Mondrian: A Two-Dimensional Graphical Specification and Design Programming Environment

by

Kenneth Lahn Fern

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Science in Computer Science and Engineering

at the

Massachusetts Institute of Technology

May 1988

Signature of Author_____
Department of Electrical Engineering and Computer Science
6 May 1988

Certified by_____
William E. Weihl
Assistant Professor of Electrical Engineering
Thesis Supervisor

Certified by_____
Dr. Gerald Fisher, Jr.
IBM Research Division, T. J. Watson Research Division
Company Supervisor

Accepted by_____
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Mondrian: A Two-Dimensional Graphical Specification and Design Programming Environment

by

Kenneth L. Fern

Submitted to the Department of
Electrical Engineering and Computer Science
on May 6, 1988 in Partial Fulfillment of the Requirements
for the Degrees of

Bachelor of Science in Computer Science and Engineering

and

Master of Science in Computer Science and Engineering

ABSTRACT

Mondrian is a prototype environment for viewing and editing a specification and design language in a simple, geometric fashion using lines and boxes to create its pictures. Specifically, Mondrian addresses the issues of data modeling, formal specifications, and structure in a programming language. Other programming environments, such as Pecan, PegaSys, and Gandalf, do not address all of these issues, and none of these environments addresses the data modeling issue. Mondrian uses SEDL (Software Engineering Design Language) as its model for a specification and design language, while it uses RPDE (Refinement-based Product Development Environment) as its development environment.

Thesis Supervisor: Prof. William E. Weihl

Title: Assistant Professor of Electrical Engineering

Contents

Chapter 1

Introduction

With today's computer systems getting larger, more powerful, and more complex, there exists a need for larger, more powerful, and more complex software systems. This evolution has caused major ramifications in the software development process.

Programming in a text editor has become increasingly more difficult, especially when one is trying to manipulate a large program that is broken down into different modules. Another major concern from industry is that there is an increasing need to write programs with an eye towards the future; that is, using good design principles and specifications so that one can maintain the software better in the future. These ideas have led researchers towards the development of graphics, programming environments, and specification and design languages to help the programmer do his job better.

1.1 What is Mondrian?

Mondrian is a prototype environment for viewing and editing a specification and design language in a simple, geometric fashion using lines and boxes. Mondrian uses SEDL (Software Engineering Design Language) as its specification and design language, and

RPDE[1] as its base for a graphical environment. SEDL was chosen for its formal use of data modeling (using an abstract model form of the data), its formal use of specifications, and its availability. RPDE was chosen for its support of the stepwise refinement process and for its availability.

More specifically, Mondrian presents a graphical relationship between a specification and its refinement (on both a procedural and statement level), while also presenting a method for viewing both the external (user's) and internal (implementor's) view of a data model. This method exploits the underlying database that both RPDE and Mondrian use by storing a data model as **one** object that is presented using **two** perspectives.

Developing SEDL programs in the traditional textual way has two main problems:

1. SEDL's textual format necessitatees many redundancies.

2. SEDL's methodology forms natural relationships between different parts of its structure, such as a specification and its refinement or the external view and the internal view of a data model. In SEDL's textual format, these relationships are not always clear.

Mondrian's graphical capabilities will present these natural relationships in a manner that simplifies developing, editing, and viewing of structured programs that use specifications. Mondrian also eliminates redundant information via its use of the underlying database.

1.2 Differences From Other Environments

Other environments such as Pecan [4] and PegaSys [3] use pictures to help describe the structure of a program. However, they do not take into account specifications or data modeling issues, and they implement a node and arc paradigm to display their programs (i.e., objects are the nodes in a directed graph structure). The Cornell Program Synthesizer [6] makes use of comment templates as a way to write psuedo-specifications, but

[1]RPDE is an open-ended acronym, variously standing for: Refinement-based Product Development Environment; Research Program Design Editor; and Re-use Paradigm Development Environment.

it does not distinguish between comments and specifications and does not use graphics. Mondrian, however, accounts for both specifications and comments, and also uses a nested box paradigm to display a program's hierarchy. That is, each displayed object is associated with a rectangular region of the display, and any child of that object is assigned a rectangular space within its parent.

1.3 Introduction to SEDL

SEDL is a specification and design language being developed at IBM Research Division, T.J. Watson Research Center [5]. It combines the software development process (specification, design, and implementation) into one language. SEDL is executable and also provides the programmer the potential to record and maintain the development process of a software product in an integrated and cohesive presentation. Underlying the language is a design methodology based on the use of abstract specification (i.e., **what** is to be done) and a process of refinement for incrementally adding program design details (i.e., **how** it should be accomplished). When applied repeatedly, this process of refinement forms a hierarchy of intermediate abstractions being recorded.

1.3.1 Behavior Modeling

A behavior specification is used to record the specification of an operation. This specification may be an initial design specification associated with a program unit such as a function or procedure (i.e., a specification that states the functionality of the entire function or procedure). It may also be a terminal design specification that is to serve as the basis for implementation (i.e., a specification for a statement, list of statements, or the body of a loop), or any step in between. This process of refining behavior specifications from an initial phase to a terminal phase is commonly known as stepwise refinement. Each refinement step should reflect a definite design decision about how a particular specified behavior is to be implemented.

1.3.2 Data Modeling

Another aspect of SEDL is its use of data modeling as a way to represent abstract data types. An abstract type specifies a logical collection of data together with the operations on that data. An abstract type is presented in two parts: the specification and the body. The specification states the expected external properties and behavior of the abstract type; the body states its internal implementation. Associated with each specification is a model, and with each body is a representation. The model specifies the external view (i.e. the user's view) of the data type in a high-level, abstract manner, and the representation specifies the internal view (i.e. the implementer's view) of the data in a lower-level fashion. For instance, suppose one were to create an abstract type, called Smallset, for a set of elements with a size limit. Furthermore, the target language to be used does not have a built-in Set type. One still could use a Set type as the model for Smallset (e.g. Set(Max) of Elements). However, the corresponding representation would have to use types available to the programmer, such as Records, Arrays, and Integers (e.g. a Record with one field being an Array of Elements and the other field being a Counter of type Integer).

1.4 Problem Statement

Along with integrating SEDL with RPDE, Mondrian will also address the following issues:

- How should Mondrian graphically represent the hierarchy of design?
- How should Mondrian display and manipulate specifications along with their respective refinements?
- How should Mondrian represent both the external view and the internal view for an abstract data model?

The major issues Mondrian will focus on are:

1. The preservation and enhancement in a graphical format of a structured design methodology that uses stepwise refinement.

2. The ability to view both the external view and the internal view of a data model via one simple command.

The basis of my project is to build Mondrian. Mondrian is an integration of SEDL and RPDE. In RPDE terminology, I will be creating an SEDL world-view. A world-view is a collection of object types suitable for representing a programming language, in this case SEDL. An object type includes the definition of the object, its methods, and its display options. The user then has a comprehensive way of dealing with the information repository he is developing [1]. In creating an SEDL world-view, I will be designing the graphical representation of the SEDL language constructs called pictures. I will be adding functionality to existing methods in order to manipulate the display of various statement pictures and data model pictures, their relationships to their specifications, and their overall presentation. I will concentrate on the graphical representation and relationship between an abstract data model and its representation, and between a specification and its refinement.

1.5 Why Call it Mondrian?

Pierre Mondrian was a 20th Century artist who based his paintings upon simple geometric principles – in particular, the line and the rectangle. Mondrian's view of nature, like that of his contemporary Cubist artists, was that the essence of nature is the simple geometry that composes every living and non-living object. Mondrian captured his ideas using lines and boxes. My project, Mondrian, like Pierre Mondrian, attempts to view a complex structure (in this case, a specification and design language) in a simple, two-dimensional, graphical format.

Chapter 2

SEDL

SEDL implements two kinds of modeling:

1. Data Modeling
2. Behavior Modeling

In this section, we will discuss both kinds of modeling, their importance to the SEDL methodology of abstraction and stepwise refinement, and the various relationships that exist within each type of modeling and between the two.

2.1 Data Modeling

SEDL uses data modeling to represent the concept of data specification and design. This concept allows the designer to separate concerns about the use of data from those involving the representation of data. When focusing on the use of data, there is a need to specify completely the type of the data and the operations allowed on the data.

2.1.1 What is Data Modeling?

The concept of data modeling, using an abstract model form of the data, allows the designer to specify fully both the interface and the behavior of the operations specified

for a data object or type. Abstract types such as sets, maps, and lists are often chosen as abstract models of data. Such models reveal the essential structure of the data type and are free from implementation bias. The operations of the abstract model types are used to specity the behavior of the operations in terms of the data being modelled. To operate on the data correctly, the user needs no further knowledge than is given by the model and the specifications of operation behavior. The actual representation chosen for the data is completely hidden and may even be changed so long as the properties of the model are preserved.

2.1.2 Smallset - An Abstract Type Example

We will be referring to the abstract type Smallset as a typical data modeling example. Smallset appears in [5] and consists of integer sets whose cardinality does not exceed a given maximum size. See Figure 2-1 and Figure 2-2 for the definition of Smallset's specification and implementation, respectively.

In particular, notice that an abstract type[1] is declared in two parts:

1. The specification part containing the model type declarations, subprograms, and behavior specifications for the operations of the type.

2. The body part containing the representation of the abstract type and the implementation of its operations.

2.1.3 Relationships

Abstraction is the main concept behind data modeling. The model corresponds to the external (or user's) view, and the representation corresponds to the internal (or implementor's) view. The model also permits the designer to select a data representation

[1]Throughout this paper, an abstract type will be used as a typical representative of a data modeling object. Other choices that SEDL provides include a package, an abstract object, and an abstract type family.

```
        - Assume Max_size is a non-negative integer constant

abstract type Smallset is

    model
        type Smallset is Set of Integer;
        constraint
            for all S in Smallset : Card(S) <= Max_Size;
        initial
            Empty;
    end model;

    function Query(I: in integer; S: in Smallset) return Boolean
        <*I in S*>;

end Smallset;
```

Figure 2-1: Smallset Specification

during a sparate phase of development. For instance, the Smallset model is a Set of Integer. The Set type is a high level construct that SEDL has provided for the user so that Smallset can be thought of in an abstract manner. On the other hand, the implementor may not have the use of a Set in his library, so Smallset must be defined in terms of lower level constructs that are available, such as the Array and the Record. This link between the model and the representation is very important; SEDL's textual format, however, does not exploit this relationship. As a matter of fact, the physical separation between the model and the representation helps to *hide* the relationship.

Another important relationship exists between the behavior specification of an operation that appears in both the model and in the representation of an abstract data type. (Behavior specifications are a special kind of statement enclosed by the delimeters <* and *> and are discussed in Section 2.2). Notice how the specification of Query in Figure 2-1 is in terms of a Set, whereas the specification of Query in Figure 2-2 is in terms of an Array and Natural index. The relationship between a model and its representation

```
abstract type body Smallset is

    representation
        type Elem_Vector is Array(1..Max_Size) of Integer;
        type Smallset is
            Record
                Size:  Natural range 0..Max_Size := 0;
                V:     Elem_Vector;
            end record;
        constraint
            for all S in Smallset and J,K in 1..S.Size :
                J /= K ==> S.V(J) /= S.V(K);
        mapping of S in Smallset'representation to Smallset'model
            <* S.V(I) : I in 1..S.Size *>;
    end representation;

    function Query(I: in integer; S: in Smallset) return Boolean
        <* exists J in 1..S.Size : S.V(J) = I *>
is Separate;

end Smallset;
```

Figure 2-2: Smallset Implementation

is formally defined by the mapping function, which appears in the representation. This function maps the representation type onto the model type, thus defining how to produce a model value from a representation value. Again, this relationship between the model specification and the representation specification is *lost* in its textual format. Mondrian solves this problem.

2.1.4 Duplication

In very much the same way that the code for an Ada[2] package is physically divided into its specification part and its body part, so is the SEDL abstract type. This stipulation causes the programmer to repeat identical information in both the model and the representation, such as the name of the abstract type, and the name and parameter lists of the abstract type's operations. Furthermore, the model and the representation are two physically distinct pieces of code, which may cause consistency problems. Mondrian solves this problem as well.

2.1.5 Other Aspects of Data Modeling

In addition to the abstract type model definition and other associated definitions, the model section also contains a constraint, which is a condition that must always hold on the objects, and an initialization, which specifies the initial value for the objects of this type. The abstract type body also contains a constraint and initialization, but in terms of the representation and not the model.

[2]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

```
procedure SimpleProc(x : in Boolean; y : out Character) is
    <* x -> y := Any('B', 'C') *>;
```

Figure 2-3: Behavior Specification for SimpleProc

2.2 Behavior Modeling and Refinements

2.2.1 What is Behavior Modeling?

Each computer language has its own set of operations, usually procedures and functions. In SEDL, these operations are specified by behavior specifications. Behavior modeling is the act of writing a specification for an operation and its subsequent refinements. The behavior specification for an operation describes the underlying mathematical behavior. It specifies **what** the operation is, rather than **how** it is to be computed. In other words, it hides design details such as the algorithm used to accomplish the specified behavior and its implementation.

2.2.2 SimpleProc - A Behavior Model Example

A behavior specification in SEDL is a special form of statement enclosed by the delimiters <* and *>. When specifying an operation, one follows the procedure heading with an appropriate behavior specification followed by a semi-colon. The semi-colon indicates that the procedure has yet to be refined into lower level behavior specifications or actual code. In our example, Figure 2-3, SimpleProc takes a boolean argument, x, and a variable of Character type, y, as input and nondeterministically assigns y to the character 'A' or 'B' if x is TRUE. If x is FALSE, the procedure is undefined.

When one refines the procedure, the semi-colon is replaced by the keyword **is**, and then the appropriate refinement is chosen (e.g. a loop, or an if-then-else clause). See Figure 2-4. If the refinement is written as an external procedure, the keyphrase **is separate** is used. Also notice how the Then statement is refined into another behavior specification,

```
procedure SimpleProc(x : in Boolean; y : out Character) is
   <* x -> y := Any('B', 'C') *> is

Begin
   If x Then
      <* y := Any('B', 'C') *>;
   Else
      y := 'Z';
   End If;

End SimpleProc;
```

Figure 2-4: Behavior Specification and Refinement of SimpleProc.

whereas the Else statement is refined directly into code.[3] The entire stepwise refinement procedure enforces a specification-refinement relationship. This relationship, however, is lost somewhat in the syntax and overall textual representation.

[3]The specification of SimpleProc defines an action only if x is TRUE. If x is FALSE, the action is undefined. The implementor of this procedure decided to simply assign y to the character 'Z' in this case.

Chapter 3

What Mondrian Has to Offer

The basic idea behind Mondrian is that each SEDL object used for data modeling is **one** object composed of two perspectives. In the data model example to be presented, Smallset (Figure 3-3 and Figure 3-4), the object is the abstract type, and the two perspectives are the model view and the representation view (or rep view for short). In this paper, two possible ways in which Mondrian can handle behavior modeling are discussed that:

1. enforce the SEDL methodology of stepwise refinement, and
2. graphically display a specification along with its refinement.

Before presenting the data model example, however, we must present some basic terminology for dealing with Mondrian.

3.1 Notation and Terminology

In order to describe the graphical constructs and abilities that Mondrian offers, we must first establish a basic dictionary of terms and commands that Mondrian uses. This information provides us with a foundation from which Mondrian builds its properties.

3.1.1 Overall Structure

There are two distinct levels to Mondrian - the user interface, which is all that the user of Mondrian has access to in order to construct his program; and the source code for Mondrian, which is an extension of the RPDE source code. Furthermore, this source code is broken into two parts - the object type definitions, which is where I define all the pictures available in the user interface; and the methods, which determine exactly how the object type definitions are to be displayed and manipulated. The object type definitions are written in an assembler-like language[1], whereas the methods are written in Pascal.

3.1.2 The User Interface - Pictures

The diagrams that the user sees on the screen are called pictures. Some of the most common pictures in Mondrian are the procedure and the function. These pictures are themselves composed of pictures (such as the name picture, the parameter list picture, the definition list picture, and the statement list picture (the body of the procedure)), resulting in a hierarchy of pictures. The major pictures that are unique to Mondrian include:

- abstract type
- abstract object
- abstract type family
- package
- model

[1]The underlying structure of an object type is a record. The assembler-like code assigns values to some of the fields in this record. It is the methods that manipulate and display the record. For instance, the box methods determine that the box object type is presented as boxes and not, for example, as a node of a tree.

- representation
- specification
- exception handler

As previously stated, each of these pictures is defined as an object type definition. The most common object types are:

- boxes
- aggregates (lists)
- names

Due to time restrictions, I was constrained to use these existing object types in new ways in order to create my pictures. I also used and modified existing methods, as well as created some of my own methods, to display my pictures.

Associated with each object type (and therefore each picture) is a set of commands. These commands can be typed in on the command line (the bottom line of the screen) or chosen from a pop-up menu. Commands act on the current picture, which is determined by which picture the cursor is on. If a command is given that does not act upon the current picture, then the parent of the current picture is checked, and so on. This action is possible because of the hierarchical nature of pictures.

3.2 The Display of Objects

The key behind the Mondrian graphic presentation is the RPDE output composition facility. Interfacing functions called methods govern the display of individual objects. These methods further interact with general algorithms to provide a coherent viewing model. This view is composed of a number of windows, each one of which is an independent view of the information in the repository. Associated with each window is a focus set, the item or items that are of main interest to the user. The item that is initially

chosen from the repository by the user is defined to be the root. The RPDE display algorithm determines the correct geographic layout of a set of objects in which all the points of focus are displayed along with as much other nearby material as will fit in the window.

Unlike most editors that use scrolling to view material that doesn't fit on the screen, RPDE uses ellision. With ellision, material that can't be displayed on the screen is replaced by "...". In order to see the material at an ellided spot, one simply has to move the cursor there and give the command to focus out. Likewise, if one has focused in and wants to see the surrounding material, he simply has to give the command to focus in. This format allows the user to see the entire structure of a program as opposed to a linear set of text.

3.2.1 The Basic Box

Presented here is an example of an object type and its corresponding picture - the box. The box picture as provided in the RPDE library is shown in Figure 3-1. We will be referring to each rectangular subsection of the box in the course of this paper, so we have labelled each subsection with a specific name. The subsection in the top left-hand corner will be referred to as **TopLeft**, the top right-hand corner will be referred to as **TopRight**, and the bands directly underneath the TopLeft and TopRight sections will be referred to as **Band1** through **Band9**, from top to bottom.[2]

Associated with each band is a command name. By issuing this command, the appearance of the corresponding band is *toggled.* That is, if the band is currently showing, the band is "turned off", otherwise, it is "turned on". When toggled off, the band still retains all information – it is just hidden from view. When defining a box object type, the programmer has the option of giving the user this toggle capability for each band of the box object.

[2]At the current time, RPDE allows only 9 Bands in this type of a box.

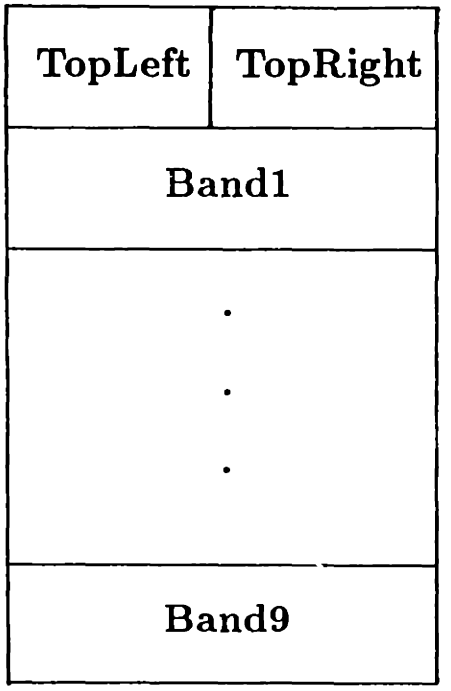

Figure 3-1: Structure of a Basic Box

3.2.2 Description Fields

Associated with many RPDE pictures is a description field. This area is a place to put informal text (e.g., comments). If the picture is a box object type, the description field is placed in a band of the box; thus, a band must be reserved for informal text for most boxes. Mondrian will make use of this RPDE feature as well.

3.3 Commands

Commands available to the Mondrian user manipulate either the Mondrian environment or a Mondrian picture, or do both. Mondrian uses existing RPDE commands, modified RPDE commands, and new commands to form its complement of commands. The following is a list of the major Mondrian commands:

- *close* - RPDE command. Closes the window containing the cursor and removes it from contention for display space.
- *distribute <command>* - RPDE command. Applies the given command to all of the immediate subpictures of the current picture.
- *mark* - RPDE command. Marks the object for use as an implicit argument to other commands. Marked objects are displayed specially on the screen. If an object is already marked, the mark command may be applied to a second object that is another element of the same aggregate as the first. In this case, the marked objects include all of the objects between the first and second mark inclusive.
- *modview* - Mondrian command. Creates a separate window with the current picture being displayed in its model perspective.
- *move* - RPDE command. Moves a marked region to the cursor position.
- *newfile <picture name>* - RPDE command. Creates the specified picture on the screen.
- *overlay* - RPDE command. Copies a marked picture to the cursor position.
- *perspective <#>* - RPDE command. Sets the perspective variable to <#> in the current window.
- *repview* - Mondrian command. Creates a separate window with the current picture being displayed in its representation perspective.
- *showmod* - Mondrian command. Sets the perspective so that pictures will appear in their model view.
- *showrep* - Mondrian command. Sets the perspective so that pictures will appear in thier representation view.
- *unmark* - RPDE command. Unmarks an object.

- *view* - RPDE command. Resets the current window's root to the picture pointed by the cursor and builds a new display in the window.
- *window* - Modified RPDE command. Opens a window for editing information. The window is opened in the same perspective for the picture at which the cursor is placed.

Another set of commands coerce existing pictures into new pictures, such as changing a simple statement into a conditional or loop statement or changing a statement into a spec-refine pair.

- *conditional, loop*, etc. - Modified RPDE commands. Replace the existing statement picture with a conditional or loop picture.
- *refine* - Modified RPDE command. Creates a spec-refine pair, with the current statement picture becoming the specification.
- *speclanguage* - Mondrian command. Changes the current statement picture into a specification picture (not implemented).

Finally, there are commands that simply toggle picture bands. These commands exist for each band in every box picture. They also exist for a refinement and its specification in a spec-refine pair.

- *description* - RPDE command. Toggles the description band of a picture.
- *refinement* - Modified RPDE command. Toggles the refinement band of a picture.
- *spec* - Modified RPDE command. Toggles the specification band of a picture.

3.4 Mondrian's Representation of an Abstract Type

Now that we have established a basic vocabulary and understanding of Mondrian's structure, we can discuss how Mondrian incorporates the ideas of data modeling and behavior modeling into its environment.

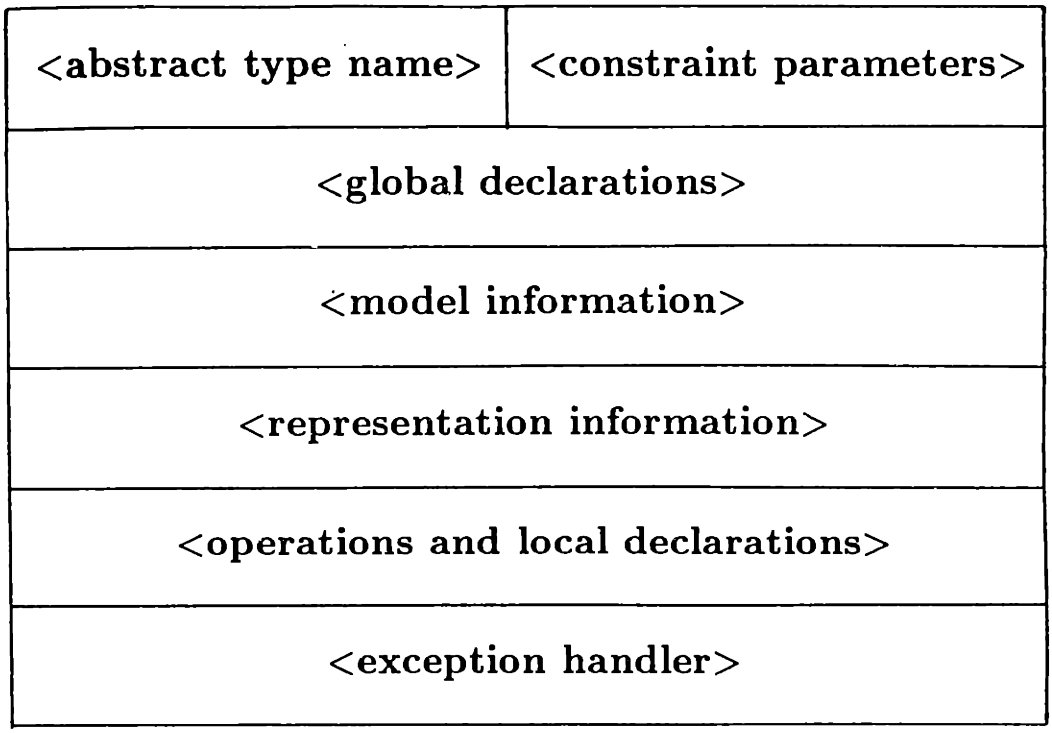

Figure 3-2: The Entire Abstract Type Picture Skeleton - Both Views Showing

Figure 3-2 is a representation of an abstract type skeleton as viewed in Mondrian. As the picture suggests, the consequence of having the abstract type picture be one object type is that the specification and the body are no longer two totally separate pieces of code. Mondrian stores both the specification and the refinement as part of **one** abstract type object. If the programmer is editing the model, he has immediate access to the representation, and vice versa. However, Figure 3-2 is **not** in either the model view or the rep view – it is a representation of what the abstract type picture would look like if both views were showing at the same time, in the same window, with all bands toggled on.

The way one uses Mondrian is to look at one view per window. For instance, Figure 3-3 shows the model view of the abstract type Smallset previously defined. Notice the differences between Figure 3-3 and Figure 3-4: Figure 3-3 is the model view and does not contain the representation picture (Band3). The function Query is also in its model

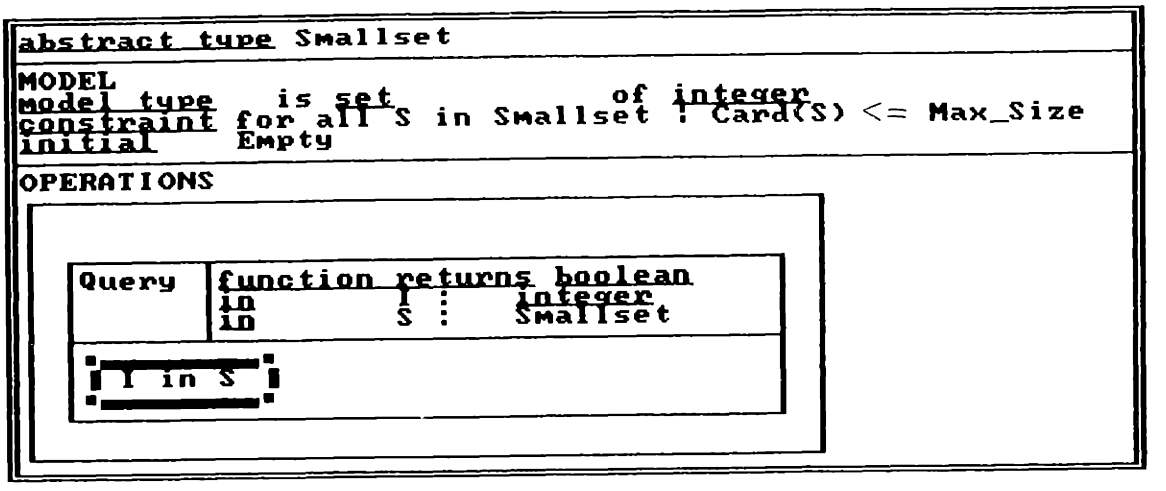

Figure 3-3: Model View of Smallset

view, showing only its name, parameter list, and its specification in terms of the model. All other information is suppressed and can never be seen[3] because it does not have any relevance in the model view.

Figure 3-4 shows the rep view of Smallset. In this view, the representation picture can be seen, but the model picture cannot. Notice that Query is now defined in terms of the representation. Also notice the bottom three bands of Query. These bands will be used to refine Query into code.

Another important aspect of Mondrian is its lack of duplication. For example, in the SEDL code for Smallset, the name "Smallset" and the name and parameter list of the function, Query, are duplicated in both the model and the body. So if the programmer

[3]This is not absolutely true. The RPDE protocol insists that a picture be shown as long as the cursor is on that picture. Therefore, if one were to give the command *repview* while the cursor was on the model picture (Band2), then the model picture initially would appear in the newly created repview window. This effect is very temporary, however, since one simply has to refresh the screen to make the model picture go away.

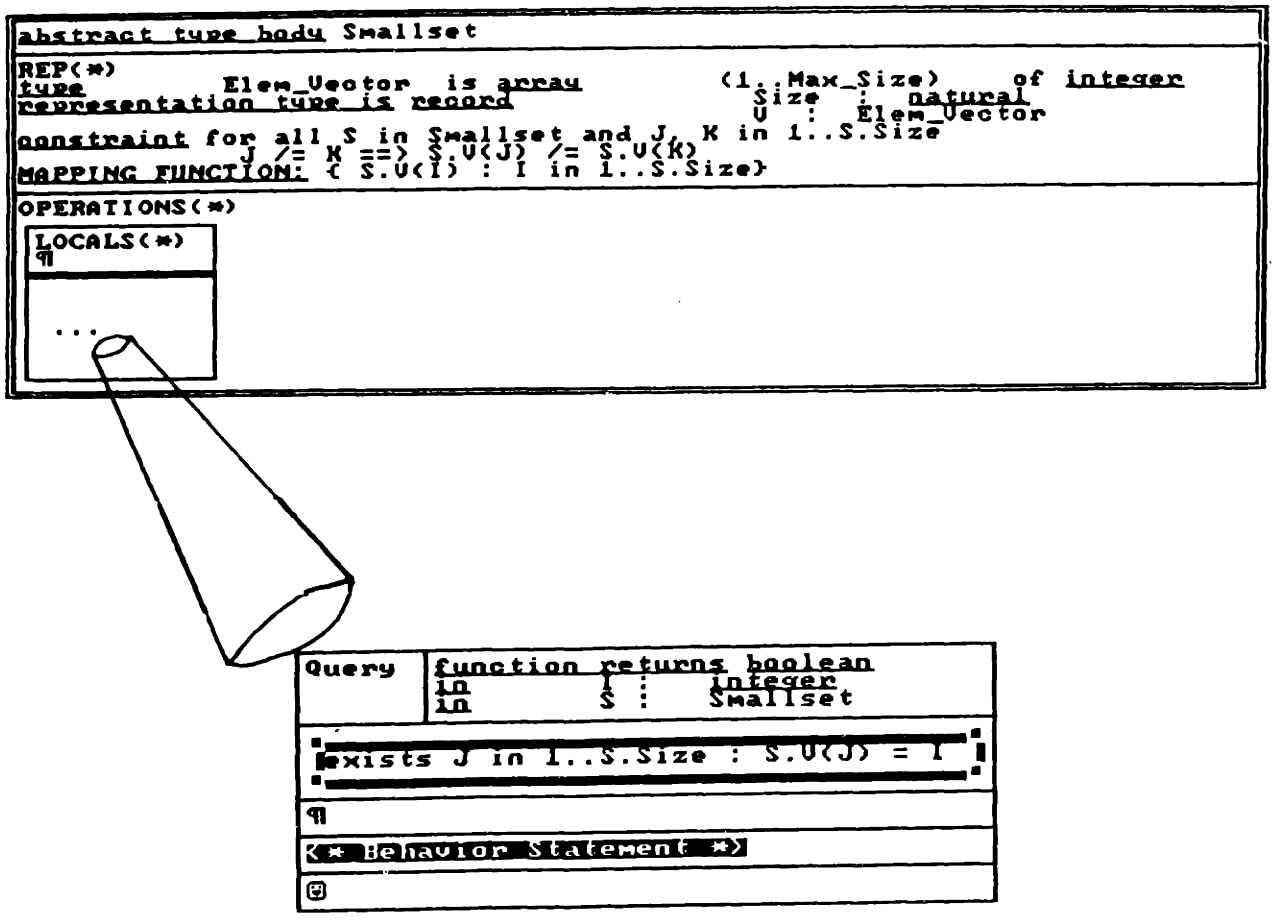

Figure 3-4: Rep View of Smallset

changes one, he has to remember to go and change the other.

This problem is solved in Mondrian. Notice that in Figure 3-2 there is only **one** band reserved for the name (TopLeft), and only **one** band reserved for the operations (Band4). However, this information appears in both the model view and the rep view (Figure 3-3 and Figure 3-4). Furthermore, since this information is actually the same object appearing in two different views, if any common information is changed in one view, the change will be propagated in the other view.[4]

3.4.1 Perspectives

A perspective is an RPDE variable that is global to all pictures (and therefore object types) in a window. This variable is set via the RPDE command *perspective*. A particular perspective setting has no effect on a picture if that picture's object type definition does not declare any perspective masks. Perspective masks are defined along with the object type. These perspective masks are used to control which parts of an object type can be shown. In particular, perspective masks can control which bands of an abstract type picture can be shown when in the model view or when in the rep view. The user controls which perspective is active at any given time via the two Mondrian commands *showrep* and *showmod*. These commands are only "sugar" for manipulating the RPDE perspective command.

For instance, the model view perspective mask allows for the TopLeft, TopRight, Band1, Band2, and Band4 bands to be viewed. The rep view perspective mask, on the other hand, allows for the TopLeft, TopRight, Band1, Band3, Band4, and Band5 bands to be shown (TopRight and Band5 are toggled off because they do not contain any relevant information in our Smallset example, Figure 3-4). In our example, the TopLeft band, which contains the name, "Smallset", appears in both views because the name of the abstract type is common information between the two views. Therefore, changing the name in one view automatically changes the name in the other view. In this fashion, the

[4]If both views are showing, one must refresh the screen first.

programmer can never enter an inconsistent state where common information between views is different. On the other hand, information concerning the representation is of no consequence to a user viewing the model; therefore, the representation picture (Band3) of the abstract type picture is not defined in the model perspective.

3.4.2 Perspectives are Global

Because perspectives are global, each picture contained within another picture will also appear in the current perspective setting. Therefore, when the model perspective is set, each picture in each band of the model view will also appear in its model view. Likewise, when the representation perspective is set, each picture in each band of the rep view will also appear in its rep view. Therefore, even though the operations picture (Band4) appears in both the model view and the rep view of Smallset, this picture has a different display in each view. Unlike the TopLeft band for which the picture is the same in both views, Band4 contains an operations picture which, in turn, has its own model views and rep views. Specifications of operations were discussed in Section 2.2.

3.4.3 Viewing and Modifying Perspectives at the Same Time

As previously mentioned, abstract types are designed to be examined in either the model view or the rep view. There is a specified way in which to view both the model view and the rep view at the same time.

To see two views simultaneously, Mondrian creates a new window, places the current picture in the new window, and sets the specified perspective. All of this can be performed by one command, which actually is the composition of four RPDE windowing, viewing, and perspective commands. For instance, the Mondrian command *repview* creates a second window that contains the current picture in its rep view. (See Figure 3-5). In both windows the programmer still has the capability to focus in and out, as well as to modify any information. Again, if any information that is modified is common information, the change is reflected in the other view.

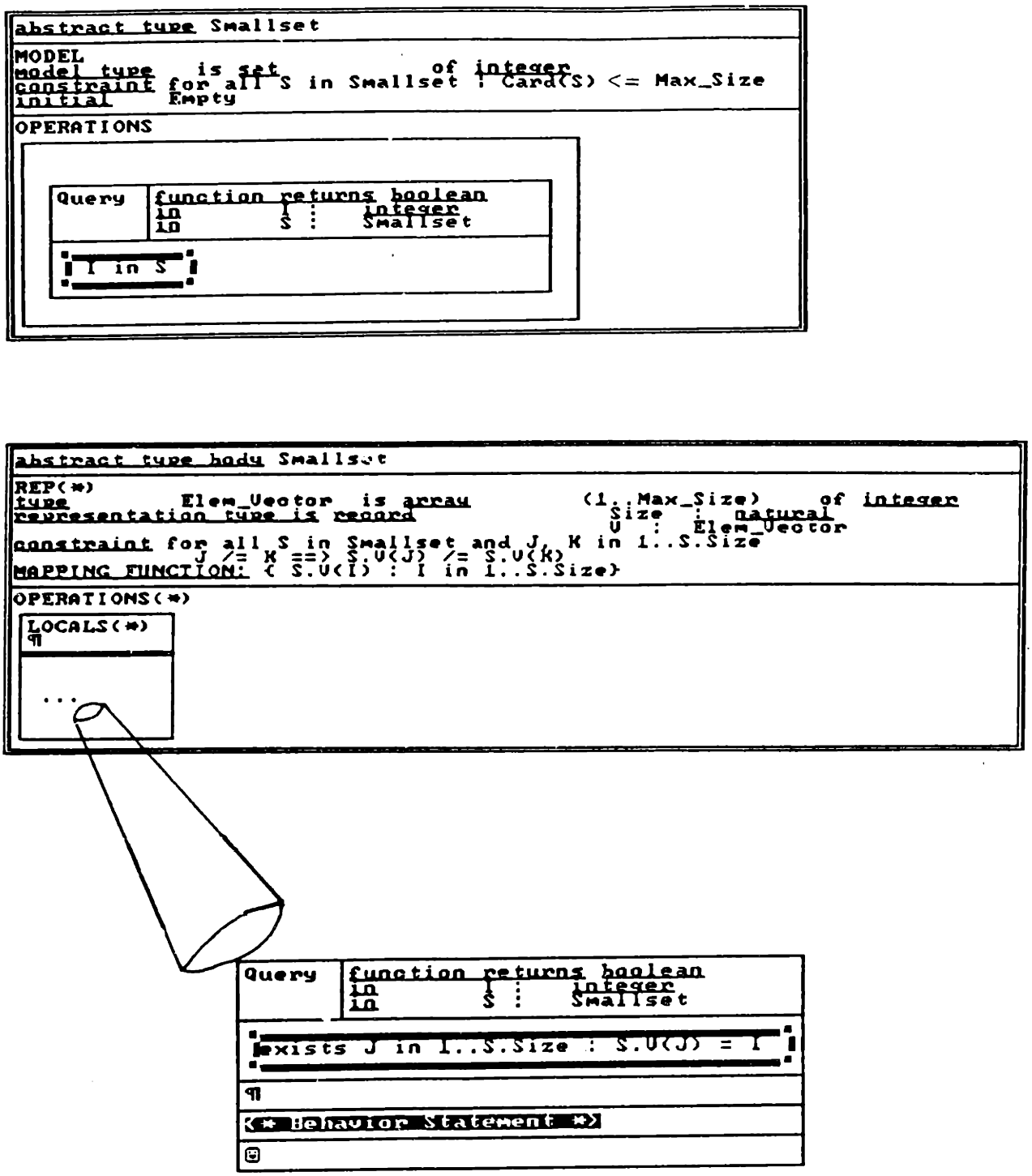

Figure 3-5: Properly Seeing Two Views of Smallset at the Same Time.

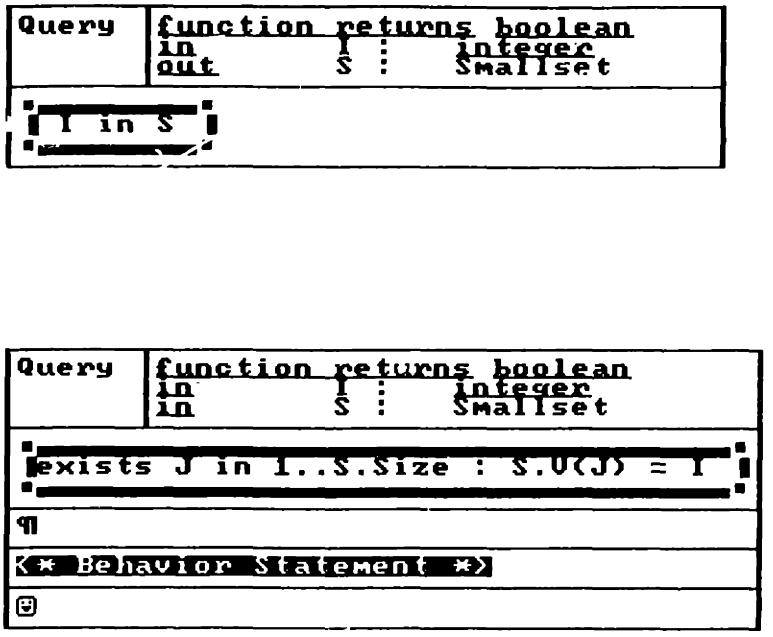

Figure 3-6: Properly Seeing Two Views of Query at the Same Time.

Unlike the textual representation, Mondrain offers a way to maintain a relationship between corresponding pictures in the model and rep views at various levels of granularity. For instance, if the current picture is an operation such as the function Query, then issuing the *repview* command will result in a picture like Figure 3-6. Furthermore, if the current picture is the behavior specification of the operation, then issuing the *repview* command will result in a picture like Figure 3-7.

By making the programmer explicitly create a new window in order to view a second perspective, the idea of modularity and abstraction between the two views is reinforced. Perspectives also ensure that it is not possible to see any model view information while in the rep view, and vice versa. There are problems with perspectives, however. These problems are addressed in detail in Section 5.

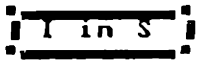

Figure 3-7: Properly Seeing Two Views of a Behavior Specification at the Same Time.

```
procedure SimpleProc(x : in Boolean; y : out Character) is
    <* x -> y := Any('B', 'C') *>;
```

Figure 3-8: Behavior Specification for SimpleProc

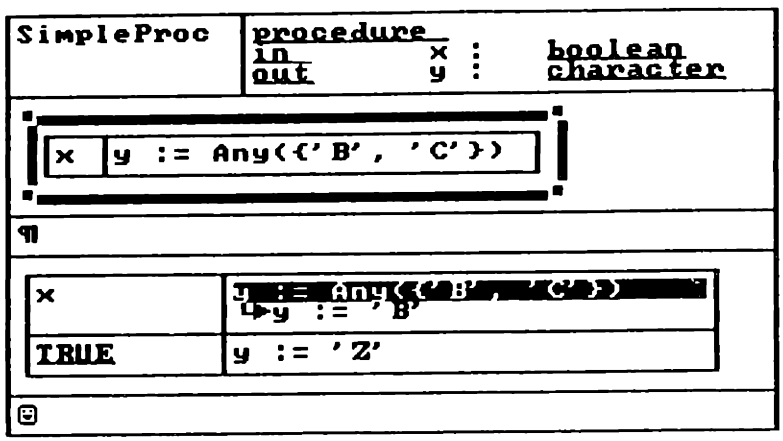

Figure 3-9: SimpleProc as Seen Using Mondrian

3.5 Behavior Refinement in Mondrian

3.5.1 The Specification

In its textual format, a SEDL behavior specification is indicated by the surrounding delimeters <* and *>. This representation is the same for both the specification of an operation and the specification of a statement. See Figure 3-8, which is our old friend SimpleProc.

Mondrian does not need the special symbols <* and *> to signify its specifications. Figure 3-9 is SimpleProc as seen in Mondrian.

First, notice the extra thick box that encloses the behavior specification in Band1. Statement pictures appearing in such a box indicate that the statement is the behavior specification for the operation. In this example, the behavior specification is a form of

conditional called a partition. The partition picture is represented as a two-column table. Conditions are listed in the left column, while the corresponding actions (or statements) are placed in the right column on the same row. If a condition is not specified, its action is considered undefined.

Second, notice the representation for the body of SimpleProc (Band3). This representation is a conditional picture. Like the partition picture, the conditional picture is represented as a two-column table with the conditions listed in the left column, and the corresponding actions (or statements) placed in the right column on the same row.

Also notice the difference between the first condition (the Then) clause and the second condition (the Else clause). The Else clause is a simple statement and appears as such. However, the Then clause is actually a *refined* statement specification.

Mondrian represents a statement specification and its refinement using reverse video and the refinement arrow, ↳. The specification (y := Any({'B', 'C'})) is put in reverse video, and the refinement arrow points to the specification's refinement (y := 'Z'), thus creating a spec-refine pair. Besides being a very recognizable symbol, this refinement arrow also indents the refinement of its specification. This indentation also makes it easy for the user to distinguish quickly between the specification and its refinement.

3.5.2 The Stepwise Refinement Process

In the stepwise refinement process, each refinement step should reflect a definite design decision as to how a specified behavior is to be implemented. The reasoning behind specific design decisions, such as, "An array was chosen over a linked list because ..." should be documented in the comment band of the appropriate picture. Two different methodologies that enforce the stepwise refinement process have been implemented in Mondrian and will be discussed.

Version 1 - The Strict Approach

In the Strict Approach, there are two kinds of statements – one for specifications, and one for code. In order to encourage users to write specifications, Mondrian prompts the user with a specification statement by default:

< Behavior Specification *>*

The user simply has to write his specification over the prompt, the input is automatically put into reverse video, and the user is done. For instance, the specification "there exists an a such that a := 5" appears as:

Exists a : a := 5

If the user wants to convert to a different kind of specification[5], he gives the appropriate command. When dealing with specifications, one can only convert to another specification or give the *refine* command.

When the user is ready to refine his specification, he gives the command *refine* at the current picture, and a spec-refine pair is created:

Exists a : a := 5

↳*<* Behavior Specification *>*

The refinement is again a specification statement prompt, and the process can be repeated. At this point, however, the user may want the refinement to be actual code. The user must first issue the command *ProgramStmt* to convert the specification statement into a program statement:

Exists a : a := 5

↳*statement*

A program statement is any kind of statement, such as a loop or a conditional. If the refinement is a simple statement, such as a := 5, then the user types his statement over the prompt:

[5]SEDL allows for four different kinds of specifications: simple, partitions, multiple, or text.

Exists a : a := 5

↳a := 5

If the refinement is to be a more complicated structure, such as a loop or conditional, then the user issues the command *loop* or *conditional*, and the simple statement is transformed into a loop or conditional picture. Once a statement has been transformed into a program statement, the *refine* command is no longer available because the user has reached the end of the stepwise refinement process.

The advantage to this version is that the approach is very methodical and strict. One either writes a specification statement, or explicitly converts it to a program statement. There is no confusion between specifications and code. The price, however, is in terms of object type definitions and inconvenience. First, two object types must be defined for statement pictures – one for a specification, and one for code. Second, it is common to have a simple statement with no specification in the lower level refinement. In such a case, it would be an inconvenience for the user first to convert the behavior statement into a program statement, and then to write the statement. This inconvenience is handled in the next approach, but other problems arise.

Version 2 - The Compromise Approach

The Compromise Approach eliminates the duplicate definition problem by allowing the user to input simple code statements without first converting to a program statement. However, the distinction between specification statements and code statements is not as clear. This approach assumes that each statement is a specification that needs to be refined. Therefore, when the prompt *statement* is on the screen, Mondrian expects a specification to be entered:

statement

However, if the user wants to write a simple program statement such as a := 5, then he can do so without first converting to a program statement:

a := 5

One can do this because Mondrian assumes that any simple statement without a refinement is **code**. This method, then, removes the inconvenience of issuing the *ProgramStmt* command as in the Strict Approach.

However, if the statement is a specification (e.g., Exist a: a := 5), then one issues the command *refine* to create a spec-refine pair. Now the programmer is left with another *statement* prompt as the refinement:

Exists a : a := 5

↳*statement*

This statement is another behavior statement. If another refinement is needed, then the process is repeated.

At this point, however, the programmer also has the option of changing the behavior specification into a program statement. As before, if the refinement is a simple statement the programmer simply types it (e.g a := 5) over the *statement* prompt:

Exists a : a := 5

↳**a := 5**

If the refinement is a more complicated structure, such as a conditional or a loop, then the programmer issues the command *conditional* or *loop*, and the simple statement is converted into a conditional or loop picture.

There is one special case during the refinement process that must be handled carefully. Suppose one wants to write a simple behavior specification with no refinement. As stated previously, one would just write the specification over the *statement* prompt. However, if left in this condition, Mondrian will assume that the statement just written is a code statement, and not a behavior specification statement. In this instance, known as the base case, the programmer must somehow indicate that his behavior specification is not the base case refinement.

One can indicate this difference in one of two ways. First, the programmer can explicitly give the *refine* command, with no intention of filling in the refinement at the present time:

Exists a : a := 5

↳*statement*

As stated in Section 3.2.1, the user can simply toggle the refinement band off if he doesn't want to look at the empty refinement. Alternatively, the programmer could give the command *speclanguage*, which would explicitly tell Mondrian that the current statement is to be viewed as a specification.

3.5.3 Viewing Specifications and Refinements

As previously stated in Section 2, underlying the SEDL methodology is the idea of abstraction. To visually enhance this methodology, Mondrian allows the user to toggle the refinement band off if he is not interested in a specification's implementation. Mondrian also allows the user to toggle the specification band of if the user is strictly interested in viewing a refinement and not the specification itself.

Furthermore, one can use the RPDE command *distribute* along with the band toggle commands *spec* or *refine* in order to distribute the band toggle command throughout the current picture and its subpictures.

Chapter 4

RPDE's Role in Mondrian

RPDE is a programming and design development environment whose heart is an editor that displays the programmer's material (code, documentation, comments, and other information) in a two-dimensional, graphic form [1]. RPDE views the creation of a program as the presentation and modification of a comprehensive data repository containing all the design and implementation information.

RPDE currently adheres to a development by refinement methodology. That is, the user is encouraged to begin with a natural language description (e.g., a comment) and refine it into an implementation. The natural language statement is automatically placed in the description field of the implementation picture (i.e., statement, conditional, loop, etc.), thus becoming a comment for the refinement.

The features of RPDE fit well with the SEDL methodology. Both methodologies are based on the ideas of abstraction and stepwise refinement. SEDL is a precise and formal design language, and RPDE presents information in a structured manner. By combining the two, one should be able to develop a programming environment for a design and specification language that is able to maintain, control, and present all of the information in a useful and coherent fashion. Thus the idea for Mondrian began.

The overall structure of Mondrian is inherited from RPDE. Due to time constraints, I concentrated mostly on object types that were defined as some sort of box, thus resulting

in a nested box paradigm. I also was able to become very familiar with methods and options associated with boxes. However, due to the variety of display flags associated with a box object type, the box picture is not necessarily displayed as a box. This allows for a greater diversity in the appearance of box object types, which makes the overall appearance more diversified and more appealing.

The use of lines and arrows to show relationships between objects was also explored. There are two cases to consider – a graph structure and a tree structure. The use of lines and arrows to show a graph structure idea met a quick death because of the spaghetti argument. This argument states that even with a small number of interconnected objects, the connecting lines soon turn into "spaghetti", making the picture unreadable and the design hierarchy unclear. The nested box approach, with the sharing of common subobjects, allows for a coherent and visually well defined picture.

On the other hand, if the structure of a program is a tree, one does not run into this spaghetti problem. Using lines and arrows to show relationships at a modular level and even a procedural level can be very informative. For instance, a common use of lines and arrows is to show the flow of data. However, using this paradigm to show the structure of low level objects, such as a loop or conditional, can confuse the user more than help him. The structure of these objects is more presentable in a box format.

Although Mondrian does not find the node and arc paradigm useful for showing relationships between objects, the line and arrow play an important role in showing the relationship between a specification and its refinement. This refinement arrow is acceptable because there is no fear of the spaghetti problem. First, the refinement arrow is very small; and second, the arrow always points to the object directly beneath it. Thus, these arrows aid the viewer rather than hinder him. Furthermore, if Mondrian did use a node and arc paradigm, the importance of the refinement arrow would be obscured.

Chapter 5

Problems with Perspectives

When dealing with a data model whose operations are strictly procedures or functions, the model views and the rep views of both the abstract type and the procedure can be shown consistently. However, problems arise when one tries to declare a data model (e.g. an abstract type) in the body of another data model. Due to time restrictions, the following solutions are discussed but were not implemented.

5.1 Global Perspectives Cause Problems

Although global perspectives work well with data models and procedural operations, a problem arises when one wants to declare another data model, say an abstract type called Inner, in the body (or rep view) of another data model, say an abstract type called Outer. Inner can only be seen in the rep view of Outer since its declaration is private to Outer's body. However, because perspectives are global, Inner will also be seen in its rep view, even though what we want to see is Inner's model view.

There is no simple solution to this problem. One solution would be to keep Mondrian as is, but make the user create a new window that contained Inner. This can be done easily with the RPDE command *view*. The user would then be responsible for issuing the Mondrian command *modview* in the newly created window, thereby making Inner appear

in its model form.

The aforementioned solution is not an ideal solution because it is based on my implementation of Mondrian. Ideally, Mondrian should handle this situation consistently and internally. One such solution is to make perspectives local to each picture. In this manner, each picture would initially appear with both views showing, waiting for the user to indicate which perspective he wants for that picture.

Another solution would rely on a smart display algorithm that could distinguish between Outer and Inner. Basically, the current picture would be considered "Outer". Outer could be seen and manipulated in both views, but Inner could only be seen and manipulated in its model view. To see the rep view of Inner, one simply has to make it the current picture.

5.2 Get Rid of Perspectives?

Another possible solution would be to get rid of perspectives altogether. The user would have to specify for each picture which bands he wants to see. Such an act could easily be tailored to a single keystroke. This solution is called the virtual perspective solution because even though it appears as if the object is being shown in a certain perspective, a band associated with the other perspective can be toggled on at any time. The drawback to this solution is that the protection offered by a global perspective is no longer available. Using global perspectives, one can toggle individual rep view bands while in the rep view, and likewise when in the model view. However, using the virtual perspective solution, one is able to toggle on a rep view band while in the model view. One could restrict the user from viewing inappropriate bands by allowing the user only two commands – one to toggle all the model view bans while suppressing the repview bands, and vice versa. However, the freedom to toggle individual bands would be lost.

Chapter 6

Other Issues

Like Ada, SEDL allows for the use of generic values and necessitates the declaration of any external operation to be used. An example of a generic value can be seen in our original example of Smallset in Figure 2-1. In this figure, the opening comment states "Assume Max_size is a non-negative integer constant". One can formalize Max_size as part of the code by making it a generic value. When one creates the abstract type Smallset, he must instantiate the declaration with a value for Max_size.

An example of the declaration of an external operation would occur if Smallset invoked another, external procedure from its body. In this case, the name of the external procedure must be declared beforehand, immediately outside of the procedure definition, so that Smallset knows of its existence.

In the textual format, both of these declarations must occur before the operation or data model heading, and as a consequence, the exact scope can be unclear. Mondrian handles both of these cases in the same way. If the user wants to add one of these features to the operation he is currently viewing, he simply has to give the *generic* command. This command forms a box "template" around the current operation picture. In the TopLeft of this template, one can enter either generic values or external operation declarations. In both cases, the idea that these values encompass the entire operation is visually reinforced. The fact that these definitions **surround** the picture instead of

appearing before the textual code makes the scope much clearer.

Chapter 7

Conclusions and Directions

7.1 Test Results

I have demonstrated Mondrian to a select group of individuals, most of whom are involved in the RPDE project. The general reaction was very positive. Most people liked the way that Mondrian manipulated the model view and the rep view of an abstract type. The idea of the spec-refine pair was not unfamiliar to this group because RPDE has a similar capability, so their reaction to this aspect of Mondrian was one of acceptance.

However, I also demonstrated Mondrian to a couple of individuals not involved with the RPDE project, but who are involved with Ada or SEDL. Their reactions were even stronger where the manipulation of the two views was concerned. And the spec-refine pair concept also was favorably accepted, although not to the same degree as the two views.

7.2 Directions for Future Work

The most important issue to be dealt with is the perspective issue. Some consistent way must be devised to handle the declaration of a data model in the body of another data model. Some possibilities were discussed in Section 5.

A follow-on project that in itself could be a thesis is to develop display algorithms specifically for SEDL. This "smart" algorithm would know about SEDL's structure and would prettyprint Mondrian in a manner that would exploit design structure to its fullest. Currently, Mondrian uses the RPDE algorithm, which fits as much nearby material on the screen as possible. This algorithm is based on the work done by Mikelson [2], which explains various ways to prettyprint a graphical environment using ellipses. This algorithm can be fine-tuned to account for all of SEDL's pictures.

Although Mondrian was not developed to be a syntax-directed editor, Mondrian could be modified in such a way as to become one. Such a project would involve creating many more object types, templates, and methods in order to handle a large language such as SEDL. Currently, Mondrian allows for the entry of illegal constructs. It does not have a template for every kind of SEDL construct. Because Mondrian is not strictly a syntax-directed editor and because it is still a prototype, Mondrian assumes some prior knowledge of SEDL for its proper use.

Concerning externally declared operations, an implementation could be developed such that an external reference to a procedure (e.g., with Proc1;) is actually a pointer to that object, and not just a named reference. Such an implementation would benefit the user because he would have direct access to that procedure wherever its name appeared. This implementation is possible because RPDE handles all its objects, which include procedures, as entries in its database; therefore, one can have multiple pointers accessing the same object.

Chapter 8

Acknowledgements

There are many people that I want to thank, for without them, Mondrian never would have been built and this thesis never would have been written. First, I want to thank Bill Harrison and the RPDE group at IBM Research for their help in teaching me about the RPDE system. Specifically, Harold Ossher and Brian Weston were invaluable in answering question after question and helping form the early versions of this draft. Second, I want to thank Dr. Gerry Fisher and the SEDL group at IBM Research for supporting my efforts as well. Third, I want to thank Bill Weihl for being my MIT advisor and for all his help in polishing both my thesis proposal and my actual thesis.

Most importantly, I want to thank my mother and father. Without their never-ending support and love in all my activities, I would not be where I am today.

Bibliography

[1] William H. Harrison. RPDE(3) - an environment framework for integrating tool fragments. Technical report, IBM Thomas J. Watson Research Center, 1987.

[2] Martin Mikelsons. Prettyprinting in an interactive programming environment. In *Proceedings of the ACM SIGSOFT/SIGOA Symposium on Text Manipulation*, pages 108–116. June 1981.

[3] Mark Moriconi and Dwight F. Hare. Pegasys: A system for graphical explanation of program designs. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 30–41, 1985.

[4] S.P. Reiss. Graphical program development with Pecan program development systems. In *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 30–41, 1984.

[5] Ann E. Kelley Sobel and Gerry Fisher. A specification and design language. Technical Report 12718(57227), IBM Thomas J. Watson Research Center, April 1987.

[6] Tim Teitelbaum and Thomas Reps. The Cornell program synthesizer: A syntax-directed programming environment. In *Communications of the ACM*, pages 563–573, September 1981.