<i>CoordFrame</i>
<i>cfid</i> : <i>ID</i>
<i>maxX</i> , <i>maxY</i> : $\mathbb{N}$

In the specification for *CoordFrame*, *maxX* and *maxY* refer to the maximum values of *x* and *y* coordinates within this frame. The variable *cfid* refers to the unique identifier of this coordinate frame.

A pixel is modeled as an entity that occupies a valid address within a frame. Note that at this level of specification, the visual contents of a pixel are not addressed.

<i>Pixel</i>
<i>CoordFrame</i>
<i>x</i> , <i>y</i> : $\mathbb{N}$
$x \leq \textit{maxX}$
$y \leq \textit{maxY}$

The coordinate frame to which a pixel belongs combined with its address within the coordinate frame gives a unique identification to the pixel. Stated otherwise, it is possible for two or more pixels to have the same pixel address but only if they belong to different coordinate frames, as specified by the following:

$$\begin{aligned} \forall \textit{Pixel}; \textit{Pixel}_1 \bullet \\ (x = x_1 \wedge y = y_1) \Rightarrow \\ \theta \textit{CoordFrame} \neq \theta \textit{CoordFrame}_1 \end{aligned}$$

The notation θS in Z refers to an unnamed instance of schema type *S*. When θ is applied to a schema, its variables must be in scope with the declaration of that schema; that is, *S* must be declared within the current scope in which θS is used. In the above predicate, the declaration

$$\forall \textit{Pixel}; \textit{Pixel}_1 \bullet$$

contains the declarations of *CoordFrame* and *CoordFrame*₁ and so $\theta \textit{CoordFrame}$ and $\theta \textit{CoordFrame}_1$ are valid. In this example, θ is used to extract the schema *CoordFrame* included as a portion of the schema *Pixel*.

3.1.1 Comparison of Pixels

Pixel addresses can be compared using standard relational operators. Intuitively, $p_1 > p_2$ is true if p_1 is below and to the right of p_2 and both are in the same coordinate frame. Conversely, $p_1 < p_2$ is true if p_1 is above and to the left of p_2 and both are in the same coordinate frame. These operations on pixels are defined as infix relational operations ($\leftrightarrow$) whose scopes extend to the entire set of specifications in this thesis.

$$\begin{array}{l}
\text{---} = \text{---} : \text{Pixel} \leftrightarrow \text{Pixel} \\
\text{---} > \text{---} : \text{Pixel} \leftrightarrow \text{Pixel} \\
\text{---} >= \text{---} : \text{Pixel} \leftrightarrow \text{Pixel} \\
\text{---} < \text{---} : \text{Pixel} \leftrightarrow \text{Pixel} \\
\text{---} <= \text{---} : \text{Pixel} \leftrightarrow \text{Pixel} \\
\hline
\forall p_1, p_2 : \text{Pixel} \bullet \\
\quad p_1 = p_2 \Leftrightarrow \\
\quad \quad (p_1.\text{cfid} = p_2.\text{cfid} \wedge p_1.x = p_2.x \wedge p_1.y = p_2.y) \wedge \\
\quad p_1 > p_2 \Leftrightarrow \\
\quad \quad (p_1.\text{cfid} = p_2.\text{cfid} \wedge \\
\quad \quad (p_1.x > p_2.x \wedge p_1.y > p_2.y \vee \\
\quad \quad p_1.x = p_2.x \wedge p_1.y > p_2.y \vee \\
\quad \quad p_1.x > p_2.x \wedge p_1.y = p_2.y)) \wedge \\
\quad p_1 >= p_2 \Leftrightarrow \\
\quad \quad (p_1 = p_2 \vee p_1 > p_2) \wedge \\
\quad p_1 < p_2 \Leftrightarrow \\
\quad \quad p_2 > p_1 \wedge \\
\quad p_1 <= p_2 \Leftrightarrow \\
\quad \quad p_2 >= p_1
\end{array}$$

Given a group of distinct pixels, the pixel that is positioned furthest from and that which is positioned closest to the origin of the coordinate frame can both be determined, as defined by the following functions:

$$\begin{array}{l}
\text{FarPixel} : \mathbb{P} \text{Pixel} \longrightarrow \text{Pixel} \\
\text{NearPixel} : \mathbb{P} \text{Pixel} \longrightarrow \text{Pixel} \\
\hline
\forall pset : \mathbb{P} \text{Pixel}; \text{maxp}, \text{minp} : \text{Pixel} \bullet \\
\quad \text{FarPixel}(pset) = \text{maxp} \Leftrightarrow \\
\quad \quad (\text{maxp} \in pset \wedge \\
\quad \quad (\forall p : pset \bullet \\
\quad \quad \quad \text{maxp} >= p)) \wedge \\
\quad \text{NearPixel}(pset) = \text{minp} \Leftrightarrow \\
\quad \quad (\text{minp} \in pset \wedge \\
\quad \quad (\forall p : pset \bullet \\
\quad \quad \quad \text{minp} <= p))
\end{array}$$

Subtraction on pixels can be performed provided the pixels are within the same coordinate frame. Addition may also be specified, however it is not used in this thesis, and so is left undefined. If two pixels can be visualized as vectors starting at the origin of the coordinate system and extending to the point located at their addresses, pixel subtraction is equivalent to vector subtraction, which is defined by subtracting each of the x and y components of the second vector from those of the first vector. Since negative pixel addresses are not permitted, the x and y values of the first pixel must be larger than or equal to those of the second pixel:

$$\begin{array}{|l} \hline _ - _ : Pixel \times Pixel \leftrightarrow Pixel \\ \hline \forall p_1, p_2, p : Pixel \bullet \\ \quad p_1 - p_2 = p \Leftrightarrow \\ \quad (p_1.cfid = p_2.cfid = p.cfid \wedge \\ \quad p.x = p_1.x - p_2.x \wedge \\ \quad p.y = p_1.y - p_2.y \wedge \\ \quad p_1 \geq p_2) \end{array}$$

It can be said that one pixel is *above* another pixel if their respective x coordinate values are the same and their y coordinates differ. The pixel occupying the top position must have a y coordinate value less than that of the bottom pixel. Similarly, one pixel is to the *left of* another pixel if their y coordinate values are equal and the x values differ with the x coordinate value of the leftmost pixel less than that of the other pixel:

$$\begin{array}{|l} \hline _ PAbove _ : Pixel \leftrightarrow Pixel \\ _ PLeftOf _ : Pixel \leftrightarrow Pixel \\ \hline \forall p, p_1 : Pixel \bullet \\ \quad p PAbove p_1 \Leftrightarrow (p.x = p_1.x \wedge p.y < p_1.y) \wedge \\ \quad p PLeftOf p_1 \Leftrightarrow (p.y = p_1.y \wedge p.x < p_1.x) \end{array}$$

From these definitions, *below* and *right of* can be defined:

$$\begin{array}{|l} \hline _ PBelow _ : Pixel \leftrightarrow Pixel \\ _ PRightOf _ : Pixel \leftrightarrow Pixel \\ \hline \forall p, p_1 : Pixel \bullet \\ \quad p PBelow p_1 \Leftrightarrow p_1 PAbove p \wedge \\ \quad p PRightOf p_1 \Leftrightarrow p_1 PLeftOf p \end{array}$$

More generally, pixels may be located along a common horizontal or vertical line:

PHorizontal_ : Pixel $\leftrightarrow$ Pixel
PVertical_ : Pixel $\leftrightarrow$ Pixel
$\forall p, p_1 : \text{Pixel} \bullet$
PHorizontal(p, p_1) $\Leftrightarrow$ (p PLeftOf $p_1 \vee p$ PRightOf p_1) $\wedge$
PVertical(p, p_1) $\Leftrightarrow$ (p PAbove $p_1 \vee p$ PBelow p_1)

Next, adjacency relationships between pixels are defined. Two pixels are *horizontally adjacent* to each other, or side-by-side, if the difference between their x coordinate values is equal to or less than one *unit of screen resolution* and their respective y coordinate values are equal. *Vertical adjacency* is defined in a similar manner. Since the number of pixels on a display depends on the display's resolution, adjacency between two pixels is strictly based on the unit of resolution. At the specification level, this is given as a global variable which will be instantiated at execution time:

$\text{unitofresolution} : \mathbb{N}_1$
PHAdjacent_ : Pixel $\leftrightarrow$ Pixel
PVAdjacent_ : Pixel $\leftrightarrow$ Pixel
$\forall p, p_1 : \text{Pixel} \bullet$
PHAdjacent(p, p_1) $\Leftrightarrow$
(PHorizontal(p, p_1) $\wedge$
$\text{Abs}(p.x - p_1.x) \leq \text{unitofresolution}$) $\wedge$
PVAdjacent(p, p_1) $\Leftrightarrow$
(PVertical(p, p_1) $\wedge$
$\text{Abs}(p.y - p_1.y) \leq \text{unitofresolution}$)

A set of pixels forms a *square* in the coordinate frame if it contains exactly $n * n$ pixels for some dimension n , and if the distance between the x coordinates of the farthest and nearest pixels in the set is equal to the distance between the y coordinates of the same pixels.

PlsSquare_ : $\mathbb{P}(\mathbb{P} \text{Pixel})$
$\forall pp : \mathbb{P} \text{Pixel} \bullet \text{PlsSquare}(pp) \Leftrightarrow$
($\exists \text{dim} : \mathbb{N} \bullet$
$\text{dim} = (\text{FarPixel}(pp)).x - (\text{NearPixel}(pp)).x =$
$(\text{FarPixel}(pp)).y - (\text{NearPixel}(pp)).y \wedge$
$\#pp = \text{Sqr}(\text{dim})$)

PlsSquare is defined as a prefix relation over a set of groups of pixels. The axiom for PlsSquare is thus valid for every member of this set, where each member itself is a set of pixels forming a square.

3.1.2 Operations on Pixels

General operations on pixels are now defined. Every operation on a pixel has an invariant property that the coordinate frame to which the pixel belongs cannot be changed. Any other pixel property may be changed by an operation unless the operation itself indicates a restriction:

$$\frac{\text{OpOnPixel} \quad \Delta\text{Pixel}}{\theta\text{CoordFrame}' = \theta\text{CoordFrame}}$$

The notation ΔPixel is used in the schema operation above as a shorthand form for the inclusion of Pixel and Pixel' , as in:

$$\frac{\Delta\text{Pixel}}{\text{Pixel} \quad \text{Pixel}'}$$

where Pixel refers to the initial state and pixel' refers to the final state of the operation.

To change the x or y coordinate value of a pixel, the new value must be within the limits of the coordinate frame:

$$\frac{\text{ChangePixelX} \quad \text{OpOnPixel} \quad \text{newx?} : \mathbb{N}}{\text{newx?} \neq x \quad \text{newx?} \leq \text{maxX} \quad x' = \text{newx?} \quad y' = y}$$

$$\frac{\text{ChangePixelY} \quad \text{OpOnPixel} \quad \text{newy?} : \mathbb{N}}{\text{newy?} \neq y \quad \text{newy?} \leq \text{maxY} \quad y' = \text{newy?} \quad x' = x}$$

3.2 Regions Containing Pixels

A simple region can be defined as a rectangular group of pixels within a coordinate frame. A region must be congruent with the coordinate frame, therefore rotations or rectangular groups at an oblique angle to an axis are not considered. The top-leftmost and bottom-rightmost pixels identify the area occupied by a simple region:

<i>SimpleRegion</i>	
<i>srid</i> : ID	
<i>cfid</i> : ID	
<i>topleft</i> : Pixel	
<i>botright</i> : Pixel	
<i>topleft</i> <= <i>botright</i>	
<i>cfid</i> = <i>topleft.cfid</i> = <i>botright.cfid</i>	

In the *SimpleRegion* schema, the *srid* variable is used to uniquely identify a simple region. The variable *cfid* refers to the coordinate frame identifier for both bounding pixels of the region. Though the value of *cfid* can be derived from *topleft* and *botright*, it is included as a variable for notational convenience.

The height and width of a simple region can be determined by the respective distances between the *x* and *y* coordinate values of the bounding pixels:

<i>SRHeight</i> : <i>SimpleRegion</i> $\rightarrow \mathbb{N}$	
<i>SRWidth</i> : <i>SimpleRegion</i> $\rightarrow \mathbb{N}$	
$\forall sr : SimpleRegion \bullet$	
$SRHeight(sr) = sr.botright.y - sr.topleft.y + 1 \wedge$	
$SRWidth(sr) = sr.botright.x - sr.topleft.x + 1$	

Although the *SimpleRegion* schema does not explicitly provide the set of pixels contained within it, this can be determined through a function. The pixels in this set must be located on or between the bounding coordinate values, and the number of pixels in this set is equal to the product of the dimensions of the region:

<i>AllSRPoints</i> : <i>SimpleRegion</i> $\rightarrow \mathbb{P} Pixel$	
$\forall sr : SimpleRegion \bullet$	
$AllSRPoints(sr) = \{p : Pixel \mid p.cfid = sr.cfid \wedge$	
$p <= sr.botright \wedge p >= sr.topleft\}$	

Two simple regions have the same *shape* if their height and width are identical:

$\text{SameShape_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$	$\text{SameShape_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$
--	--

$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $\text{SameShape}(sr, sr_1) \Leftrightarrow$ $(SRHeight(sr) = SRHeight(sr_1) \wedge$ $SRWidth(sr) = SRWidth(sr_1))$	$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $\text{SameShape}(sr, sr_1) \Leftrightarrow$ $(SRHeight(sr) = SRHeight(sr_1) \wedge$ $SRWidth(sr) = SRWidth(sr_1))$
--	--

Since the set of pixels contained within a simple region can be computed, containment and overlapping between simple regions can be specified. A simple region is *contained* within another if the set of pixels to which the first region maps is a proper subset of the set of pixels to which the second region maps. These simple regions must be defined in the same coordinate frame:

$\text{_SRContains_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$	$\text{_SRContains_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$
---	---

$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $sr \text{ SRContains } sr_1 \Leftrightarrow$ $(sr.cfid = sr_1.cfid \wedge$ $AllSRPoints(sr_1) \subset AllSRPoints(sr))$	$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $sr \text{ SRContains } sr_1 \Leftrightarrow$ $(sr.cfid = sr_1.cfid \wedge$ $AllSRPoints(sr_1) \subset AllSRPoints(sr))$
---	---

A simple region is equal to another simple region if the pixels to which each region maps form the same set. This need not be specified since type equality is automatically defined in Z for all data types.

Two screen regions *overlap* if the pixel sets to which they map have at least one pixel in common. Again, both simple regions must be defined in the same coordinate frame:

$\text{_SROverlaps_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$	$\text{_SROverlaps_} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion}$
---	---

$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $sr \text{ SROverlaps } sr_1 \Leftrightarrow$ $(sr.cfid = sr_1.cfid \wedge$ $AllSRPoints(sr) \cap AllSRPoints(sr_1) \neq \emptyset)$	$\forall sr, sr_1 : \text{SimpleRegion} \bullet$ $sr \text{ SROverlaps } sr_1 \Leftrightarrow$ $(sr.cfid = sr_1.cfid \wedge$ $AllSRPoints(sr) \cap AllSRPoints(sr_1) \neq \emptyset)$
---	---

Determining the relative position of two or more simple regions in their coordinate frame is important if visual characteristics of interface components must be specified. For example, interface requirements may indicate that one image must appear above another image on the screen. If each image can be defined in terms of the simple region that bounds it, these requirements can be expressed in terms of the positioning of the simple regions. Simple region relationships can be built using pixel functions and operations combined with primitive simple region functions.

Every simple region has a leftmost column of pixels, a rightmost column of pixels, and a top and bottom row of pixels. These correspond to the four edges of the region and are also of type *SimpleRegion* (with a width or height of one pixel). Given a simple region, its four edges can be returned by the following functions:

$ \begin{aligned} &SRLeftmostCol : SimpleRegion \rightarrow SimpleRegion \\ &SRRightmostCol : SimpleRegion \rightarrow SimpleRegion \\ &SRTopmostRow : SimpleRegion \rightarrow SimpleRegion \\ &SRBottommostRow : SimpleRegion \rightarrow SimpleRegion \\ &\forall sr : SimpleRegion \bullet \\ &\quad AllSRPoints(SRLeftmostCol(sr)) = \\ &\quad \quad \{p : AllSRPoints(sr) \mid p.x = sr.toleft.x\} \wedge \\ &\quad AllSRPoints(SRRightmostCol(sr)) = \\ &\quad \quad \{p : AllSRPoints(sr) \mid p.x = sr.botrigh.x\} \wedge \\ &\quad AllSRPoints(SRTopmostRow(sr)) = \\ &\quad \quad \{p : AllSRPoints(sr) \mid p.y = sr.toleft.y\} \wedge \\ &\quad AllSRPoints(SRBottommostRow(sr)) = \\ &\quad \quad \{p : AllSRPoints(sr) \mid p.y = sr.botrigh.y\} \end{aligned} $

One simple region is *above* another if the bottom row of pixels in the first region is adjacent to the top row of pixels in the second and both regions have the same dimension along the touching side. This is specified below, along with similar kinds of relationships (*below*, *left of*, and *right of*) between simple regions:

$ \begin{aligned} &_SRAbove_ : SimpleRegion \leftrightarrow SimpleRegion \\ &_SRLeftOf_ : SimpleRegion \leftrightarrow SimpleRegion \\ &\forall sr, sr_1 : SimpleRegion \bullet \\ &\quad sr \text{ SRAbove } sr_1 \Leftrightarrow \\ &\quad \quad (\exists m : Pixel \succ\!\!\rightarrow Pixel \bullet \\ &\quad \quad \quad dom\ m = AllSRPoints(SRBottommostRow(sr)) \wedge \\ &\quad \quad \quad ran\ m = AllSRPoints(SRTopmostRow(sr_1)) \wedge \\ &\quad \quad \quad (\forall p : dom\ m \bullet \\ &\quad \quad \quad \quad p \text{ PAbove } m(p) \wedge PVAadjacent(p, m(p)))) \wedge \\ &\quad sr \text{ SRLeftOf } sr_1 \Leftrightarrow \\ &\quad \quad (\exists m : Pixel \succ\!\!\rightarrow Pixel \bullet \\ &\quad \quad \quad dom\ m = AllSRPoints(SRLeftmostCol(sr)) \wedge \\ &\quad \quad \quad ran\ m = AllSRPoints(SRRightmostCol(sr_1)) \wedge \\ &\quad \quad \quad (\forall p : dom\ m \bullet \\ &\quad \quad \quad \quad p \text{ PABove } m(p) \wedge PHAadjacent(p, m(p)))) \end{aligned} $

The specification for *SRAbove* asserts the following: there is a bijective function m (indicated by $\succ\!\!\rightarrow$) that maps each pixel in the bottom row of the top simple region

to a pixel in the top row of the bottom simple region. This function indicates that for every pixel pair p and $m(p)$, p is PAbove $m(p)$ and they are vertically adjacent. The *below* and *right of* simple region relationships can be defined in terms of the relationships defined above:

$$\begin{array}{|l} \text{-- SRBelow --} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion} \\ \text{-- SRRightOf --} : \text{SimpleRegion} \leftrightarrow \text{SimpleRegion} \\ \hline \forall sr, sr_1 : \text{SimpleRegion} \bullet \\ \quad sr \text{ SRBelow } sr_1 \Leftrightarrow sr_1 \text{ SRAbove } sr \wedge \\ \quad sr \text{ SRRightOf } sr_1 \Leftrightarrow sr_1 \text{ SRLeftOf } sr \end{array}$$

A sequence of simple regions is *aligned vertically* if one region in the sequence appears above the next, forming a stack. All regions in the sequence must have the same width. Similarly, a sequence of simple regions is *aligned horizontally* if one region in the sequence appears to the left of the next. These characteristics can be asserted by using the SRLeftOf and SRAbove relationships just defined:

$$\begin{array}{|l} \text{SRHAdjacentLine--} : \mathbb{P}(\text{seq SimpleRegion}) \\ \text{SRVAdjacentLine--} : \mathbb{P}(\text{seq SimpleRegion}) \\ \hline \forall sr : \text{seq SimpleRegion} \bullet \\ \quad \text{SRHAdjacentLine}(sr) \Leftrightarrow \\ \quad (\forall i : 1 \dots (\#sr) - 1 \bullet sr(i) \text{ SRLeftOf } sr(i + 1)) \wedge \\ \quad \text{SRVAdjacentLine}(sr) \Leftrightarrow \\ \quad (\forall i : 1 \dots (\#sr) - 1 \bullet sr(i) \text{ SRAbove } sr(i + 1)) \end{array}$$

Because of the congruency relationship between simple regions and the coordinate frame axes, there can exist only horizontal and vertical region orientations:

$$\text{lineorient} ::= \text{horizontal} \mid \text{vertical}$$

Thus, the previous two functions may be generalized as:

$$\begin{array}{|l} \text{SRAdjacentLine--} : \mathbb{P}(\text{seq SimpleRegion} \times \text{lineorient}) \\ \hline \forall sr : \text{seq SimpleRegion}; ori : \text{lineorient} \bullet \\ \quad \text{SRAdjacentLine}(sr, ori) \Leftrightarrow \\ \quad (ori = \text{horizontal} \Leftrightarrow \text{SRHAdjacentLine}(sr) \wedge \\ \quad \quad ori = \text{vertical} \Leftrightarrow \text{SRVAdjacentLine}(sr)) \end{array}$$

3.2.1 Operations on Simple Regions

Several operations may be defined on simple regions. These include operations for altering the corner pixels and thus changing the shape of the region, and operations

for enlarging, shrinking and moving a simple region. First, it must be asserted that an operation on *SimpleRegion* cannot change its identifier or the coordinate frame to which it belongs, but may change any other property:

<i>OpOnSimpleRegion</i>	_____
$\Delta SimpleRegion$	
$srid' = srid$	
$cfid' = cfid$	

The x coordinate value of the top left pixel in a simple region can be changed by applying the *ChangePixelX* schema operation defined earlier to the *opleft* property of the initial state. The bottom right pixel remains unchanged:

<i>ChTopLeftX</i>	_____
<i>OpOnSimpleRegion</i>	
$newx? : \mathbb{N}$	
$opleft' = (\mu \text{ ChangePixelX} \mid \theta Pixel = topleft \bullet \theta Pixel')$	
$botright' = botright$	

In a similar fashion, the top left y coordinate and both bottom right coordinate values can be changed:

<i>ChTopLeftY</i>	_____
<i>OpOnSimpleRegion</i>	
$newy? : \mathbb{N}$	
$opleft' = (\mu \text{ ChangePixelY} \mid \theta Pixel = topleft \bullet \theta Pixel')$	
$botright' = botright$	

<i>ChBotRightX</i>	_____
<i>OpOnSimpleRegion</i>	
$newx? : \mathbb{N}$	
$botright' = (\mu \text{ ChangePixelX} \mid \theta Pixel = botright \bullet \theta Pixel')$	
$topleft' = topleft$	

<i>ChBotRightY</i>	_____
<i>OpOnSimpleRegion</i>	
$newy? : \mathbb{N}$	
$botright' = (\mu \text{ ChangePixelY} \mid \theta Pixel = botright \bullet \theta Pixel')$	
$topleft' = topleft$	

The μ construct in the above specifications indicate the creation of a temporary instance of a schema satisfying the conditions stated in the μ expression. The result of such a μ expression is identified after the $\bullet$ symbol.

Using these operations as building blocks, operations for enlarging a region can be specified. A simple region can be enlarged along each of the eight directions; hence there are eight enlarging operations specified for simple regions. Each of these operations has a direction of enlargement and a pixel provided as input, where the direction is one of those indicated by the following global identifier:

$$\text{direction} ::= \text{Top} \mid \text{Bottom} \mid \text{Left} \mid \text{Right} \mid \text{TopLeft} \mid \text{TopRight} \mid \\ \text{BottomLeft} \mid \text{BottomRight}$$

As an example of enlargement, the top side of a simple region can be changed so that its y coordinate value is equal to the y coordinate value of the input pixel. In this case, the input pixel's y coordinate value must be less than that of the top side of the region, and the direction of enlargement must indicate enlargement towards the top side of the region. The $ChTopLeftY$ schema operation is used to change the top left pixel's value:

$\frac{\text{EnlargeTopSide} \quad \text{OpOnSimpleRegion} \quad \text{pixel?} : \text{Pixel} \quad \text{direction?} : \text{direction}}{\text{pixel?.y} < \text{toleft.y} \quad \text{direction?} = \text{Top} \quad \exists \text{newy} : \mathbb{N} \bullet \quad \text{newy} = \text{pixel?.y} \wedge \quad \text{ChTopLeftY}[\text{newy}/\text{newy?}]}$
--

Enlarging in the bottom, left and right directions are specified in a similar way:

$\frac{\text{EnlargeBottomSide} \quad \text{OpOnSimpleRegion} \quad \text{pixel?} : \text{Pixel} \quad \text{direction?} : \text{direction}}{\text{pixel?.y} > \text{botright.y} \quad \text{direction?} = \text{Bottom} \quad \exists \text{newy} : \mathbb{N} \bullet \quad \text{newy} = \text{pixel?.y} \wedge \quad \text{ChBotRightY}[\text{newy}/\text{newy?}]}$

EnlargeLeftSide OpOnSimpleRegion $\text{pixel?} : \text{Pixel}$ $\text{direction?} : \text{direction}$ $\text{pixel?.}x < \text{topleft}.x$ $\text{direction?} = \text{Left}$ $\exists \text{newx} : \mathbb{N} \bullet$ $\quad \text{newx} = \text{pixel?.}x \wedge$ $\quad \text{ChTopLeftX}[\text{newx}/\text{newx?}]$

EnlargeRightSide OpOnSimpleRegion $\text{pixel?} : \text{Pixel}$ $\text{direction?} : \text{direction}$ $\text{pixel?.}x > \text{botright}.x$ $\text{direction?} = \text{Right}$ $\exists \text{newx} : \mathbb{N} \bullet$ $\quad \text{newx} = \text{pixel?.}x \wedge$ $\quad \text{ChBotRightX}[\text{newx}/\text{newx?}]$

Enlarging towards a corner direction implies that the simple region is enlarged simultaneously in two of the four side directions. These operations can be constructed by combining the schema operations just specified. For example, enlarging a simple region towards the top left side uses both the *EnlargeTopSide* and *EnlargeLeftSide* schema operations:

EnlargeTopLeft OpOnSimpleRegion $\text{pixel?} : \text{Pixel}$ $\text{direction?} : \text{direction}$ $\text{direction?} = \text{TopLeft}$ $\exists \text{dir} : \text{direction} \bullet$ $\quad \text{dir} = \text{Top} \wedge \text{EnlargeTopSide}[\text{dir}/\text{direction?}]$ $\exists \text{dir} : \text{direction} \bullet$ $\quad \text{dir} = \text{Left} \wedge \text{EnlargeLeftSide}[\text{dir}/\text{direction?}]$
--

It must be noted that the two schema operations that are used in the predicate part of the *EnlargeTopLeft* operation, namely *EnlargeTopSide* and *EnlargeLeftSide*, both require the inclusion of the pair of states *SimpleRegion* and *SimpleRegion'*. These two

states are in scope in the *EnlargeTopLeft* schema operation since *OpOnSimpleRegion* is included in the declaration part. It follows that this specification indicates a change in state of a simple region by the simultaneous application of two separate state changes. Specifications for enlarging in the other three corner directions are built in a similar manner:

$\begin{array}{l} \text{EnlargeTopRight} \\ \text{OpOnSimpleRegion} \\ \text{pixel?} : \text{Pixel} \\ \text{direction?} : \text{direction} \end{array}$
$\begin{array}{l} \text{direction?} = \text{TopRight} \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Top} \wedge \text{EnlargeTopSide}[\text{dir}/\text{direction?}] \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Right} \wedge \text{EnlargeRightSide}[\text{dir}/\text{direction?}] \end{array}$

$\begin{array}{l} \text{EnlargeBottomLeft} \\ \text{OpOnSimpleRegion} \\ \text{pixel?} : \text{Pixel} \\ \text{direction?} : \text{direction} \end{array}$
$\begin{array}{l} \text{direction?} = \text{BottomLeft} \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Bottom} \wedge \text{EnlargeBottomSide}[\text{dir}/\text{direction?}] \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Left} \wedge \text{EnlargeLeftSide}[\text{dir}/\text{direction?}] \end{array}$

$\begin{array}{l} \text{EnlargeBottomRight} \\ \text{OpOnSimpleRegion} \\ \text{pixel?} : \text{Pixel} \\ \text{direction?} : \text{direction} \end{array}$
$\begin{array}{l} \text{direction?} = \text{BottomRight} \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Bottom} \wedge \text{EnlargeBottomSide}[\text{dir}/\text{direction?}] \\ \exists \text{ dir} : \text{direction} \bullet \\ \quad \text{dir} = \text{Right} \wedge \text{EnlargeRightSide}[\text{dir}/\text{direction?}] \end{array}$

Enlargement of a simple region can be generalized as enlargement in one of eight possible directions. This can be specified as follows:

$$\begin{aligned}
\text{EnlargeSimpleRegion} \cong & \text{EnlargeTopSide} \vee \text{EnlargeBottomSide} \vee \\
& \text{EnlargeLeftSide} \vee \text{EnlargeRightSide} \vee \text{EnlargeTopLeft} \vee \\
& \text{EnlargeTopRight} \vee \text{EnlargeBottomLeft} \vee \text{EnlargeBottomRight}
\end{aligned}$$

The *EnlargeSimpleRegion* specification uses *schema disjunction* to combine the eight individual schema operations. For schemas operations S and S_1 , the operation implied by $S \vee S_1$ is valid if it is equivalent to the application of operation S or operation S_1 or both simultaneously. Since each of the eight operations being combined above requires a unique value for the *direction?* input, only one of the eight operations can be applied at one time. In practice, schema disjunction is used to combine two or more alternative schema operations that may be applied in order to create the specification for a more abstract operation.

The shrinking operation applied to a simple region can be specified in a manner similar to the enlarge operation:

$ \begin{aligned} & \text{ShrinkTopSide} \text{ ---} \\ & \text{OpOnSimpleRegion} \\ & \text{pixel?} : \text{Pixel} \\ & \text{direction?} : \text{direction} \end{aligned} $ $ \begin{aligned} & \text{pixel?.y} > \text{topleft.y} \\ & \text{direction?} = \text{Top} \\ & \exists \text{newy} : \mathbb{N} \bullet \\ & \quad \text{newy} = \text{pixel?.y} \wedge \\ & \quad \text{ChTopLeftY}[\text{newy}/\text{newy?}] \end{aligned} $
--

$ \begin{aligned} & \text{ShrinkBottomSide} \text{ ---} \\ & \text{OpOnSimpleRegion} \\ & \text{pixel?} : \text{Pixel} \\ & \text{direction?} : \text{direction} \end{aligned} $ $ \begin{aligned} & \text{pixel?.y} < \text{botright.y} \\ & \text{direction?} = \text{Bottom} \\ & \exists \text{newy} : \mathbb{N} \bullet \\ & \quad \text{newy} = \text{pixel?.y} \wedge \\ & \quad \text{ChBotRightY}[\text{newy}/\text{newy?}] \end{aligned} $
--

ShrinkLeftSide

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

$pixel?.x > topleft.x$
 $direction? = Left$
 $\exists newx : \mathbb{N} \bullet$
 $newx = pixel?.x \wedge$
 $ChTopLeftX[newx/newx?]$

ShrinkRightSide

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

$topleft.x \leq pixel?.x < botright.x$
 $direction? = Right$
 $\exists newx : \mathbb{N} \bullet$
 $newx = pixel?.x \wedge$
 $ChBotRightX[newx/newx?]$

ShrinkTopLeft

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

$direction? = TopLeft$
 $\exists dir : direction \bullet$
 $dir = Top \wedge ShrinkTopSide[dir/direction?]$
 $\exists dir : direction \bullet$
 $dir = Left \wedge ShrinkLeftSide[dir/direction?]$

ShrinkTopRight

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

$direction? = TopRight$
 $\exists dir : direction \bullet$
 $dir = Top \wedge ShrinkTopSide[dir/direction?]$
 $\exists dir : direction \bullet$
 $dir = Right \wedge ShrinkRightSide[dir/direction?]$

ShrinkBottomLeft

OpOnSimpleRegion

pixel? : *Pixel*

direction? : *direction*

direction? = *BottomLeft*

$\exists \text{ dir} : \text{direction} \bullet$

$\text{dir} = \text{Bottom} \wedge \text{ShrinkBottomSide}[\text{dir} / \text{direction?}]$

$\exists \text{ dir} : \text{direction} \bullet$

$\text{dir} = \text{Left} \wedge \text{ShrinkLeftSide}[\text{dir} / \text{direction?}]$

ShrinkBottomRight

OpOnSimpleRegion

pixel? : *Pixel*

direction? : *direction*

direction? = *BottomRight*

$\exists \text{ dir} : \text{direction} \bullet$

$\text{dir} = \text{Bottom} \wedge \text{ShrinkBottomSide}[\text{dir} / \text{direction?}]$

$\exists \text{ dir} : \text{direction} \bullet$

$\text{dir} = \text{Right} \wedge \text{ShrinkRightSide}[\text{dir} / \text{direction?}]$

$$\text{ShrinkSimpleRegion} \hat{=} \text{ShrinkTopSide} \vee \text{ShrinkBottomSide} \vee \text{ShrinkLeftSide} \vee$$
$$\text{ShrinkRightSide} \vee \text{ShrinkTopLeft} \vee \text{ShrinkTopRight} \vee$$
$$\text{ShrinkBottomLeft} \vee \text{ShrinkBottomRight}$$

It is interesting to note that the same set of primitive operations can be used for both scaling and moving a region. For example, consider a simple region R to be moved to a new position p . Assume that the move is specified in such a way that p represents the new top left corner of R , and that the x and y coordinates of p are both less than the current x and y coordinates of the top left corner of R . That is, the region is to be moved diagonally towards its top left corner. Such a move can be broken into two parts: i) enlarge towards the top left of the region; and ii) shrink from the bottom right of the region by the same distance. As in the case of enlargement and shrinking, moving a symbol will be specified in each of the eight possible directions and then as a general operation.

To move a simple region in the direction of its top side (or *up*), the top side of the region is first moved so that its new top left corner has a y coordinate value equal to the y coordinate value of the pixel provided as input. Since the simple region's

final state must have the same shape as the simple region in the initial state, the bottom side of the region must be moved by the same distance:

<i>MoveUp</i>
<i>OpOnSimpleRegion</i>
<i>pixel?</i> : <i>Pixel</i>
<i>direction?</i> : <i>direction</i>
<i>direction?</i> = <i>Top</i>
<i>EnlargeTopSide</i>
$\exists pixel : Pixel \bullet$
<i>ShrinkBottomSide</i> [<i>pixel/pixel?</i>]
<i>SameShape</i> ($\theta SimpleRegion, \theta SimpleRegion'$)

Movement in the other seven directions is specified in a similar way:

<i>MoveDown</i>
<i>OpOnSimpleRegion</i>
<i>pixel?</i> : <i>Pixel</i>
<i>direction?</i> : <i>direction</i>
<i>direction?</i> = <i>Bottom</i>
<i>EnlargeBottomSide</i>
$\exists pixel : Pixel \bullet$
<i>ShrinkTopSide</i> [<i>pixel/pixel?</i>]
<i>SameShape</i> ($\theta SimpleRegion, \theta SimpleRegion'$)

<i>MoveLeft</i>
<i>OpOnSimpleRegion</i>
<i>pixel?</i> : <i>Pixel</i>
<i>direction?</i> : <i>direction</i>
<i>direction?</i> = <i>Left</i>
<i>EnlargeLeftSide</i>
$\exists pixel : Pixel \bullet$
<i>ShrinkRightSide</i> [<i>pixel/pixel?</i>]
<i>SameShape</i> ($\theta SimpleRegion, \theta SimpleRegion'$)

MoveRight

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

direction? = *Right*
EnlargeRightSide
 $\exists \text{ pixel} : \text{Pixel} \bullet$
 ShrinkLeftSide[*pixel/pixel?*]
SameShape($\theta \text{SimpleRegion}$, $\theta \text{SimpleRegion}'$)

MoveUpAndLeft

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

direction? = *TopLeft*
EnlargeTopLeft
 $\exists \text{ pixel} : \text{Pixel} \bullet$
 ShrinkBottomRight[*pixel/pixel?*]
SameShape($\theta \text{SimpleRegion}$, $\theta \text{SimpleRegion}'$)

MoveDownAndRight

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

direction? = *BottomRight*
EnlargeBottomRight
 $\exists \text{ pixel} : \text{Pixel} \bullet$
 ShrinkTopLeft[*pixel/pixel?*]
SameShape($\theta \text{SimpleRegion}$, $\theta \text{SimpleRegion}'$)

MoveUpAndRight

OpOnSimpleRegion
pixel? : *Pixel*
direction? : *direction*

direction? = *TopRight*
EnlargeTopRight
 $\exists \text{ pixel} : \text{Pixel} \bullet$
 ShrinkBottomLeft[*pixel/pixel?*]
SameShape($\theta \text{SimpleRegion}$, $\theta \text{SimpleRegion}'$)

<i>MoveDownAndLeft</i> <i>OpOnSimpleRegion</i> <i>pixel? : Pixel</i> <i>direction? : direction</i>
<i>direction? = BottomLeft</i> <i>EnlargeBottomLeft</i> $\exists pixel : Pixel \bullet$ <i>ShrinkTopRight</i> [<i>pixel/pixel?</i>] <i>SameShape</i> (<i>SimpleRegion</i> , <i>SimpleRegion'</i>)

Now, a general move can be specified as follows:

$$\begin{aligned}
 MoveSimpleRegion \cong & MoveUp \vee MoveDown \vee MoveLeft \vee MoveRight \vee \\
 & MoveUpAndLeft \vee MoveDownAndRight \vee MoveUpAndRight \vee \\
 & MoveDownAndLeft
 \end{aligned}$$

3.3 Virtual Screen Pixels and Regions

The specifications for pixels and simple regions discussed thus far are abstract in the sense that they do not describe an actual display or visual surface. These specifications are intended to be generic enough so that several distinct visual coordinate frames can be easily specified through reuse. The first among these coordinate frames is called the *virtual screen* which models the coordinate frame of the display area on a physical screen.

A computer screen is comprised of a fixed-size array of pixels. The dimensions of this array are a function of the interface software and the resolution capability of the hardware being used. For now, screen pixels will be discussed in terms of a *virtual screen* that has an unknown size. This is done for two reasons: i) many screen objects may be mapped to the same virtual screen pixel. The objects that actually appear on the screen can only be determined when all mapped objects are known and their layering and overlapping is considered; and ii) a screen object may only be partially mapped to a physical screen, and hence partially visible, but it must be mapped as a whole to some larger coordinate system so that its visual information is not lost.

Virtual screen pixels and *virtual screen regions* can be specified by reusing the specifications for the types and operations just defined for pixels and simple regions.

These new types will later be used to visually represent window controls, windows and other information presented to the user of a GUI.

The virtual screen coordinate system has bounding coordinate values, and a virtual screen pixel must be located in a coordinate frame having these bounds:

$vscreenXmax : \mathbb{N}_1$ $vscreenYmax : \mathbb{N}_1$	
$VSscreenPixel$ $Pixel$	
$maxX = vscreenXmax$ $maxY = vscreenYmax$	

Every instance of type $VSscreenPixel$ must belong to the same coordinate frame since there is only one virtual screen:

$$\forall vsp, vsp_1 : VSscreenPixel \bullet \\ vsp.cfid = vsp_1.cfid$$

Several functions and relations can be defined on virtual screen regions. Since these were specified earlier for the schema type $Pixel$, and $VSscreenPixel$ contains $Pixel$ as an included schema type, these new specifications can be created through reuse with some notational conversions. First, the $PixelFromVSP$ function abbreviation is defined in order to extract the $Pixel$ component of $VSscreenPixel$:

$$PixelFromVSP == (\lambda VSscreenPixel \bullet \theta Pixel)$$

This function takes an argument whose *shape* is indicated after the λ symbol, and returns the value indicated after the $\bullet$.

Now, relations, axioms and operations can be specified on a $VSscreenPixel$ using those specifications defined on $Pixel$:

$FarVSscreenPixel : \mathbb{P} VSscreenPixel \rightarrow VSscreenPixel$ $NearVSscreenPixel : \mathbb{P} VSscreenPixel \rightarrow VSscreenPixel$	
$\forall vspset : \mathbb{P} VSscreenPixel; vsp : VSscreenPixel \bullet$ $FarVSscreenPixel(vspset) = vsp \Leftrightarrow$ $PixelFromVSP(vsp) = FarPixel(PixelFromVSP(vspset)) \wedge$ $NearVSscreenPixel(vspset) = vsp \Leftrightarrow$ $PixelFromVSP(vsp) = NearPixel(PixelFromVSP(vspset))$	

The notation $\langle \rangle$ in Z stands for relational image; typically, $R \langle S \rangle$ returns the image of the relation R for all inputs in the set S , where S is a subset of the domain of R .

$$\begin{array}{|l}
\hline
_ - _ : VScreenPixel \times VScreenPixel \leftrightarrow VScreenPixel \\
\hline
\forall vsp_1, vsp_2, vsp : VScreenPixel \bullet \\
\quad vsp_1 - vsp_2 = vsp \Leftrightarrow \\
\quad (PixelFromVSP(vsp_1) - PixelFromVSP(vsp_2) = PixelFromVSP(vsp)) \\
\\
VSPHAdjacent_ : VScreenPixel \leftrightarrow VScreenPixel \\
VSPVAdjacent_ : VScreenPixel \leftrightarrow VScreenPixel \\
\hline
\forall vsp, vsp_1 : VScreenPixel \bullet \\
\quad VSPHAdjacent(vsp, vsp_1) \Leftrightarrow \\
\quad PHAdjacent(PixelFromVSP(vsp), PixelFromVSP(vsp_1)) \wedge \\
\quad VSPVAdjacent(vsp, vsp_1) \Leftrightarrow \\
\quad PVAJacent(PixelFromVSP(vsp), PixelFromVSP(vsp_1))
\end{array}$$

A rectangular virtual screen region can now be defined and several functions and relations previously defined on *SimpleRegion* can be reused:

$$\begin{array}{|l}
VScreenRegion \text{ --- } \\
SimpleRegion \\
vsrid : ID \\
rtopleft : VScreenPixel \\
rbotright : VScreenPixel \\
\hline
topleft = PixelFromVSP(rtopleft) \\
botright = PixelFromVSP(rbotright)
\end{array}$$

The simple region component of a virtual screen region can be extracted using the function *SRFromVSR* which is defined as follows:

$$SRFromVSR == (\lambda VScreenRegion \bullet \theta SimpleRegion)$$

The functions and relations that follow are self explanatory in that they are based on the operations specified for a simple region:

$$\begin{array}{|l}
AllVSRPoints : VScreenRegion \rightarrow \mathbb{P} VScreenPixel \\
\hline
\forall vsr : VScreenRegion; vsp : VScreenPixel \bullet \\
\quad vsp \in AllVSRPoints(vsr) \Rightarrow \\
\quad PixelFromVSP(vsp) \in AllSRPoints(SRFromVSR(vsr))
\end{array}$$

$$\begin{aligned}
& \text{SameShapeVSR_} : V\text{ScreenRegion} \leftrightarrow V\text{ScreenRegion} \\
& \text{_VSRContains_} : V\text{ScreenRegion} \leftrightarrow V\text{ScreenRegion} \\
& \text{_VSROverlaps_} : V\text{ScreenRegion} \leftrightarrow V\text{ScreenRegion} \\
& \text{_VSRAbove_} : V\text{ScreenRegion} \leftrightarrow V\text{ScreenRegion} \\
& \text{_VSRLeftOf_} : V\text{ScreenRegion} \leftrightarrow V\text{ScreenRegion}
\end{aligned}$$

$$\begin{aligned}
& \forall v\text{sr}, v\text{sr}_1 : V\text{ScreenRegion} \bullet \\
& \quad \text{SameShapeVSR}(v\text{sr}, v\text{sr}_1) \Leftrightarrow \\
& \quad \quad \text{SameShape}(\text{SRFromVSR}(v\text{sr}), \text{SRFromVSR}(v\text{sr}_1)) \wedge \\
& \quad v\text{sr VSRContains } v\text{sr}_1 \Leftrightarrow \\
& \quad \quad \text{SRFromVSR}(v\text{sr}) \text{ SRContains } \text{SRFromVSR}(v\text{sr}_1) \wedge \\
& \quad v\text{sr VSROverlaps } v\text{sr}_1 \Leftrightarrow \\
& \quad \quad \text{SRFromVSR}(v\text{sr}) \text{ SROverlaps } \text{SRFromVSR}(v\text{sr}_1) \wedge \\
& \quad v\text{sr VSRAbove } v\text{sr}_1 \Leftrightarrow \text{SRFromVSR}(v\text{sr}) \text{ SRAbove } \text{SRFromVSR}(v\text{sr}_1) \wedge \\
& \quad v\text{sr VSRLeftOf } v\text{sr}_1 \Leftrightarrow \text{SRFromVSR}(v\text{sr}) \text{ SRLeftOf } \text{SRFromVSR}(v\text{sr}_1)
\end{aligned}$$

$$\begin{aligned}
& \text{VSRHAdjacentLine_} : \mathbb{P}(\text{seq } V\text{ScreenRegion}) \\
& \text{VSRVAdjacentLine_} : \mathbb{P}(\text{seq } V\text{ScreenRegion})
\end{aligned}$$

$$\begin{aligned}
& \forall \text{svsr} : \text{seq } V\text{ScreenRegion} \bullet \\
& \quad \text{VSRHAdjacentLine}(\text{svsr}) \Leftrightarrow \\
& \quad \quad (\exists \text{ssr} : \text{seq } \text{SimpleRegion} \bullet \\
& \quad \quad \quad \# \text{ssr} = \# \text{svsr} \wedge \\
& \quad \quad \quad (\forall i : 1 \dots \# \text{ssr} \bullet \\
& \quad \quad \quad \quad \text{ssr}(i) = \text{SRFromVSR}(\text{svsr}(i))) \wedge \\
& \quad \quad \quad \text{SRHAdjacentLine}(\text{ssr})) \wedge \\
& \quad \text{VSRVAdjacentLine}(\text{svsr}) \Leftrightarrow \\
& \quad \quad (\exists \text{ssr} : \text{seq } \text{SimpleRegion} \bullet \\
& \quad \quad \quad \# \text{ssr} = \# \text{svsr} \wedge \\
& \quad \quad \quad (\forall i : 1 \dots \# \text{ssr} \bullet \\
& \quad \quad \quad \quad \text{ssr}(i) = \text{SRFromVSR}(\text{svsr}(i))) \wedge \\
& \quad \quad \quad \text{SRVAdjacentLine}(\text{ssr}))
\end{aligned}$$

$$\text{VSRAdjacentLine_} : \mathbb{P}(\text{seq } V\text{ScreenRegion} \times \text{lineorient})$$

$$\begin{aligned}
& \forall \text{svsr} : \text{seq } V\text{ScreenRegion}; \text{ori} : \text{lineorient} \bullet \\
& \quad (\exists \text{ssr} : \text{seq } \text{SimpleRegion} \bullet \\
& \quad \quad \# \text{ssr} = \# \text{svsr} \wedge \\
& \quad \quad (\forall i : 1 \dots \# \text{ssr} \bullet \\
& \quad \quad \quad \text{ssr}(i) = \text{SRFromVSR}(\text{svsr}(i))) \wedge \\
& \quad \quad \text{VSRAdjacentLine}(\text{svsr}, \text{ori}) \Leftrightarrow \text{SRAdjacentLine}(\text{ssr}, \text{ori}))
\end{aligned}$$

Finally, operations on a virtual screen region can be specified in which operations for a simple region are reused. An operation on *VScreenRegion* cannot change its identifier:

$OpOnVScreenRegion$ $\Delta VScreenRegion$ $OpOnSimpleRegion$
$vsrid' = vsrid$

Enlarging, shrinking or moving a virtual screen region is equivalent to enlarging, shrinking or moving its *SimpleRegion* component:

$EnlargeVSR$ $OpOnVScreenRegion$ $vspixel? : VScreenPixel$ $direction? : direction$
$\exists pixel : Pixel \bullet$ $pixel = PixelFromVSP(vspixel?) \wedge$ $EnlargeSimpleRegion[pixel/pixel?]$

$ShrinkVSR$ $OpOnVScreenRegion$ $vspixel? : VScreenPixel$ $direction? : direction$
$\exists pixel : Pixel \bullet$ $pixel = PixelFromVSP(vspixel?) \wedge$ $ShrinkSimpleRegion[pixel/pixel?]$

$MoveVSR$ $OpOnVScreenRegion$ $vspixel? : VScreenPixel$ $direction? : direction$
$\exists pixel : Pixel; newx, newy : \mathbb{N} \bullet$ $pixel = PixelFromVSP(vspixel?) \wedge$ $MoveSimpleRegion[pixel/pixel?]$

3.4 Summary

The specifications introduced in this chapter define pixels and simple regions, and from these virtual pixels and virtual screen regions were defined. The latter two types will be used extensively in this thesis as the basis by which visual characteristics of

GUI objects are defined. That is, every GUI object that may be visible to the user of the GUI-based system will have its visual characteristics defined in terms of virtual screen pixels and virtual screen regions.

Chapter 4

The Mouse and Its Behaviour

The mouse is the primary device used for interacting with a graphical user interface. Portions of an application's interface can be manipulated on the display by using a combination of mouse movement and mouse button manipulation. For example, if the user points the mouse cursor at a push button control on the screen and clicks the left mouse button, a new window may appear prompting the user for relevant information. A system supporting this type of user interaction is often referred to as an *event-driven* system.

Mouse actions are temporal in the sense that the mouse's state changes over time. Depending on the type of change and the time interval between these changes, different actions can be performed. A common example is the distinction between a mouse "double-click" and two "single-click" actions; the action interpreted by the interface is a function of the precise length of time between the click actions, and the timing specifications as defined in the system. In order to examine the specifications of a GUI, it is necessary to discuss the mouse, its actions and its temporal properties.

4.1 Time

The Z specification language does not have an inherent capacity for modeling time as in temporal and real-time specification languages such as CSP [8]. Although rigorous specifications of time have been proposed using the Z language [11], the specifications in this thesis require only a simple formal definition.

Formally, time can be viewed as a sequence of distinct "instances", where the instant at the head of the sequence is the most recent. In this model, it is important

to assume that a new instant is added to the sequence on a regular and constant basis. An instant is formally defined as a basic type:

$[INSTANT]$

Time is a type corresponding to a sequence of distinct instances, as asserted by:

$$\begin{aligned} time &== \text{seq } INSTANT \\ \forall t : time \bullet \#t &= \#(\text{ran } t) \end{aligned}$$

The last predicate ensures that the length of the timing sequence is the same as the cardinality of the set of instances found in that sequence. This, in turn, asserts that *time* is composed of distinct instances. Note that the $\#$ operator is overloaded in Z .

For simplicity, the entire session of a GUI is modeled within one large interval, represented by the global variable *globaltime*:

$$\mid \quad globaltime : time$$

Two instances in *globaltime* can be compared using the **after** relation to determine the order in which the instances occurred:

$$\left| \begin{array}{l} \text{_ after _} : INSTANT \leftrightarrow INSTANT \\ \hline \forall i, i_1 : INSTANT \bullet \\ \quad i \text{ after } i_1 \Leftrightarrow \\ \quad \quad globaltime^{\sim}(i) < globaltime^{\sim}(i_1) \end{array} \right|$$

The notation $R^{\sim}$ in Z refers to relational inverse. That is, $R^{\sim}$ stands for a relation whose domain is the range of R and whose range is the domain of R . Moreover, a sequence in Z is also treated as a function from its index (modeled as $\mathbb{N}_1$) to the type of elements of the sequence. Hence, in Z :

$$\text{seq } INSTANT \equiv \mathbb{N}_1 \rightarrow INSTANT$$

Using this concept, $globaltime^{\sim}$ in the above predicate refers to the relation whose domain is the set of instances of *globaltime* and $globaltime^{\sim}(i)$ refers to one particular element of this relation.

The *interval* between instances of time is a *duration* and is computed as the number of intervals between two given instances:

$duration == \mathbb{N}_1$

$$\begin{array}{|l} Interval : (INSTANT \times INSTANT) \leftrightarrow duration \\ \hline \forall i, i_1 : INSTANT; int : duration \bullet \\ \quad Interval(i, i_1) = int \Rightarrow \\ \quad \quad (i_1 \text{ after } i \wedge \\ \quad \quad int = globaltime^\sim(i_1) - globaltime^\sim(i)) \end{array}$$

4.2 The Mouse

A mouse allows a GUI user to *point* at a location on the screen with a *mouse cursor*, to move the mouse cursor on the screen by moving the mouse, and to perform actions on the object to which the cursor is pointing. The action is controlled by the way in which the user manipulates the mouse button or buttons, the simultaneous movement of the mouse, and the properties of the object itself.

Contemporary mouse design provides up to three buttons

$ButtonPosition ::= LeftButton \mid CentreButton \mid RightButton$

each of which can be depressed. Thus, each button has two states at any instant

$MouseButtonState ::= down \mid up$

and so a mouse button can be represented as a function between its position and its state

$MouseButtons == ButtonPosition \leftrightarrow MouseButtonState$

A mouse's cursor acts as an on-screen pointer that moves as the user moves the mouse. It is assumed that the cursor always points at exactly one pixel. Although the cursor is defined in terms of virtual screen pixels, the valid locations must be restricted to pixels actually displayed on the physical screen. These will be discussed further in Chapter 7. The X and Y axis limits for a physical display are given as global variables:

$$\begin{array}{|l} displayXmax : \mathbb{N}_1 \\ displayYmax : \mathbb{N}_1 \end{array}$$

A mouse has up to three buttons and a cursor that is limited to pointing to screen pixels on the physical display, as indicated by the following schema:

<i>Mouse</i>	
<i>buttons</i> : <i>MouseButtons</i>	
<i>cursor</i> : <i>VScreenPixel</i>	
<i>cursor.x</i> ≤ <i>displayXmax</i>	
<i>cursor.y</i> ≤ <i>displayYmax</i>	

The state of a mouse at any time is captured by augmenting *Mouse* with an instant:

<i>MouseState</i>	
<i>Mouse</i>	
<i>t</i> : <i>INSTANT</i>	
⟨ <i>t</i> ⟩ in <i>globaltime</i>	

The predicate part asserts that ⟨*t*⟩ is a subsequence of *globaltime*; in other words, the instant *t* has occurred somewhere in *globaltime*.

A history of mouse states is required to determine a mouse's action in some cases. The global variable *mousehistory* retains a history of *MouseState* instances for each instant of time in *globaltime*:

<i>mousehistory</i> : seq <i>MouseState</i>
<i>mousehistory</i> = # <i>globaltime</i> ∧
(∀ <i>i</i> : 1 .. # <i>globaltime</i> •
(<i>mousehistory</i> (<i>i</i>)). <i>t</i> = <i>globaltime</i> (<i>i</i>))

The most recent state of the mouse can be specified using the following:

$$\text{mousenow} == \text{head}(\text{mousehistory})$$

Next, a schema is specified which asserts that for any mouse operation, the final state of the given *MouseState* type should occur after its initial state in *mousehistory*:

<i>OpOnMouseState</i>	
Δ <i>MouseState</i>	
<i>t'</i> after <i>t</i>	
dom <i>buttons'</i> = dom <i>buttons</i>	
⟨θ <i>MouseState</i> ⟩ in <i>mousehistory</i>	
⟨θ <i>MouseState'</i> ⟩ in <i>mousehistory</i>	

For notational convenience, the following axioms are provided to confirm the state of a given mouse button:

$$\begin{array}{|l}
 \text{ButtonUp}_- : \mathbb{P}(\text{MouseState} \times \text{ButtonPosition}) \\
 \text{ButtonDown}_- : \mathbb{P}(\text{MouseState} \times \text{ButtonPosition}) \\
 \hline
 \forall ms : \text{MouseState}; bp : \text{ButtonPosition} \bullet \\
 \quad \text{ButtonUp}(ms, bp) \Leftrightarrow ms.buttons(bp) = up \wedge \\
 \quad \text{ButtonDown}(ms, bp) \Leftrightarrow ms.buttons(bp) = down
 \end{array}$$

Mouse operations are defined as follows: if a mouse button is depressed and released within a short period of time, the operation performed is regarded as a mouse button *click*. However, if the button is not released soon enough, the mouse button is *held down*. The difference between these two operations is the duration of time between the mouse button being pressed and being released. Similarly, a *double click* of a mouse button occurs when two button clicks are performed in rapid succession, and the duration between these two clicks is shorter than a predetermined length of time. It can be seen that the distinction between clicking and holding down a mouse button, and between double clicking and clicking a mouse button twice is based on the timing between actions.

Formally, a mouse button is clicked if the length of time between its depression and release is less than that given by the global variable *clickspeed*. If the length of time is greater than that given by *clickspeed*, the mouse button is being held down:

$$| \text{clickspeed} : \text{duration}$$

To single click a mouse button requires the state of the button to change from “up” to “down” and back to “up” within the interval required by the *clickspeed* variable:

$$\begin{array}{|l}
 \text{Click} \\
 \hline
 \text{OpOnMouseState} \\
 \text{posn?} : \text{ButtonPosition} \\
 \hline
 \exists ms : \text{MouseState} \bullet \\
 \quad \langle ms \rangle \text{ in mousehistory} \wedge \\
 \quad \text{ButtonUp}(\theta \text{MouseState}, \text{posn?}) \wedge \\
 \quad \text{ButtonDown}(ms, \text{posn?}) \wedge \\
 \quad \text{ButtonUp}(\theta \text{MouseState}', \text{posn?}) \wedge \\
 \quad t' \text{ after } ms.t \wedge \\
 \quad ms.t \text{ after } t \wedge \\
 \quad \text{Interval}(t, t') \leq \text{clickspeed}
 \end{array}$$

The mouse button being clicked is indicated by the *posn?* input variable. *MouseState* is the state of the mouse the instant before the mouse button is depressed, and *MouseState'* is the state of the mouse the instant after the mouse button is released. The existentially qualified variable *ms* used in the predicate part represents the state of the mouse at the instant when the mouse button is “down”. These three states must occur in the proper order, and the interval between the first and last states must be less than or equal to the specified minimum.

Similarly, the mouse button is held down if its position changes from “up” to “down” and is held down for a period of time greater than that implied by *clickspeed*:

<i>HeldDown</i>
<i>OpOnMouseState</i>
<i>posn?</i> : <i>ButtonPosition</i>
<i>ButtonUp</i> (θ <i>MouseState</i> , <i>posn?</i>)
<i>ButtonDown</i> (θ <i>MouseState'</i> , <i>posn?</i>)
<i>Interval</i> (<i>t</i> , <i>t'</i>) > <i>clickspeed</i>

Double clicking a mouse button is really two successive clicks occurring within a given length of time; the global variable *doubleclickspeed* indicates this value:

| *doubleclickspeed* : *duration*

$$DblClick \cong Click ; Click \wedge [OpOnMouseState \mid Interval(t, t') \leq doubleclickspeed]$$

The double click operation is defined using schema composition between two *Click* schema operations, with an additional condition indicating that the interval between clicks must be less than that indicated by *doubleclickspeed*.

A mouse’s presence is represented on the display by its cursor, which always points to some virtual screen pixel. This association between the mouse and its cursor requires that the cursor can be moved on the screen only when the mouse is physically moved. This is represented by the following schemas:

<i>MouseMoveCursor</i>
<i>OpOnMouseState</i>
<i>cursor</i> $\neq$ <i>cursor'</i>

$$\text{MouseStaticCursor} \triangleq \neg \text{MouseMoveCursor}$$

An object on the display occupies an area bounded by a virtual screen region. This will be discussed further in the next chapter. To click, double click, or hold down the mouse button while the mouse cursor rests in some virtual screen region requires that the cursor remain static within the region while the button operation is performed. Given a virtual screen region, the mouse button, and the mouse state occurring immediately after the operation, these operations can be formally specified as follows:

$\begin{aligned} &\text{MouseClickedObject}_- : \mathbb{P}(V\text{ScreenRegion} \times \text{ButtonPosition} \times \text{MouseState}) \\ &\forall \text{vsr} : V\text{ScreenRegion}; \text{posn} : \text{ButtonPosition}; \text{ms} : \text{MouseState} \bullet \\ &\quad \text{MouseClickedObject}(\text{vsr}, \text{posn}, \text{ms}) \Leftrightarrow \\ &\quad (\exists \text{OpOnMouseState} \bullet \\ &\quad \quad \text{ms} = \theta \text{MouseState}' \wedge \\ &\quad \quad \text{Click}[\text{posn}/\text{posn?}] \wedge \\ &\quad \quad \text{MouseStaticCursor} \wedge \\ &\quad \quad \text{cursor} \in \text{AllVSRPoints}(\text{vsr})) \end{aligned}$
--

$\begin{aligned} &\text{MouseHoldDownObject}_- : \mathbb{P}(V\text{ScreenRegion} \times \text{ButtonPosition} \times \text{MouseState}) \\ &\forall \text{vsr} : V\text{ScreenRegion}; \text{posn} : \text{ButtonPosition}; \text{ms} : \text{MouseState} \bullet \\ &\quad \text{MouseHoldDownObject}(\text{vsr}, \text{posn}, \text{ms}) \Leftrightarrow \\ &\quad (\exists \text{OpOnMouseState} \bullet \\ &\quad \quad \text{ms} = \theta \text{MouseState}' \wedge \\ &\quad \quad \text{HeldDown}[\text{posn}/\text{posn?}] \wedge \\ &\quad \quad \text{MouseStaticCursor} \wedge \\ &\quad \quad \text{cursor} \in \text{AllVSRPoints}(\text{vsr})) \end{aligned}$

$\begin{aligned} &\text{MouseDbClickObject}_- : \mathbb{P}(V\text{ScreenRegion} \times \text{ButtonPosition} \times \text{MouseState}) \\ &\forall \text{vsr} : V\text{ScreenRegion}; \text{posn} : \text{ButtonPosition}; \text{ms} : \text{MouseState} \bullet \\ &\quad \text{MouseDbClickObject}(\text{vsr}, \text{posn}, \text{ms}) \Leftrightarrow \\ &\quad (\exists \text{OpOnMouseState} \bullet \\ &\quad \quad \text{ms} = \theta \text{MouseState}' \wedge \\ &\quad \quad \text{DbClick}[\text{posn}/\text{posn?}] \wedge \\ &\quad \quad \text{MouseStaticCursor} \wedge \\ &\quad \quad \text{cursor} \in \text{AllVSRPoints}(\text{vsr})) \end{aligned}$
--

Formally, these operations are represented as prefix relations. Using `MouseDbClickObject` as an example, an instance of `OpOnMouseState` is created where the final

state is equal to the given mouse state, during which the given mouse button is double clicked (as indicated by the *DblClick* schema operation). The mouse cursor must be static as indicated by the *MouseStaticCursor* schema and must be located within the specified virtual screen region.

Many GUI-based application support *drag and drop* actions. This action is performed by pointing at an object on the display, holding down the mouse button while moving the mouse to a new position, and then releasing the mouse button. Such an application can interpret this action in a number of ways which are application specific. For example, an application might redisplay the information contained within the object in the position to which it was dragged. Since the actual result of this operation is application dependent, only simple aspects are formalized by the following global function:

$$\begin{array}{l}
 \text{MouseDown_Drop} : V\text{ScreenRegion} \times \text{ButtonPosition} \leftrightarrow \\
 \quad (V\text{ScreenRegion} \times \text{MouseState}) \\
 \hline
 \forall \text{vsr}, \text{vsr}_1 : V\text{ScreenRegion}; \text{posn} : \text{ButtonPosition}; \text{ms} : \text{MouseState} \bullet \\
 \quad \text{MouseDown_Drop}(\text{vsr}, \text{posn}) = (\text{vsr}_1, \text{ms}) \Rightarrow \\
 \quad (\exists \text{OpOnMouseState}; \text{OpOnVScreenRegion}; \text{ms}_1 : \text{MouseState} \bullet \\
 \quad \quad \text{ms} = \theta \text{MouseState}' \wedge \\
 \quad \quad \text{ms}_1.t \text{ after } t \wedge \\
 \quad \quad t' \text{ after } \text{ms}_1.t \wedge \\
 \quad \quad \text{vsr} = \theta V\text{ScreenRegion} \wedge \\
 \quad \quad \text{vsr}_1 = \theta V\text{ScreenRegion}' \wedge \\
 \quad \quad \text{MouseHoldDownObject}(\text{vsr}, \text{posn}, \text{ms}_1) \wedge \\
 \quad \quad \text{MouseMoveCursor} \wedge \\
 \quad \quad \text{ButtonUp}(\theta \text{MouseState}', \text{posn}) \wedge \\
 \quad \quad (\exists \text{vspixel} : \text{AllVSRPoints}(\text{vsr}_1); \text{dir} : \text{direction} \bullet \\
 \quad \quad \quad \text{MoveVSR}[\text{vspixel}/\text{vspixel}', \text{dir}/\text{direction}']) \wedge \\
 \quad \quad \text{SameShapeVSR}(\theta V\text{ScreenRegion}, \theta V\text{ScreenRegion}') \wedge \\
 \quad \quad \text{cursor} - \text{rtopleft} = \text{cursor}' - \text{rtopleft}')
 \end{array}$$

Given a virtual screen region and a mouse button, this function returns a virtual screen region bounding the object in its new position and the final mouse state. Since the operation changes the mouse state and the state of a virtual screen region it is necessary to include *OpOnMouseState* and *OpOnVScreenRegion* in the axiom so that all variables in *Mouse* and *VScreenRegion* are in scope. The predicates in the function assert the following: i) at some intermediate mouse state ms_1 , the mouse's button is held down on an object; ii) in the final mouse state, the mouse button must

be in the “up” position; iii) the mouse cursor must be moved between the first and final mouse states; iv) the virtual screen region is moved between the first and final states to a position indicated by the position of the mouse cursor in its final state; v) the cursor must be at the same offset in the virtual screen region before and after it has moved; vi) the virtual screen region retains its original shape after it’s moved; and vii) the virtual screen region is moved to the new location.

The `MouseClickedObject`, `MouseHoldDownObject`, `MouseDownClickObject` and `MouseDrag_n_Drop` functions will be used in this thesis extensively. If a visual object has an operation that involves a mouse action, the specifications given in this chapter will be incorporated in the new specifications to illustrate the required action of the mouse.

Chapter 5

Controls

Most user interaction with a GUI-based application is managed through a set of *controls*. These controls are usually consistent across all applications using the same GUI, which is valuable since a consistent “look and feel” is attained, and the advantages of reduced training time and increased usability are realized. Additionally, software developers need only concern themselves with accessing the GUI’s interface routines rather than writing interface routines from scratch. It is interesting to note that there are a handful of primitive controls common to almost all GUIs currently available. Therefore, it is sufficient to specify these primitive controls from which the behaviour of a generic GUI can be inferred.

In this chapter, the different types of user interface controls for the GUI under development are specified. These controls can be divided into four categories: the first type, *primitive controls*, are those controls that are atomic in nature; that is, they are not constructed using other controls as components. The second type, *composite controls*, may have both primitive and composite controls as properties. A *scroll region control* is the third type. It is a special type of composite control whose behaviour is especially relevant for the CASE tool’s GUI, and for this reason, it is treated separately. Finally, there are three varieties of *menu controls*, each of which is discussed in turn.

The specifications in this chapter are partitioned into three major categories: representational abstraction of the controls, visual aspects of these controls, and behavioural abstraction. Since Z supports a modular specification approach, it is easy to separate behavioural abstraction from visual representation. Moreover, visual rep-

resentation of controls may not be of interest to others who wish to reuse these specifications.

5.1 Representational Abstraction of Controls

For each of the four types of controls mentioned earlier, representational abstraction includes the abstract structure and invariants of the state space.

5.1.1 Primitive Controls

There are three types of primitive controls discussed in this chapter - *push buttons*, *check boxes* and *radio button groups*.

Push Button control Generally, a push button control is represented as a rectangle on the screen and serves as a trigger for some action to take place. It can be enabled or disabled. When enabled, clicking on the push button causes some pre-defined action to be invoked. Since virtually any action can be associated with the clicking of a push button, only its enabling mechanism is specified:

<i>PushButton</i> <i>enabled : Boolean</i>

Check Box control A check box control is used to toggle a boolean value whose purpose is application-specific. It is always associated with a text label. The check box takes on different appearances depending on the value of the variable to which the control is associated. A check box control allows the value of the variable to be toggled by clicking on the control with the mouse. Like a push button control, a check box control can be enabled or disabled:

<i>CheckBox</i> <i>label : STRING</i> <i>selected : Boolean</i> <i>enabled : Boolean</i>

Radio Button Group control A radio button has almost the same functionality as a check box control, except it always appears in a group with other radio buttons.

Only one radio button in a group can be selected at any time. A radio button itself is not considered to be a control since its behaviour and appearance are defined in terms of other radio buttons in the same group. Moreover, a group should always have more than one radio button:

<i>RadioButton</i>	_____
<i>label</i> :	<i>STRING</i>
<i>selected</i> :	<i>Boolean</i>
<i>RadioButtonGroup</i>	_____
<i>rbuttons</i> :	$\mathbb{P}$ <i>RadioButton</i>
<i>enabled</i> :	<i>Boolean</i>
$\exists_1$ <i>rb</i> :	<i>rbuttons</i> •
	<i>rb.selected</i> = <i>Yes</i>
<i>#rbuttons</i>	> 1

5.1.2 Composite Controls

The following composite controls are discussed in this chapter - *scroll bar*, *text edit*, *list box* and *combo box*.

Scroll Bar control A scroll bar control is used to graphically depict the value of some application-specific variable within a range of possible values. It has four components - a *track*, a *thumb*, and two push button controls. The maximum and minimum values are visually represented by the end points of the track, and the position of the thumb within the track represents the variable's current value. Push button controls are located at either end of the track which, when clicked or held down, will advance the thumb towards the respective end of the track. By changing the position of the thumb, the user can change the value of the variable to which the thumb position is dependent. The mapping between the thumb position and the actual value in the range position depends on the resolution of the track. That is, each step of thumb movement may be mapped to a single or a subrange of values.

As an example, a scroll bar may be used to control the scrolling of a document. In this case, the control's range would be defined by the line count of the document where the minimum value is 0 and the maximum value is the line count of the document. The

position of the thumb would indicate visually the subset of the document information that is currently visible. Similarly, a scroll bar control may be used to manage a speaker's volume. The minimum value of the range would represent no activation of the speakers, while the maximum value would represent the loudest volume that the speakers can achieve. The thumb would indicate the current volume with respect to its range.

ScrollBar

minbutton : *PushButton*

maxbutton : *PushButton*

thumb : *VScreenRegion*

rangemin : $\mathbb{N}$

rangemax : $\mathbb{N}$

rangeposn : $\mathbb{N}$

enabled : *Boolean*

$rangemin \leq rangeposn \leq rangemax$

$minbutton.enabled = maxbutton.enabled = enabled$

$rangemin \geq rangemax \Rightarrow$

$enabled = No$

To specify the dependency between the thumb position and the current value in the range requires specifications for the visual aspects of the scroll bar control. These will be provided in the next section.

Text Edit control A text edit control allows for the entry of text on a single line, or spanning over several lines. If the text being manipulated contains more lines than can be displayed, then a scroll bar will be provided as part of the text edit control.

TextEdit

height : $\mathbb{N}_1$
text : seq *STRING*
visibletext : seq *STRING*
vscrollbar : *ScrollBar*
enabled : *Boolean*

visibletext in *text*
 $\#visibletext \leq height$
 $\#visibletext = \#text \Rightarrow$
 $vscrollbar \in \emptyset[ScrollBar]$
 $\#visibletext < \#text \Rightarrow$
 ($vscrollbar.enabled = enabled \wedge$
 $vscrollbar.rangemin = 1 \wedge$
 $vscrollbar.rangemax = \#text \wedge$
 $vscrollbar.rangeposn \in \text{dom } text \wedge$
 $text(vscrollbar.rangeposn) = \text{head}(visibletext)$)

In the specification for *TextEdit*, *text* refers to the sequence of text lines that can be displayed and *visibletext* contains the subsequence of *text* which is currently visible on the screen. The variable *height* limits the number of lines that can be visible. The invariants assert the following: i) *visibletext* is a subsequence of *text* whose cardinality is limited by *height*; ii) if all of *text* is visible, then no scroll bar is necessary (indicated by an empty set of scroll bars in the specification); and iii) the scroll bar's range and thumb position control the visibility of text, when *visibletext* is a proper subsequence of *text*. It is assumed here that the scroll bar's range is from 1 to the length of *text*.

List Box control A list box control is a scrollable list that presents lines of text for browsing or selection by the user. A line is selected by clicking on it with the mouse. This control might be used, for example, to select one or more file names from a list of files for deletion. Depending on the characteristics and intended purpose of the control, several selections may be made at one time, or the user may be restricted to selecting at most one item. Each item that can appear in a list box control may be represented formally as:

ListItem

text : *STRING*
selected : *Boolean*

Like a *TextEdit* control, the list box contains a sequence of text items, a subsequence of which may be visible at any time. If only a subset is visible, a scroll bar can be used to move through the items.

<i>ListBox</i> <i>height</i> : $\mathbb{N}_1$ <i>items</i> : seq <i>ListItem</i> <i>visibleitems</i> : seq <i>ListItem</i> <i>selecteditems</i> : $\mathbb{P}$ <i>ListItem</i> <i>vscrollbar</i> : <i>ScrollBar</i> <i>multiple</i> : <i>Boolean</i> <i>enabled</i> : <i>Boolean</i> <i>visibleitems</i> in <i>items</i> $\#visibleitems \leq height$ $\#visibleitems = \#items \Rightarrow$ <i>vscrollbar</i> $\in \emptyset[ScrollBar]$ $\#visibleitems < \#items \Rightarrow$ (<i>vscrollbar.enabled</i> = <i>enabled</i> $\wedge$ <i>vscrollbar.rangemin</i> = 1 $\wedge$ <i>vscrollbar.rangemax</i> = $\#items$ $\wedge$ <i>vscrollbar.rangeposn</i> $\in \text{dom } items$ $\wedge$ <i>items</i>(<i>vscrollbar.rangeposn</i>) = <i>head(visibleitems)</i>) <i>selecteditems</i> = {<i>i</i> : ran <i>items</i> <i>i.selected</i> = <i>Yes</i>} <i>multiple</i> = <i>No</i> $\Rightarrow$ $\#selecteditems \leq 1$ $\#items = 0 \Rightarrow$ <i>enabled</i> = <i>No</i>
--

The invariants in *ListBox* are similar to those for the *TextEdit* control with the addition of the following: i) *selecteditems* must contain only those list items that are selected; ii) if multiple items cannot be selected, then *selecteditems* contains at most one item; and iii) the list box is disabled if there are no items to list.

Combo Box controls A combo box control combines a list box control with a text edit control. Generally, when a row in the list box is selected, the text that is associated with the selected list item appears in the text edit region. Since each list item can contain at most one line of text (as specified earlier in *ListBox*), the height of the *TextEdit* region within a combo box is always one. Information appearing in the text edit part may be static and unchangeable, or the user may be able to

alter it directly. This behaviour is set by the boolean variable *static* in the following specification. As with all other controls, a combo box control can be enabled or disabled. There are two flavours of combo boxes; the “simple” type displays a list box at all times, and the “drop down” type displays a list box only when its appearance is requested by the user. This is accomplished by clicking on a special “drop down” push button control associated with a drop down combo box. The specification for the simple combo box control follows:

ComboBoxSimple

textfield : *TextEdit*

picklist : *ListBox*

static : *Boolean*

enabled : *Boolean*

textfield.height = 1

picklist.multiple = *No*

textfield.enabled = *picklist.enabled* = *enabled*

$(static = Yes \wedge \#(picklist.selecteditems) = 0) \Rightarrow$

$head(textfield.text) \in \emptyset[STRING] \wedge$

$\#(picklist.selecteditems) = 1 \Rightarrow$

$(\exists li : picklist.selecteditems \bullet$

$head(textfield.text) = li.text)$

The invariants in *ComboBoxSimple* assert the following: i) the text edit control permits only one line of text; ii) at most one list box control item can be selected; iii) if the simple combo box control is enabled or disabled, the associated text edit and list box controls must also be enabled or disabled; iv) if the *static* property is true and no items are selected, the text displayed is the empty string; and v) if an item is selected from the list box control, the text field displays the text for that item.

A drop down combo box control does not have a static list box - instead, it may be displayed or removed as desired. This control would be used if the functionality of a simple combo box is desired but the screen space necessary to permanently display a list box is not available. Clicking the push button control attached to this control causes a list box to appear if one did not previously exist, or causes the list box that currently exists to vanish:

ComboBoxDrop

textfield : *TextEdit*

picklist : *ListBox*

static : *Boolean*

dropbutton : *PushButton*

enabled : *Boolean*

textfield.height = 1

dropbutton.enabled = *textfield.enabled* = *enabled*

picklist $\notin \emptyset[\textit{ListBox}] \Rightarrow$

(*picklist.enabled* = *enabled* $\wedge$

picklist.multiple = *No* $\wedge$

(*static* = *Yes* $\wedge$ $\#(\textit{picklist.selecteditems}) = 0) \Rightarrow$

$\textit{head}(\textit{textfield.text}) \in \emptyset[\textit{STRING}] \wedge$

$\#(\textit{picklist.selecteditems}) = 1 \Rightarrow$

($\exists li : \textit{picklist.selecteditems} \bullet$

$\textit{head}(\textit{textfield.text}) = li.\textit{text}$)

5.1.3 Scrollable Region Control

A *scrollable region* control is a special type of composite control that enables the user of the GUI to view and navigate through a large amount of visual information. Typically, this information cannot be displayed on the screen at one time because of space restrictions. Navigation through this information is accomplished by using scroll bar controls attached to the scrollable region control. This method of displaying information would be used, for example, in a word processing system where the entire document is too large to be displayed on the screen at one time.

It is assumed that all visual information associated with this control is available in a different coordinate frame called *work area* in which only a portion, called a *work region*, is presented to the user. It is the mapping of this work region to a virtual screen region, and hence to virtual screen pixels, that actually allows work area information to become visible to the user. Manipulation of the associated scroll bars simply changes the position of the work region within the work area.

In addition to the mapping and navigation, the size of the work region may be changed. Characteristics of the mapping between the two coordinate frames may also be controlled through a zooming factor associated with the control.

It is necessary to first introduce specifications for the work area coordinate system,

its pixels, a region in the frame, and the associated operations. Next, a zooming function will be specified which will map information between the coordinate frames. Finally, the scroll region control itself will be specified.

Work Area

In this section, the work pixel and work region types are specified. These specifications are derived from the simple pixel and simple region types discussed earlier. Even though it is a straightforward task to derive the new types, they have to be written explicitly because each schema and function introduced in this section defines a new type or operation. For brevity, the informal descriptions are ignored.

A work area has a finite size that is limited by two global variables:

$$\left| \begin{array}{l} workXmax : \mathbb{N}_1 \\ workYmax : \mathbb{N}_1 \end{array} \right.$$

and a pixel within a work area is defined within these limits:

$\begin{array}{l} \text{WorkPixel} \\ \text{Pixel} \end{array}$
$\begin{array}{l} maxX = workXmax \\ maxY = workYmax \end{array}$

The coordinate frame in which work pixels are defined must be different from that used in the definition of virtual screen pixels. The following asserts these facts:

$$\forall wp : \text{WorkPixel}; vsp : \text{VScreenPixel} \bullet \\ wp.cfid \neq vsp.cfid$$

The variable *cfid* used in the specifications above are in scope through the declaration of *Pixel* in both *WorkPixel* and *VScreenPixel*.

Once again, several axioms defined on pixels can be redefined to operate on work pixels:

$$\text{PixelFromWorkPixel} == (\lambda \text{WorkPixel} \bullet \theta \text{Pixel})$$

$$\begin{aligned}
WPAbove &: WorkPixel \leftrightarrow WorkPixel \\
WPBelow &: WorkPixel \leftrightarrow WorkPixel \\
WPLeftOf &: WorkPixel \leftrightarrow WorkPixel \\
WPRightOf &: WorkPixel \leftrightarrow WorkPixel \\
WPHAdjacent_ &: WorkPixel \leftrightarrow WorkPixel \\
WPVAdjacent_ &: WorkPixel \leftrightarrow WorkPixel
\end{aligned}$$

$$\begin{aligned}
&\forall wp, wp_1 : WorkPixel \bullet \\
&\quad wp \text{ WPAbove } wp_1 \Leftrightarrow \\
&\quad \quad PixelFromWorkPixel(wp) \text{ PAbove } PixelFromWorkPixel(wp_1) \wedge \\
&\quad wp \text{ WPBelow } wp_1 \Leftrightarrow \\
&\quad \quad PixelFromWorkPixel(wp) \text{ PBelow } PixelFromWorkPixel(wp_1) \wedge \\
&\quad wp \text{ WPLeftOf } wp_1 \Leftrightarrow \\
&\quad \quad PixelFromWorkPixel(wp) \text{ PLeftOf } PixelFromWorkPixel(wp_1) \wedge \\
&\quad wp \text{ WPRightOf } wp_1 \Leftrightarrow \\
&\quad \quad PixelFromWorkPixel(wp) \text{ PRightOf } PixelFromWorkPixel(wp_1) \wedge \\
&\quad WPHAdjacent(wp, wp_1) \Leftrightarrow \\
&\quad \quad PHAdjacent(PixelFromWorkPixel(wp), PixelFromWorkPixel(wp_1)) \wedge \\
&\quad WPVAdjacent(wp, wp_1) \Leftrightarrow \\
&\quad \quad PVAdjacent(PixelFromWorkPixel(wp), PixelFromWorkPixel(wp_1))
\end{aligned}$$

$$WPlsSquare_ : \mathbb{P}(\mathbb{P} WorkPixel)$$

$$\begin{aligned}
&\forall pwp : \mathbb{P} WorkPixel \bullet WPlsSquare(pwp) \Leftrightarrow \\
&\quad (\exists f : WorkPixel \multimap Pixel \bullet \\
&\quad \quad \text{dom } f = pwp \wedge \\
&\quad \quad (\forall wp : \text{dom } f \bullet \\
&\quad \quad \quad f(wp) = PixelFromWorkPixel(wp)) \wedge \\
&\quad \quad PlsSquare(\text{ran } f))
\end{aligned}$$

A work region is a rectangular subset of the work area. It is defined in a manner similar to the definition of *VScreenRegion*:

$$\begin{aligned}
&\text{WorkRegion} \\
&\quad SimpleRegion \\
&\quad wrid : ID \\
&\quad wtopleft : WorkPixel \\
&\quad wbotright : WorkPixel \\
&\quad \text{topleft} = PixelFromWorkPixel(wtopleft) \\
&\quad \text{botright} = PixelFromWorkPixel(wbotright)
\end{aligned}$$

Several functions and relations follow which are derived from those defined on the *SimpleRegion* type:

$SRFromWR == (\lambda WorkRegion \bullet \theta SimpleRegion)$

$AllWRPoints : WorkRegion \leftrightarrow \mathbb{P} WorkPixel$
$\forall wr : WorkRegion; wp : WorkPixel \bullet$ $wp \in AllWRPoints(wr) \Rightarrow$ $PixelFromWorkPixel(wp) \in AllSRPoints(SRFromWR(wr))$

$WRHeight : WorkRegion \rightarrow \mathbb{N}$ $WRWidth : WorkRegion \rightarrow \mathbb{N}$
$\forall wr : WorkRegion \bullet$ $WRHeight(wr) = SRHeight(SRFromWR(wr)) \wedge$ $WRWidth(wr) = SRWidth(SRFromWR(wr))$

$_ WRAbove _ : WorkRegion \leftrightarrow WorkRegion$ $_ WRBelow _ : WorkRegion \leftrightarrow WorkRegion$ $_ WRLeftOf _ : WorkRegion \leftrightarrow WorkRegion$ $_ WRRightOf _ : WorkRegion \leftrightarrow WorkRegion$
$\forall wr, wr_1 : WorkRegion \bullet$ $wr WRAbove wr_1 \Leftrightarrow SRFromWR(wr) SRAbove SRFromWR(wr_1) \wedge$ $wr WRBelow wr_1 \Leftrightarrow SRFromWR(wr) SRBelow SRFromWR(wr_1) \wedge$ $wr WRLeftOf wr_1 \Leftrightarrow SRFromWR(wr) SRLeftOf SRFromWR(wr_1) \wedge$ $wr WRRightOf wr_1 \Leftrightarrow SRFromWR(wr) SRRightOf SRFromWR(wr_1)$

$OpOnWorkRegion$
$\Delta WorkRegion$ $OpOnSimpleRegion$
$wrid' = wrid$

$EnlargeWR$
$OpOnWorkRegion$ $wpixel? : WorkPixel$ $direction? : direction$
$\exists pixel : Pixel \bullet$ $pixel = PixelFromWorkPixel(wpixel?) \wedge$ $EnlargeSimpleRegion[pixel/pixel?]$

ShrinkWR

OpOnWorkRegion

wpixel? : *VScreenPixel*

direction? : *direction*

$\exists pixel : Pixel \bullet$

$pixel = PixelFromWorkPixel(wpixel?) \wedge$

$ShrinkSimpleRegion[pixel/pixel?]$

MoveWR

OpOnWorkRegion

wpixel? : *VScreenPixel*

direction? : *direction*

$\exists pixel : Pixel; newx, newy : \mathbb{N} \bullet$

$pixel = PixelFromWorkPixel(wpixel?) \wedge$

$MoveSimpleRegion[pixel/pixel?]$

Zooming Operations

The mapping between work pixels and virtual screen pixels can take three forms: i) each work pixel maps to exactly one virtual screen pixel (a one-to-one mapping); ii) each work pixel maps to a group of virtual screen pixels enabling one to *zoom in* on the image contained within the work area, or iii) a group of work pixels maps to one virtual screen pixel, enabling one to *zoom out* an image.

The mapping process for each of these three alternatives operates on the same premise: the work area contains an image represented by work pixels bounded by a work region; these work pixels may be transformed through a zooming process to create an instance of *SimpleRegion*, and finally the *SimpleRegion* is mapped directly to a virtual screen region. The transitional mapping is necessary because in zooming a set of pixels may be created with coordinate values larger than those that are allowed by the coordinate frame occupied by work region. The final mapping will crop this transitional image to fit into a virtual screen region.

The zooming process is controlled by a *zoom factor* such that a positive value implies the *zooming in* operation where each work region pixel is mapped to a square group of pixels in the transitional *SimpleRegion* having a dimension equal to the value of *zoom factor*. A negative value implies that the *zooming out* operation is performed

where one pixel is the transitional region maps to a square group of work pixels in the original work area having dimension equal to the value of *zoom factor*. A zero value of *zoom factor* implies a one-to-one mapping. Obviously, relative positioning of pixels in the original work region and in the transformed *SimpleRegion* must be maintained.

First, a function called *ScreenView* is specified, which maps a *SimpleRegion* to a *VScreenRegion*:

$$\begin{array}{|l}
 \text{ScreenView} : \text{SimpleRegion} \leftrightarrow \text{VScreenRegion} \\
 \hline
 \forall sr : \text{SimpleRegion}; vsr : \text{VScreenRegion} \bullet \\
 \quad \text{ScreenView}(sr) = vsr \Leftrightarrow \\
 \quad (\exists f : \text{AllSRPoints}(sr) \rightarrow \text{AllVSRPoints}(vsr) \bullet \\
 \quad (\forall p, p_1 : \text{dom } f \bullet \\
 \quad \quad \text{PHAdjacent}(p, p_1) \Leftrightarrow \text{VSPHAdjacent}(f(p), f(p_1)) \wedge \\
 \quad \quad \text{PVAdjacent}(p, p_1) \Leftrightarrow \text{VSPVAdjacent}(f(p), f(p_1))))
 \end{array}$$

The function *ScreenView* asserts that the simple pixels and the virtual screen pixels are associated using a bijective mapping such that the adjacency relationships between every pair of pixels in the simple region are maintained between the respective pixels in the virtual screen region. All pixels in the simple region and virtual screen region participate in the mapping.

Next, zooming functions are specified which map a work region to a simple region. Since there are three cases to consider as outlined above, three separate functions will be provided. All three functions will then be combined to create one general purpose zooming function.

First, the “zooming in” function can be specified as follows:

$$\text{ZoomIn} : \text{WorkRegion} \times \mathbb{Z} \leftrightarrow \text{VScreenRegion}$$

$$\forall wr : \text{WorkRegion}; \text{zoomfactor} : \mathbb{Z}; vsr : \text{VScreenRegion} \bullet$$

$$\text{ZoomIn}(wr, \text{zoomfactor}) = vsr \Leftrightarrow$$

$$(\text{zoomfactor} > 0 \wedge$$

$$(\exists sr : \text{SimpleRegion}; pset : \mathbb{P}(\mathbb{P} \text{Pixel}) \bullet$$

$$\text{ScreenView}(sr) = vsr \wedge$$

$$\bigcup pset = \text{AllSRPoints}(sr) \wedge$$

$$\bigcap pset = \emptyset \wedge$$

$$(\forall ps : pset \bullet$$

$$\text{PlsSquare}(ps) \wedge$$

$$\#ps = \text{Sqr}(\text{zoomfactor})) \wedge$$

$$(\exists f : \text{AllWRPoints}(wr) \rightarrow pset \bullet$$

$$(\forall wp, wp_1 : \text{dom } f \bullet$$

$$(\exists_1 sr_1, sr_2 : \text{SimpleRegion} \bullet$$

$$\text{AllSRPoints}(sr_1) = f(wp) \wedge$$

$$\text{AllSRPoints}(sr_2) = f(wp_1) \wedge$$

$$wp \text{ WPAbove } wp_1 \Leftrightarrow sr_1 \text{ SRAbove } sr_2 \wedge$$

$$wp \text{ WPBelow } wp_1 \Leftrightarrow sr_1 \text{ SRBelow } sr_2 \wedge$$

$$wp \text{ WPLeftOf } wp_1 \Leftrightarrow sr_1 \text{ SRLeftOf } sr_2 \wedge$$

$$wp \text{ WPRightOf } wp_1 \Leftrightarrow sr_1 \text{ SRRightOf } sr_2))))))$$

The specification for the *ZoomIn* function is written to establish a mapping between the work pixels and the simple pixels. Every work pixel is mapped to a square region of simple pixels whose dimension is equal to *zoomfactor*. There exists a bijection between the work pixels and the square sets of simple pixels such that adjacency relationships between every pair of work pixels is preserved between the respective square sets of simple pixels. By the property of bijection, the mapping becomes one-to-one. The other two cases can be specified in a similar manner:

ZoomOut : *WorkRegion* $\times \mathbb{Z} \leftrightarrow$ *VScreenRegion*

$\forall wr : \text{WorkRegion}; \text{zoomfactor} : \mathbb{Z}; \text{vsr} : \text{VScreenRegion} \bullet$
 $\text{ZoomOut}(wr, \text{zoomfactor}) = \text{vsr} \Leftrightarrow$
 $(\text{zoomfactor} < 0 \wedge$
 $(\exists sr : \text{SimpleRegion}; \text{wpset} : \mathbb{P}(\mathbb{P} \text{WorkPixel}) \bullet$
 $\text{ScreenView}(sr) = \text{vsr} \wedge$
 $\bigcup \text{wpset} = \text{AllWRPoints}(wr) \wedge$
 $\bigcap \text{wpset} = \emptyset \wedge$
 $(\forall \text{wps} : \text{wpset} \bullet$
 $\text{WPIsSquare}(\text{wps}) \wedge$
 $\# \text{wps} = \text{Sqr}(\text{zoomfactor})) \wedge$
 $(\exists f : \text{wpset} \rightarrow \text{AllSRPoints}(sr) \bullet$
 $(\forall p, p_1 : \text{ran } f \bullet$
 $(\exists_1 wr_1, wr_2 : \text{WorkRegion} \bullet$
 $\text{AllWRPoints}(wr_1) = f^{-1}(p) \wedge$
 $\text{AllWRPoints}(wr_2) = f^{-1}(p_1) \wedge$
 $p \text{ PAbove } p_1 \Leftrightarrow wr_1 \text{ WRAbove } wr_2 \wedge$
 $p \text{ PBelow } p_1 \Leftrightarrow wr_1 \text{ WRBelow } wr_2 \wedge$
 $p \text{ PLeftOf } p_1 \Leftrightarrow wr_1 \text{ WRLeftOf } wr_2 \wedge$
 $p \text{ PRightOf } p_1 \Leftrightarrow wr_1 \text{ WRRightOf } wr_2))))))$

A one-to-one mapping between work area pixels and virtual screen pixels is specified by the following:

NoZoom : *WorkRegion* $\times \mathbb{Z} \leftrightarrow$ *VScreenRegion*

$\forall wr : \text{WorkRegion}; \text{zoomfactor} : \mathbb{Z}; \text{vsr} : \text{VScreenRegion} \bullet$
 $\text{NoZoom}(wr, \text{zoomfactor}) = \text{vsr} \Leftrightarrow$
 $(\text{zoomfactor} = 0 \wedge$
 $(\exists sr : \text{SimpleRegion} \bullet$
 $\text{ScreenView}(sr) = \text{vsr} \wedge$
 $(\exists f : \text{AllWRPoints}(wr) \rightarrow \text{AllSRPoints}(sr) \bullet$
 $(\forall p, p_1 : \text{ran } f \bullet$
 $(\exists_1 \text{wp}, \text{wp}_1 : \text{WorkPixel} \bullet$
 $f(\text{wp}) = p \wedge$
 $f(\text{wp}_1) = p_1 \wedge$
 $p \text{ PAbove } p_1 \Leftrightarrow \text{wp} \text{ WPAbove } \text{wp}_1 \wedge$
 $p \text{ PBelow } p_1 \Leftrightarrow \text{wp} \text{ WPBelow } \text{wp}_1 \wedge$
 $p \text{ PLeftOf } p_1 \Leftrightarrow \text{wp} \text{ WPLeftOf } \text{wp}_1 \wedge$
 $p \text{ PRightOf } p_1 \Leftrightarrow \text{wp} \text{ WPRightOf } \text{wp}_1))))))$

A generalized zoom function can be written by combining the *ZoomIn*, *ZoomOut* and *NoZoom* functions:

$$\text{Zoom} : \text{WorkRegion} \times \mathbb{Z} \rightarrow \text{VScreenRegion}$$

$$\forall wr : \text{WorkRegion}; \text{zoomfactor} : \mathbb{Z}; vsr : \text{VScreenRegion} \bullet$$

$$\begin{aligned} &\text{Zoom}(wr, \text{zoomfactor}) = vsr \Rightarrow \\ &\quad (\text{ZoomIn}(wr, \text{zoomfactor}) = vsr \vee \\ &\quad \text{ZoomOut}(wr, \text{zoomfactor}) = vsr \vee \\ &\quad \text{NoZoom}(wr, \text{zoomfactor}) = vsr) \end{aligned}$$

Finally, the specification for a scroll region control can be given, which uses the *Zoom* function to relate information in the work area to the virtual screen. It includes a zooming factor, the work region within the work area, and the virtual screen region. It also provides scroll bars to allow the user to maneuver through the work area. The scroll bars may be disabled if the entire work area can be mapped to the virtual screen region.

$$\text{ScrollRegion}$$

$$\text{viewportVSR} : \text{VScreenRegion}$$

$$\text{viewportWR} : \text{WorkRegion}$$

$$\text{hscrollbar} : \text{ScrollBar}$$

$$\text{vscrollbar} : \text{ScrollBar}$$

$$\text{zoomfactor} : \mathbb{Z}$$

$$\text{enabled} : \text{Boolean}$$

$$\text{Zoom}(\text{viewportWR}, \text{zoomfactor}) = \text{viewportVSR}$$

$$\text{WRWidth}(\text{viewportWR}) = \text{workXmax} \Leftrightarrow \text{hscrollbar} \in \emptyset[\text{ScrollBar}]$$

$$\text{WRHeight}(\text{viewportWR}) = \text{workYmax} \Leftrightarrow \text{vscrollbar} \in \emptyset[\text{ScrollBar}]$$

$$\text{hscrollbar} \notin \emptyset[\text{ScrollBar}] \Rightarrow$$

$$(\text{hscrollbar.rangemin} = 1 \wedge$$

$$\text{hscrollbar.rangemax} = \text{workXmax} \wedge$$

$$\text{hscrollbar.rangeposn} = \text{viewportWR.wtopleft.x} \wedge$$

$$\text{hscrollbar.enabled} = \text{enabled})$$

$$\text{vscrollbar} \notin \emptyset[\text{ScrollBar}] \Rightarrow$$

$$(\text{vscrollbar.rangemin} = 1 \wedge$$

$$\text{vscrollbar.rangemax} = \text{workYmax} \wedge$$

$$\text{vscrollbar.rangeposn} = \text{viewportWR.wtopleft.y} \wedge$$

$$\text{vscrollbar.enabled} = \text{enabled})$$

The variables *viewportVSR* and *viewportWR* represent the virtual screen region and the work region respectively.

5.1.4 Menus

In a GUI environment, a *menu* provides a variety of selections or *menu items* from which the user can choose a path of execution. There are a number of different menu types to choose from, depending on the desired characteristics of the menu. The first, a *popup menu*, may temporarily appear anywhere on the screen, and is simply a list of currently available options that may be selected. It disappears when the user no longer requires it, or when an option is selected. The second type, a *menu bar*, appears as a horizontal list of high level descriptions of actions that may be selected. It usually appears fixed under the caption bar of a window. Generally, only the main window of an application will have a menu bar associated with it. The third type, a *drop down menu*, is similar to a popup menu except that its menu items may invoke additional drop down menus, so that several drop down menus may be active simultaneously. The initial drop down menu can only be created by selecting a specific type of menu item from a menu bar. Consequently, drop down menus are actually components of the menu bar control.

Menu items may perform four broad types of actions when selected. They are: i) to initiate the display of a new drop down menu; ii) to toggle an application specific value; iii) to initiate the display of a window; or iv) to invoke some application specific behaviour, such as shutting down the application. Points one and two will be discussed in detail in this section. Point three will be discussed in Chapter 7 where multiple windows in a display are discussed. Point four is beyond the scope of this thesis. The characteristics of each type of menu item vary; however, all types have two properties in common - a text label and a means of enabling or disabling the item. Specifications of the menu item types follow:

InvokeWindowMenuItem _____

label : *STRING*

enabled : *Boolean*

windowid : *ID*

The variable *windowid* refers to a unique identifier for the window to be displayed when the menu item is selected.

InvokeDropDownMenuItem _____

label : *STRING*
enabled : *Boolean*
ddmid : *ID*

The variable *ddmid* refers to an identifier for the drop down menu to be displayed when the menu item is selected.

ExecCommandMenuItem _____

label : *STRING*
enabled : *Boolean*

ToggleValueMenuItem _____

label : *STRING*
enabled : *Boolean*
applvar : *Boolean*

The variable *applvar* is an application-specific boolean variable that may be toggled.

These menu item types can be combined into one schema type using Hall's style. This is useful for clarity of the specifications and provides a means by which the object-oriented "isa" relationship can be expressed. First, using schema disjunction, a new intermediate schema type is created:

$$Item \cong InvokeWindowMenuItem \vee InvokeDropDownMenuItem \vee \\ ExecCommandMenuItem \vee ToggleValueMenuItem$$

Next, a global identifier is required, with a value describing each menu item type:

$$MenuItemType ::= IsaInvokeWindowMenuItem \mid \\ IsaInvokeDropDownMenuItem \mid \\ IsaExecCommandMenuItem \mid \\ IsaToggleValueMenuItem$$

Finally, an abstract menu item can be specified by the following:

MenuItem _____

miid : *ID*
Item
type : *MenuItemType*

type = *IsaInvokeWindowMenuItem* $\wedge$ *InvokeWindowMenuItem* $\vee$
type = *IsaInvokeDropDownMenuItem* $\wedge$ *InvokeDropDownMenuItem* $\vee$
type = *IsaExecCommandMenuItem* $\wedge$ *ExecCommandMenuItem* $\vee$
type = *IsaToggleValueMenuItem* $\wedge$ *ToggleValueMenuItem*

An abstract menu contains a non-empty series of unique menu items which appear in a specific order:

<i>Menu</i>
<i>menuitems</i> : $ID \leftrightarrow MenuItem$
<i>disorder</i> : $seq\ ID$
$dom\ menuitems = \{m : MenuItem \mid m \in \text{ran } menuitems \bullet m.miid\}$
$dom\ menuitems = \text{ran } disorder$
$\#menuitems > 0$

In the specification *Menu*, the variable *menuitems* refers to a set of menu items uniquely identified by the domain. The variable *disorder* denotes the order in which the menu items appear.

Popup Menu control A popup menu contains no menu item capable of invoking a drop down menu. It is a window control, and so may be disabled:

<i>PopupMenu</i>
<i>Menu</i>
<i>enabled</i> : <i>Boolean</i>
$\forall m : \text{ran } menuitems \bullet$ $m.type \neq IsaInvokeDropDownMenuItem$

$$MenuFromPopupMenu == (\lambda PopupMenu \bullet \theta Menu)$$

Drop Down Menus and the Menu Bar control A single drop down menu is identical to a popup menu except that there are no restrictions on its menu item types. However, if a menu item exists that can invoke another drop down menu, the two menus must be different. This prevents a drop down menu from invoking itself:

<i>DropDownMenu</i>
<i>ddmid</i> : <i>ID</i>
<i>Menu</i>
$\forall m : \text{ran } menuitems \bullet$ $m.type = IsaInvokeDropDownMenuItem \Rightarrow$ $m.ddmid \neq ddmid$

$$MenuFromDropDownMenu == (\lambda DropDownMenu \bullet \theta Menu)$$

If several drop down menus are active at some point in time, they must have been invoked in a sequence. The behaviour of these menus as a group is specified as follows:

<i>DropDownMenusSeq</i>
$menus : ID \leftrightarrow DropDownMenu$ $dispor\text{order} : seq\ ID$
$dom\ menus = \{ ddm : DropDownMenu \mid ddm \in ran\ menus \bullet ddm.ddmid \}$ $\#menus \geq 0$ $dom\ menus = ran\ dispor\text{order}$ $\forall i : 1 .. (\#menus - 1) \bullet$ $(\exists mi : ran((dispor\text{order} ; menus)(i)).menuitems \bullet$ $mi.type = IsaInvokeDropDownMenuItem \wedge$ $mi.ddmid = ((dispor\text{order} ; menus)(i + 1)).ddmid)$

A menu bar, contains a sequence of menu items, none of which can toggle values. Since a sequence of drop down menus must be initially invoked from selection of a menu bar item, a property of type *DropDownMenusSeq* is included in the *MenuBar* schema type:

<i>MenuBar</i>
$Menu$ $enabled : Boolean$ $dropdownmenus : DropDownMenusSeq$
$\forall m : ran\ menuitems \bullet$ $m.type \neq IsaToggleValueMenuItem$

$$MenuFromMenuBar == (\lambda\ MenuBar \bullet \theta Menu)$$

5.1.5 Summary of Control Types

For notational convenience, it is desirable to create a general specification for a window control. Since there are many kinds of window controls, this specification must be expressed in a hierarchical style. Following Hall's style, this representation can be done in Z by using schema disjunction and schema inclusion.

First, a transitional schema type called *Control* is created as the disjunction of all window control schema types:

$$Control \cong PushButton \vee CheckBox \vee RadioButtonGroup \vee ScrollBar \vee \\ TextEdit \vee ListBox \vee ComboBoxSimple \vee ComboBoxDrop \vee \\ ScrollRegion \vee PopupMenu \vee MenuBar$$

Next, an identifier is specified whose valid values represent a “tag” for each of the window control types:

$$\begin{aligned} \text{ControlType} ::= & \text{IsaPushButton} \mid \text{IsaCheckBox} \mid \text{IsaRadioButtonGroup} \mid \\ & \text{IsaScrollBar} \mid \text{IsaTextEdit} \mid \text{IsaListBox} \mid \text{IsaComboBoxSimple} \mid \\ & \text{IsaComboBoxDrop} \mid \text{IsaScrollRegion} \mid \text{IsaPopupMenu} \mid \text{IsaMenuBar} \end{aligned}$$

Finally, a *WindowControl* schema type can be specified as follows:

<i>WindowControl</i>	
<i>wcid</i> : <i>ID</i>	
<i>Control</i>	
<i>type</i> : <i>ControlType</i>	
<i>type</i> = <i>IsaPushButton</i> $\wedge$ <i>PushButton</i> $\vee$	
<i>type</i> = <i>IsaCheckBox</i> $\wedge$ <i>CheckBox</i> $\vee$	
<i>type</i> = <i>IsaRadioButtonGroup</i> $\wedge$ <i>RadioButtonGroup</i> $\vee$	
<i>type</i> = <i>IsaScrollBar</i> $\wedge$ <i>ScrollBar</i> $\vee$	
<i>type</i> = <i>IsaTextEdit</i> $\wedge$ <i>TextEdit</i> $\vee$	
<i>type</i> = <i>IsaListBox</i> $\wedge$ <i>ListBox</i> $\vee$	
<i>type</i> = <i>IsaComboBoxSimple</i> $\wedge$ <i>ComboBoxSimple</i> $\vee$	
<i>type</i> = <i>IsaComboBoxDrop</i> $\wedge$ <i>ComboBoxDrop</i> $\vee$	
<i>type</i> = <i>IsaScrollRegion</i> $\wedge$ <i>ScrollRegion</i> $\vee$	
<i>type</i> = <i>IsaPopupMenu</i> $\wedge$ <i>PopupMenu</i> $\vee$	
<i>type</i> = <i>IsaMenuBar</i> $\wedge$ <i>MenuBar</i>	

5.2 Visual Representation of Controls

Thus far, visual characteristics of window controls have not been addressed. This has been done deliberately so that the true functionality of various window controls could be specified first, without being distracted by their visual representations. In this section, a separate function is provided for each control that specifies how the control is mapped to the screen. The relative placement of the components of each control is also specified.

String and Push Button control The screen space occupied by a string and by a push button control are given by the following functions for which only the signatures are provided:

<i>StringVSR</i> : <i>STRING</i> $\leftrightarrow$ <i>VScreenRegion</i>
<i>PushButtonVSR</i> : <i>PushButton</i> $\leftrightarrow$ <i>VScreenRegion</i>

Check Box control A check box control is represented as a box with the associated text displayed directly to its right. Although the box's appearance may differ depending on the value of the boolean variable to which it is associated, such intrinsic details are not necessary at this specification level:

$$\begin{array}{|l}
 \hline
 \text{CheckBoxVSR} : \text{CheckBox} \leftrightarrow \text{VScreenRegion} \\
 \hline
 \forall cb : \text{CheckBox}; vsr : \text{VScreenRegion} \bullet \\
 \quad \text{CheckBoxVSR}(cb) = vsr \Rightarrow \\
 \quad (\exists box : \text{VScreenRegion} \bullet \\
 \quad \quad box \text{ VSRLeftOf } \text{StringVSR}(cb.\text{label}) \wedge \\
 \quad \quad vsr \text{ VSRContains } \text{StringVSR}(cb.\text{label}) \wedge \\
 \quad \quad vsr \text{ VSRContains } box)
 \end{array}$$

The *CheckBoxVSR* function asserts that a check box control is contained within a rectangular virtual screen region; contained within this region are two smaller regions, with one appearing directly to the left of the other. The region at the left contains the box, and the other contains the text from the check box label.

Radio Button Group control A radio button is represented as a circle with its text label immediately to the right:

$$\begin{array}{|l}
 \hline
 \text{RadioButtonVSR} : \text{RadioButton} \leftrightarrow \text{VScreenRegion} \\
 \hline
 \forall rb : \text{RadioButton}; vsr : \text{VScreenRegion} \bullet \\
 \quad \text{RadioButtonVSR}(rb) = vsr \Rightarrow \\
 \quad (\exists circle : \text{VScreenRegion} \bullet \\
 \quad \quad circle \text{ VSRLeftOf } \text{StringVSR}(rb.\text{label}) \wedge \\
 \quad \quad vsr \text{ VSRContains } \text{StringVSR}(rb.\text{label}) \wedge \\
 \quad \quad vsr \text{ VSRContains } circle)
 \end{array}$$

Radio buttons in a group cannot overlap and must be contained within the virtual screen region mapped to the radio button group as a whole:

$$\begin{array}{|l}
 \hline
 \text{RadioButtonGroupVSR} : \text{RadioButtonGroup} \leftrightarrow \text{VScreenRegion} \\
 \hline
 \forall rg : \text{RadioButtonGroup}; vsr : \text{VScreenRegion} \bullet \\
 \quad \text{RadioButtonGroupVSR}(rg) = vsr \Rightarrow \\
 \quad (\forall rb, rb_1 : rg.\text{rbuttons} \bullet \\
 \quad \quad vsr \text{ VSRContains } \text{RadioButtonVSR}(rb) \wedge \\
 \quad \quad rb \neq rb_1 \Rightarrow \\
 \quad \quad \neg (\text{RadioButtonVSR}(rb) \text{ VSROverlaps } \\
 \quad \quad \quad \text{RadioButtonVSR}(rb_1)))
 \end{array}$$

Scroll Bar control A scroll bar control may appear horizontally or vertically on the screen. The orientation determines the visual relationship between components of the control. The difficult aspect of formally specifying the visual characteristics of this control lies in the positioning of the thumb within the track according to the range position value of the control. This has been accomplished by considering the three cases separately: i) the thumb is at the “low” end of the track indicating the range value is equal to the minimum range value; ii) the thumb is at the “high” end of the track; or iii) the thumb is at some point between the endpoints. In order to reduce the size of the specification, the desired orientation of the scroll bar control is passed (indicated by *lineorient* in the following specification) with the scroll bar itself to the function *ScrollBarVSR*:

$$\text{ScrollBarVSR} : (\text{ScrollBar} \times \text{lineorient}) \rightarrow \text{VScreenRegion}$$

$$\begin{aligned} &\forall sb : \text{ScrollBar}; ori : \text{lineorient}; vsr : \text{VScreenRegion} \bullet \\ &\quad \text{ScrollBarVSR}(sb, ori) = vsr \Rightarrow \\ &\quad \quad (vsr \text{ VSRContains } \text{PushButtonVSR}(sb.\text{minbutton}) \wedge \\ &\quad \quad vsr \text{ VSRContains } \text{PushButtonVSR}(sb.\text{maxbutton}) \wedge \\ &\quad \quad vsr \text{ VSRContains } sb.\text{thumb} \wedge \\ &\quad \quad \text{SameShapeVSR}(\text{PushButtonVSR}(sb.\text{minbutton}), \\ &\quad \quad \quad \text{PushButtonVSR}(sb.\text{maxbutton})) \wedge \\ &\quad (\exists \text{trackminside} : \text{VScreenRegion} \bullet \\ &\quad \quad (vsr \text{ VSRContains } \text{trackminside} \wedge \\ &\quad \quad \text{VSRAdjacentLine}(\langle \text{PushButtonVSR}(sb.\text{minbutton}) \rangle^\sim \\ &\quad \quad \quad \langle \text{trackminside} \rangle^\sim \langle sb.\text{thumb} \rangle^\sim \\ &\quad \quad \quad \langle \text{PushButtonVSR}(sb.\text{maxbutton}) \rangle, ori)) \Rightarrow \\ &\quad \quad \quad sb.\text{rangeposn} = sb.\text{rangemax}) \wedge \\ &\quad (\exists \text{trackmaxside} : \text{VScreenRegion} \bullet \\ &\quad \quad (vsr \text{ VSRContains } \text{trackmaxside} \wedge \\ &\quad \quad \text{VSRAdjacentLine}(\langle \text{PushButtonVSR}(sb.\text{minbutton}) \rangle^\sim \\ &\quad \quad \quad \langle sb.\text{thumb} \rangle^\sim \langle \text{trackmaxside} \rangle^\sim \\ &\quad \quad \quad \langle \text{PushButtonVSR}(sb.\text{maxbutton}) \rangle, ori)) \Rightarrow \\ &\quad \quad \quad sb.\text{rangeposn} = sb.\text{rangemin}) \wedge \\ &\quad (\exists \text{trackminside}, \text{trackmaxside} : \text{VScreenRegion} \bullet \\ &\quad \quad (vsr \text{ VSRContains } \text{trackminside} \wedge \\ &\quad \quad vsr \text{ VSRContains } \text{trackmaxside} \wedge \\ &\quad \quad \text{VSRAdjacentLine}(\langle \text{PushButtonVSR}(sb.\text{minbutton}) \rangle^\sim \\ &\quad \quad \quad \langle \text{trackminside} \rangle^\sim \langle sb.\text{thumb} \rangle^\sim \\ &\quad \quad \quad \langle \text{trackmaxside} \rangle^\sim \\ &\quad \quad \quad \langle \text{PushButtonVSR}(sb.\text{maxbutton}) \rangle, ori)) \Rightarrow \\ &\quad \quad \quad \#(\text{AllVSRPoints}(\text{trackminside})) \text{div} \\ &\quad \quad \quad \#(\text{AllVSRPoints}(\text{trackmaxside})) = \\ &\quad \quad \quad (sb.\text{rangeposn} - sb.\text{rangemin} + 1) \text{div} \\ &\quad \quad \quad (sb.\text{rangemax} - sb.\text{rangemin} + 1))) \end{aligned}$$

The definition of the function *ScrollBarVSR* has four main parts. The first asserts that the virtual screen region bounding the entire control contains two push button controls and a thumb, and that the two push buttons have the same size. Each of the next three sections deals with one of the three cases for thumb positioning. Taking the final case for illustration, the specification asserts that the virtual screen region bounding the entire control must also contain two virtual screen regions corresponding to the two visible sections of the track bisected by the thumb. The “minimum” push button, a portion of the track, the thumb, the second portion of the track, and the “maximum” push button must appear either side by side in order from left to right,

or stacked from top to bottom, depending on the orientation passed to the function. Finally, the areas of the two visible track segments must have the same ratio as the value of the current range position within the range. This ensures that the thumb is positioned appropriately.

Text Edit control A text edit control contains a vertical list of strings, with a scroll bar, if required, appearing to the right of the text area:

$$\begin{array}{l}
 \hline
 \textit{TextEditVSR} : \textit{TextEdit} \leftrightarrow \textit{VScreenRegion} \\
 \hline
 \forall te : \textit{TextEdit}; vsr : \textit{VScreenRegion} \bullet \\
 \quad \textit{TextEditVSR}(te) = vsr \Rightarrow \\
 \quad (\exists \textit{textarea} : \textit{VScreenRegion} \bullet \\
 \quad \quad (\forall i : 1 \dots \#te.\textit{visibletext} \bullet \\
 \quad \quad \quad \textit{textarea VSRContains StringVSR}(te.\textit{visibletext}(i)) \wedge \\
 \quad \quad \quad i < \#te.\textit{visibletext} \Rightarrow \\
 \quad \quad \quad \quad \textit{StringVSR}(te.\textit{visibletext}(i))\textit{VSRAbove} \\
 \quad \quad \quad \quad \textit{StringVSR}(te.\textit{visibletext}(i + 1))) \wedge \\
 \quad \quad vsr \textit{VSRContains textarea} \wedge \\
 \quad \quad te.\textit{vscrollbar} \notin \emptyset[\textit{ScrollBar}] \Rightarrow \\
 \quad \quad \quad (vsr \textit{VSRContains ScrollBarVSR}(te.\textit{vscrollbar}, \textit{vertical}) \wedge \\
 \quad \quad \quad \textit{textarea VSRLeftOf ScrollBarVSR}(te.\textit{vscrollbar}, \textit{vertical})))
 \end{array}$$

List Box control A list box control has the same visual characteristics as a text edit control:

$$\begin{array}{l}
 \hline
 \textit{ListBoxVSR} : \textit{ListBox} \leftrightarrow \textit{VScreenRegion} \\
 \hline
 \forall lb : \textit{ListBox}; vsr : \textit{VScreenRegion} \bullet \\
 \quad \textit{ListBoxVSR}(lb) = vsr \Rightarrow \\
 \quad (\exists \textit{listarea} : \textit{VScreenRegion} \bullet \\
 \quad \quad (\forall i : 1 \dots \#lb.\textit{visibleitems} \bullet \\
 \quad \quad \quad \textit{listarea VSRContains StringVSR}((lb.\textit{visibleitems}(i)).\textit{text}) \wedge \\
 \quad \quad \quad i < \#lb.\textit{visibleitems} \Rightarrow \\
 \quad \quad \quad \quad \textit{StringVSR}((lb.\textit{visibleitems}(i)).\textit{text})\textit{VSRAbove} \\
 \quad \quad \quad \quad \textit{StringVSR}((lb.\textit{visibleitems}(i + 1)).\textit{text})) \wedge \\
 \quad \quad vsr \textit{VSRContains listarea} \wedge \\
 \quad \quad lb.\textit{vscrollbar} \notin \emptyset[\textit{ScrollBar}] \Rightarrow \\
 \quad \quad \quad (vsr \textit{VSRContains ScrollBarVSR}(lb.\textit{vscrollbar}, \textit{vertical}) \wedge \\
 \quad \quad \quad \textit{listarea VSRLeftOf ScrollBarVSR}(lb.\textit{vscrollbar}, \textit{vertical})))
 \end{array}$$

Combo Box controls A simple combo box control appears as a text edit control with a list box control appearing below it:

$$\text{ComboBoxSimpleVSR} : \text{ComboBoxSimple} \leftrightarrow \text{VScreenRegion}$$

$$\begin{aligned} \forall cbs : \text{ComboBoxSimple}; vsr : \text{VScreenRegion} \bullet \\ \text{ComboBoxSimpleVSR}(cbs) = vsr \Rightarrow \\ (vsr \text{ VSRContains } \text{TextEditVSR}(cbs.\text{textfield}) \wedge \\ vsr \text{ VSRContains } \text{ListBoxVSR}(cbs.\text{picklist}) \wedge \\ \text{VSRAdjacentLine}(\langle \text{TextEditVSR}(cbs.\text{textfield}) \rangle^\sim \\ \langle \text{ListBoxVSR}(cbs.\text{picklist}) \rangle, \text{vertical})) \end{aligned}$$

A drop down combo box control has a list box component that may appear anywhere on the screen, or may not be visible at all. Thus, the control must be mapped to one virtual screen region that contains the text edit control and push button, and a second virtual screen region that contains the list box control if it is visible:

$$\text{ComboBoxDropVSR} : \text{ComboBoxDrop} \leftrightarrow \mathbb{P} \text{VScreenRegion}$$

$$\begin{aligned} \forall cbd : \text{ComboBoxDrop}; pvsr : \mathbb{P} \text{VScreenRegion} \bullet \\ \text{ComboBoxDropVSR}(cbd) = pvsr \Rightarrow \\ (\exists vsr : \text{VScreenRegion} \bullet \\ vsr \text{ VSRContains } \text{TextEditVSR}(cbd.\text{textfield}) \wedge \\ vsr \text{ VSRContains } \text{PushButtonVSR}(cbd.\text{dropbutton}) \wedge \\ \text{VSRAdjacentLine}(\langle \text{TextEditVSR}(cbd.\text{textfield}) \rangle^\sim \\ \langle \text{PushButtonVSR}(cbd.\text{dropbutton}) \rangle, \text{horizontal}) \wedge \\ cbd.\text{picklist} \in \emptyset[\text{ListBox}] \Rightarrow \\ pvsr = \{vsr\} \wedge \\ cbd.\text{picklist} \notin \emptyset[\text{ListBox}] \Rightarrow \\ pvsr = \{vsr\} \cup \{\text{ListBoxVSR}(cbd.\text{picklist})\}) \end{aligned}$$

Scroll Region control The scroll region control appears as a screen area with scroll bars to the right, below, or both as required:

$$\text{ScrollRegionVSR} : \text{ScrollRegion} \leftrightarrow \text{VScreenRegion}$$

$$\begin{aligned} \forall sr : \text{ScrollRegion}; vsr : \text{VScreenRegion} \bullet \\ \text{ScrollRegionVSR}(sr) = vsr \Rightarrow \\ (vsr \text{ VSRContains } sr.\text{viewportVSR} \wedge \\ sr.\text{hscrollbar} \notin \emptyset[\text{ScrollBar}] \Rightarrow \\ (vsr \text{ VSRContains } \text{ScrollBarVSR}(sr.\text{hscrollbar}, \text{horizontal}) \wedge \\ \text{VSRAdjacentLine}(\langle sr.\text{viewportVSR} \rangle^\sim \\ \langle \text{ScrollBarVSR}(sr.\text{hscrollbar}, \text{horizontal}) \rangle, \text{horizontal})) \wedge \\ sr.\text{vscrollbar} \notin \emptyset[\text{ScrollBar}] \Rightarrow \\ (vsr \text{ VSRContains } \text{ScrollBarVSR}(sr.\text{vscrollbar}, \text{vertical}) \wedge \\ \text{VSRAdjacentLine}(\langle sr.\text{viewportVSR} \rangle^\sim \\ \langle \text{ScrollBarVSR}(sr.\text{vscrollbar}, \text{vertical}) \rangle, \text{vertical}))) \end{aligned}$$

Popup Menu control A popup menu appears as a list of strings, one per menu item, whose ordering is defined by the *disorder* property. Note that relational composition is used in the following definition to order the menu items:

$$\begin{array}{l}
\hline
\text{PopupMenuVSR} : \text{PopupMenu} \rightarrow \text{VScreenRegion} \\
\hline
\forall pum : \text{PopupMenu}; vsr : \text{VScreenRegion} \bullet \\
\quad \text{PopupMenuVSR}(pum) = vsr \Rightarrow \\
\quad ((\forall i : 1 \dots \#(pum.disorder) \bullet \\
\quad \quad vsr \text{VSRContains} \\
\quad \quad \quad \text{StringVSR}(((pum.disorder \mathbin{\text{\textasciitilde}} pum.menuitems)(i)).label)) \wedge \\
\quad (\forall i : 1 \dots \#(pum.disorder) - 1 \bullet \\
\quad \quad \text{StringVSR}(((pum.disorder \mathbin{\text{\textasciitilde}} pum.menuitems)(i)).label) \\
\quad \quad \quad \text{VSRAbove} \\
\quad \quad \quad \text{StringVSR}(((pum.disorder \mathbin{\text{\textasciitilde}} pum.menuitems)(i + 1)).label)))
\end{array}$$

Menu Bar control Visual aspects of a drop down menu and a sequence of drop down menus must be specified first, since drop down menus are components of a menu bar control. The specification for a single drop down menu is identical to that for a popup menu control:

$$\begin{array}{l}
\hline
\text{DropDownMenuVSR} : \text{DropDownMenu} \rightarrow \text{VScreenRegion} \\
\hline
\forall ddm : \text{DropDownMenu}; vsr : \text{VScreenRegion} \bullet \\
\quad \text{DropDownMenuVSR}(ddm) = vsr \Rightarrow \\
\quad ((\forall i : 1 \dots \#(ddm.disorder) \bullet \\
\quad \quad vsr \text{VSRContains} \\
\quad \quad \quad \text{StringVSR}(((ddm.disorder \mathbin{\text{\textasciitilde}} ddm.menuitems)(i)).label)) \wedge \\
\quad (\forall i : 1 \dots \#(ddm.disorder) - 1 \bullet \\
\quad \quad \text{StringVSR}(((ddm.disorder \mathbin{\text{\textasciitilde}} ddm.menuitems)(i)).label) \\
\quad \quad \quad \text{VSRAbove} \\
\quad \quad \quad \text{StringVSR}(((ddm.disorder \mathbin{\text{\textasciitilde}} ddm.menuitems)(i + 1)).label)))
\end{array}$$

A set of drop down menus maps to a set of virtual screen regions. The screen regions in this set cannot overlap:

$$\begin{array}{l}
\hline
\text{DropDownMenusSeqVSR} : \text{DropDownMenusSeq} \rightarrow \mathbb{P} \text{VScreenRegion} \\
\hline
\forall ddms : \text{DropDownMenusSeq}; pvsr : \mathbb{P} \text{VScreenRegion} \bullet \\
\quad \text{DropDownMenusSeqVSR}(ddms) = pvsr \Rightarrow \\
\quad ((\forall ddm, ddm_1 : \text{ran } ddms.menu \bullet \\
\quad \quad ddm \neq ddm_1 \Rightarrow \\
\quad \quad \quad \neg (\text{DropDownMenuVSR}(ddm) \text{VSROverlaps} \\
\quad \quad \quad \quad \text{DropDownMenuVSR}(ddm_1))) \wedge \\
\quad \quad pvsr = \text{DropDownMenuVSR}(\text{ran } ddms.menu))
\end{array}$$

Visually, a menu bar consists of a sequence of labels in a horizontal line, plus the regions to which any drop down menus invoked from the menu bar are mapped:

$$\begin{array}{l}
\text{MenuBarVSR} : \text{MenuBar} \rightarrow \mathbb{P} \text{VScreenRegion} \\
\hline
\forall mb : \text{MenuBar}; pvsr : \mathbb{P} \text{VScreenRegion} \bullet \\
\quad \text{MenuBarVSR}(mb) = pvsr \Rightarrow \\
\quad (\exists vsrseq : \text{seq VScreenRegion}; barvsr : \text{VScreenRegion} \bullet \\
\quad \quad pvsr = \{barvsr\} \cup \\
\quad \quad \quad \text{DropDownMenusSeqVSR}(mb.\text{dropdownmenus}) \wedge \\
\quad (\forall i : 1 \dots \#(mb.\text{disorder}) \bullet \\
\quad \quad vsrseq(i) \text{VSRContains} \\
\quad \quad \quad \text{StringVSR}(((mb.\text{disorder} \# mb.\text{menuitems})(i)).\text{label}) \wedge \\
\quad \quad \quad barvsr \text{VSRContains } vsrseq(i)) \wedge \\
\quad (\forall i : 1 \dots \#(mb.\text{disorder}) - 1 \bullet \\
\quad \quad \text{VSRAdjacentLine}(\langle vsrseq(i) \rangle \smallfrown \langle vsrseq(i+1) \rangle, \text{horizontal})))
\end{array}$$

5.2.1 Summary of Visual Characteristics

Making use of the axioms and the *WindowControl* type defined earlier, one axiom can be specified that will return the set of virtual screen pixels to which a given window control maps. This function is very important in controlling the placement of window controls, as will be discussed in Chapter 6.

$$\text{WindowControlPoints} : \text{WindowControl} \rightarrow \mathbb{P} \text{VScreenPixel}$$

$$\forall wc : \text{WindowControl}; pvsp : \mathbb{P} \text{VScreenPixel} \bullet$$

$$\text{WindowControlPoints}(wc) = pvsp \Rightarrow$$

$$(wc.type = \text{IsaPushButton} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{PushButtonVSR}(wc.\theta \text{PushButton})) \wedge$$

$$wc.type = \text{IsaCheckBox} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{CheckBoxVSR}(wc.\theta \text{CheckBox})) \wedge$$

$$wc.type = \text{IsaRadioButtonGroup} \Rightarrow$$

$$psvp =$$

$$\text{AllVSRPoints}(\text{RadioButtonGroupVSR}(wc.\theta \text{RadioButtonGroup})) \wedge$$

$$wc.type = \text{IsaScrollBar} \Rightarrow$$

$$(psvp =$$

$$\text{AllVSRPoints}(\text{ScrollBarVSR}(wc.\theta \text{ScrollBar}, \text{horizontal})) \vee$$

$$psvp =$$

$$\text{AllVSRPoints}(\text{ScrollBarVSR}(wc.\theta \text{ScrollBar}, \text{vertical}))) \wedge$$

$$wc.type = \text{IsaTextEdit} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{TextEditVSR}(wc.\theta \text{TextEdit})) \wedge$$

$$wc.type = \text{IsaListBox} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{ListBoxVSR}(wc.\theta \text{ListBox})) \wedge$$

$$wc.type = \text{IsaComboBoxSimple} \Rightarrow$$

$$psvp =$$

$$\text{AllVSRPoints}(\text{ComboBoxSimpleVSR}(wc.\theta \text{ComboBoxSimple})) \wedge$$

$$wc.type = \text{IsaComboBoxDrop} \Rightarrow$$

$$psvp =$$

$$\cup (\text{AllVSRPoints}(\text{ComboBoxDropVSR}(wc.\theta \text{ComboBoxDrop}))) \wedge$$

$$wc.type = \text{IsaScrollRegion} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{ScrollRegionVSR}(wc.\theta \text{ScrollRegion})) \wedge$$

$$wc.type = \text{IsaPopupMenu} \Rightarrow$$

$$psvp = \text{AllVSRPoints}(\text{PopupMenuVSR}(wc.\theta \text{PopupMenu})) \wedge$$

$$wc.type = \text{IsaMenuBar} \Rightarrow$$

$$psvp = \cup (\text{AllVSRPoints}(\text{MenuBarVSR}(wc.\theta \text{MenuBar})))$$

Note that for those controls mapping to more than one virtual screen region, the *generalized union* ($\cup$) and *relational image* ($\Downarrow$) operators are required.

5.3 Behavioural Abstraction of Controls

In this section, the representational and visual aspects of the various controls are combined with the mouse specifications. Almost all operations defined on window controls make use of mouse actions either to initiate the action or to control its effect. Each control may use a different mouse button for its actions, and the mouse button

to be used for each control is specified globally. The GUI developer or user is free to choose any of the three mouse buttons for each control:

<i>ButtonForPushButton</i> : <i>ButtonPosition</i> <i>ButtonForCheckBox</i> : <i>ButtonPosition</i> <i>ButtonForRadioButton</i> : <i>ButtonPosition</i> <i>ButtonForScrollBar</i> : <i>ButtonPosition</i> <i>ButtonForListBox</i> : <i>ButtonPosition</i> <i>ButtonForMenus</i> : <i>ButtonPosition</i>
--

The operations discussed in this section are not exhaustive. Many operations defined for primitive controls can be reused for defining similar operations on composite controls in a straightforward manner. Unless a composite control's operation requires additional behaviour, the operation will be omitted from the specifications. For example, the operation of selecting an item from a list box will be specified as a list box operation, but the operation of selecting a list box item from a combo box control will be omitted since it adds no additional behaviour.

Operations on a window control can be performed only if the control is enabled. The operations must leave the control enabled and preserve the value of the identifier and the control type:

<i>OpOnWindowControl</i> _____ <i>Δ WindowControl</i>
<i>wcid'</i> = <i>wcid</i> <i>type'</i> = <i>type</i> <i>enabled'</i> = <i>enabled</i> = <i>Yes</i>

Push Button Control A push button control can be selected by moving the mouse cursor over its virtual screen region and clicking the mouse button. Similarly, a push button control may be held down by the user if the mouse button is held down and the cursor is positioned over the control's virtual screen region:

<i>ButtonIsPushed</i> _____ <i>OpOnWindowControl</i>
<i>type</i> = <i>IsaPushButton</i> <i>MouseClickedObject</i> (<i>PushButtonVSR</i> (<i>θ PushButton</i>), <i>ButtonForPushButton</i> , <i>mousenow</i>)

ButtonIsHeldDown

OpOnWindowControl

type = IsaPushButton

*MouseHoldDownObject(PushButtonVSR(θ PushButton),
ButtonForPushButton, mousenow)*

Check Box control A check box control can be toggled by clicking on the virtual screen region of the control with the mouse button:

ToggleCheckBox

OpOnWindowControl

type = IsaCheckBox

*MouseClickedObject(CheckBoxVSR(θ CheckBox),
ButtonForCheckBox, mousenow)*

selected' = Toggle(selected)

label' = label

Radio Button Group control A radio button in a radio button group control is selected by clicking on its virtual screen region with the mouse button. The radio button previously selected becomes unselected and the radio button on which the mouse clicks becomes selected:

SelectRadioButton

OpOnWindowControl

rb? : RadioButton

type = IsaRadioButtonGroup

rb? $\in$ rbuttons

rb?.selected = No

*MouseClickedObject(RadioButtonVSR(rb?),
ButtonForRadioButton, mousenow)*

$\exists_1 rb : \text{RadioButton} \bullet$

rb $\in$ rbuttons $\wedge$

rb.selected = Yes $\wedge$

rbuttons' = rbuttons $\setminus$ {rb?, rb} $\cup$

*{(μ rb₁ : RadioButton |
rb₁.selected = Yes $\wedge$ rb₁.label = rb?.label $\bullet$ rb₁)}* $\cup$

*{(μ rb₂ : RadioButton |
rb₂.selected = No $\wedge$ rb₂.label = rb.label $\bullet$ rb₂)}*

The radio button that is to be selected is passed as an input to the operation. It must be a valid button in the group and should not be selected in the initial state. The specification also ensures that in the final state the radio button passed as input is selected and the others are not selected.

Scroll Bar control The thumb of a scroll bar control may be moved in several ways: i) by dragging the thumb to a new position; ii) by clicking on either the “max” push button or the “min” push button to move the thumb slowly; or iii) by holding down either push button to move the thumb rapidly. The specification for the first operation takes the new range position as input, and asserts that the thumb was dragged to the new position with the mouse. In the final state, the thumb of the scroll bar control has this new position. The dependency between the range value and the thumb position is enforced by the *ScrollBarVSR* specification:

<i>MoveThumb</i> <i>OpOnWindowControl</i> <i>rangeposn?</i> : $\mathbb{N}$ <i>type</i> = <i>IsaScrollBar</i> <i>rangeposn'</i> = <i>rangeposn?</i> $(thumb', mousenow) = \text{MouseDrag_n_Drop}(thumb, \text{ButtonForScrollBar})$ <i>minbutton'</i> = <i>minbutton</i> <i>maxbutton'</i> = <i>maxbutton</i> <i>rangemin'</i> = <i>rangemin</i> <i>rangemax'</i> = <i>rangemax</i>

The push button at either end of the scroll bar control may be clicked to change the range position slowly. The new range value is taken as input and it must be one greater than or less than the current value. The operation ensures that the new position is achieved by clicking the appropriate end button:

PressScrollBar

OpOnWindowControl

rangeposn? : $\mathbb{N}$

type = *IsaScrollBar*

Abs(*rangeposn* - *rangeposn?*) = 1

rangeposn? < *rangeposn* $\Rightarrow$

$(\mu \text{ ButtonIsPushed}_1 \mid$
 $\theta \text{ PushButton}_1 = \text{minbutton} \bullet \theta \text{ PushButton}'_1) \wedge$
 $\text{maxbutton}' = \text{maxbutton})$

rangeposn? > *rangeposn* $\Rightarrow$

$(\mu \text{ ButtonIsPushed}_1 \mid$
 $\theta \text{ PushButton}_1 = \text{maxbutton} \bullet \theta \text{ PushButton}'_1) \wedge$
 $\text{minbutton}' = \text{minbutton})$

rangeposn' = *rangeposn?*

rangemin' = *rangemin*

rangemax' = *rangemax*

In the *PressScrollBar* specification, the range value may either increase or decrease depending on the input value. Each case is specified in separate predicates since a different push button must be clicked for each case. If the input range position is less than the current position, then the “min” push button has been clicked and the “max” push button is ignored. The act of clicking the push button is specified using the μ expression where the initial and final states of the *ButtonIsPushed* operation corresponds to the states of the “min” push button control in the initial and final states of *WindowControl*. A similar method is used for increasing the range position value by clicking the “max” push button control.

A scroll bar control’s range value may be moved in larger increments by holding down the push button at either end. The specification for this operation is similar to that defined above, except that the new range position is unrestricted:

HoldScrollBar

OpOnWindowControl

rangeposn? : $\mathbb{N}$

type = *IsaScrollBar*

rangeposn? < *rangeposn* $\Rightarrow$

$(\mu \text{ ButtonIsHeldDown}_1 \mid$
 $\theta \text{ PushButton}_1 = \text{minbutton} \bullet \theta \text{ PushButton}'_1) \wedge$
 $\text{maxbutton}' = \text{maxbutton})$

rangeposn? > *rangeposn* $\Rightarrow$

$(\mu \text{ ButtonIsHeldDown}_1 \mid$
 $\theta \text{ PushButton}_1 = \text{maxbutton} \bullet \theta \text{ PushButton}'_1) \wedge$
 $\text{minbutton}' = \text{minbutton})$

rangeposn' = *rangeposn?*

rangemin' = *rangemin*

rangemax' = *rangemax*

In summary, the range position of a scroll bar control may be changed in three different ways:

$\text{ChangeRangePosn} \hat{=} \text{MoveThumb} \vee \text{PressScrollBar} \vee \text{HoldScrollBar}$

Text Edit control A new line of text may be added to a text edit control at a given line, or a given line of text may be deleted. For these operations, state change in the *visibletext* and *vscrollbar* properties are not restricted since their values are calculated in the *TextEdit* schema for all bindings. The specifications for adding a new line of text follow:

AddNewTextLine

OpOnWindowControl

str? : *STRING*

lineposn? : $\mathbb{N}_1$

type = *IsaTextEdit*

lineposn? $\leq \# \text{text}$

$\text{text}' = \{i : 1 \dots \text{lineposn}' - 1\} \upharpoonright \text{text} \hat{\ } \langle \text{str}' \rangle \hat{\ }$
 $\{i : \text{lineposn}' \dots \# \text{text}\} \upharpoonright \text{text}$

height' = *height*

In *AddNewTextLine*, the binary infix operator $\upharpoonright$ represents *sequence extraction*. Given a set of indices *I* and a sequence of elements *E*, $I \upharpoonright E$ returns a sequence of elements from *E* having indices in *I*, in the same order as in *I*. Its function in the above

specification is to extract two subsequences from *text*, where the division between subsequences occurs at the element indicated by *lineposn?*. The symbol $\hat{}$ denotes *sequence concatenation* in Z. If S_1 and S_2 are two sequences, $S_1 \hat{} S_2$ contains the elements of S_1 , in order, followed by the elements of S_2 , in order.

A line of text may be deleted from a text edit control as specified by the following:

<i>DeleteTextLine</i>
<i>OpOnWindowControl</i>
<i>lineposn?</i> : $\mathbb{N}_1$
<i>type</i> = <i>IsaTextEdit</i>
<i>lineposn?</i> $\leq$ <i>#text</i>
<i>text'</i> = $\{i : 1 \dots \text{lineposn?} - 1\} \upharpoonright \text{text} \hat{}$ $\{i : \text{lineposn?} + 1 \dots \#text\} \upharpoonright \text{text}$
<i>height'</i> = <i>height</i>

Changing a text line can be specified as a composition of the text line deletion operation followed by the text line addition operation:

$$\textit{ChangeTextLine} \hat{=} \textit{DeleteTextLine} ; \textit{AddNewTextLine}$$

List Box control Clicking on an item in a list box control causes the item to become selected if it was not already selected, or unselected if it was previously selected. The item to be selected or unselected by the user must appear on the screen since it must be clicked by the mouse. The *visibleitems* and *selecteditems* properties are calculated by the *ListBox* schema for all bindings, so their values need not be restricted for the state change implied by the operation:

<i>ToggleItem</i>
<i>OpOnWindowControl</i>
<i>li?</i> : <i>ListItem</i>
<i>type</i> = <i>IsaListBox</i>
$\langle li? \rangle$ in <i>visibleitems</i>
<i>MouseClickedObject</i> (<i>StringVSR</i> (<i>li?.text</i>), <i>ButtonForListBox</i> , <i>mousenow</i>)
<i>ran items'</i> = <i>ran items</i> $\setminus \{li?\} \cup$ $\{(\mu li : \textit{ListItem} \mid$ $li.\textit{text} = li?.\textit{text} \wedge li.\textit{selected} = \textit{Toggle}(li?.\textit{selected}))\}$
<i>height'</i> = <i>height</i>
<i>vscrollbar'</i> = <i>vscrollbar</i>
<i>multiple'</i> = <i>multiple</i>

Combo Box controls If a drop down combo box control's push button is clicked with the mouse, its list box will appear if it is not currently visible, or disappear if it is currently visible:

<i>ComboBoxToggleList</i> <i>OpOnWindowControl</i>
$ \begin{aligned} &type = IsaComboBoxDrop \\ &dropbutton' = (\mu ButtonIsPushed_1 \mid \\ &\quad \theta PushButton_1 = dropbutton \bullet \theta PushButton'_1) \wedge \\ &picklist \in \emptyset[ListBox] \Rightarrow \\ &\quad picklist' \notin \emptyset[ListBox] \\ &picklist \notin \emptyset[ListBox] \Rightarrow \\ &\quad picklist \in \emptyset[ListBox] \\ &textfield' = textfield \\ &static' = static \end{aligned} $

Text may be changed within a text edit region for either type of combo box control, provided it is not static and no items are selected from the list box. If text is entered into the text field of a drop down combo box, the list box is automatically removed.

<i>ChangeComboBoxText</i> <i>OpOnWindowControl</i> <i>lineposn?</i> : $\mathbb{N}_1$ <i>str?</i> : <i>STRING</i>
$ \begin{aligned} &type = IsaComboBoxSimple \vee type = IsaComboBoxDrop \\ &static = No \\ &\#picklist.selecteditems = 0 \\ &textfield' = (\mu ChangeTextLine_1 \mid \theta TextEdit_1 = textfield \bullet \theta TextEdit'_1) \wedge \\ &type = IsaComboBoxDrop \Rightarrow \\ &\quad picklist' \in \emptyset[ListBox] \\ &type = IsaComboBoxSimple \Rightarrow \\ &\quad picklist' = picklist \\ &static' = static \\ &type = IsaComboBoxDrop \Leftrightarrow dropbutton' = dropbutton \end{aligned} $

Scroll Region control A scroll region control may be enlarged, shrunk or moved when the window in which it is contained is scaled or moved (as discussed in Chapter 6). It cannot be scaled or moved directly with the mouse. It is to be noted that enlarging a scroll region control exposes more information from the work area, shrink-

ing it causes a portion of the previously displayed information to disappear from view, and moving it exposes the same work area information as before the move:

<i>EnlargeScrollRegion</i> <i>OpOnWindowControl</i> <i>direction?</i> : <i>direction</i>
<i>type</i> = <i>IsaScrollRegion</i> <i>viewportVSR'</i> = (μ <i>EnlargeVSR</i> ₁ <i>direction?</i> ₁ = <i>direction?</i> $\wedge$ θ <i>VScreenRegion</i> ₁ = <i>viewportVSR</i> $\bullet$ θ <i>VScreenRegion'</i> ₁) <i>AllWRPoints</i> (<i>viewportWR</i>) $\subset$ <i>AllWRPoints</i> (<i>viewportWR'</i>) <i>zoomfactor'</i> = <i>zoomfactor</i>
<i>ShrinkScrollRegion</i> <i>OpOnWindowControl</i> <i>direction?</i> : <i>direction</i>
<i>type</i> = <i>IsaScrollRegion</i> <i>viewportVSR'</i> = (μ <i>ShrinkVSR</i> ₁ <i>direction?</i> ₁ = <i>direction?</i> $\wedge$ θ <i>VScreenRegion</i> ₁ = <i>viewportVSR</i> $\bullet$ θ <i>VScreenRegion'</i> ₁) <i>AllWRPoints</i> (<i>viewportWR'</i>) $\subset$ <i>AllWRPoints</i> (<i>viewportWR</i>) <i>zoomfactor'</i> = <i>zoomfactor</i>
<i>MoveScrollRegion</i> <i>OpOnWindowControl</i> <i>direction?</i> : <i>direction</i>
<i>type</i> = <i>IsaScrollRegion</i> <i>viewportVSR'</i> = (μ <i>MoveVSR</i> ₁ <i>direction?</i> ₁ = <i>direction?</i> $\wedge$ θ <i>VScreenRegion</i> ₁ = <i>viewportVSR</i> $\bullet$ θ <i>VScreenRegion'</i> ₁) <i>AllWRPoints</i> (<i>viewportWR</i>) = <i>AllWRPoints</i> (<i>viewportWR'</i>) <i>zoomfactor'</i> = <i>zoomfactor</i>

Generic menu item operations Of the four menu item types discussed earlier, three exhibit a behaviour when selected that can be modeled independent of the application using the menu item. The fourth menu item type, namely that which indicates the execution of some application-specific action, has an application dependent behaviour when selected, and hence is beyond the scope of this thesis. After the selection operation on the three menu item types is specified, they will be promoted to define menu operations. First, every operation on a menu item must ensure that the menu item remain enabled and its type and identifier are unchanged:

<i>OpOnMenuItem</i>	_____
$\Delta MenuItem$	
$enabled' = enabled = Yes$	
$miid' = miid$	
$type' = type$	

A menu item can be selected by pointing to its text label with the mouse cursor and clicking on it with the mouse button indicated by the variable *ButtonForMenus*:

$SelectMenuItem_ : \mathbb{P} MenuItem$	_____
$\forall mi : MenuItem \bullet$	
$SelectMenuItem(mi) \Leftrightarrow$	
$MouseClickedObject(StringVSR(mi.label), ButtonForMenus, mousenow)$	

A menu item can be toggled by changing the boolean variable *applvar* specified in *MenuItem*:

<i>ToggleMenuItem</i>	_____
<i>OpOnMenuItem</i>	
$type = IsaToggleValueMenuItem$	
$SelectMenuItem(\theta MenuItem)$	
$applvar' = Toggle(applvar)$	

At this stage, only the selection action for menu items that display windows or menus when selected is specified. The operations for creating new menus and selecting a window will be specified in Chapter 7.

$NewWindowID : MenuItem \leftrightarrow ID$	_____
$\forall mi : MenuItem \bullet$	
$NewWindowID(mi) = mi.windowid \wedge$	
$mi.type = IsaInvokeWindowMenuItem \wedge$	
$SelectMenuItem(mi)$	

$NewMenuID : MenuItem \leftrightarrow ID$	_____
$\forall mi : MenuItem \bullet$	
$NewMenuID(mi) = mi.ddmid \wedge$	
$mi.type = IsaInvokeDropDownMenuItem \wedge$	
$SelectMenuItem(mi)$	

Generic menu operations Operations on menus can change the states of its menu items, but cannot add or remove menu items. As well, the displayed order of the menu items cannot be changed. This is asserted by the following schema:

<i>OpOnMenu</i>	_____
$\Delta Menu$	
$\text{dom } menuitems' = \text{dom } menuitems$ $disporder' = disporder$	

Toggling a menu item contained within a simple menu can be specified by promoting the *ToggleMenuItem* schema operation using Hall's style:

<i>MToggleMenuItem</i>	_____
<i>OpOnMenu</i>	
$miid? : ID$	
$menuitems' = menuitems \oplus \{miid? \mapsto (\mu \text{ ToggleMenuItem} \mid \theta MenuItem = menuitems(miid?) \bullet \theta MenuItem')\}$	

In the *MToggleMenuItem* operation, all menu items from the initial state remain unaffected except for the menu item indicated by the *miid?* input. For this menu item the *ToggleMenuItem* state transition is applied where the initial state of the menu item is used as input. The result of this promoted operation is used as the final state value for the menu item.

Simple functions are provided next for menu items that invoke a new window or a new menu when selected:

$MNewWindowID : Menu \times ID \rightarrow ID$	_____
$\forall m : Menu; miid : ID \bullet$	
$MNewWindowID(m, miid) = NewWindowID(m.menuitems(miid))$	
$MNewMenuID : Menu \times ID \rightarrow ID$	_____
$\forall m : Menu; miid : ID \bullet$	
$MNewMenuID(m, miid) = NewWindowID(m.menuitems(miid))$	

These three simple menu operations can now be reused to create operations for each of the three menu types.

Popup Menu control Toggling within a popup menu can be specified as follows:

<i>PToggleMenuItem</i>	_____
<i>OpOnWindowControl</i>	
<i>miid? : ID</i>	
<i>type = IsaPopupMenu</i>	
<i>MenuFromPopupMenu(θPopupMenu') = (μ MToggleMenuItem </i>	
<i>θMenu = MenuFromPopupMenu(θPopupMenu) • θMenu')</i>	

Again, the full specifications for creating a new window when a menu item is selected will be deferred until Chapter 7. For now, only a promoted version of the more generic function is specified.

<i>PNewWindowID : WindowControl $\times$ ID $\leftrightarrow$ ID</i>	
$\forall wc : WindowControl; miid : ID \bullet$	
<i>PNewWindowID($wc, miid$) = MNewWindowID($wc.\theta$Menu, $miid$) $\wedge$</i>	
<i>$wc.type = IsaPopupMenu$</i>	

Menu Bar control Operations must first be specified for a drop down menu and for drop down menu sequences since they are components of the menu bar control.

Operations on a drop down menu cannot alter the menu's identifier, as asserted by the following:

<i>OpOnDropDownMenu</i>	_____
Δ <i>DropDownMenu</i>	
<i>OpOnMenu</i>	
<i>ddmid' = ddmid</i>	

The next three specifications are straightforward promotions of the operations defined on *Menu*:

<i>DToggleMenuItem</i>	_____
<i>OpOnDropDownMenu</i>	
<i>miid? : ID</i>	
<i>MenuFromDropDownMenu(θDropDownMenu') = (μ MToggleMenuItem </i>	
<i>θMenu = MenuFromDropDownMenu(θDropDownMenu) • θMenu')</i>	
<i>DNewWindowID : DropDownMenu $\times$ ID $\leftrightarrow$ ID</i>	
$\forall ddm : DropDownMenu; miid : ID \bullet$	
<i>DNewWindowID($ddm, miid$) = MNewWindowID($ddm.\theta$Menu, $miid$)</i>	

$$\begin{array}{|l}
DNewMenuID : DropDownMenu \times ID \leftrightarrow ID \\
\hline
\forall ddm : DropDownMenu; miid : ID \bullet \\
DNewMenuID(ddm, miid) = MNewMenuID(ddm.\theta Menu, miid)
\end{array}$$

Since drop down menus may appear in groups as indicated by the *DropDownMenusSeq* schema type, the operations just defined for a single drop down menu must be redefined within the larger context of multiple active drop down menus. The action of creating a new drop down menu can also be fully specified at this point.

All operations performed on a sequence of drop down menus exhibit a common trait: if a menu item is selected from drop down menu i , any drop down menus dependent on drop down menu i are removed. To capture this, primitive operations on a *DropDownMenusSeq* type will be defined first, and then using schema composition this additional behaviour will be added.

There are no restrictions on changing the state of a *DropDownMenusSeq* schema type:

$$OpOnDropDownMenusSeq \triangleq \Delta DropDownMenusSeq$$

Given a drop down menu indicated by the $ddmid?$ input, and a menu item from this menu indicated with the $miid?$ input, the menu item can be toggled:

$$\begin{array}{|l}
DDMToggleMenuItem0 \\
OpOnDropDownMenusSeq \\
ddmid? : ID \\
miid? : ID \\
\hline
ddmid? \in \text{dom } menus \\
menus' = menus \oplus \{ddmid? \mapsto (\mu DToggleMenuItem \mid \\
\theta DropDownMenu = menus(ddmid?) \bullet \theta DropDownMenu')\} \\
disporder' = disporder
\end{array}$$

Given the same input as above, the window identifier for the window to be displayed is provided as output:

<i>DDMNewWindowID0</i>
<i>OpOnDropDownMenusSeq</i>
<i>ddmid?</i> : <i>ID</i>
<i>miid?</i> : <i>ID</i>
<i>windowid!</i> : <i>ID</i>
$ddmid? \in \text{dom } menus$
$windowid! = DNewWindowID(menus(ddmid?), miid?)$
$menus' = menus$
$disorder' = disorder$

A new drop down menu will be displayed if the correct menu item type is selected from another drop down menu in the group. This menu item must be contained within the last drop down menu in the sequence of menus. The new menu will be added to the end of the sequence, thus becoming dependent on the menu whose item was selected. The menu identifier, menu item identifier and the new drop down menu are passed to this operation as input:

<i>DDMNewMenu0</i>
<i>OpOnDropDownMenusSeq</i>
<i>newddm?</i> : <i>DropDownMenu</i>
<i>ddmid?</i> : <i>ID</i>
<i>miid?</i> : <i>ID</i>
$newddm?.ddmid \notin \text{dom } menus$
$newddm?.ddmid = DNewMenuID(menus(ddmid?), miid?)$
$menus' = menus \cup \{newddm?.ddmid \mapsto newddm?\}$
$disorder' = disorder \hat{\ } \langle newddm?.ddmid \rangle$

Removal of trailing drop down menus requires manipulation of the *disorder* property:

<i>RemoveTrailingDropDownMenus</i>
<i>OpOnDropDownMenusSeq</i>
<i>ddmid?</i> : <i>ID</i>
$\exists_1 i : 1 \dots \#disorder \bullet$
$((disorder \circ menus)(i)).ddmid = ddmid? \wedge$
$\#disorder' = i \wedge$
$(\forall j : 1 \dots i \bullet$
$disorder'(j) = disorder(j))$

These operations can be composed into three new operations, ensuring that the drop down menu on which each operation is applied, represented by the input *ddmid?*, is always the last menu in the sequence of drop down menus:

$$\begin{aligned}
DDMToggleMenuItem &\triangleq \\
&\quad RemoveTrailingDropDownMenus ; DDMToggleMenuItem0 \\
DDMNewWindowID &\triangleq \\
&\quad RemoveTrailingDropDownMenus ; DDMNewWindowID0 \\
DDMNewMenu &\triangleq \\
&\quad RemoveTrailingDropDownMenus ; DDMNewMenu0
\end{aligned}$$

Finally, selection of menu items on a menu bar can be specified. Selection of a menu bar menu item can invoke a window, perform some operation, or invoke a drop down menu. Since drop down menus are part of *MenuBar*, operations defined on *DropDownMenusSeq* can be used to specify the operations within the menu bar context.

As with the other menu types, complete specifications for invocation of a window are deferred:

$$\begin{array}{|l}
MBNewWindowID : WindowControl \times ID \rightarrow ID \\
\hline
\forall wc : WindowControl; miid : ID \bullet \\
\quad MBNewWindowID(wc, miid) = MNewWindowID(wc.\theta Menu, miid) \wedge \\
\quad wc.type = IsaMenuBar
\end{array}$$

The operation for invoking the initial drop down menu accepts the new drop down menu as input, and ensures that the menu item selected must invoke a drop down menu with a matching identifier:

$$\begin{array}{|l}
MBNewDropDownMenu \text{ -----} \\
OpOnWindowControl \\
newddm? : DropDownMenu \\
miid? : ID \\
\hline
type = IsaMenuBar \\
newddm?.ddmid = MNewWindowID(\theta Menu, miid?) \\
\exists ddms : DropDownMenusSeq \bullet \\
\quad ddms.menus = \{newddm?.ddmid \mapsto newddm?\} \wedge \\
\quad ddms.disporder = \langle newddm?.ddmid \rangle \wedge \\
\quad dropdownmenus' = ddms \\
menuitems' = menuitems \\
disporder' = disporder
\end{array}$$

Operations defined on *DropDownMenusSeq* can be used to create operations on *MenuBar*:

MB_DDMToggleMenuItem

OpOnWindowControl

miid? : *ID*

ddmid? : *ID*

type = *IsaMenuBar*

dropdownmenus' = (μ *DDMToggleMenusItem* |
 θ *DropDownMenusSeq* = *dropdownmenus* • θ *DropDownMenusSeq'*)

menuitems' = *menuitems*

disporder' = *disporder*

MB_DDMNewWindowID

OpOnWindowControl

miid? : *ID*

ddmid? : *ID*

windowid! : *ID*

type = *IsaMenuBar*

dropdownmenus' = (μ *DDMNewWindowID* |
 θ *DropDownMenusSeq* = *dropdownmenus* • θ *DropDownMenusSeq'*)

menuitems' = *menuitems*

disporder' = *disporder*

MB_DDMNewMenu

OpOnWindowControl

miid? : *ID*

ddmid? : *ID*

newddm? : *DropDownMenu*

type = *IsaMenuBar*

dropdownmenus' = (μ *DDMNewMenu* |
 θ *DropDownMenusSeq* = *dropdownmenus* • θ *DropDownMenusSeq'*)

menuitems' = *menuitems*

disporder' = *disporder*

5.4 Summary

In this chapter, a variety of window controls were defined including a multitude of operations, many using the mouse. These specifications are sufficiently generic to be incorporated as part of the specifications for a large GUI-based system.

Chapter 6

Windows

A *window* is the primary mechanism in a GUI through which application information and controls are grouped and presented to the user. The collection and the respective semantics of the controls within a window are dependent on the application and are selected by the application designer. This chapter contains detailed specifications for windows, including their behaviour, visual characteristics and operations that can be performed on them.

6.1 The Window

In general, a window has a caption explaining its role or purpose in the application domain. Every window has a set of controls which enable the user to interact with the window. At any one time, at most one of these controls can have operations performed on it. Such a control is said to *have focus*. It may also be possible that none of the controls have focus at a given time. There are two types of windows - *modal* and *modeless*. A modal window must be closed before another window can be selected by the user, whereas a modeless window can become inactive and placed into the background temporarily. A window also has a unique identifier. These characteristics are specified in the following schema:

Window

wid : *ID*

caption : *STRING*

controls : *ID* $\leftrightarrow$ *WindowControl*

focuscontrol : *ID*

modal : *Boolean*

$\text{dom } controls = \{wc : WindowControl \mid wc \in \text{ran } controls \bullet wc.wcid\}$

$focuscontrol \in \text{dom } controls \cup \emptyset[ID]$

6.2 Visual Characteristics of Windows

This section describes the relationship between the visual characteristics of controls in a window and that of the window itself. Generally, the following constraints are imposed on the placement of controls within a window: i) except for the control having focus, no two controls can overlap each other; ii) each window has a group of pixels contained in the *move handle*. A window can be moved by dragging one of the pixels in its move handle to a new position. For each window, the pixels forming its move handle can be computed based on the size of the window; iii) similar to the move handle, there exists eight groups of distinct pixels which enable scaling of the window. Each such group of pixels is known as a *scale handle*, and the precise pixels contained within the scales handles can also be computed based on the size of the window; iv) the handles of a window can neither overlap among themselves, nor with any of the controls not having focus; and v) the control having focus in a window, if one exists, can overlap with any other control or any handle.

These restrictions are specified below. For ease of understanding, each restriction is specified independently as a global function on a window.

Control positioning Unfocused controls in a window cannot overlap. This is asserted by the following function that returns the set of virtual screen pixels to which all unfocused controls map:

$$\begin{array}{|l}
\hline
\text{UnfocusedControlPoints} : \text{Window} \rightarrow \mathbb{P} \text{VScreenPixel} \\
\hline
\forall w : \text{Window}; \text{pvsp} : \mathbb{P} \text{VScreenPixel} \bullet \\
\quad \text{UnfocusedControlPoints}(w) = \text{pvsp} \Rightarrow \\
\quad (\text{pvsp} = \bigcup (\text{WindowControlPoints}(\text{ran } w.\text{controls} \setminus \\
\quad \quad \{w.\text{controls}(w.\text{focuscontrol})\})) \wedge \\
\quad (\forall \text{wcid}, \text{wcid}_1 : \text{dom } w.\text{controls} \setminus \{w.\text{focuscontrol}\} \bullet \\
\quad \quad \text{wcid} \neq \text{wcid}_1 \Rightarrow \\
\quad \quad \quad \text{WindowControlPoints}(w.\text{controls}(\text{wcid})) \cap \\
\quad \quad \quad \text{WindowControlPoints}(w.\text{controls}(\text{wcid}_1)) = \emptyset))
\end{array}$$

In the *UnfocusedControlPoints* function, the predicate

$$\text{pvsp} = \bigcup (\text{WindowControlPoints}(\text{ran } w.\text{controls} \setminus \{w.\text{controls}(w.\text{focuscontrol})\}))$$

computes the set of all pixels to which all controls in the window, except for the control having focus, are mapped. The expression

$$\text{ran } w.\text{controls} \setminus \{w.\text{controls}(w.\text{focuscontrol})\}$$

computes the set of all controls in the window excluding the control having focus. The function *WindowControlPoints* is then applied successively to each element of this set. The result is a set of pixel groups, one group per control. The application of the $\bigcup$ operation to this set results in a simple set of distinct virtual screen pixels.

The function *FocusedControlPoints* given below returns the virtual screen pixels occupied by the control having focus in a window. Note that this function returns an empty set of pixels if no control has focus, otherwise it returns the set of pixels to which the control maps:

$$\begin{array}{|l}
\hline
\text{FocusedControlPoints} : \text{Window} \rightarrow \mathbb{P} \text{VScreenPixel} \\
\hline
\forall w : \text{Window}; \text{pvsp} : \mathbb{P} \text{VScreenPixel} \bullet \\
\quad \text{FocusedControlPoints}(w) = \text{pvsp} \Rightarrow \\
\quad (w.\text{focuscontrol} \in \emptyset[\text{ID}] \Rightarrow \\
\quad \quad \text{pvsp} = \emptyset \wedge \\
\quad \quad w.\text{focuscontrol} \notin \emptyset[\text{ID}] \Rightarrow \\
\quad \quad \quad \text{pvsp} = \text{WindowControlPoints}(w.\text{controls}(w.\text{focuscontrol})))
\end{array}$$

Typically, a window itself occupies a virtual screen region. This view of a window as a distinct virtual screen region greatly helps in understanding the rest of the specifications in this chapter. Since the schemas *Window* and *VScreenRegion* refer to

different types in Z , the following function is required to support this view; this function is assumed to be understood by the reader and hence is not further elaborated:

$$\mid \text{WindowVSR} : \text{Window} \rightarrow \text{VScreenRegion}$$

Note that WindowVSR is a total function; this means that every window can be mapped to a virtual screen region.

Move Handle A window's move handle pixels are contained in a horizontal region that in turn contains the window caption text as a subset. This is asserted by the following:

$$\begin{array}{l} \text{MoveHandlePixels} : \text{Window} \leftrightarrow \mathbb{P} \text{VScreenPixel} \\ \hline \forall w : \text{Window}; pvsp : \mathbb{P} \text{VScreenPixel} \bullet \\ \quad \text{MoveHandlePixels}(w) = pvsp \Rightarrow \\ \quad (\exists vsr : \text{VScreenRegion} \bullet \\ \quad \quad \text{WindowVSR}(w) \text{VSRContains } vsr \wedge \\ \quad \quad pvsp = \text{AllVSRPoints}(vsr) \wedge \\ \quad \quad vsr \text{VSRContains StringVSR}(w.\text{caption})) \end{array}$$

Scale Handle The eight scale handles are contained within the border pixels of a window. The set of pixels comprising a window can be thought of as being contained in a virtual screen region surrounded by four borders (of equal thickness) on the four sides. The relationship between the border pixels, the window pixels and the virtual screen region inside a window can be specified. Given

$$\mid \text{borderwidth} : \mathbb{N}_1$$

as a global variable, the function $\text{WindowBorderPoints}$ given below identifies the border pixels of a window:

$$\begin{array}{l} \text{WindowBorderPoints} : \text{Window} \leftrightarrow \mathbb{P} \text{VScreenPixel} \\ \hline \forall w : \text{Window}; pvsp : \mathbb{P} \text{VScreenPixel} \bullet \\ \quad \text{WindowBorderPoints}(w) = pvsp \Rightarrow \\ \quad (\exists invsr, bigvsr : \text{VScreenRegion} \bullet \\ \quad \quad bigvsr = \text{WindowVSR}(w) \wedge \\ \quad \quad invsr.\text{topleft}.x = bigvsr.\text{topleft}.x + \text{borderwidth} \wedge \\ \quad \quad invsr.\text{topleft}.y = bigvsr.\text{topleft}.y + \text{borderwidth} \wedge \\ \quad \quad invsr.\text{botright}.x = bigvsr.\text{botright}.x - \text{borderwidth} \wedge \\ \quad \quad invsr.\text{botright}.y = bigvsr.\text{botright}.y - \text{borderwidth} \wedge \\ \quad \quad pvsp = \text{AllVSRPoints}(bigvsr) \setminus \text{AllVSRPoints}(invsr)) \end{array}$$

The *WindowBorderPoints* function asserts that there is a smaller virtual screen region contained within the window's virtual screen region, such that the top left corner pixel and the bottom right corner pixel of the smaller region have coordinate values offset exactly *borderwidth* pixels from the larger region's top left and bottom right corner pixels.

Handles for scaling a window towards the top, bottom, left or right sides are located along the top, bottom and side borders of the window, however, they do not occupy all border points along each side. Handles for moving in the four diagonal directions are located in the corner border positions of the window and along each of the two nearest side borders for some distance from the corner of the window. The top left scale handle, for example, is located along the left portion of the top border and along the top portion of the left border. As such, none of the eight scale handles share pixels, and they collectively occupy all window border pixels. The distance that a corner scale handle extends into the side border points is indicated by the following global variable:

| *cornerdistance* : $\mathbb{N}_1$

Given a window and the direction of a scale handle, the set of virtual screen pixels for the side scale handles can be identified as follows:

$SideScaleHandlePixels : Window \times direction \rightarrow \mathbb{P} VScreenPixel$

$\forall w : Window; dir : direction; pvsp : \mathbb{P} VScreenPixel \bullet$
 $SideScaleHandlePixels(w, dir) = pvsp \Rightarrow$
 $(let\ topleftvsp == NearVScreenPixel(WindowBorderPoints(w));$
 $botrightvsp == FarVScreenPixel(WindowBorderPoints(w)) \bullet$
 $dir = Top \Rightarrow pvsp = \{vsp : VScreenPixel \mid$
 $vsp \in WindowBorderPoints(w) \wedge$
 $topleftvsp.x + cornerdistance \leq vsp.x \leq$
 $botrightvsp.x - cornerdistance \wedge$
 $topleftvsp.y \leq vsp.y \leq topleftvsp.y + borderwidth \bullet vsp\} \wedge$
 $dir = Right \Rightarrow pvsp = \{vsp : VScreenPixel \mid$
 $vsp \in WindowBorderPoints(w) \wedge$
 $botrightvsp.x - borderwidth \leq vsp.x \leq botrightvsp.x \wedge$
 $topleftvsp.y + cornerdistance \leq vsp.y \leq$
 $botrightvsp.y - cornerdistance \bullet vsp\} \wedge$
 $dir = Bottom \Rightarrow pvsp = \{vsp : VScreenPixel \mid$
 $vsp \in WindowBorderPoints(w) \wedge$
 $topleftvsp.x + cornerdistance \leq vsp.x \leq$
 $botrightvsp.x - cornerdistance \wedge$
 $botrightvsp.y - borderwidth \leq vsp.y \leq$
 $botrightvsp.y \bullet vsp\} \wedge$
 $dir = Left \Rightarrow pvsp = \{vsp : VScreenPixel \mid$
 $vsp \in WindowBorderPoints(w) \wedge$
 $topleftvsp.x \leq vsp.x \leq topleftvsp.x + borderwidth \wedge$
 $topleftvsp.y + cornerdistance \leq vsp.y \leq$
 $botrightvsp.y - cornerdistance \bullet vsp\})$

The set of virtual screen pixels for the corner scale handles can be identified as follows:

CornerScaleHandlePixels : *Window* $\times$ *direction* $\leftrightarrow \mathbb{P}$ *VScreenPixel*

$\forall w : \text{Window}; \text{dir} : \text{direction}; \text{pvsp} : \mathbb{P} \text{VScreenPixel} \bullet$
CornerScaleHandlePixels(*w*, *dir*) = *pvsp* $\Rightarrow$
 (let *toleftvsp* == *NearVScreenPixel*(*WindowBorderPoints*(*w*));
 botrightvsp == *FarVScreenPixel*(*WindowBorderPoints*(*w*)) $\bullet$
dir = *TopRight* $\Rightarrow$ *pvsp* = {*vsp* : *VScreenPixel* |
 vsp $\in$ *WindowBorderPoints*(*w*) $\wedge$
 (*FarVScreenPixel*(*SideScaleHandlePixels*(*w*, *Top*))).*x* + 1 $\leq$
 vsp.x $\leq$ *botrightvsp.x* $\wedge$
 (*NearVScreenPixel*(*SideScaleHandlePixels*(*w*, *Right*))).*y* - 1 $\leq$
 vsp.y $\leq$ *toleftvsp.y* $\bullet$ *vsp*} $\wedge$
dir = *TopLeft* $\Rightarrow$ *pvsp* = {*vsp* : *VScreenPixel* |
 vsp $\in$ *WindowBorderPoints*(*w*) $\wedge$
 toleftvsp.x $\leq$ *vsp.x* $\leq$
 (*NearVScreenPixel*(*SideScaleHandlePixels*(*w*, *Top*))).*x* - 1 $\wedge$
 toleftvsp.y $\leq$ *vsp.y* $\leq$
 (*NearVScreenPixel*(*SideScaleHandlePixels*(*w*, *Left*))).*y* - 1 $\bullet$ *vsp*} $\wedge$
dir = *BottomRight* $\Rightarrow$ *pvsp* = {*vsp* : *VScreenPixel* |
 vsp $\in$ *WindowBorderPoints*(*w*) $\wedge$
 (*FarVScreenPixel*(*SideScaleHandlePixels*(*w*, *Bottom*))).*x* + 1 $\leq$
 vsp.x $\leq$ *botrightvsp.x* $\wedge$
 (*FarVScreenPixel*(*SideScaleHandlePixels*(*w*, *Right*))).*y* + 1 $\leq$
 vsp.y $\leq$ *toleftvsp.y* $\bullet$ *vsp*} $\wedge$
dir = *BottomLeft* $\Rightarrow$ *pvsp* = {*vsp* : *VScreenPixel* |
 vsp $\in$ *WindowBorderPoints*(*w*) $\wedge$
 toleftvsp.x $\leq$ *vsp.x* $\leq$
 (*NearVScreenPixel*(*SideScaleHandlePixels*(*w*, *Bottom*))).*x* - 1 $\wedge$
 (*FarVScreenPixel*(*SideScaleHandlePixels*(*w*, *Left*))).*y* + 1 $\leq$
 vsp.y $\leq$ *botrightvsp.y* $\bullet$ *vsp*}}

The general function for returning scale handle pixels for any of the eight directions can be specified as follows:

ScaleHandlePixels : *Window* $\times$ *direction* $\leftrightarrow \mathbb{P}$ *VScreenPixel*

$\forall w : \text{Window}; \text{dir} : \text{direction}; \text{pvsp} : \mathbb{P} \text{VScreenPixel} \bullet$
ScaleHandlePixels(*w*, *dir*) = *pvsp* $\Rightarrow$
 (*SideScaleHandlePixels*(*w*, *dir*) = *pvsp* $\vee$
 CornerScaleHandlePixels(*w*, *dir*) = *pvsp*)

Overlapping Components of a window cannot overlap, with the exception of the control that has focus. This control may overlap any part of the window, or any other window as well. This property is asserted by the following function which

returns the virtual screen pixels to which a window maps in addition to specifying this overlapping restriction:

$$\begin{array}{|l}
\hline
AllWindowPoints : Window \leftrightarrow \mathbb{P} VScreenPixel \\
\hline
\forall w : Window; pvsp : \mathbb{P} VScreenPixel \bullet \\
\quad AllWindowPoints(w) = pvsp \Rightarrow \\
\quad (\cap \{ UnfocusedControlPoints(w), MoveHandlePixels(w), \\
\quad \quad \cup (ScaleHandlePixels(\{ \{ dir : direction \bullet (w, dir) \} \}) \}) = \emptyset \wedge \\
\quad pvsp = AllVSRPoints(WindowVSR(w)) \cup FocusedControlPoints(w))
\end{array}$$

The *AllWindowPoints* function ensures that the set of pixels to which all unfocused controls map, the set of pixels to which the move handle maps, and the set of pixels to which each of the eight scale handles maps do not intersect. Since there are eight distinct directions, the set of all scale handle pixels groups is first computed using the relational image operation ($\Downarrow$) on the set of ordered pairs containing all possible directions as the first element and the window as the second element. The generalized union operation ($\cup$) is then applied to the set of pixel groups to return the set of all pixels comprising the window's scale handles.

6.3 Operations On A Window

In this section, two main categories of window operations are discussed. The first is concerned with manipulating the controls associated with a window, which include operations to manage the focus of controls. The second category includes operations that scale and move a window.

Any operation performed on a window preserves the window identifier, the caption text, and the value of the *modal* property. The states for any controls associated with a window may be altered by an operation, however controls cannot be added to or removed from the window. These restrictions can be specified by the following:

$$\begin{array}{|l}
\hline
OpOnWindow \text{ ————— } \\
\Delta Window \\
\hline
wid' = wid \\
caption' = caption \\
dom\ controls' = dom\ controls \\
modal' = modal
\end{array}$$

As with control operations, all window operations use a specific mouse button, as indicated by a global identifier:

| *ButtonForWindowOps : ButtonPosition*

Manipulation of Controls The control that is currently being manipulated by the user with the mouse becomes the focused control in the window. When focus is switched, two separate events occur: the control originally having focus loses focus and the control to which focus moves receives focus. These two events can be specified separately, and then combined to specify the switching operation.

To give focus to a control, its identifier must be assigned to the window's *focus-control* property:

<i>GiveFocus</i> <i>OpOnWindow</i> <i>wcid? : ID</i>
<i>wcid? ∈ dom controls \ {focuscontrol}</i> <i>focuscontrol' = wcid?</i> <i>controls' = controls</i>

Removing focus from a control is less straightforward. When a drop down combo box control loses focus, its floating list box component must vanish. Similarly, drop down menus associated with a menu bar control must disappear from view. All other control types have no special properties when focus is removed from them. The removal of focus from a control implies that no control will have focus in the final state. The following operation specifies the removal of focus from the control which had focus in the initial state:

<i>RemoveFocus</i>	
<i>OpOnWindow</i>	
$ \begin{aligned} & (controls(focuscontrol)).type = IsaComboBoxDrop \Rightarrow \\ & \quad controls' = controls \oplus \{focuscontrol \mapsto (\mu OpOnWindowControl \mid \\ & \quad \quad \theta WindowControl = controls(focuscontrol) \wedge \\ & \quad \quad picklist' \in \emptyset[ListBox] \wedge \\ & \quad \quad dropbutton' = dropbutton \wedge \\ & \quad \quad textfield' = textfield \wedge \\ & \quad \quad static' = static \wedge \\ & \quad \quad enabled' = enabled \bullet \theta WindowControl')\} \wedge \\ & (controls(focuscontrol)).type = IsaMenuBar \Rightarrow \\ & \quad controls' = controls \oplus \{focuscontrol \mapsto (\mu OpOnWindowControl \mid \\ & \quad \quad \theta WindowControl = controls(focuscontrol) \wedge \\ & \quad \quad menuitems' = menuitems \wedge \\ & \quad \quad disporde' = disporde \wedge \\ & \quad \quad dropdownmenus' \in \emptyset[DropDownMenusSeq] \bullet \theta WindowControl')\} \wedge \\ & (controls(focuscontrol)).type \neq IsaComboBoxDrop \wedge \\ & \quad (controls(focuscontrol)).type \neq IsaMenuBar \Rightarrow \\ & \quad \quad controls' = controls \wedge \\ & \quad focuscontrol' \in \emptyset[ID] \end{aligned} $	

As an illustration, if a drop down combo box control has focus removed, this operation has the effect of assigning an “empty” value to the *picklist* property through the expression $picklist' \in \emptyset[ListBox]$.

Switching focus is accomplished by removing focus from the control currently having focus and giving it to another control. This operation can be specified by using schema composition of the *RemoveFocus* and *GiveFocus* schema operations with the additional criteria that the control receiving focus did not originally have focus. This restriction is added to the schema composition by use of schema conjunction:

$$\begin{aligned}
SwitchFocus \triangleq & [OpOnWindow; wcid? : ID \mid wcid? \neq focuscontrol] \wedge \\
& (RemoveFocus \ ; \ GiveFocus)
\end{aligned}$$

Most operations defined on controls in Chapter 5 can be promoted to window operations in a straightforward manner. However, focus management must be added to these specifications. Since the promotion technique is similar for all controls, the *ButtonIsPushed* operations will be promoted to illustrate the technique. Using Hall’s style, an operation that specifies the state change of a window control can be promoted to specify the state change of a window provided that the initial window control state

is included as a part of the initial state of the given window. In its simplest form, with focus management not included, this can be specified as follows:

$WButtonIsPushed0$	_____
$OpOnWindow$ $wcid? : ID$	
$wcid? = focuscontrol$ $controls' = controls \oplus \{wcid? \mapsto (\mu ButtonIsPushed_1 \mid$ $\theta WindowControl_1 = controls(wcid?) \bullet \theta WindowControl'_1)\}$	

The addition of focus management requires that focus changes to the control on which the operation is performed just prior to the operation being performed. This can be specified as follows:

$$WButtonIsPushed \triangleq SwitchFocus ; WButtonIsPushed0$$

Scaling and Moving a Window A window can be scaled or moved. When a window is enlarged, a number of its associated scrollable regions may be enlarged as well. Similarly, when a window is shrunk, some of its scrollable regions may shrink. When a window is moved, all controls must be moved along with the window, and so their screen pixels must move by the same offsets as the window. In all of these cases, the properties of all controls in the window remain unchanged and no control has focus after these operations are performed. First, the basic operations are specified:

EnlargeWindow0

OpOnWindow

direction? : *direction*

$$\begin{aligned} & \exists vsp, vsp_1 : VScreenPixel; vsr, vsr_1 : VScreenRegion \bullet \\ & \quad \{vsp\} = AllVSRPoints(vsr) \wedge \{vsp_1\} = AllVSRPoints(vsr_1) \wedge \\ & \quad vsp \in ScaleHandlePixels(\theta Window, direction?) \wedge \\ & \quad vsp_1 \in ScaleHandlePixels(\theta Window', direction?) \wedge \\ & \quad (vsr_1, mousenow) = MouseDrag_n_Drop(vsr, ButtonForWindowOps) \wedge \\ & \quad WindowVSR(\theta Window') = \\ & \quad \quad (\mu EnlargeVSR \mid \theta VScreenRegion = WindowVSR(\theta Window) \wedge \\ & \quad \quad \quad vspixel? = vsp_1 \bullet \theta VScreenRegion') \\ & \quad direction? \in \{Left, Right, TopLeft, TopRight\} \Rightarrow \\ & \quad \quad \#(MoveHandlePixels(\theta Window')) > \#(MoveHandlePixels(\theta Window)) \\ & \exists ids : \mathbb{P} ID \bullet \\ & \quad ids \subseteq \text{dom controls} \wedge \\ & \quad (\forall id : ids \bullet \\ & \quad \quad (controls(id)).type = IsaScrollRegion \wedge \\ & \quad \quad controls'(id) = (\mu EnlargeScrollRegion \mid \\ & \quad \quad \quad \theta WindowControl = controls(id) \bullet \theta WindowControl')) \\ & controls' = controls \end{aligned}$$

This schema asserts the following: i) the window is enlarged by the drag and drop action taking place on a one-pixel region in the handle given by *direction?*. The actual enlargement of the window is performed by the *EnlargeVSR* operation applied against the window's virtual screen region; ii) since the move handle is located as a horizontal bar across the top of the window, any enlarging in the left or right direction will also enlarge the move handle; and iii) a subset of the scroll region controls associated with the window may also be enlarged.

The shrink operation is specified in a similar manner:

ShrinkWindow0

OpOnWindow

direction? : *direction*

$$\begin{aligned} & \exists \text{vsp}, \text{vsp}_1 : \text{VScreenPixel}; \text{vsr}, \text{vsr}_1 : \text{VScreenRegion} \bullet \\ & \quad \{ \text{vsp} \} = \text{AllVSRPoints}(\text{vsr}) \wedge \{ \text{vsp}_1 \} = \text{AllVSRPoints}(\text{vsr}_1) \wedge \\ & \quad \text{vsp} \in \text{ScaleHandlePixels}(\theta \text{Window}, \text{direction?}) \wedge \\ & \quad \text{vsp}_1 \in \text{ScaleHandlePixels}(\theta \text{Window}', \text{direction?}) \wedge \\ & \quad (\text{vsr}_1, \text{mousenow}) = \text{MouseDown_Drop}(\text{vsr}, \text{ButtonForWindowOps}) \wedge \\ & \quad \text{WindowVSR}(\theta \text{Window}') = \\ & \quad \quad (\mu \text{ShrinkVSR} \mid \theta \text{VScreenRegion} = \text{WindowVSR}(\theta \text{Window}) \wedge \\ & \quad \quad \text{vspixel?} = \text{vsp}_1 \bullet \theta \text{VScreenRegion}') \\ & \text{direction?} \in \{ \text{Left}, \text{Right}, \text{TopLeft}, \text{TopRight} \} \Rightarrow \\ & \quad \#(\text{MoveHandlePixels}(\theta \text{Window}')) < \#(\text{MoveHandlePixels}(\theta \text{Window})) \\ & \exists \text{ids} : \mathbb{P} \text{ID} \bullet \\ & \quad \text{ids} \subseteq \text{dom controls} \wedge \\ & \quad (\forall id : \text{ids} \bullet \\ & \quad \quad (\text{controls}(id)).\text{type} = \text{IsaScrollRegion} \wedge \\ & \quad \quad \text{controls}'(id) = (\mu \text{ShrinkScrollRegion} \mid \\ & \quad \quad \quad \theta \text{WindowControl} = \text{controls}(id) \bullet \theta \text{WindowControl}')) \\ & \text{controls}' = \text{controls} \end{aligned}$$

The window move operation is accomplished by dragging a pixel in the move handle to a new position on the screen. The window will move such that the relative position of the pixel in the window will remain the same before and after the move. All pixels in the window will be remapped to the new position. The move operation can be asserted by the following:

MoveWindow0

OpOnWindow

$$\begin{aligned} & \exists \text{vsp} : \text{VScreenPixel} \bullet \\ & \quad \text{vsp} \in \text{MoveHandlePixels}(\theta \text{Window}) \wedge \\ & \quad (\text{WindowVSR}(\theta \text{Window}'), \text{mousenow}) = \\ & \quad \quad \text{MouseDown_Drop}(\text{WindowVSR}(\theta \text{Window}), \text{ButtonForWindowOps}) \wedge \\ & \quad \text{WindowVSR}(\theta \text{Window}') = \\ & \quad \quad (\mu \text{MoveVSR} \mid \theta \text{VScreenRegion} = \text{WindowVSR}(\theta \text{Window}) \wedge \\ & \quad \quad \quad \text{vspixel?} = \text{vsp} \bullet \theta \text{VScreenRegion}') \\ & \quad \#(\text{MoveHandlePixels}(\theta \text{Window}')) = \#(\text{MoveHandlePixels}(\theta \text{Window})) \\ & \quad (\text{let } x\text{move} == (\text{NearVScreenPixel}(\text{AllWindowsPoints}(\theta \text{Window}'))).x - \\ & \quad \quad (\text{NearVScreenPixel}(\text{AllWindowsPoints}(\theta \text{Window}))).x; \\ & \quad \quad y\text{move} == (\text{FarVScreenPixel}(\text{AllWindowsPoints}(\theta \text{Window}'))).y - \\ & \quad \quad (\text{FarVScreenPixel}(\text{AllWindowsPoints}(\theta \text{Window}))).y \bullet \\ & \quad (\forall \text{vsp} : \text{AllWindowsPoints}(\theta \text{Window}) \bullet \\ & \quad \quad (\exists_1 \text{vsp}_1 : \text{AllWindowsPoints}(\theta \text{Window}') \bullet \\ & \quad \quad \quad \text{vsp}_1.x - \text{vsp}.x = x\text{move} \wedge \\ & \quad \quad \quad \text{vsp}_1.y - \text{vsp}.y = y\text{move}))) \\ & \quad \text{controls}' = \text{controls} \end{aligned}$$

The operations to enlarge, shrink and move a window assume that no control currently has focus. To include this assumption as part of the specification, all three operation are composed with the *RemoveFocus* operation.

$\text{EnlargeWindow} \triangleq \text{RemoveFocus} ; \text{EnlargeWindow0}$

$\text{ShrinkWindow} \triangleq \text{RemoveFocus} ; \text{ShrinkWindow0}$

$\text{MoveWindow} \triangleq \text{RemoveFocus} ; \text{MoveWindow0}$

The window is an important component of a GUI since it gathers together all controls. In the next chapter, the window type will be used to complete the basic GUI specifications in this thesis.

Chapter 7

The Display

A *display* is an abstract model of the physical screen on which windows and their components appear and on which the mouse cursor moves. Many windows can appear simultaneously on a display, and they may overlap or even hide one another. A detailed specification of a display that permits overlapping and occlusion of windows requires a three-dimensional model. That is, a depth parameter must be added to each overlapping or occluding window that shows the relative ordering of windows in the display. For simplicity, such a detailed specification is not covered in this thesis, and readers are referred to [12] for further details.

In order to precisely specify the relationship between windows in a display, a global binary operator, namely `_WOverlaps_` is introduced here; only the signature of `_WOverlaps_` is given for brevity:

| `_WOverlaps_ : Window  $\leftrightarrow$  Window`

7.1 Structure

A display contains a number of distinct windows. These windows may overlap with each other, except for the *active* window which must not be obscured. The active window is the one which contains a control currently having focus; it is dynamic in the sense that another window can be made active by the user provided that the active window is not modal. Only one window can be active on a display.

Display

$windows : ID \leftrightarrow Window$

$points : VScreenPixel \rightarrow seq ID$

$activewindow : ID$

$dom\ windows = \{w : ran\ windows \bullet w.wid\}$

$activewindow \in dom\ windows$

$\forall s : ran\ points \bullet$

$\#s = \#(ran\ s)$

$\bigcup \{s : ran\ points \bullet ran\ s\} = dom\ windows$

$\forall vsp : dom\ points \bullet$

$(\forall i : 1 .. \#(points(vsp)) \bullet$

$vsp \in AllWindowPoints(windows(points(vsp)(i))) \wedge$

$vsp.x \leq displayXmax \wedge$

$vsp.y \leq displayYmax) \wedge$

$(\forall i : 1 .. \#(points(vsp)) - 1 \bullet$

$windows(points(vsp)(i)) WOverlaps\ windows(points(vsp)(i + 1))) \wedge$

$head(points(vsp)) = activewindow$

In the schema *Display*, the variable *windows* indicates the set of windows on the display. The variable *activewindow* refers to the window in this group on which operations are currently being performed. An interesting part of this specification is the way in which multiple pixel mapping is specified. Each virtual screen pixel on the display is associated with a sequence of window identifiers. This sequence represents, from front to back, the order of the windows that map to the pixel. All windows in the display participate in this mapping.

7.2 Operations on a Display

Operations that can be performed on a display fall into three categories: i) the operation to change the active window; ii) operations that promote window operations to display operations; and iii) menu operations to invoke new windows. For all operations, the pixels in the display remain unchanged:

OpOnDisplay

$\Delta Display$

$dom\ points' = dom\ points$

As with control and window operations, the mouse button used for display operations

is indicated by a global identifier:

| *ButtonForWindowSelection* : *ButtonPosition*

Changing the Active Window A window becomes active in a display when it is clicked, double clicked or the mouse button is held down over a portion of it, provided that the currently active window is not *modal*. If this is the case, the modal window can become inactivated only by the application itself:

<i>ChangeActiveWindow</i> <i>OpOnDisplay</i> <i>wid?</i> : <i>ID</i> $wid? \in \text{dom } windows \setminus \{activewindow\}$ $(windows(activewindow)).modal = No$ let <i>vsr</i> == <i>WindowVSR</i>(<i>windows</i>(<i>wid?</i>)) • $(MouseClickObject(vsr, ButtonForWindowSelection, mousenow) \vee$ $MouseHoldDownObject(vsr, ButtonForWindowSelection, mousenow) \vee$ $MouseDbClickObject(vsr, ButtonForWindowSelection, mousenow))$ $activewindow' = wid?$ $windows' = windows$

Promoted Window Operations There are many window operations, and all are promoted in the same manner. Due to space limitations, only promotion of the *WButtonIsPushed* schema operation is illustrated here. Given a window identifier corresponding to the active window in the display and a push button control identifier corresponding to the control having focus in the window, a push button control can be pushed as asserted by the following schema:

<i>DButtonIsPushed</i> <i>OpOnDisplay</i> <i>wid?</i> : <i>ID</i> <i>wcid?</i> : <i>ID</i> $wid? = activewindow$ $wcid? = (windows(wid?)).focuscontrol$ $windows' = windows \oplus \{wid? \mapsto (\mu WButtonIsPushed \mid$ $\theta Window = windows(wid?) \bullet \theta Window')\}$ $activewindow' = activewindow$

Menu Control Operations In Chapter 5, a menu item type was defined that could change the active window of an application when selected. Partial specifications were provided for selection of this menu item type in a popup menu, a drop down menu and a menu bar. Operations for the selection of this menu item type for each of the menu types can now be elaborated further. First, the operations as defined previously must be promoted to window operations. This is performed as illustrated in Chapter 6. Then, the window operations are promoted to display operations. This will connect the window identifier associated with the menu item that has been selected to the identifier of the new active window in the display.

The promoted window operations for each of the menu types follow:

$WPNewWindowID0$ <i>OpOnWindow</i> <i>wcid?</i> : <i>ID</i> <i>miid?</i> : <i>ID</i> <i>windowid!</i> : <i>ID</i> <i>wcid?</i> = <i>focuscontrol</i> <i>windowid!</i> = <i>PNewWindowID</i> (<i>controls</i> (<i>wcid?</i>), <i>miid?</i>) <i>controls'</i> = <i>controls</i>
--

$WMBNewWindowID0$ <i>OpOnWindow</i> <i>wcid?</i> : <i>ID</i> <i>miid?</i> : <i>ID</i> <i>windowid!</i> : <i>ID</i> <i>wcid?</i> = <i>focuscontrol</i> <i>windowid!</i> = <i>MBNewWindowID</i> (<i>controls</i> (<i>wcid?</i>), <i>miid?</i>) <i>controls'</i> = <i>controls</i>
--

$WMB_DDMNewWindowID0$ <i>OpOnWindow</i> <i>wcid?</i> : <i>ID</i> <i>miid?</i> : <i>ID</i> <i>windowid!</i> : <i>ID</i> <i>wcid?</i> = <i>focuscontrol</i> <i>controls'</i> = <i>controls</i> $\oplus$ { <i>wcid?</i> $\mapsto$ (μ <i>MB_DDMNewWindowID</i> θ <i>WindowControl</i> = <i>controls</i> (<i>wcid?</i>) $\bullet$ θ <i>WindowControl'</i>) }

$WPNewWindowID \triangleq SwitchFocus \ ; \ WPNewWindowID0$
 $WMBNewWindowID \triangleq SwitchFocus \ ; \ WMBNewWindowID0$
 $WMB_DDMNewWindowID \triangleq SwitchFocus \ ; \ WMB_DDMNewWindowID0$

The window operations can now be promoted to display operations. Given the identifier for the popup menu, the identifier for the menu item selected from the popup menu, and the window to display as input, the operations must ensure that the newly active window corresponds to that indicated by the menu item. This window may or may not be already contained in the set of windows associated with the display; if it is not, it is added to the set:

$DNewWindowFromPopupMenu$
$OpOnDisplay$ $window? : Window$ $wcid? : ID$ $miid? : ID$
$window?.wid \neq (windows(activewindow)).wid \wedge$ $windows' = windows \oplus \{activewindow \mapsto (\mu \ WPNewWindowID \mid$ $\quad \theta Window = windows(activewindow) \wedge$ $\quad windowid! = window?.wid \bullet \theta Window')\} \cup$ $\quad \{window?.wid \mapsto window?\}$ $activewindow' = window?.wid$

Activating a window from a menu bar selection and from a drop down menu selection is specified in a similar way:

$DNewWindowFromMenuBar$
$OpOnDisplay$ $window? : Window$ $wcid? : ID$ $miid? : ID$
$window?.wid \neq (windows(activewindow)).wid \wedge$ $windows' = windows \oplus \{activewindow \mapsto (\mu \ WMBNewWindowID \mid$ $\quad \theta Window = windows(activewindow) \wedge$ $\quad windowid! = window?.wid \bullet \theta Window')\} \cup$ $\quad \{window?.wid \mapsto window?\}$ $activewindow' = window?.wid$

DNewWindowFromDropDownMenu _____

OpOnDisplay

window? : *Window*

wcid? : *ID*

miid? : *ID*

$window?.wid \neq (windows(activewindow)).wid \wedge$
 $windows' = windows \oplus \{activewindow \mapsto (\mu WMB_DDMNewWindowID \mid$
 $\theta Window = windows(activewindow) \wedge$
 $windowid! = window?.wid \bullet \theta Window')\} \cup$
 $\{window?.wid \mapsto window?\}$
 $activewindow' = window?.wid$

Chapter 8

A Simple Diagramming Tool

In this chapter, the specifications for a simple diagramming tool are described. These specifications can be reused for any diagramming tool including the CASE tool to which this thesis is targeted. A diagramming tool has several characteristics: i) a diagram is composed of three kinds of visible objects, namely *symbols*, *connections* and *text blocks*; ii) the creation, deletion, modification and movement of these objects are all performed using the mouse and other GUI controls; iii) the tool must be able to display an entire diagram. If it is too large to be displayed at one time, then scrolling facilities must be provided so that the user can navigate through the diagram; and iv) several restrictions on the placement of objects within the diagram may be imposed to ensure that the diagram follows a simple format. For example, a connection object must be placed between two symbols, and it cannot overlap with any other symbol or connection. The specifications defined up to this point in the thesis can be used as a framework around which a simple diagramming tool can be specified.

This chapter is organized as follows: first, diagram objects and a simple diagram type will be defined. The second section details the visual aspects and restrictions placed on the positioning of the objects within a diagram, culminating in the specification of an axiom that determines the validity of a simple diagram. Operations on individual diagram objects are specified in the third section, and in the fourth section, these operations are used in part to create operations on a diagram as a whole. In the final section, the simple diagram itself is linked together with the scroll region control specified in Chapter 5, and the full set of operations that can be performed by the user with the mouse on the diagram through the control is specified.

8.1 Diagram Objects and a Simple Diagram

There are three kinds of objects in a simple diagram. The first object, the *text block*, can be used almost anywhere in a diagram to provide a narrative or a description. The second object type, the *symbol*, has a shape associated with it that can be used to convey its meaning in the diagram. Since the meaning of a shape depends on the domain in which the diagramming tool is used, this property is not elaborated in this chapter. A text block label is also associated with a symbol to provide a description. The third type of diagram object, the *connection*, can be placed between symbols. It has a shape attribute that represents the visual appearance of the connection. For example, the connection may appear as a single directed line, or as an undirected line. It also has a text block label associated with it that indicates the purpose of the connection.

A text block is comprised of one or more lines of text, and can be specified as follows:

<i>TextBlock</i>
<i>cfid</i> : <i>ID</i>
<i>lines</i> : seq <i>STRING</i>
<i>#lines</i> > 0

The variable *cfid* indicates the coordinate frame identifier to which the text block object belongs. This property will be used later to group objects together in a diagram and to provide a means by which the objects map to virtual screen pixels.

Symbols and connections have shapes which are assumed to belong to a basic type as stated below:

[*OBJECTSHAPE*]

A symbol has a shape and a text block associated with it. The text block can be used to label the symbol:

<i>Symbol</i>
<i>cfid</i> : <i>ID</i>
<i>shape</i> : <i>OBJECTSHAPE</i>
<i>text</i> : <i>TextBlock</i>
<i>text.cfid</i> = <i>cfid</i>

The predicate in the above specification ensures that the text block component of the symbol has the same coordinate frame identifier as the symbol.

A connection also has a shape and an associated text block. Connections are defined as a series of connected line segments; this is represented as a sequence of distinct pixels where every pair of adjacent pixels indicates the end points in one line segment. There must be a minimum of two distinct points to establish a connection between two symbols:

<i>Connection</i>
<i>cfid</i> : <i>ID</i> <i>shape</i> : <i>OBJECTSHAPE</i> <i>text</i> : <i>TextBlock</i> <i>points</i> : seq <i>WorkPixel</i>
$head(points) \neq last(points)$ $\#points = \#(ran\ points)$ $\#points \geq 2$ $\forall wp : WorkPixel \bullet$ $\langle wp \rangle \text{ in } points \Rightarrow$ $wp.cfid = cfid$ $text.cfid = cfid$

The predicates in *Connection* assert that: i) the connection starts and terminates at two different points; ii) there are at least two points associated with a connection, representing the end points on one line segment; iii) all pixels in the connection belong to the same coordinate frame as the connection object; and iv) the text block belongs to the same coordinate frame as the connection.

Using Hall's style, a single *DiagramObject* type can be specified as follows:

$PObject \cong Symbol \vee Connection \vee TextBlock$
 $PObjectType ::= IsaSymbol \mid IsaConnection \mid IsaTextBlock$

<i>DiagramObject</i>
<i>doid</i> : <i>ID</i> <i>PObject</i> <i>type</i> : <i>PObjectType</i>
$type = IsaSymbol \wedge Symbol \vee$ $type = IsaConnection \wedge Connection \vee$ $type = IsaTextBlock \wedge TextBlock$

A *simple diagram* can now be specified in non-visual terms. It consists of a number of diagram objects, all defined within the same coordinate frame:

<i>SimpleDiagram</i>
$cfid : ID$
$diagramobjects : ID \rightarrow DiagramObject$
$dom\ diagramobjects =$ $\{do : DiagramObject \mid do \in \text{ran } diagramobjects \bullet do.doid\}$ $\forall do : \text{ran } diagramobjects \bullet$ $do.cfid = cfid$

8.2 Semantics of a Simple Diagram

The semantics of a diagram created using a diagramming tool include the semantics of each diagram object in the diagram, the semantic relationships between the diagram objects, and any constraints on the visual characteristics of these diagram objects on the display. Each of these issues are addressed in this section.

8.2.1 Diagram Object Pixel Mappings

In this section, the visual characteristics of the three diagram objects are specified.

Unlike the specifications for windows and controls discussed in the previous chapters, the visual aspects of diagram objects describe the mapping of these objects to work pixels, rather than to virtual screen pixels. There are two reasons for this; first, mapping of work pixels to virtual screen pixels has already been specified in the previous chapters. Therefore, it is not difficult to derive the mapping of a diagram object to a set of virtual screen pixels when the object appears on the screen. Second, even when a diagram object is occluded from the display, the constraints on its placement are still valid as long as the object exists in the work area and hence the diagram.

Image Points A text block consists of a sequence of words or strings. Hence, it is appropriate to extend the mapping of a string as such:

$$\mid \text{StringImagePoints} : STRING \rightarrow \mathbb{P} \text{WorkPixel}$$

Note that this function is not complete since it depends on *STRING*, which is assumed to be a basic type in this thesis.

A text block may contain a number of lines of text, represented as a sequence of strings, none of which may overlap:

$$\begin{array}{|l}
\text{TextBlockImagePoints} : \text{TextBlock} \leftrightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall tb : \text{TextBlock}; wps : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{TextBlockImagePoints}(tb) = wps \Rightarrow \\
\quad (\cup \{l : \text{STRING} \mid \langle l \rangle \text{ in } tb.\text{lines} \bullet \text{StringImagePoints}(l)\} \subseteq wps \wedge \\
\quad (\forall l, l_1 : \text{STRING} \bullet \\
\quad \quad \{l, l_1\} \subseteq \text{ran } tb.\text{lines} \wedge \\
\quad \quad \text{StringImagePoints}(l) \cap \text{StringImagePoints}(l_1) = \emptyset))
\end{array}$$

The set of work pixels containing a symbol's image must also contain the symbol's text label since the label is part of the image. Although the shape of a symbol determines the exact pixels included in this set, this level of detail is not included in these specifications. The pixels to which a symbol maps can be specified as follows:

$$\begin{array}{|l}
\text{SymbolImagePoints} : \text{Symbol} \leftrightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall s : \text{Symbol}; wps : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{SymbolImagePoints}(s) = wps \Rightarrow \\
\quad \text{TextBlockImagePoints}(s.\text{text}) \subset wps
\end{array}$$

A connection's image must contain the text label, as well as the pixels to which each of its line segments map. The image points for a single line segment can be viewed as a set of n pixels located between the two end points, where n is equal in value to the change in x or the change in y , whichever is greater. At this level of specification, the connection's shape is not considered in determining a connection's image points:

$$\begin{array}{|l}
\text{LineSegmentImagePoints} : \text{WorkPixel} \times \text{WorkPixel} \leftrightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall wpfrom, wpto : \text{WorkPixel}; wps : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{LineSegmentImagePoints}(wpfrom, wpto) = wps \Rightarrow \\
\quad (\exists wr : \text{WorkRegion} \bullet \\
\quad \quad \{wr.wtopleft, wr.wbotright\} = \{wpfrom, wpto\} \wedge \\
\quad \quad wps \subseteq \text{AllWRPoints}(wr) \wedge \\
\quad \quad \#wps = \max(\{\text{WRHeight}(wr), \text{WRWidth}(wr)\}))
\end{array}$$

The set of pixels to which a connection maps includes the set of pixels to which each of the individual line segments map. As well, line segments in a connection cannot cross:

$$\begin{array}{|l}
\hline
\text{ConnectionImagePoints} : \text{Connection} \rightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall c : \text{Connection}; wps : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{ConnectionImagePoints}(c) = wps \Rightarrow \\
\quad (\text{TextBlockImagePoints}(c.\text{text}) \subseteq wps \wedge \\
\quad (\forall i, j : 1 \dots \#c.\text{points} - 1 \bullet \\
\quad \quad \text{LineSegmentImagePoints}(c.\text{points}(i), c.\text{points}(i+1)) \subseteq wps \wedge \\
\quad \quad \text{LineSegmentImagePoints}(c.\text{points}(i), c.\text{points}(i+1)) \cap \\
\quad \quad \text{LineSegmentImagePoints}(c.\text{points}(j), c.\text{points}(j+1)) = \emptyset))
\end{array}$$

Finally, the set of work pixels to which a diagram object maps, in general, can be specified as follows:

$$\begin{array}{|l}
\hline
\text{DOImagePoints} : \text{DiagramObject} \rightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall do : \text{DiagramObject}; wps : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{DOImagePoints}(do) = wps \Rightarrow \\
\quad (do.\text{type} = \text{IsaTextBlock} \Rightarrow \\
\quad \quad wps = \text{TextBlockImagePoints}(do.\theta \text{TextBlock}) \wedge \\
\quad do.\text{type} = \text{IsaSymbol} \Rightarrow \\
\quad \quad wps = \text{SymbolImagePoints}(do.\theta \text{Symbol}) \wedge \\
\quad do.\text{type} = \text{IsaConnection} \Rightarrow \\
\quad \quad wps = \text{ConnectionImagePoints}(do.\theta \text{Connection}))
\end{array}$$

Edge Points The pixels that correspond to a diagram object's *contour* or outer edge can be determined by the following axiom:

$$\begin{array}{|l}
\hline
\text{DOEdgePoints} : \text{DiagramObject} \rightarrow \mathbb{P} \text{WorkPixel} \\
\hline
\forall do : \text{DiagramObject}; pwp : \mathbb{P} \text{WorkPixel} \bullet \\
\quad \text{DOEdgePoints}(do) = pwp \Rightarrow \\
\quad (pwp \subseteq \text{DOImagePoints}(do) \wedge \\
\quad (\forall wp : pwp \bullet \\
\quad \quad (\exists wpneighbour : \text{WorkPixel} \bullet \\
\quad \quad \quad wpneighbour \notin \text{DOImagePoints}(do) \wedge \\
\quad \quad \quad (\text{WPHAdjacent}(wp, wpneighbour) \vee \\
\quad \quad \quad \text{WPVAdjacent}(wp, wpneighbour))))))
\end{array}$$

The function *DOEdgePoints* returns the set of pixels to which a diagram object maps where each point in the set has a neighbouring pixel that is not part of the image.

Bounding Regions The *bounding region* of a diagram object can be defined as the smallest work region that contains all of the object's image points. It is mainly

used in diagram object operations that require the use of the mouse, since it is easier to manipulate a diagram object through the object's bounding region than directly through its image pixels. Specifications to determine the bounding regions for each of the diagram object types follow. Each of these axioms follows the same structure:

$$\begin{array}{|l}
 \hline
 \textit{TextBlockWorkRegion} : \textit{TextBlock} \leftrightarrow \textit{WorkRegion} \\
 \hline
 \forall tb : \textit{TextBlock}; wr : \textit{WorkRegion} \bullet \\
 \quad \textit{TextBlockWorkRegion}(tb) = wr \Rightarrow \\
 \quad (\textit{TextBlockImagePoints}(tb) \subseteq \textit{AllWRPoints}(wr) \wedge \\
 \quad \neg (\exists wr_1 : \textit{WorkRegion} \bullet \\
 \quad \quad \#(\textit{AllWRPoints}(wr_1)) < \#(\textit{AllWRPoints}(wr)) \wedge \\
 \quad \quad \textit{TextBlockImagePoints}(tb) \subseteq \textit{AllWRPoints}(wr_1)))
 \end{array}$$

$$\begin{array}{|l}
 \hline
 \textit{SymbolWorkRegion} : \textit{Symbol} \leftrightarrow \textit{WorkRegion} \\
 \hline
 \forall s : \textit{Symbol}; wr : \textit{WorkRegion} \bullet \\
 \quad \textit{SymbolWorkRegion}(s) = wr \Rightarrow \\
 \quad (\textit{SymbolImagePoints}(s) \subseteq \textit{AllWRPoints}(wr) \wedge \\
 \quad \neg (\exists wr_1 : \textit{WorkRegion} \bullet \\
 \quad \quad \#(\textit{AllWRPoints}(wr_1)) < \#(\textit{AllWRPoints}(wr)) \wedge \\
 \quad \quad \textit{SymbolImagePoints}(s) \subseteq \textit{AllWRPoints}(wr_1)))
 \end{array}$$

$$\begin{array}{|l}
 \hline
 \textit{ConnectionWorkRegion} : \textit{Connection} \leftrightarrow \textit{WorkRegion} \\
 \hline
 \forall c : \textit{Connection}; wr : \textit{WorkRegion} \bullet \\
 \quad \textit{ConnectionWorkRegion}(c) = wr \Rightarrow \\
 \quad (\textit{ConnectionImagePoints}(c) \subseteq \textit{AllWRPoints}(wr) \wedge \\
 \quad \neg (\exists wr_1 : \textit{WorkRegion} \bullet \\
 \quad \quad \#(\textit{AllWRPoints}(wr_1)) < \#(\textit{AllWRPoints}(wr)) \wedge \\
 \quad \quad \textit{ConnectionImagePoints}(c) \subseteq \textit{AllWRPoints}(wr_1)))
 \end{array}$$

In general, the smallest work region containing a diagram object is specified as:

$$\begin{array}{|l}
 \hline
 \textit{DOWorkRegion} : \textit{DiagramObject} \leftrightarrow \textit{WorkRegion} \\
 \hline
 \forall do : \textit{DiagramObject}; wr : \textit{WorkRegion} \bullet \\
 \quad \textit{DOWorkRegion}(do) = wr \Rightarrow \\
 \quad (do.type = \textit{IsaTextBlock} \Rightarrow \\
 \quad \quad wr = \textit{TextBlockWorkRegion}(do.\theta \textit{TextBlock}) \wedge \\
 \quad do.type = \textit{IsaSymbol} \Rightarrow \\
 \quad \quad wr = \textit{SymbolWorkRegion}(do.\theta \textit{Symbol}) \wedge \\
 \quad do.type = \textit{IsaConnection} \Rightarrow \\
 \quad \quad wr = \textit{ConnectionWorkRegion}(do.\theta \textit{Connection}))
 \end{array}$$

8.2.2 Relationships Between Diagram Objects

The relationships between diagram objects are specified based on the set of work pixels to which each object is mapped.

The images of two diagram objects *overlap* if the intersection of their image pixel sets is non-empty:

$$\begin{array}{|l} \hline _ DOOverlapsDO _ : DiagramObject \leftrightarrow DiagramObject \\ \hline \forall do_1, do_2 : DiagramObject \bullet \\ do_1 DOOverlapsDO do_2 \Leftrightarrow \\ DOImagePoints(do_1) \cap DOImagePoints(do_2) \neq \emptyset \end{array}$$

Two diagram objects *touch* one another if, graphically, their edge points overlap. Note that this is a special case of overlapping:

$$\begin{array}{|l} \hline _ DOTouchesDO _ : DiagramObject \leftrightarrow DiagramObject \\ \hline \forall do_1, do_2 : DiagramObject \bullet \\ do_1 DOTouchesDO do_2 \Leftrightarrow \\ DOEdgePoints(do_1) \cap DOEdgePoints(do_2) \neq \emptyset \end{array}$$

A connection can be attached to a symbol with either its “to” end point or with its “from” end point touching the symbol. A connection’s end point is attached to a symbol if the point is also one of the symbol’s edge points. First, an axiom for each end point connection is provided, followed by a generalized axiom indicating a connection of either end point to a symbol.

$$\begin{array}{|l} \hline _ CFromEndAttachedToS _ : DiagramObject \leftrightarrow DiagramObject \\ \hline \forall c : DiagramObject; s : DiagramObject \bullet \\ c CFromEndAttachedToS s \Leftrightarrow \\ (c.type = IsaConnection \wedge \\ s.type = IsaSymbol \wedge \\ head(c.points) \in DOEdgePoints(s)) \end{array}$$

$$\begin{array}{|l} \hline _ CToEndAttachedToS _ : DiagramObject \leftrightarrow DiagramObject \\ \hline \forall c : DiagramObject; s : DiagramObject \bullet \\ c CToEndAttachedToS s \Leftrightarrow \\ (c.type = IsaConnection \wedge \\ s.type = IsaSymbol \wedge \\ last(c.points) \in DOEdgePoints(s)) \end{array}$$

$$\begin{array}{|l}
\hline
_ CAttachedToS _ : DiagramObject \leftrightarrow DiagramObject \\
\hline
\forall c : DiagramObject; s : DiagramObject \bullet \\
\quad c CAttachedToS s \Leftrightarrow \\
\quad (c CToEndAttachedToS s \vee c CFromEndAttachedToS s)
\end{array}$$

Two symbols are connected to each other if they share a common connection. There are two types of connections defined - directed and undirected. The directed connection can be specified as follows:

$$\begin{array}{|l}
\hline
_ CConnectsFromSToS _ : DiagramObject \leftrightarrow (DiagramObject \times DiagramObject) \\
\hline
\forall do_1, do_2, do_3 : DiagramObject \bullet \\
\quad do_1 CConnectsFromSToS (do_2, do_3) \Leftrightarrow \\
\quad (\neg do_2 DOOverlapsDO do_3 \wedge \\
\quad \quad do_1 CFromEndAttachedToS do_2 \wedge \\
\quad \quad do_1 CToEndAttachedToS do_3)
\end{array}$$

The undirected connection is specified as:

$$\begin{array}{|l}
\hline
_ CConnectsSS _ : DiagramObject \leftrightarrow (DiagramObject \times DiagramObject) \\
\hline
\forall do_1, do_2, do_3 : DiagramObject \bullet \\
\quad do_1 CConnectsSS (do_2, do_3) \Leftrightarrow \\
\quad (\neg do_2 DOOverlapsDO do_3 \wedge \\
\quad \quad do_1 CAttachedToS do_2 \wedge \\
\quad \quad do_1 CAttachedToS do_3)
\end{array}$$

The simple diagram can be validated using the relationships between diagram objects that were just specified. The requirements for a valid diagram include the following rules:

- Symbols and text blocks cannot overlap.
- All connections must be defined between exactly two existing, but not necessarily distinct symbols.
- Connections cannot overlap symbols.

These requirements are formally specified below:

$\text{ValidSimpleDiagram_} : \mathbb{P} \text{ SimpleDiagram}$

$\forall sd : \text{SimpleDiagram} \bullet$
 $\text{ValidSimpleDiagram}(sd) \Leftrightarrow$
 $(\forall do, do_1 : \text{ran } sd.\text{diagramobjects} \bullet$
 $do \neq do_1 \wedge$
 $do.type = do_1.type = \text{IsaSymbol} \vee$
 $do.type = do_1.type = \text{IsaTextBlock} \Rightarrow$
 $\neg do \text{ DOOverlapsDO } do_1 \wedge$
 $do.type = \text{IsaConnection} \Rightarrow$
 $((\exists_1 s_1, s_2 : \text{ran } sd.\text{diagramobjects} \bullet$
 $do \text{ CConnectsSS } (s_1, s_2)) \wedge$
 $(\forall s : \text{ran } sd.\text{diagramobjects} \bullet$
 $s.type = \text{IsaSymbol} \wedge$
 $\neg do \text{ DOOverlapsDO } s)))$

8.3 Diagram Object Operations

There are a number of operations that can be performed on diagram objects. These include enlarging, shrinking or moving a symbol, moving text blocks, or changing end points of a connection. At this stage of specification, mouse actions are not included in the specifications. A number of other operations such as adding text to a symbol or connection label are not specified for brevity. Scaling of text blocks, and scaling and moving connections are not defined operations for this simple tool, and so are not specified.

Operations on a diagram object cannot change its identifier, type or coordinate frame:

OpOnDiagramObject _____
 $\Delta \text{DiagramObject}$

$doid' = doid$
 $type' = type$
 $cfid' = cfid$

Move Handles and Scale Handles The moving and scaling of symbols and moving of text blocks are performed through the manipulation of the move handles and scale handles of the diagram objects. These handles perform a similar function in this context as in the window context discussed in Chapter 6. Scale handles for a

symbol appear as squares in each of the four corners of the bounding region, and at the midway point of each of the four sides of the region. The dimension of these square handles is given by the following global identifier:

$$\mid \quad scalehandlesize : \mathbb{N}_1$$

The work pixels in a scale handle for a symbol can be computed by the following axiom:

$SymbolScaleHandlePixels : Symbol \times direction \leftrightarrow \mathbb{P} WorkPixel$

$\forall s : Symbol; dir : direction; pwp : \mathbb{P} WorkPixel \bullet$
 $SymbolScaleHandlePixels(s, dir) = pwp \Rightarrow$
 $(\exists handlewr : WorkRegion; handlexmid, handleymid,$
 $symxmid, symymid : \mathbb{N}_1 \bullet$
 $pwp = AllWRPoints(handlewr) \wedge$
 $WPlsSquare(pwp) \wedge$
 $\#pwp = Sqr(scalehandlesize) \wedge$
 $symxmid = ((SymbolWorkRegion(s)).botright.x -$
 $(SymbolWorkRegion(s)).topleft.x + 1) \text{ div } 2 \wedge$
 $symymid = ((SymbolWorkRegion(s)).botright.y -$
 $(SymbolWorkRegion(s)).topleft.y + 1) \text{ div } 2 \wedge$
 $handlexmid =$
 $(handlewr.botright.x - handlewr.topleft.x + 1) \text{ div } 2 \wedge$
 $handleymid =$
 $(handlewr.botright.y - handlewr.topleft.y + 1) \text{ div } 2 \wedge$
 $dir = Top \Rightarrow$
 $(handlexmid = symxmid \wedge$
 $handleymid = handlewr.topleft.y) \wedge$
 $dir = Bottom \Rightarrow$
 $(handlexmid = symxmid \wedge$
 $handleymid = handlewr.botright.y) \wedge$
 $dir = Left \Rightarrow$
 $(handleymid = symymid \wedge$
 $handlexmid = handlewr.topleft.x) \wedge$
 $dir = Right \Rightarrow$
 $(handleymid = symymid \wedge$
 $handlexmid = handlewr.botright.x) \wedge$
 $dir = TopLeft \Rightarrow$
 $(handlexmid = handlewr.topleft.x \wedge$
 $handleymid = handlewr.topleft.y) \wedge$
 $dir = TopRight \Rightarrow$
 $(handlexmid = handlewr.botright.x \wedge$
 $handleymid = handlewr.topleft.y) \wedge$
 $dir = BottomLeft \Rightarrow$
 $(handlexmid = handlewr.topleft.x \wedge$
 $handleymid = handlewr.botright.y) \wedge$
 $dir = BottomRight \Rightarrow$
 $(handlexmid = handlewr.botright.x \wedge$
 $handleymid = handlewr.botright.y))$

The move handle for a symbol is composed of all pixels in the bounding region except those in the scale handles:

$$\begin{array}{l}
\hline
\textit{SymbolMoveHandlePixels} : \textit{Symbol} \rightarrow \mathbb{P} \textit{WorkPixel} \\
\hline
\forall s : \textit{Symbol} \bullet \\
\quad \textit{SymbolMoveHandlePixels}(s) = \\
\quad \quad \textit{AllWRPoints}(\textit{SymbolWorkRegion}(s)) \setminus \\
\quad \quad \cup (\textit{SymbolScaleHandlePixels}(\{\textit{dir} : \textit{direction} \bullet (s, \textit{dir})\}))
\end{array}$$

The relational image operation in Z is used in the specification above to compute the scale handle pixels for all scale handles of the given symbol. The set given by $\{\textit{dir} : \textit{direction} \bullet (s, \textit{dir})\}$ denotes a set of eight ordered pairs, where the first element in each pair is the symbol under consideration, and the second element is a direction.

Since a text block has no scale handles, its move handle is composed of the set of pixels inside its bounding region:

$$\begin{array}{l}
\hline
\textit{TextBlockMoveHandlePixels} : \textit{TextBlock} \rightarrow \mathbb{P} \textit{WorkPixel} \\
\hline
\forall tb : \textit{TextBlock} \bullet \\
\quad \textit{TextBlockMoveHandlePixels}(tb) = \\
\quad \quad \textit{AllWRPoints}(\textit{TextBlockWorkRegion}(tb))
\end{array}$$

Text Block Operations Moving a text block is accomplished by moving its bounding *WorkRegion*:

$$\begin{array}{l}
\hline
\textit{MoveTextBlock} \\
\textit{OpOnDiagramObject} \\
\textit{wpixel?} : \textit{WorkPixel} \\
\textit{direction?} : \textit{direction} \\
\hline
\textit{type} = \textit{IsaTextBlock} \\
\textit{DOWorkRegion}(\theta \textit{DiagramObject}') = (\mu \textit{MoveWR} \mid \\
\quad \theta \textit{WorkRegion} = \textit{DOWorkRegion}(\theta \textit{DiagramObject}) \bullet \theta \textit{WorkRegion}') \\
\textit{lines}' = \textit{lines} \\
\hline
\end{array}$$

Symbol Operations To enlarge, shrink or move a symbol, the work region that contains the symbol is scaled or moved:

EnlargeSymbol

OpOnDiagramObject
wpixel? : *WorkPixel*
direction? : *direction*

type = *IsaSymbol*
 $DOWorkRegion(\theta DiagramObject') = (\mu EnlargeWR \mid$
 $\theta WorkRegion = DOWorkRegion(\theta DiagramObject) \bullet \theta WorkRegion')$
text' = *text*

ShrinkSymbol

OpOnDiagramObject
wpixel? : *WorkPixel*

type = *IsaSymbol*
 $DOWorkRegion(\theta DiagramObject') = (\mu ShrinkWR \mid$
 $\theta WorkRegion = DOWorkRegion(\theta DiagramObject) \bullet \theta WorkRegion')$
text' = *text*

MoveSymbol

OpOnDiagramObject
wpixel? : *WorkPixel*

type = *IsaSymbol*
 $DOWorkRegion(\theta DiagramObject') = (\mu MoveWR \mid$
 $\theta WorkRegion = DOWorkRegion(\theta DiagramObject) \bullet \theta WorkRegion')$
text' = *text*

Connection Operations Either end point of a line segment in a connection can be changed, as asserted by the following:

ChangeConnectionPoint

OpOnDiagramObject
wpixel? : *WorkPixel*

type = *IsaConnection*
 $\exists i : \mathbb{N} \bullet$
 $points(i) \neq wpixel? \wedge$
 $points' = points \oplus \{i \mapsto wpixel?\}$
shape' = *shape*
text' = *text*

Changing either of a connection's end points is a special case of the *ChangeConnectionPoint* operation:

<i>ChangeConnectionFromEnd</i>	_____
<i>OpOnDiagramObject</i>	
<i>wpixel? : WorkPixel</i>	
<i>ChangeConnectionPoint</i>	
<i>head(points') = wpixel?</i>	

<i>ChangeConnectionToEnd</i>	_____
<i>OpOnDiagramObject</i>	
<i>wpixel? : WorkPixel</i>	
<i>ChangeConnectionPoint</i>	
<i>last(points') = wpixel?</i>	

8.4 Diagram Operations

A number of operations on a simple diagram are specified in this section, including:

- Adding a text block to a diagram, removing a text block from a diagram, moving a text block, and moving multiple text blocks.
- Adding a symbol to a diagram, removing a symbol from a diagram, enlarging, shrinking and moving a symbol, and moving multiple symbols.
- Connecting and disconnecting two symbols.

Operations on a simple diagram preserve its coordinate frame:

<i>OpOnSimpleDiagram</i>	_____
$\Delta SimpleDiagram$	
<i>cfid' = cfid</i>	

Text Block Operations A text block may be added to or removed from a simple diagram by simply manipulating the *diagramobjects* property of the simple diagram:

<i>AddTextBlock</i>	_____
<i>OpOnSimpleDiagram</i>	
<i>newtextblock? : DiagramObject</i>	
<i>newtextblock?.type = IsaTextBlock</i>	
<i>newtextblock?.doid $\notin$ dom diagramobjects</i>	
<i>diagramobjects' = diagramobjects $\oplus$ {newtextblock?.doid $\mapsto$ newtextblock?}</i>	

RemoveTextBlock

OpOnSimpleDiagram

textblockid? : *ID*

$(\text{diagramobjects}(\text{textblockid?})) . \text{type} = \text{IsaTextBlock}$

$\text{dom diagramobjects}' = \text{dom diagramobjects} \setminus \{\text{textblockid?}\}$

Moving one or more text blocks within a simple diagram is a straightforward promotion of the *MoveTextBlock* operation:

DMoveTextBlock

OpOnSimpleDiagram

textblockid? : *ID*

wpixel? : *WorkPixel*

direction? : *direction*

$\text{diagramobjects}' = \text{diagramobjects} \oplus \{\text{textblockid?} \mapsto (\mu \text{ MoveTextBlock} \mid \theta \text{ DiagramObject} = \text{diagramobjects}(\text{textblockid?}) \bullet \theta \text{ DiagramObject}')\}$

To move multiple text blocks, the identifiers for the text blocks, the direction in which to move them, and the work pixel to use for the move operations are provided as input to the operation. Each text block is then moved using the *MoveTextBlock* operation.

DMoveTextBlocks

OpOnSimpleDiagram

wpixel? : *WorkPixel*

textblockids? : $\mathbb{P} \text{ ID}$

direction? : *direction*

$\text{textblockids?} \subseteq \text{dom diagramobjects}$

$\forall id : \text{dom diagramobjects} \bullet$

$id \in \text{textblockids?} \Rightarrow$

$\text{diagramobjects}'(id) = (\mu \text{ MoveTextBlock} \mid$

$\theta \text{ DiagramObject} = \text{diagramobjects}(id) \bullet \theta \text{ DiagramObject}') \wedge$

$id \notin \text{textblockids?} \Rightarrow$

$\text{diagramobjects}'(id) = \text{diagramobjects}(id)$

$\text{dom diagramobjects}' = \text{dom diagramobjects}$

Symbol Operations Adding a new symbol to the picture is similar to adding a new text block:

AddSymbol

OpOnSimpleDiagram

newsymbol? : DiagramObject

newsymbol?.type = IsaSymbol

newsymbol?.doid $\notin$ dom diagramobjects

diagramobjects' = diagramobjects $\oplus$ {newsymbol?.doid $\mapsto$ newsymbol?}

A symbol can be removed only if it already exists in the picture and there are no connections to it:

RemoveSymbol

OpOnSimpleDiagram

symbolid? : ID

(diagramobjects(symbolid?)).type = IsaSymbol

$\neg (\exists c : \text{ran diagramobjects} \bullet$

$c \text{ AttachedToS diagramobjects(symbolid?)}$

dom diagramobjects' = dom diagramobjects $\setminus$ {symbolid?}

When a symbol is scaled, the connections to the symbol must change their end points to remain attached to the symbol:

DEnlargeSymbol

OpOnSimpleDiagram

symbolid? : *ID*

wpixel? : *WorkPixel*

direction? : *direction*

```
let enlargedsymbol == (μ EnlargeSymbol |
  θ DiagramObject = diagramobjects(symbolid?) • θDiagramObject') •
  (∃ connections : ID → DiagramObject •
    connections ⊂ diagramobjects ∧
    (∀ cid : dom connections •
      diagramobjects(cid) CAttachedToS diagramobjects(symbolid?) ∧
      connections(cid) CAttachedToS enlargedsymbol ∧
      (∃ wpixel : WorkPixel •
        wpixel ∈ DOIimagePoints(enlargedsymbol) ∧
        (connections(cid) = (μ ChangeConnectionToEnd |
          θ DiagramObject = diagramobjects(cid) ∧
          wpixel? = wpixel • θDiagramObject') ∨
          connections(cid) = (μ ChangeConnectionFromEnd |
            θ DiagramObject = diagramobjects(cid) ∧
            wpixel? = wpixel • θDiagramObject')))) ∧
      diagramobjects' = diagramobjects ⊕
        {symbolid? ↦ enlargedsymbol} ⊕ connections)
```

In *DEnlargeSymbol*, the symbol to enlarge is indicated by the *symbolid?* input. The *enlargedsymbol* variable contains this symbol's final state after enlarging has taken place through the *EnlargeSymbol* operation. The connections that were attached to the symbol in the initial state are transformed through the application of *ChangeConnectionToEnd*, *ChangeConnectionFromEnd*, or both. These connections are indicated by the function *connections*. Finally, the enlarged symbol and the modified connections must be updated in the set of diagram objects in the final state of the diagram.

The shrink and move operations of a symbol can be expressed in a similar manner:

DShrinkSymbol

OpOnSimpleDiagram

symbolid? : *ID*

wpixel? : *WorkPixel*

direction? : *direction*

let *shrunksymbol* == (μ *ShrinkSymbol* |
 θ *DiagramObject* = *diagramobjects*(*symbolid?*) • θ *DiagramObject'*) •
 ($\exists$ *cf* : *ID* $\leftrightarrow$ *DiagramObject* •
 dom *cf* $\subset$ dom *diagramobjects* $\wedge$
 ($\forall$ *cid* : dom *cf* •
 diagramobjects(*cid*) CAttachedToS *diagramobjects*(*symbolid?*) $\wedge$
 cf(*cid*) CAttachedToS *shrunksymbol* $\wedge$
 ($\exists$ *wpixel* : *WorkPixel* •
 wpixel $\in$ *DOImagePoints*(*shrunksymbol*) $\wedge$
 (*cf*(*cid*) = (μ *ChangeConnectionToEnd* |
 θ *DiagramObject* = *diagramobjects*(*cid*) $\wedge$
 wpixel? = *wpixel* • θ *DiagramObject'*) $\vee$
 cf(*cid*) = (μ *ChangeConnectionFromEnd* |
 θ *DiagramObject* = *diagramobjects*(*cid*) $\wedge$
 wpixel? = *wpixel* • θ *DiagramObject'*)))) $\wedge$
 diagramobjects' = *diagramobjects* $\oplus$
 {*symbolid?* $\mapsto$ *shrunksymbol*} $\oplus$ *cf*)

DMoveSymbol
OpOnSimpleDiagram
symbolid? : *ID*
wpixel? : *WorkPixel*
direction? : *direction*

let *movedsymbol* == (μ *MoveSymbol* |
 θ *DiagramObject* = *diagramobjects*(*symbolid?*) • θ *DiagramObject'*) •
 $(\exists cf : ID \mapsto DiagramObject \bullet$
 $\text{dom } cf \subset \text{dom } diagramobjects \wedge$
 $(\forall cid : \text{dom } cf \bullet$
 $diagramobjects(cid) \text{ CAttachedToS } diagramobjects(symbolid?) \wedge$
 $cf(cid) \text{ CAttachedToS } movedsymbol \wedge$
 $(\exists wpixel : WorkPixel \bullet$
 $wpixel \in DOImagePoints(movedsymbol) \wedge$
 $(cf(cid) = (\mu \text{ ChangeConnectionToEnd } |$
 $\theta \text{ DiagramObject} = diagramobjects(cid) \wedge$
 $wpixel? = wpixel \bullet \theta \text{ DiagramObject'}) \vee$
 $cf(cid) = (\mu \text{ ChangeConnectionFromEnd } |$
 $\theta \text{ DiagramObject} = diagramobjects(cid) \wedge$
 $wpixel? = wpixel \bullet \theta \text{ DiagramObject'}))))) \wedge$
 $diagramobjects' = diagramobjects \oplus$
 $\{symbolid? \mapsto movedsymbol\} \oplus cf)$

As with text blocks, many symbols can be moved at one time through the repeated application of the *MoveSymbol* operation:

DMoveSymbols
OpOnSimpleDiagram
wpixel? : *WorkPixel*
symbolids? : $\mathbb{P} ID$
direction? : *direction*

$symbolids? \subseteq \text{dom } diagramobjects$
 $\forall id : \text{dom } diagramobjects \bullet$
 $id \in symbolids? \Rightarrow$
 $diagramobjects'(id) = (\mu \text{ MoveSymbol } |$
 $\theta \text{ DiagramObject} = diagramobjects(id) \bullet \theta \text{ DiagramObject'}) \wedge$
 $id \notin symbolids? \Rightarrow$
 $diagramobjects'(id) = diagramobjects(id)$
 $\text{dom } diagramobjects' = \text{dom } diagramobjects$

Connection Operations Two symbols can be connected by creating a new connection between them:

<i>AddConnection</i> <i>OpOnSimpleDiagram</i> <i>symboltoid?</i> : <i>ID</i> <i>symbolfromid?</i> : <i>ID</i> <i>newconnection?</i> : <i>DiagramObject</i>
$\{symboltoid?, symbolfromid?\} \subseteq \text{dom } diagramobjects$ $newconnection? \notin \text{ran } diagramobjects$ $newconnection? \text{CConnectsSS}$ $\quad (diagramobjects(symboltoid?), diagramobjects(symbolfromid?))$ $diagramobjects' = diagramobjects \oplus$ $\quad \{newconnection?.doid \mapsto newconnection?\}$

A connection may be removed by removing it from the set of diagram objects:

<i>RemoveConnection</i> <i>OpOnSimpleDiagram</i> <i>connectionid?</i> : <i>ID</i>
$(diagramobjects(connectionid?)).type = IsaConnection$ $\text{dom } diagramobjects' = \text{dom } diagramobjects \setminus \{connectionid?\}$

The end point for any line segment within a connection can be changed as follows:

<i>DChangeConnection</i> <i>OpOnSimpleDiagram</i> <i>connectionid?</i> : <i>ID</i> <i>wpixel?</i> : <i>WorkPixel</i>
$diagramobjects' = diagramobjects \oplus$ $\quad \{connectionid? \mapsto (\mu \text{ ChangeConnectionPoint } $ $\quad \theta \text{ DiagramObject} = diagramobjects(connectionid?) \bullet \theta \text{ DiagramObject}')\}$

8.4.1 A Simple Diagramming Tool

Diagram objects have been defined in terms of work pixels because they are mapped to the display through the use of a scroll region control. It is through this control that operations on a diagram must occur in a simple diagramming tool. Given a scroll region control, a diagram is associated with the control if the coordinate frame identifier for the diagram is equal to the coordinate frame identifier for the work area associated with the control. This can be specified as follows:

$$\text{ScrollRegionPicture} : \text{WindowControl} \leftrightarrow \text{SimpleDiagram}$$

$$\forall wc : \text{WindowControl}; sp : \text{SimpleDiagram} \bullet$$

$$\begin{aligned} \text{ScrollRegionPicture}(wc) = sp \Rightarrow \\ (wc.type = \text{IsaScrollRegion} \wedge \\ wc.viewportWR.cfid = sp.cfid) \end{aligned}$$

Since a simple diagram is mapped to the work area component of a scroll region control, the functions defined on the control can be used to navigate through the diagram and zoom the image.

8.4.2 Operations of a Diagramming Tool

Operations performed on diagram objects within a diagramming tool must be defined in terms of operations on the object's image contained within the diagram's scroll region control. Hence, they are really scroll region control operations. A mouse must be used to manipulate a diagram object through this control.

To indicate to the tool that an object is to be included in the next operation, the user must click a mouse button while the cursor rests within the bounding region of the object. Once selected, the user can then manipulate the object with the mouse or invoke some action. The mouse buttons used for these purposes are indicated by global identifiers:

$$\text{ButtonForObjectSelection} : \text{ButtonPosition}$$

$$\text{ButtonForObjectManipulation} : \text{ButtonPosition}$$

The bounding region of a diagram object is mapped to virtual screen pixels using the *Zoom* function defined on the scroll region control in which the diagram object's work area resides. Clicking on a diagram object does not change the state of associated scroll region control:

<i>ClickOnDiagramObject</i> _____ <i>OpOnWindowControl</i> <i>doid? : ID</i>
<i>type = IsaScrollRegion</i> let <i>po == (ScrollRegionPicture(θ WindowControl)).diagramobjects(doid?) • <i>MouseClickedObject(Zoom(DOWorkRegion(po), zoomfactor),</i> <i>ButtonForObjectSelection, mousenow)</i> <i>viewportVSR' = viewportVSR</i> <i>viewportWR' = viewportWR</i> <i>zoomfactor' = zoomfactor</i> </i>

Several diagram objects can be selected at one time by clicking on them sequentially:

<i>ClickOnDiagramObjects</i> _____ <i>OpOnWindowControl</i> <i>doids? : $\mathbb{P}$ ID</i>
<i>type = IsaScrollRegion</i> $\forall$ <i>doid : doids? •</i> <i>ClickOnDiagramObject[doid/doid?]</i>

The diagramming tool has several global variables affecting its functionality:

<i>defaultsize : OBJECTSHAPE $\rightarrow \mathbb{N}_1$</i> <i>snaptogrid : Boolean</i> <i>gridsize : $\mathbb{N}_1$</i>
--

The *defaultsize* function ensures that new symbols added to the diagram have a default size. For example, a diagram may have a symbol that is square. When a square symbol is added to the diagram, it must have some initial size. In a diagramming tool, the default size of a symbol could likely be modified by the user. Given a shape, the *defaultsize* function returns a positive natural number that represents the cardinality of the symbol's image point set.

The *snaptogrid* global variable is a boolean variable that controls the positioning of new symbols, and scaling and movement of existing symbols and text blocks by making them “snap” to a location that is evenly divisible by some number, represented by *gridsize*. For example, in the diagramming tool there may be a requirement that symbols must be located at each horizontal and vertical centimetre. In this case, when a symbol is moved or scaled, valid locations will only be defined for every one

centimeter along each axis. While not critical to a diagramming tool, this feature is common in many drawing tools. A relation follows for checking the positioning of the picture objects based on these settings:

PositionValid_ : $\mathbb{P}$ <i>DiagramObject</i>
$\forall do : \text{DiagramObject} \bullet \text{PositionValid}(do) \Leftrightarrow$ $\text{snaptogrid} = \text{Yes} \Rightarrow$ $((\text{DOWorkRegion}(do)).\text{wtopleft}.x \bmod \text{gridsize} = 0 \wedge$ $(\text{DOWorkRegion}(do)).\text{wtopleft}.y \bmod \text{gridsize} = 0)$

Text Block Operations Next, a specification for adding a new text block to the diagram is given. There are four steps involved in this specification: i) clicking on a location in the scroll region control's work area defines the position of the new text block; in this specification, a one-pixel region is defined, located where the user has clicked, whose address represents the top left corner of the new text block's virtual screen region; ii) adding the text block to the simple diagram using the *AddTextBlock* schema operation; and iii) ensuring that the position of the text block agrees with the *snaptogrid* and *gridsize* setting. This operation can be specified as follows:

<i>MouseAddTextBlock</i> <i>OpOnWindowControl</i> <i>newtextblock?</i> : <i>DiagramObject</i>
$\text{type} = \text{IsaScrollRegion}$ $\exists \text{vsr} : \text{VScreenRegion} \bullet$ $\text{viewportVSR VSRContains vsr} \wedge$ $\#(\text{AllVSRPoints}(\text{vsr})) = 1 \wedge$ $\text{MouseClicked}(\text{vsr}, \text{ButtonForObjectManipulation}, \text{mousenow}) \wedge$ $(\text{Zoom}(\text{DOWorkRegion}(\text{newtextblock?}), \text{zoomfactor})).\text{rtopleft} =$ $\text{vsr}.\text{rtopleft}$ $\text{ScrollRegionPicture}(\theta \text{WindowControl}') = (\mu \text{AddTextBlock} \mid$ $\theta \text{SimpleDiagram} = \text{ScrollRegionPicture}(\theta \text{WindowControl}) \bullet$ $\theta \text{SimpleDiagram}')$ $\text{PositionValid}(\text{newtextblock?})$

Removing a text block from a diagram is considerably easier to specify than adding a new text block. The text block must be first selected by the user with the mouse, and then removed from the diagram using the *RemoveTextBlock* operation defined previously. This operation is specified in two phases; the first details the removal of a

text block from the diagram, and the second ensures that the text block has actually been selected before the operation is performed:

<i>MouseRemoveTextBlock0</i> <i>OpOnWindowControl</i> <i>doid? : ID</i>
<i>type = IsaScrollRegion</i> <i>ScrollRegionPicture</i> (θ <i>WindowControl'</i>) = (μ <i>RemoveTextBlock</i> θ <i>SimpleDiagram</i> = <i>ScrollRegionPicture</i> (θ <i>WindowControl</i>) $\wedge$ <i>textblockid?</i> = <i>doid?</i> • θ <i>SimpleDiagram'</i>)

MouseRemoveTextBlock $\hat{=}$ *ClickOnDiagramObject* ; *MouseRemoveTextBlock0*

A text block is moved in a simple diagram by selecting a pixel in the set of pixels corresponding to the text block's *movehandle* with the mouse, and dragging the text block to a new position. The *MouseDown_Drop* function, and the *DMoveTextBlock* operation are used for this purpose. The location to which a text block is moved is also constrained by the *snaptogrid* and *gridsize* settings:

<i>MouseMoveTextBlock0</i> <i>OpOnWindowControl</i> <i>doid? : ID</i> <i>wpixel? : WorkPixel</i> <i>direction? : direction</i>
<i>type = IsaScrollRegion</i> let <i>dobefore</i> == (<i>ScrollRegionPicture</i> (θ <i>WindowControl</i>)). <i>diagramobjects</i> (<i>doid?</i>); <i>doafter</i> == (<i>ScrollRegionPicture</i> (θ <i>WindowControl'</i>)). <i>diagramobjects</i> (<i>doid?</i>) • ($\exists$ <i>wr</i> , <i>wr</i> ₁ : <i>WorkRegion</i> ; <i>wp</i> , <i>wp</i> ₁ : <i>WorkPixel</i> • { <i>wp</i> } = <i>AllWRPoints</i> (<i>wr</i>) $\wedge$ { <i>wp</i> ₁ } = <i>AllWRPoints</i> (<i>wr</i> ₁) $\wedge$ <i>wp</i> $\in$ <i>TextBlockMoveHandlePixels</i> (<i>dobefore</i> . θ <i>TextBlock</i>) $\wedge$ <i>wp</i> ₁ $\in$ <i>TextBlockMoveHandlePixels</i> (<i>doafter</i> . θ <i>TextBlock</i>) $\wedge$ (<i>Zoom</i> (<i>wr</i> ₁ , <i>zoomfactor</i>), <i>mousenow</i>) = <i>MouseDown_Drop</i> (<i>Zoom</i> (<i>wr</i> , <i>zoomfactor</i>), <i>ButtonForObjectManipulation</i>) $\wedge$ <i>ScrollRegionPicture</i> (θ <i>WindowControl'</i>) = (μ <i>DMoveTextBlock</i> θ <i>SimpleDiagram</i> = <i>ScrollRegionPicture</i> (θ <i>WindowControl</i>) $\wedge$ <i>textblockid?</i> = <i>doid?</i> • θ <i>SimpleDiagram'</i>) $\wedge$ <i>PositionValid</i> (<i>doafter</i>))

MouseMoveTextBlock $\hat{=}$ *ClickOnDiagramObject* ; *MouseMoveTextBlock0*

Symbol Operations Adding a symbol to a diagram is almost identical to adding a new text block, except that the size of the new symbol must be the appropriate default size:

<i>MouseAddSymbol</i> <i>Op On WindowControl</i> <i>newsymbol? : DiagramObject</i> <i>type = IsaScrollRegion</i> $\exists \text{ vsr} : \text{VScreenRegion} \bullet$ $\text{viewportVSR VSRContains vsr} \wedge$ $\#(\text{AllVSRPoints}(\text{vsr})) = 1 \wedge$ $\text{MouseClickedObject}(\text{vsr}, \text{ButtonForObjectManipulation}, \text{mousenow}) \wedge$ $(\text{Zoom}(\text{DOWorkRegion}(\text{newsymbol?}), \text{zoomfactor})).\text{rtopleft} =$ vsr.rtopleft $\text{ScrollRegionPicture}(\theta \text{ WindowControl}') = (\mu \text{ AddSymbol} \mid$ $\theta \text{ SimpleDiagram} = \text{ScrollRegionPicture}(\theta \text{ WindowControl}) \bullet$ $\theta \text{ SimpleDiagram}')$ $\#(\text{DOImagePoints}(\text{newsymbol?})) = \text{defaultsize}(\text{newsymbol?.shape})$ $\text{PositionValid}(\text{newsymbol?})$

Specifications for removing and moving a symbol are almost identical to those for the corresponding text block operations:

<i>MouseRemoveSymbol0</i> <i>Op On WindowControl</i> <i>doid? : ID</i> <i>type = IsaScrollRegion</i> $\text{ScrollRegionPicture}(\theta \text{ WindowControl}') = (\mu \text{ RemoveSymbol} \mid$ $\theta \text{ SimpleDiagram} = \text{ScrollRegionPicture}(\theta \text{ WindowControl}) \wedge$ $\text{symbolid?} = \text{doid?} \bullet \theta \text{ SimpleDiagram}')$
--

$\text{MouseRemoveSymbol} \triangleq \text{ClickOnDiagramObject} \ ; \ \text{MouseRemoveSymbol0}$

```

MouseMoveSymbol0
OpOnWindowControl
doid? : ID
wpixel? : WorkPixel
direction? : direction

```

```

type = IsaScrollRegion
let dobefore ==
    (ScrollRegionPicture( $\theta$  WindowControl)).diagramobjects(doid?);
doafter ==
    (ScrollRegionPicture( $\theta$  WindowControl')).diagramobjects(doid?) •
    ( $\exists$  vsrbefore, vsrafter : VScreenRegion; wpixel : WorkPixel •
        vsrbefore = Zoom(DOWorkRegion(dobefore), zoomfactor)  $\wedge$ 
        vsrafter = Zoom(DOWorkRegion(doafter), zoomfactor)  $\wedge$ 
        (vsrafter, mousenow) =
            MouseDown_n_Drop(vsrbefore, ButtonForObjectManipulation)  $\wedge$ 
            ScrollRegionPicture( $\theta$  WindowControl') = ( $\mu$  DMoveSymbol |
                 $\theta$  SimpleDiagram = ScrollRegionPicture( $\theta$  WindowControl)  $\wedge$ 
                symbolid? = doid? •  $\theta$  SimpleDiagram')  $\wedge$ 
                PositionValid(doafter))

```

$\textit{MouseMoveSymbol} \cong \textit{ClickOnDiagramObject} \ ; \ \textit{MouseMoveSymbol0}$

Enlarging and shrinking a symbol is accomplished by selecting a pixel in the set of pixels corresponding to the symbol's *scalehandle* with the mouse, and dragging the side or corner of the symbol's bounding region to make it larger or smaller. In the specifications, the pixel being dragged is defined as being contained within a one-pixel virtual screen region which is dragged to a new location. The final location of this pixel within the handle is the location used by the *DEnlargeSymbol* or the *DShrinkSymbol* operation to enlarge or shrink the symbol:

MouseEnlargeSymbol0

OpOnWindowControl

doid? : *ID*

wpixel? : *WorkPixel*

direction? : *direction*

type = *IsaScrollRegion*

let *dobefore* ==

$(\text{ScrollRegionPicture}(\theta \text{WindowControl})).\text{diagramobjects}(\text{doid?});$

doafter ==

$(\text{ScrollRegionPicture}(\theta \text{WindowControl}')).\text{diagramobjects}(\text{doid?}) \bullet$

$\text{DOIImagePoints}(\text{dobefore}) \subset \text{DOIImagePoints}(\text{doafter}) \wedge$

$(\exists \text{wr}, \text{wr}_1 : \text{WorkRegion}; \text{wp}, \text{wp}_1 : \text{WorkPixel} \bullet$

$\{\text{wp}\} = \text{AllWRPoints}(\text{wr}) \wedge \{\text{wp}_1\} = \text{AllWRPoints}(\text{wr}_1) \wedge$

$\text{wp} \in \text{SymbolScaleHandlePixels}(\text{dobefore}.\theta \text{Symbol}, \text{direction?}) \wedge$

$\text{wp}_1 \in \text{SymbolScaleHandlePixels}(\text{doafter}.\theta \text{Symbol}, \text{direction?}) \wedge$

$(\text{Zoom}(\text{wr}_1, \text{zoomfactor}), \text{mousenow}) =$

$\text{MouseDown_n_Drop}(\text{Zoom}(\text{wr}, \text{zoomfactor}),$

$\text{ButtonForObjectManipulation}) \wedge$

$\text{ScrollRegionPicture}(\theta \text{WindowControl}') = (\mu \text{DEnlargeSymbol} \mid$

$\theta \text{SimpleDiagram} = \text{ScrollRegionPicture}(\theta \text{WindowControl}) \wedge$

$\text{symbolid?} = \text{doid?} \bullet \theta \text{SimpleDiagram}') \wedge$

$\text{PositionValid}(\text{doafter}))$

$\text{MouseEnlargeSymbol} \triangleq \text{ClickOnDiagramObject} \ ; \ \text{MouseEnlargeSymbol0}$

MouseShrinkSymbol0

OpOnWindowControl

doid? : *ID*

wpixel? : *WorkPixel*

direction? : *direction*

type = *IsaScrollRegion*

let *dobefore* ==

(ScrollRegionPicture(θ WindowControl)).diagramobjects(doid?);

doafter ==

(ScrollRegionPicture(θ WindowControl')).diagramobjects(doid?) •

DOIImagePoints(dobefore) $\subset$ DOIImagePoints(doafter) $\wedge$

*($\exists$ wr, wr_1 : *WorkRegion*; wp, wp_1 : *WorkPixel* •*

{wp} = AllWRPoints(wr) $\wedge$ {wp₁} = AllWRPoints(wr₁) $\wedge$

wp $\in$ SymbolScaleHandlePixels(dobefore. θ Symbol, direction?) $\wedge$

wp₁ $\in$ SymbolScaleHandlePixels(doafter. θ Symbol, direction?) $\wedge$

(Zoom(wr₁, zoomfactor), mousenow) =

MouseDownDrag_n_Drop(Zoom(wr, zoomfactor),

ButtonForObjectManipulation) $\wedge$

ScrollRegionPicture(θ WindowControl') = (μ DShrinkSymbol |

θ SimpleDiagram = ScrollRegionPicture(θ WindowControl) $\wedge$

symbolid? = doid? • θ SimpleDiagram')

PositionValid(doafter))

MouseShrinkSymbol $\equiv$ *ClickOnDiagramObject* ; *MouseShrinkSymbol0*

Connection Operations Establishing a connection between two symbols requires that the symbols must first be selected by the user:

MouseAddConnection

OpOnWindowControl

symboltoid? : *ID*

symbolfromid? : *ID*

newconnection? : *DiagramObject*

type = *IsaScrollRegion*

let *symbolids* == {*symboltoid?*, *symbolfromid?*} •

ClickOnDiagramObjects[symbolids/doids?]

ScrollRegionPicture(θ WindowControl') = (μ AddConnection |

θ SimpleDiagram = ScrollRegionPicture(θ WindowControl) •

θ SimpleDiagram')

Removing a connection is straightforward:

MouseRemoveConnection0

OpOnWindowControl

doid? : ID

type = IsaScrollRegion

ScrollRegionPicture(θ *WindowControl'*) = (μ *RemoveConnection* |
 θ *SimpleDiagram* = *ScrollRegionPicture*(θ *WindowControl*) $\wedge$
connectionid? = *doid?* • θ *SimpleDiagram'*)

MouseRemoveConnection $\hat{=}$ *ClickOnDiagramObject* ; *MouseRemoveConnection0*

Diagram Object Operations To make a diagram more presentable, the user of the tool might want to select a number of symbols or text blocks in the diagram and *align* them along a selected side. This would move all selected objects such that their respective bounding regions would align along the one side. This operation can be defined in the top, bottom, left and right directions. For simplicity, only the specifications for “align top” are presented here. In this operation, all selected objects are moved so that the top sides of their bounding regions have equal *y* coordinate values; equal to the *y* coordinate value of some selected object whose bounding region has the lowest *y* coordinate value. Since this operation can move both symbols and text blocks, for convenience one “move” operation is defined on picture objects using two operations specified earlier:

MovePictureObjects $\hat{=}$ *DMoveSymbols* ; *DMoveTextBlocks*

MouseAlignObjectsTop0

OpOnWindowControl

doids? : $\mathbb{P}$ ID

type = IsaScrollRegion

let *miny* == *min*($\{po : \text{DiagramObject} \mid po.doid \in doids? \bullet$
 $(DOWorkRegion(po)).wtopleft.y\}$);
symids == $\{po : \text{DiagramObject} \mid$
 $po.doid \in doids? \wedge po.type = \text{IsaSymbol} \bullet po.doid\}$;
tbids == $\{po : \text{DiagramObject} \mid$
 $po.doid \in doids? \wedge po.type = \text{IsaTextBlock} \bullet po.doid\} \bullet$
ScrollRegionPicture(θ *WindowControl'*) =
 $(\mu$ *MovePictureObjects* |
 θ *SimpleDiagram* = *ScrollRegionPicture*(θ *WindowControl*) $\wedge$
symbolids? = *symids* $\wedge$
textblockids? = *tbids* • θ *SimpleDiagram'*)

$$\textit{MouseAlignObjectsTop} \triangleq \textit{ClickOnDiagramObjects} \ ; \ \textit{MouseAlignObjectsTop0}$$

The operation *MouseAlignObjectsTop* aligns only symbols and their text block, and not connections to the symbols. The specification partitions the set of diagram object identifiers that are to participate in this operation, given by *doids?*, into two sets, *symids* and *tbids*. Each of these sets are then used as input into the *MovePictureObjects* operation.

8.5 Summary

In this chapter, specifications for simple diagram object types were provided. These objects were then grouped together to create a simple diagram and visual characteristics and constraints in object positioning were then outlined. A number of operations on diagram objects, and on a simple diagram followed. Specifications were given that linked together a simple diagram with a scroll region control, allowing manipulation and display of the simple diagram through a GUI window control. Finally, operations on this tool were specified which make use of the mouse and mouse operations. In the next chapter, these specifications will be expanded upon in a very simple way to create the full set of formal specifications for a specific diagramming tool - a tool used for object-oriented software design.

Chapter 9

A Specialized Diagramming Tool

A special purpose diagramming tool can be developed using the specifications for a simple diagramming tool provided in the last chapter and enriching the semantics for diagram objects using application-dependent information. To illustrate this concept, a simple tool for object-oriented design is specified in this chapter. The informal semantics of diagram objects are borrowed from [1].

Since the purpose of this chapter is to merely illustrate the enrichment of the specifications from the previous chapter towards a specialized application, the entire domain of object-orientation is not specified here. Consequently, this chapter includes only a few of the concepts described in [1].

To start with, specifications for a class are described. A symbol for a class in an object-oriented diagram has a specific shape (represented as a cloud in [1]):

| *ClassShape* : *OBJECTSHAPE*

A class has a number of attributes, two of which are represented visually: a class name, and a *cardinality* indicating the permitted number of instances of the class. The remaining attributes, which describe interface, implementation and visibility information, are not represented visually in this notation.

A class can have three types of cardinality:

ClassCardinality ::= *ZeroInstance* | *OneInstance* | *NInstances*

A class symbol is represented by a schema derived from *DiagramObject* with an additional attribute that represents cardinality. The shape of this symbol must be specific to class symbols and the label must contain at least two lines - one for the class name and another for an indication of the cardinality:

<i>ClassSymbol</i>
<i>DiagramObject</i>
<i>cardinality</i> : <i>ClassCardinality</i>
<i>type</i> = <i>IsaSymbol</i>
<i>shape</i> = <i>ClassShape</i>
<i>#text.lines</i> ≥ 2

$$DOFromClassSymbol == (\lambda \text{ ClassSymbol} \bullet \theta \text{ DiagramObject})$$

A connection indicating inheritance between classes has a specific shape and a non-empty label. Like class symbols, this connection is also derived from the *DiagramObject* schema type.

| *InheritsConnectionShape* : *OBJECTSHAPE*

<i>InheritsConnection</i>
<i>DiagramObject</i>
<i>type</i> = <i>IsaConnection</i>
<i>shape</i> = <i>InheritsConnectionShape</i>
<i>#text.lines</i> > 0

$$DOFromInheritsConnection == (\lambda \text{ InheritsConnection} \bullet \theta \text{ DiagramObject})$$

An object-oriented diagram representing a class hierarchy consists of class symbols, inheritance connections and text blocks for any documentation purposes. This specialized diagram type is derived from *SimpleDiagram*:

<i>OODiagram</i>
<i>SimpleDiagram</i>
<i>classsymbols</i> : $\mathbb{P} \text{ ClassSymbol}$
<i>inheritsconnections</i> : $\mathbb{P} \text{ InheritsConnection}$
<i>textblocks</i> : $\mathbb{P} \text{ DiagramObject}$
<i>ran diagramobjects</i> =
$DOFromClassSymbol(\text{classsymbols}) \cup$
$DOFromInheritsConnection(\text{inheritsconnections}) \cup$
textblocks
$\forall \text{ do} : \text{textblocks} \bullet$
$\text{do.type} = \text{IsaTextBlock}$

Based on the specific semantics for class symbols and inherits connections, one can produce a validation relationship that ensures the validity of the class hierarchy. First, a connection relationship is defined between the inherits connection and two distinct class symbols. This must be a directed connection:

$$\begin{array}{|l}
 \text{-- InhConnectsClasses -- : } \textit{InheritsConnection} \leftrightarrow (\textit{ClassSymbol} \times \textit{ClassSymbol}) \\
 \hline
 \forall ic : \textit{InheritsConnection}; cs_1, cs_2 : \textit{ClassSymbol} \bullet \\
 \quad ic \text{ InhConnectsClasses } (cs_1, cs_2) \Leftrightarrow \\
 \quad \quad (\textit{DOFromInheritsConnection}(ic) \text{ CConnectsFromSToS } \\
 \quad \quad \quad (\textit{DOFromClassSymbol}(cs_1), \textit{DOFromClassSymbol}(cs_2)) \wedge \\
 \quad \quad \quad cs_1 \neq cs_2)
 \end{array}$$

A valid class hierarchy diagram must adhere to the placement restrictions defined on a simple diagram as well as ensuring that the inherits connections are placed logically. For this simple example, multiple inherits connections cannot be defined between two class symbols. In addition, no cycles are permitted in the inheritance hierarchy.

$$\begin{array}{|l}
 \text{ValidOODDiagram -- : } \mathbb{P} \textit{OODDiagram} \\
 \hline
 \forall ood : \textit{OODDiagram} \bullet \\
 \quad \text{ValidOODDiagram}(ood) \Leftrightarrow \\
 \quad \quad (\text{ValidSimpleDiagram}(ood.\theta \textit{SimpleDiagram}) \wedge \\
 \quad \quad \neg (\exists_1 ic_1, ic_2 : ood.\textit{inheritsconnections}; \\
 \quad \quad \quad cs_1, cs_2 : ood.\textit{classsymbols} \bullet \\
 \quad \quad \quad \quad ic_1 \neq ic_2 \wedge cs_1 \neq cs_2 \wedge \\
 \quad \quad \quad \quad (ic_1 \text{ InhConnectsClasses } (cs_1, cs_2) \wedge \\
 \quad \quad \quad \quad \quad ic_2 \text{ InhConnectsClasses } (cs_1, cs_2)) \\
 \quad \quad \quad \quad \vee \\
 \quad \quad \quad \quad (ic_1 \text{ InhConnectsClasses } (cs_2, cs_1) \wedge \\
 \quad \quad \quad \quad \quad ic_2 \text{ InhConnectsClasses } (cs_2, cs_1)) \\
 \quad \quad \quad \quad \vee \\
 \quad \quad \quad \quad (ic_1 \text{ InhConnectsClasses } (cs_1, cs_2) \wedge \\
 \quad \quad \quad \quad \quad ic_2 \text{ InhConnectsClasses } (cs_2, cs_1)) \\
 \quad \quad \quad \quad \vee \\
 \quad \quad \quad \quad (ic_1 \text{ InhConnectsClasses } (cs_2, cs_1) \wedge \\
 \quad \quad \quad \quad \quad ic_2 \text{ InhConnectsClasses } (cs_1, cs_2))) \wedge \\
 \quad \quad \neg (\exists \textit{classseq} : \textit{seq ClassSymbol} \bullet \\
 \quad \quad \quad \text{ran classseq} \subseteq ood.\textit{classsymbols} \wedge \\
 \quad \quad \quad \#(\text{tail}(\textit{classseq})) = \# \textit{classseq} - 1 \wedge \\
 \quad \quad \quad \text{head}(\textit{classseq}) = \text{last}(\textit{classseq}) \wedge \\
 \quad \quad \quad (\forall i : 1 \dots \# \textit{classseq} \bullet \\
 \quad \quad \quad \quad (\exists ic : ood.\textit{inheritsconnections} \bullet \\
 \quad \quad \quad \quad \quad ic \text{ InhConnectsClasses } (\textit{classseq}(i + 1), \textit{classseq}(i))))))
 \end{array}$$

Chapter 10

Refinement

In this chapter, the refinement of specifications towards implementation is discussed. The specifications described in previous chapters represent an abstract model of a system whose implementation as a genuine product typically requires several levels of refinement. However, the theme of this thesis is to investigate the behaviour of a GUI and not its final implementation. As a result, this chapter discusses the refinement of the GUI specifications towards a prototype which illustrates that the specifications can be transformed into a final product.

10.1 Design and Implementation From Formal Specifications

Formal specifications are abstract and are not biased towards any particular design and/or implementation. Even though model-based specifications do describe an abstract model of a system, such a model does not include any design or implementation details. However, it might provide hints to the designer about the appropriate design or implementation model. This is one of the advantages of model-based specifications over axiomatic or algebraic specifications.

Refinement of a model-based specification may be carried out through several levels. Each level of refinement adds new design details into the specification such as data structures, algorithms and environmental considerations. Lower levels of refinement add details concerning implementation such as specific syntactic structures in the chosen programming language, constraints regarding expected efficiency and anticipated expectations. The final level of refinement will generally have a close

match with the chosen programming language, enabling an easy and partially verified implementation.

There are two well known techniques of refinement for model-based specifications - data refinement and operational decomposition. These two techniques are not complements of each other; rather, both are used at various stages of refinement. In data refinement, the abstract data types in the higher level specifications are transformed into more concrete data types in the lower level specifications. For example, a “set” data type can be transformed into a list or array at the lower level. During operational decomposition, an operation or a function in the higher level specifications is decomposed into one or more operations or functions in the lower level. Both types of refinement techniques require proof obligations to ensure that the lower level specifications are consistent with the higher level specifications. More about these two techniques can be found in [10, 14, 16].

10.2 Simple Prototype of a GUI

Since complete refinement and proof obligations of the GUI specifications are beyond the scope of this thesis, only a simple prototype of the GUI has been implemented. Typically, many window controls are implemented as part of a GUI subsystem, and so most development time will be concentrated on building the simple or specialized diagramming tool from these components. The prototype was developed on a PC running Microsoft Windows using Borland C++ with Application Frameworks. Borland C++ provides rich class libraries for developing Windows-based applications, and so was selected to quickly implement the prototype. The development time for the prototype was much shorter than the time taken for the formal design of the GUI, since the specifications for the GUI address many low level details such as pixels and mappings.

Though many of the abstract data types (schema structures) of the GUI specification have direct mappings to the classes in the Borland libraries, their behaviours are not identical. The specifications were developed with the CASE tool as the target and with the concept of reuse as the partial benefit. Thus, the specifications address the expected behaviours of the window controls under the CASE tool cur-

rently being developed, and do not address an existing implementation such as that in the Borland libraries. Based on the author's experience, the specifications are easily implementable, and these implementations are consistent with the rest of the class library. Some of the structures such as combo boxes are not fully supported in the class libraries. There are no direct implementations for certain push button actions in the specifications. Rather, these were implemented using timers defined in Windows.

Chapter 11

Conclusion

A graphical user interface (GUI) plays a major role in any CASE tool, and hence must be easy to understand and manipulate. Design and implementation of such a GUI is therefore not a trivial task.

This thesis is a contribution to the development of an abstract GUI, targeted towards the primary interface of a CASE tool for object-oriented design. Though there are several GUI toolkits available, this GUI was developed from scratch in order to provide a foundation for the development of the CASE tool. Therefore, the GUI is expected to behave in a manner amenable towards its manipulation in the CASE tool. This necessitated the investigation of the behaviours of individual components in the GUI and its high level behaviour derived from those components. Thus, the focus of this thesis was to provide a behavioural study of a GUI for the CASE tool being targeted.

Formal specifications are used in this thesis to describe the behaviour of the GUI. Recently, formal specification techniques have been introduced into the software development process for behavioural study, design, derivation and verification, and for maintenance. In this thesis, the Z notation is used to develop an abstract model of the GUI. Z's modular approach enabled the author to specify the GUI's components in a fairly independent manner, and aggregate them for the final abstract model of the GUI. Moreover, the specifications for the components were written in such a way that they can be reused in other applications. Hence, they provide a toolkit of primitive GUI types and operations.

The specifications in this thesis address the structure and behaviour of pixels,

window controls, the mouse, the screen address space, the work area address space (where application dependent information is stored), the mapping between these two address spaces and finally the behaviour of all controls united along with the mouse. Both simple and specialized diagramming tools are specified in the later chapters. The thesis ended with a discussion about the development of a prototype of the GUI, derived from the specifications.

Continuation of work presented in this thesis can be carried out in several ways. For example, a detailed design can be derived from the specifications using refinement techniques that are outlined in [10]. This requires proof obligations on the part of the designer. Alternatively, some properties of the window controls can be stated and proved using the specifications in this thesis. As an example, the property "moving a window within the screen does not change the contents of the window" can be easily proved from the specifications. Such proofs are beyond the scope of this thesis and are left as continuing work.

Bibliography

- [1] Grady Booch, *Object-Oriented Design With Applications*, Benjamin-Cummings, 1991.
- [2] Jonathan Bowen, "Formal Specification of Window Systems", Technical Monograph PRG-74, Oxford University Computing Laboratory, Oxford, UK, 1989.
- [3] Peter Coad and Ed Yourdon, *Object-Oriented Analysis*, Prentice Hall/Yourdon Press, 1990.
- [4] D. DeChampeaux and P. Faure, "A Comparative Study of Object-Oriented Analysis Methods", in *Harnessing the Object Revolution: Workshop on Object-Oriented Software Engineering Practice*, Denver, CO, Feb 1992.
- [5] Tom DeMarco, *Structured Analysis and System Specification*, Yourdon Press, 1978.
- [6] Chris Gane and Trish Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, 1979.
- [7] J. Anthony Hall, "Using Z as a Specification Calculus for Object-Oriented Systems" in Bjørner D., Hoare C.A.R. and Langmaack, H. (Eds.) *VDM '90: VDM and Z - formal methods in software development*, Lecture Notes Computer Science 428, Springer-Verlag, 1990.
- [8] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985.
- [9] "CASE At the Start of the 1990's", *13th International Conference on Software Engineering*, Los Alamitos, CA, 1991.

- [10] Cliff Jones, *Systematic Software Development Using VDM* (Second Edition), Prentice Hall International Series in Computer Science, 1991.
- [11] B.P. Mahony and I.J. Hayes, "A Case-Study in Timed Refinement: A Mine Pump", *IEEE Transactions on Software Engineering*, Vol. 18, No. 9, Sept 1992, pp 817-826.
- [12] K.T. Narayana and S. Dharap, "Invariant Properties in a Dialog System", in *Proc. ACM Int. Workshop Formal Methods*, 1990.
- [13] Meilir Page-Jones, *The Practical Guide to Structured Systems Design*, Yourdon Press, 1980.
- [14] Ben Potter, Jane Sinclair and David Till, *An Introduction to Formal Specification and Z*, Prentice Hall International Series in Computer Science, 1991.
- [15] Ian Sommerville, *Software Engineering*, (Fourth Edition), Addison-Wesley Publishing Company, 1992.
- [16] Mike Spivey, *The Z Notation: A Reference Manual* (Second Edition), Prentice Hall International Series in Computer Science, 1992.
- [17] Susan Stepney, Rosalind Barden, David Cooper (Eds.), *Object Orientation in Z*, Springer-Verlag, 1992.
- [18] Ed Yourdon and Larry Constantine, *Structured Design*, Prentice-Hall, 1979.

Appendix A

Z Notations

This appendix includes only those notations of Z used in this thesis; readers are referred to [16] for a complete set of Z notations.

Definitions

$[]$	given set brackets for given types
$==$	abbreviation definition
$::=$	free type definition

Logic

$\neg$	logical negation
$\wedge$	logical conjunction
$\vee$	logical disjunction
$\Rightarrow$	logical implication
$\Leftrightarrow$	logical equivalence
$\forall d \bullet p$	universal quantification (d is declaration, p is predicate)
$\exists d \bullet p$	existential quantification (d is declaration, p is predicate)
$\exists_1 d \bullet p$	unique existential quantification (d is declaration, p is predicate)
$<, \leq, =, \geq, >$	logical operators

(note: $=$ is defined for all data types)

Sets

$\in$	set membership
$\notin$	set non-membership
$\emptyset, \{\}$	empty set
$\subseteq$	subset
$\subset$	proper subset
$\{\}$	set brackets
$()$	tuple brackets
$\times$	Cartesian product
$\mathbb{P}$	power set
$\cap$	set intersection
$\cup$	set union
$\setminus$	set difference
$\bigcap$	generalized intersection
$\bigcup$	generalized union
$\#$	cardinality

Relations

$\leftrightarrow$	binary relation
$\mapsto$	maplet
dom	domain of a relation
ran	range of a relation
$\circ$	forward relational composition
$\parallel$	relational image
$\sim$	relational inverse

Functions

$\mapsto$	partial function
$\rightarrow$	total function
$\twoheadrightarrow$	bijective function

$f(x), f\ x$	function application
$\oplus$	functional overriding

Numbers

$\mathbb{N}$	set of natural numbers
$\mathbb{N}_1$	set of non-zero natural numbers
$\mathbb{Z}$	set of integers
$+$	addition
$-$	subtraction
$*$	multiplication
div	division
$..$	number range

Sequences

seq	sequence as a type
$\langle \rangle$	sequence
$\wedge$	sequence concatenation
<i>head</i>	first element of a sequence
<i>tail</i>	all but the head element of a sequence
<i>last</i>	last element of a sequence
<i>front</i>	all but the last element of a sequence
<i>in</i>	contiguous subsequence
$\upharpoonright$	sequence extraction
$\upharpoonright$	sequence filtering

Schema Notation

S	
d	
p	

$$S \triangleq [d \mid p]$$

The former denotes vertical notation of a schema and the latter denotes its horizontal version.

.	component selection in a schema
θ	binding
'	after state decoration
$[a/b]$	renaming schema components (rename b to a)
$\wedge$	schema conjunction
$\vee$	schema disjunction
$\neg$	schema negation
$;$	schema composition
λ	lambda-expression
μ	mu-expression
$?$	decoration of input variable
$!$	decoration of output variable

Axiomatic Notation

$$\frac{d}{p}$$

Appendix B

Preliminary Functions and Data Types

It is assumed that *STRING* and *ID* are two basic types representing character strings and object identifiers respectively:

$$[STRING, ID]$$

$\mathbb{Z}$ does not have a boolean type. However, some of the specifications in this thesis use a boolean type which is defined below:

$$Boolean ::= Yes \mid No$$

A *toggling* function on a boolean variable is defined next:

$$\begin{array}{|l} Toggle : Boolean \rightarrow Boolean \\ \hline \forall b : Boolean \bullet \\ \quad b = Yes \Rightarrow Toggle(b) = No \wedge \\ \quad b = No \Rightarrow Toggle(b) = Yes \end{array}$$

The following functions specify two simple arithmetic operations used within the specifications. *Sqr* represents the square of a natural number and *Abs* is used to retrieve the absolute value of an integer:

$$\begin{array}{|l} Sqr : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall o : \mathbb{N} \bullet \\ \quad Sqr(o) = o * o \\ \\ Abs : \mathbb{Z} \rightarrow \mathbb{Z} \\ \hline \forall z : \mathbb{Z} \bullet \\ \quad z < 0 \Rightarrow Abs(z) = -z \wedge \\ \quad z \geq 0 \Rightarrow Abs(z) = z \end{array}$$