Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-87-30

1987-12-01

The SMGJ Segmentation System: Users' Manual

Will D. Gillett

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Gillett, Will D., "The SMGJ Segmentation System: Users' Manual" Report Number: WUCS-87-30 (1987). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/816

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

THE SMGJ SEGMENTATION SYSTEM: USERS' MANUAL

Will D. Gillett

WUCS-87-30

December 1987

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

This research was supported by the Defense Mapping Agency under contract number DMA800-85-C-0010.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. Introduction

The SMGJ System is intended to be a testbed for the discovery and investigation of algorithm for segmenting images using a Split-Merge-Group-Join approach (to be discussed subsequently) based on arbitrary but specific properties (such as the size of a region, the difference in grey-level in a region, or the texture in a region) of the image. Predicates based on these properties can be defined and used to "drive" the segmentation algorithms intrinsically present in the system (split, merge, group, and join). These predicates may have variable parameters which can be changed to selectively alter the sensitivity of the predicates, thus affecting the specific segmentation of an image. The system can be used to implement and evaluate algorithms based on different strategies and methodologies which use these properties/predicates (and their parameters) applied in different ways.

Given a specific *strategy*, the user can refine the strategy into a specific *methodology*. As an example, two different strategies might be: (a) apply different properties/predicates sequentially, and (b) apply different properties/predicates simultaneously. Of course, these two strategies can be merged by combining sequential and simultaneous applications of properties/predicates. Two specific methodologies might be: (a) to oversegment the image and subsequently combine regions that have been oversegmented, and (b) undersegment the image and continue to separate regions that have not been segmented enough.

Given a specific methodology, the user can instanciate it by developing specific properties/predicates (with parameters), implement them in C code, and install them into the system. Sequencing and parameter adjustments can then be used to determine empirically what specific actions produce appropriate segmentations. In this manner the methodology can be refined into a specific *algorithm* potentially capable of performing the segmentation desired. An example of a specific algorithm might be: (a) *split* the original rectangular image into systematically placed regions of 4×4 pixels, (b) *merge* these regions into other rectangular

regions (composed entirely of the original 4×4 pixel regions) so that within any newly merged region the grey-scale levels differ by no more than 60, (c) *group* the new rectangular regions into (potentially non-rectangular) groups whose grey-scale levels differ by no more than 80, and (d) *join* these groups together by combining small groups of say 25 pixels or less with adjacent groups. Splitting and merging is done on a quad tree representation of the image in which the original rectangular image is recursively subdivided into 4 successively smaller rectangular subregions, a north-west region, a north-east region, a south-west region, and a south-east region. Grouping and joining produce groups of arbitrary shape, composed of components defined by the quad tree produced during the splitting and merging operations.

As an example, consider the abstracted 4×4 image shown in Figure 1. The grey-scale intensity is indicated by the number inside the (dashed box) pixel; the grey-scale intensities vary from 1 to 4. If we were to split this image into its quad tree representation to a level of size 1×1, the result would look like that shown in Figure 2. This is a "flattened out" pictorial representation of the quad tree; there are actually 21 nodes in the quad tree: the top level containing one node representing all 16 pixels, the intermediate level containing 4 nodes (4 pixels each), and the bottom level containing 16 nodes (1 pixel each). The corresponding tree structure is shown in Figure 3. If we were to now merge regions of the quad tree so that within

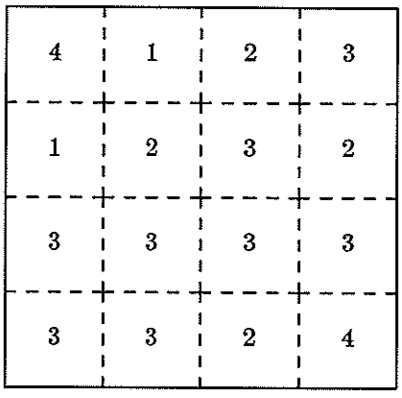

Figure 1: A Sample Abstracted Image

4	1	2	3
1	2	3	2
3	3	3	3
3	3	2	4

Figure 2: A Split Quad Tree

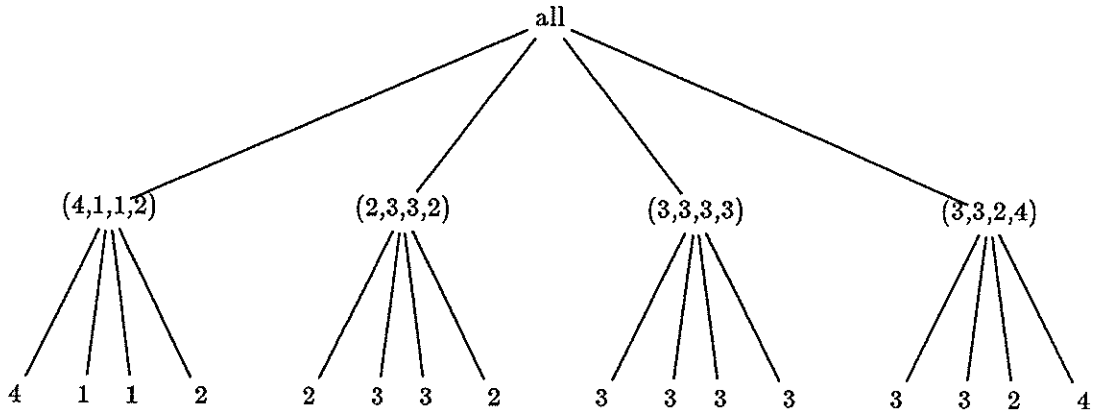

Figure 3: Quad Tree Structure of Figure 2

any merged region the grey-scale levels differs by no more than 1, the resulting quad tree would be as shown in Figure 4. This quad tree has 13 nodes; its structure is shown in Figure 5. (Splitting and merging is done on quad trees because the nature of the recursive decomposition of the rectangular image allows the processing to be done very efficiently.) If we were to now take the quad tree from Figure 4 and group the regions so that the grey-scale levels in grouped regions differs by no more than 2, we might obtain the result shown in Figure 6. This segmentation has 4 groups in its group set. Note that the result is somewhat non-deterministic; the sequence in which the regions are chosen for potential combination does affect the final result. If we were to now join the groups in Figure 6 to eliminate groups of size 1 or less, we might obtain the result shown in Figure 7. This segmentation has 2 groups in its group set.

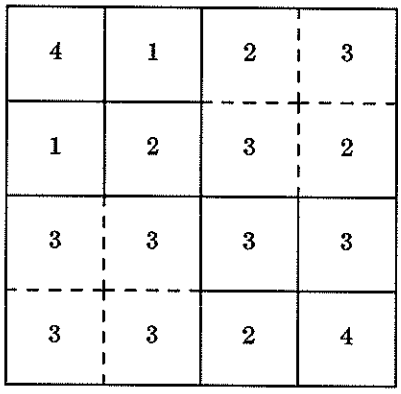

Figure 4: A Merged Quad Tree

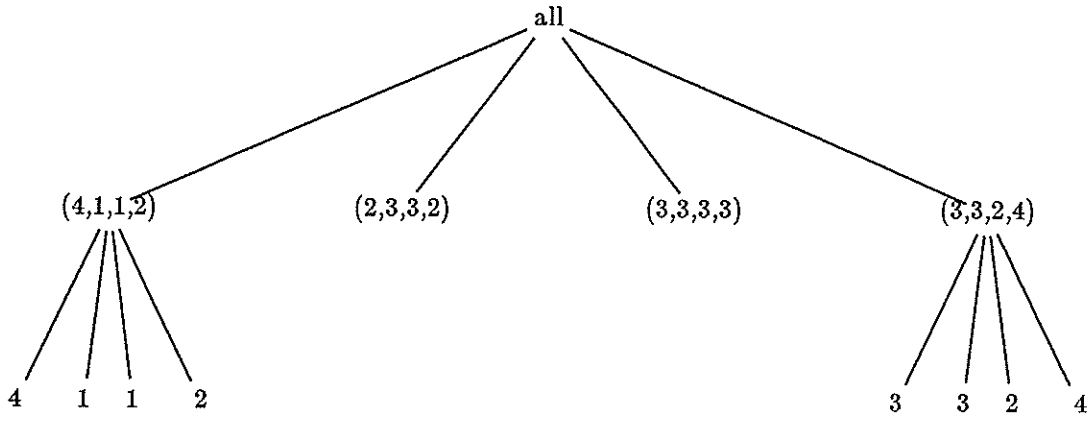

Figure 5: Quad Tree Structure of Figure 4

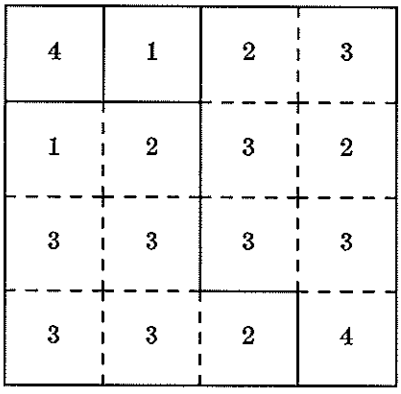

Figure 6: A Group Set from the Quad Tree in Figure 4

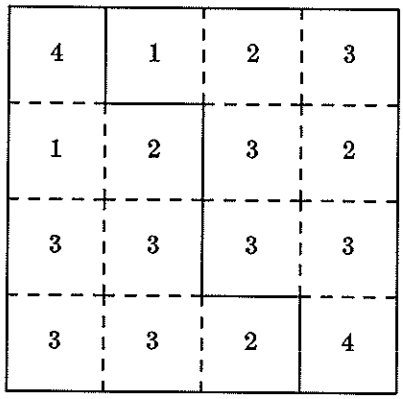

Figure 7: Small Group Elimination on Figure 6

Again, this process is somewhat non-deterministic.

Given these conceptual operations of split, merge, group, and join, it is possible to develop strategies, methodologies, and algorithms for segmenting large classes of images. The SMGJ Segmentation System is a skeletal software system which physically implements these operations and makes them available to the user through an interface language. The system allows the user to define and implement user defined properties/predicates and install them within the system. The user's task then becomes one of (a) deciding which properties/predicates may be of use for segmenting images, (b) implementing them in C, (c) installing them in the system, and (d) developing algorithms based on these properties/predicates (along with appropriate parameter selection) sequencing their use within the framework of the split-merge-group-join paradigm.

The interface language allows for arbitrary sequencing of applications of these operations using the user defined properties/predicates. It has facilities for (a) reading in images, (b) displaying images, (c) displaying the results of operations using a multi-color overlay facility, (d) printing the results of operations on the terminal, (e) setting parameters for the predicates, (f) manipulating multiple internal data structures (quad trees, group sets, and regions of interest), (g) obtaining statistics about the state of the system, (h) directing the low-level selection activities of the system, and (i) obtaining on-line help about the syntactic structure of the

interface language. In some cases, information can be input either through the interface language or through use of a track ball (or mouse) and buttons. There are also certain utilities available to help the user achieve his/her goal: zoom and roam facilities are available; timing information can be obtained; a UNIX system command escape facility is available; the system can be put to sleep for reactivation later; a pause facility is available; and group sets can be written to files to be read back into the system later.

2. Report Notation

2.1. Fonts Used

Different fonts will be used to indicate concepts in different realms. The **bold font** will be used to indicate user interface concepts. For example, everything that the user might enter using the interface language will be in this font. The `listing font` will be used to indicate programming, language, and UNIX system concepts. For example, internal data structures, language concepts, and file names will be in this font. The *italics font* will be used to indicate the first occurrence of important words and to emphasise words.

2.2. The Meta-language

A meta-language will be used to describe the syntax of the user interface language. Meta-characters in this language are: '<', '>', '(', ')', '[', ']', '{', '}', '*', '→', and '|'. The symbols '<' and '>' are used to delimit the names of nonterminals. The symbols '(' and ')' are used to indicate groupings for possible alternatives. The symbols '[' and ']' are used to indicate optional occurrences. The symbols '{' and '}' are used to indicate that a user supplied name is to be inserted. it preceeds. The symbol '→' is the rewriting symbol, and indicates what a nonterminal can be replaced with. The symbol '|' is the disjunction symbol which indicates

alternative choices. Whenever a character appears by itself (usually in conjunction with others), it in general mean itself. When a meta-character is to be used as an object character, it will be embedded between apostrophes ("'").

3. System Overview

3.1. Image Format - PDS

All images to be read into the system are held in files in PDS format. This PDS format has a 512-byte header preceding the image pixel data. There is standard software for creating, manipulating, and displaying PDS format images. For completeness the structure of the PDS header is shown in Figure 8.

3.2. Display Devices

The SMGJ Segmentation System is currently running in two different hardware configurations: a VAX11/750 with a Gould DeAnza IP8500, and a GPX with color monitor.

The DeAnza implementation uses two 8-bit memory planes for display. One memory plane is used for the grey-scale image; the other is used as a color overlay channel. These two memory planes are independent and can be displayed simultaneously. Thus, when displaying boundaries of quad tree and group sets in the overlay channel, such boundaries can be displayed and cleared independent of the grey-scale image. The overlay channel is configured so that each bit corresponds to a different color. If multiple bits are on, the colors are combined, i.e., red and green produce yellow.

The GPX color monitor has only one 8-bit memory plane. Thus, the grey-scale image and the overlay color display are held in the same memory. Displaying the boundary of a quad tree

```
/*
 *    pds.h -- pds picture header description.
 *
 */

#define twobyte short              /* for VAX and 11/70 */

#define PDSMAGIC 052525            /* pds file indentification code */

#define P_TSIZE 40                 /* size of title */
#define P_DSIZE 128                /* size of additional information */
                                   /* these will not be easy to change */

struct pds_hdr {
      twobyte hdr;               /* magic number to indicate pds file */
      twobyte fmt;               /* format descriptor of this file */
      twobyte xsize;             /* number pixels/line */
      twobyte ysize;             /* number of lines */
      twobyte zsize;             /* number of bits/pixel */
      twobyte nchan;             /* number multi-spectral channels in file */
      twobyte uchan;             /* number multi-spectral channels in use */
      twobyte crdate[2];         /* creation date */
      twobyte uid;               /* owner/creater's UID */
      twobyte modified;          /* flag -- file has a physical header */
      char title[P_TSIZE];       /* 40 character title for identif. */

      char descrip[P_DSIZE];     /* 128 chars of picture description info */

      twobyte extdesc;           /* flag for extended description */
      twobyte _filler_[32];      /* extra for header expansion */
      twobyte extra[128];        /* extra for "format specific" data */
};
```

Figure 8: PDS Header Format

or group set will destroy the grey-scale information present in the pixels on the boundary. Thus, clearing the overlay may require redisplaying the grey-scale image.

3.3. Coordinate System

The coordinate system used both by the user to input boundaries and by the system to report information back to the user is a rectangular coordinate system. The origin (i.e., (0,0)) is the upper lefthand pixel of the display on the graphics device. The x coordinate grows to the right. The y coordinate grows down. In general, when specifying the boundaries of regions, the x coordinate is given first. For example, when specifying the boundary of the root of a quad

tree, the specification (0,0,128,256) means that the boundary starts at position (0,0) (i.e., the upper lefthand corner of the display) and extends 128 pixels to the right and 256 pixels down.

3.4. Current Objects and States

Within the system, there are multiple instanciations of certain objects and multiple discrete states that certain variables can take on. In order to releave the user from continually specifying the data structure and/or state, the system incorporates the concepts of "current object" and "current state".

The user has three major system-defined data structures (or objects) that can be referenced directly: quad trees, group sets, and regions of interest. Quad trees are referenced by the reserved word: **qt**, **quadtree**, **tree**, or **root**. All of these terms or synonymous to the system. Group sets are referenced by the reserved word: **group_set**, **groupset**, **gs**, **groups**, **grpset**, or **grps**. Regions are referenced by the reserved word: **region** or **reg**. Each of these objects has 11 instanciations referencable by the user. Internally, these are organized as an array from 0 to 10, and the notation for referencing them is similar (eg., **qt**[3]). By default, referencing the reserved word without appending a subscript selects the 11th element (eg., **qt** references `qt[10]`). If a subscript is appended, the corresponding element is selected (eg., **qt**[0] references `qt[0]`). In this way, if the use of multiple objects is not required, the user need not address the fact that there are multiple instanciations available by supplying the subscript of the one desired.

When manipulating quad trees and group sets, the system treats them as objects and not as values. For example, if we want to move `qt[1]` to `qt[3]`, we might enter

qt[3] = qt[1]

However, after execution of this statement, there is no quad tree in `qt[1]`. (Actually, there is one -- the null tree.) The object that was in `qt[1]` has been moved to `qt[3]`. (If there was a

quad tree in `qt[3]`, it is removed and deallocated.) In other words, the object itself has been moved -- not a copy of the object. (The reason for this approach is that these data structures are very large and complicated; they may have specific properties attached to them. The copying of these data structures could be very waistful of time and space.)

However, when manipulating regions of interest, the system treats them as values and not as objects. Thus, a statement such as

region[5] = region[2]

does copy the value of `region[2]` into `region[5]`, but the value of `region[2]` remains unchanged. This is done because these data structures are very simple -- they are just a 4-tuple specifying the upper left corner and the extent of the region of interest.

Quad trees and group sets can be discarded (and deallocated) by assigning them the value **nil** or **null**, as in

qt[3] = nil

Quad trees and group sets can be displayed (on the graphics device), printed (on the terminal), and used implicitly or explicitly as input and output operands of appropriate operations. There is the concept of a "current object" for quad trees, group sets and regions of interest. If a specific instanciation is explicitly reference in a command, that object is made the current object (for subsequent implicit processing). For instance, if the command sequence

qt[3] = qt

merge on grey(60)

is given, the first command moves `qt[10]` to `qt[3]` (making `qt[3]` the current quad tree), and the second command does a merge on `qt[3]`, the new current quad tree.

There is a concept of a current image (`curr_img`). This is set by a **read** or **display** command. Once the current image has been established, all processing is done on that image until a new different current image is established by another **read** or **display** command.

There is the concept of a current overlay channel, which is used to indicate what color should be used when displaying the boundary of user manipulatable data structures. This is set by the **display** command, the **clear** command, and the explicit assignment of a value to the overlay channel variable. Once the current overlay channel is established, it remains in tact until explicitly changed. For instance, the command

clear(red) overlay

clears the red overlay channel (channel 0); a side affect is to set the current overlay channel to 0. This is equivalent to the two following commands.

overlay = 0
clear overlay

This current overlay channel will remain set until changed by some other command. For instance, a subsequent command, such as

display gs

would display the boundaries of `gs[10]` in red. However the command

display(green) gs

would display the boundary of `gs[10]` in the green overlay channel (channel 1); a side affect is to set the overlay channel to 1 for all subsequent displays (until changed).

3.5. Execution Directory Configuration

This subsection describes the directory configuration in which the SMGJ Segmentation System must reside at execution. Two directories must be present in the context of the execution directory: IMAGES and LOG. IMAGES must be a "brother" directory of the execution directory. This directory contains all the images that can be accessed by the system. LOG must be a "son" directory of the execution directory. This directory is used as a repository for a log of all commands entered by the user during each session. The directory configuration

is pictorially represented in Figure 9.

4. The User Interface Language

Internally the language is divided into a number of different categories of statements; these categories include:

- Help Facilities
- Setting system parameters, user referencable data structures, defining user referencable data structures, reading images, and reading and writing groups sets from and to files
- Printing data structures and system information on the terminal
- Displaying results on the graphics device

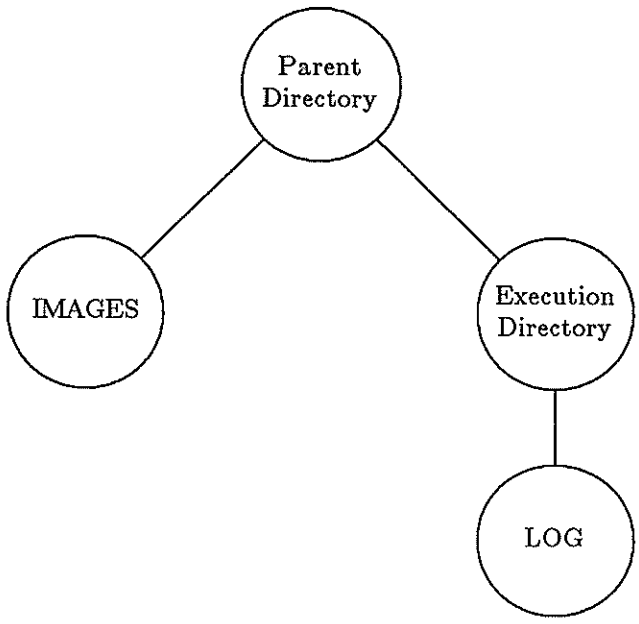

Figure 9: Execution Directory Configuration

- Clearing information from the graphics device
- Verb commands
- Utility commands

The description of the language here will be presented using this same organization. The exact syntax will not always be specified here completely. To obtain the exact syntax, the user can read the LEX and YACC programs that drive the system. The purpose here is to show the overall structure and capabilities of the language, and to give the user an overview of how to use the system through the use of the language. The following subsections describe the capabilities of language in each of the categories above.

4.1. Help Facilities

There is an on-line syntax help facility in the language. To access this facility, type **help** (this will give information on how to use help) or **help** followed by one of the following reserved words: **help, set, print, display, clear, command, or utility.**

- **help**

  Gives information about how to use the help facility.

- **set**

  Gives information about how to set system parameters, manipulate user referencable data structures, define user referencable data structures, read in images, and read and write group sets from and to files.

- **print**

  Gives information about how to print user referencable data structures, information about user defined processes (sorts, properties, and predicates), and information about system

parameters.

- **display**

  Gives information about how to display images and the boundaries of user referencable data structures.

- **clear**

  Give information about how to clear images and color overlay channels from the graphics device.

- **command**

  Gives information about how to execute verb commands (split, merge, etc.).

- **utility**

  Gives information about the utility functions.

4.2. Setting Things

The general syntax for setting things is:

```
    (display|show|time|bell|verbose)=(true|false)
    (debug|overlay|alloc_chunk'('{int}')')={int}
    <root>=(<root>|null|<glue>)
    <grpset>=(<grpset>|null)
    <region>=<region>
    define <root>
    define <root>'<'{int},{int},{int},{int}'>'
    define <region>
    define <region>'('{int},{int},{int},{int}')'
    read {file name root}
    write <grpset> {file name}
    read <grpset> {file name}
    <defs>
where:
    <root> → root['['0-9']']
    <grpset> → groupset['['0-9']']
    <region> → region['['0-9']']
    <glue> → glue'('<root>,<root>,<root>,<root>')'
    <defs> → '('{int},{int},{int},{int}')'
           | '<'{int},{int},{int},{int}'>'
```

4.2.1. System Parameters

There are a number of system parameters that the user can set. The user can indicate whether display activities should actually occur or not; this is usually a reflection of whether there is a graphics device currently available or not. This is done by one of the two commands below. The first causes display commands to be active; the second causes display commands to be inactive.

display = true
display = false

The user can request to have certain activity (caused by verb commands) displayed on the graphics device during the processing of the command. This is currently available only for the **split** verb and the **grow** verb. The activity is displayed in the overlay channel of the graphics device. This is requested by one of the two commands below. The first command turns on the show activity; the second turns it off.

show = true
show = false

The user can request timing information to be printed or not to be printed after certain activities. This is requested by one of the two commands below.

time = true
time = false

The user can request a bell to be sounded or not to be sounded after the completion of certain commands. This is useful for directing the user's attention back to the terminal after the lengthy execution of a command. This is requested by one of the two commands below.

bell = true
bell = false

The user can request that certain activities be reported or not reported at the terminal. This is requested by one of the two commands below.

verbose = true
verbose = false

The user may request debugging information about certain activities of the system. Such debugging can be incorporated into user installed code, such as property calculation code, predicated code, and sorting code. Debugging can be performed at several different levels, specified by an integer. The level of debugging is specified by the command

debug = {int}

where **{int}** specifies the level of debugging.

The user can specify the overlay channel (color) into which graphics information should be written. The overlay channel is used for displaying the boundaries of quad trees, group sets and regions of interest. The overlay channel is set by the command

overlay = {int}

where **{int}** is an integer between 0 and 7 indicating which overlay channel to use.

The system does dynamic allocation of all the major data structures. For efficiency purposes, a large block of memory is allocated whenever new memory is needed for a specific type of data structure. The user may want to control the size of how much memory is allocated in these large blocks. Each data structure is represented internally by an integer, unique to each data structure. (The integer associated with any specific data structure can be determined by doing a **print all** command.) The user can specify how much memory will be allocated for a specific data structure in subsequent allocations by the command

alloc_chunk({int1}) = {int2}

where **{int1}** indicates the data structure and **{int2}** indicates how much memory should be

allocated as a function of the size of the data structure itself.

4.2.2. Manipulating User Referencable Data Structures

The user can manipulate three different system maintained data structures: quad trees, group sets, and regions of interest. There are 11 instanciations of each of these data structures, and the user can transfer them by use of an assignment-like statement. The 11 quad trees are referenced by **qt[0]** through **qt[9]** (quad trees `qt[0]` through `qt[9]`) and **qt** (quad tree `qt[10]`). A similar notation is used for group sets (**gs[i]** and **gs**) and regions of interest (**region[i]** and **region**). Thus, the command

gs[3] = gs[8]

transfers the group set that was in `gs[8]` into `gs[3]`. If there was a non-null group set in `gs[3]`, it is removed and deallocated. After completion of this command, group set `gs[8]` is the null group set. Quad trees are handled in a similar manner. However, the assignment of regions of interest causes the values (not the objects) of the regions to be copied. Thus the command

region[1] = region[2]

causes `region[1]` to have the boundary that `region[2]` used to have, but the old content of `region[2]` is not modified.

Besides moving quad trees around, quad trees can also be "glued together" if their boundaries are compatible. This is accomplished by a command such as

qt = glue(qt[1],qt[2],qt[3],qt[4])

The **glue** routine will combine the constituent quad trees into a compound quad tree if they have the appropriate boundaries. The arguments to **glue** are assumed to be the north-west, north-east, south-west, south-east, respectively by order, components of the larger quad tree. If

they "fit" together, the operation is performed; if the do not "fit" together, an error message is given, and not action takes place.

4.2.3. Defining Roots and Regions

The boundary of regions of interest and roots of quad trees is specified by the **define** command. This can be done by specifying the coordinate information through the terminal, or by use of the track ball (or mouse) associated with the graphics device. The command

define qt[1](5,6,128,256)

defined the root of `qt[1]` to have its upper lefthand corner at position (5,6) (i.e., x=5, y=6) and to be of size 128×256 (i.e., it extends 128 pixels in the x direction and 256 pixels in the y direction). A similar command defines a region, replacing the reserved word **qt** with **region**. The command

define qt[1]

indicates that the boundary of `qt[1]` should be taken from the track ball (or mouse). The boundary is specified by track ball (or mouse) movement and button pushes. Follow the terminal screen and/or graphics device instructions given.

There is no way to define the boundary of a group or group set; this is always extracted from a quad tree or another group set.

4.2.4. Reading an Image from a File

All images read from external files must be in PDS format, and their names must have the extension ".pds". Thus the file `cell1.pds` might contain the image for cell1 held in PDS format. When specifying the name of a file, only the root of the name (e.g., **cell1**) is given. Thus the command

read cell1

reads an image from the file `cell1.pds` and place that image in the current internal image (`curr_img`) to be processed.

4.2.5. Writing and Reading Group Sets

A group set can be written to a file or read from a file. This is done by commands such as

write gs[1] gsfile
read gs[2] gsfile

The first command writes `gs[1]` to the file `gsfile`; a side affect is to set the current group set to `gs[1]`. The second command reads the file `gsfile` and places the corresponding group set in `gs[2]`; a side affect is to set the current group set to `gs[2]`.

4.3. Printing Things

The general syntax for printing things is:

```
    print (<root>|<grpset>|all|sorts|props|preds)
where:
    <root> → root['['0-9']']
    <grpset> → groupset['['0-9']']
```

There are a number of things that can be printed on the terminal. For instance, the command

print qt[3]

prints the structure of qt[3]. The recursive decomposition of the quad tree is represented by indentation. If any properties are attached to quad tree nodes, they are also printed.

The command

print gs[9]

prints the structure of `gs[9]`. This is presented in a linear manner, listing the components of each group of the group set. Again, if any properties are attached to the groups, they will also be printed.

Information about user installed routines can also be printed. For instance, the command

print sorts

prints all sorting routines that have been installed by the user and indicate the verbs with which they can be used. The command

print properties

prints all the user installed properties. The command

print predicates

prints all the user installed predicates and indicates, (a) which properties they are associated with, (b) what verbs they can be used with, and (c) the current values of any parameters.

The command

print all

prints all the system parameters (time, verbose, bell, etc.), the sort, properties, and predicates information, a table of statistics about system maintained data structures, information about the current user manipulatable data structures, and the total amount of memory that has been dynamically allocated.

4.4. Displaying Things

The general syntax for displaying things is:

```
    display ['('<color>')'] <root>
    display ['('<color>')'] <region>
    display ['('<color>')'] <grpset>
    display image [{file_name_root}]
    display {file_name_root}
where:
    <root> → root['['0-9']']
    <grpset> → groupset['['0-9']']
    <region> → region['['0-9']']
    <color> → red|green|blue|turquoise|purple|white|yellow|magenta|0-7
```

The user can display images and user manipulatable data structures on the graphics device. For instance, the command

display image

will display (or redisplay) an image which has been previously displayed or read in (by the **read** command). Either of the commands

display image {file_name}
display {file_name}

reads in a new current image from the file `file_name.pds` and displays it. This is equivalent to the following two commands.

read {file_name}
display image

User manipulatable data structures are displayed by commands like the following.

display(red) qt[3]
display(2) gs[2]
display region

The first displays the boundary of `qt[3]` in the red channel (channel 0); two side affects are to change the current overlay channel to 0 and the current quad tree to `qt[3]`. The second displays the boundaries of the groups in `gs[2]` in overlay channel 2 (blue); two side affects are to change the current overlay channel to 2 and the current group set to `gs[2]`. The third displays `region[10]` in whatever the current overlay channel happens to be. This may have

been set by other display commands, clear commands, or an assignment to the overlay variable. A side affect is to change the current region of interest to `region[10]`.

4.5. Clearing Things

The general syntax for clearing things is:

```
    clear image
    clear ['('(all|<color>)')'] overlay
where:
    <color> → red|green|blue|turquoise|purple|white|yellow|magenta|0-7
```

Images and overlay channels can be cleared from the graphics device. The command

clear image

clears the image from the screen but leaves the overlay channels in tact. The internal memory copy of the image is retained, so that a subsequent

display image

will redisplay the image on the graphics device. This ability to clear the image is useful for trying to visualize the original image given only the overlay channel information describing boundaries in the segmentation. The content of the overlay channels can be cleared by commands such as

clear(red) overlay
clear(2) overlay
clear(all) overlay
clear overlay

The first command clears only the red overlay channel (channel 0); a side affect is to set the current overlay channel to 0. The second command clears only overlay channel 2 (blue); a side affect is to set the current overlay channel to 2. The third command clears all eight of the overlay channels; there is no internal side affect. The fourth command clears the current overlay channel and does not change the current overlay channel.

4.6. Utilities

The general syntax for the utilities is:

```
time
input {file_name}
'!'{system command}
pause
zoom
sleep
```

The command

time

prints out at the terminal the user, system, and total cpu time since the process began and since the last invocation of **time**.

User interface language input can be redirected to come from a file. This is accomplished by the command

input {file_name}

When the content of the file `file_name` is exhaust, control returns to the terminal. Control can be explicitly returned to the terminal by the command

input terminal

The user can request that the system pause so that output on the screen can be viewed or photographs of the graphics device can be taken. This is accomplished by the command

pause

The system pauses until a nonblank character and a cariage return are entered at the terminal. This command is useful when executing commands redirected by the **input** command.

The user can zoom and roam through the display on the graphics device. This is accomplished by the command

zoom

Zooming and roaming is done through use of the track ball and buttons. Follow instructions presented on the graphics device. Different exits from this utility will allow subsequent displays to be performed while the graphics device is in a zoomed state.

The user can request that the system "go to sleep" until it receives a signal. This is accomplished by the command

sleep

This is useful for running jobs over night and putting the job into background mode at the end of a script (by use of the **input** command) so that the process can be brought back to forground mode later.

The user has the ability to perform UNIX system commands from within the system. This is accomplished by the usual "bang" notation of

!{system command}

4.7. Verb Commands

The general syntax for these verb commands is:

```
    split [<root>][[using]<region>][on]{predicated name}[<arg_list>]
                [redefine <defs>*]
    merge [<root>][[using]<region>][on]{predicated name}[<arg_list>]
                [redefine <defs>*]
    group [<root>][[using]<region>][on]{predicated name}[<arg_list>]
                [[with]{sort name}][redefine <defs>*]
    grow [<root>][[using]<region>][on]{predicated name}[<arg_list>]
                [[with]{sort name}][redefine <defs>*]
    join [<grpset>][[with]{sort name}][[using]<region>]
                [on]{predicated name}[<arg_list>]
                [[with]{sort name}][redefine <defs>*]
    regrow [<grpset>][[using]<region>][on]{predicated name}[<arg_list>]
                [[with]{sort name}][redefine <defs>*]
    strip [<root>][of]{property name}
where:
    <root> → root['['0-9']']
    <region> → region['['0-9']']
    <grpset> → groupset['['0-9']']
    <arg_list> → '('({real}|{int})(,({real}|{int}))*')'
    <defs> → '('{int},{int},{int},{int}')'
           | '<'{int},{int},{int},{int}'>'
```

There are four basic verb commands: **split, merge, group**, and **join**. These are the operations for which the system is named. Besides these, there are two auxiliary commands: **grow** and **regrow**. There is also a property manipulation verb: **strip**.

For a general description of the four basic verbs, see Section 1. The **grow** verb is a variation of the **group** verb which produces one and only one group given a seed quad tree node from which to start. The **grow** verb is associated with the graphics device specifically; the input quad tree seed node is input by use of the track ball (or mouse) and the resulting group is displayed in the overlay channel of the graphics device. The **regrow** verb is similar, but is a variation of the **join** verb. It takes a seed group and applies the **join** algorithm producing one and only one group as output; input and output are again via the graphics device. The **strip** verb allows the user to remove property information from quad trees. The purpose for this is to reclaim (deallocate) memory for subsequent allocation and use in processing.

In general, the optional [<root>] and [<grpset>] clauses allow the user to specify a specific quad tree or group set to be used; if the clause is not present the current quad tree or

current group set is used. The [[using]<region>] clause allows the user to specify a specific region of interest; if the clause is not present the current region of interest is used. The [on]{pred}[<arg_list>] clause specifies the predicate (along with parameters) to be used in performing the action of the verb. If the [<arg_list>] portion is not present, then the current values of the parameters are used; otherwise, the actual parameters are changed and used. It is not possible to omit a parameter; if one is given, they must all be given. The [redefine <def>*] clause allows the user the redefine the boundary of the root of the quad tree node or the boundary of the region of interest (as an after thought). The [[with]{sort}] clauses allow the user to control, to some degree, the order in which groups will be considered during the execution of the **group** verb and the **join** verb. A sort in front of the predicate specifies the order in which seed groups should be selected around which candidate groups will be considered for combination. A sort after the predicate specifies the order in which candidate nodes should be selected for potential combination with the (modified) seed node.

Note that the only non-optional clauses are the command verb name itself and the specification of the predicate; all other clauses help the user to refine the specific meaning or intent.

4.7.1. Split

The **split** verb performs the split operation as a function of the predicate specified. The conceptual input is a quad tree; the conceptual output is the same quad tree modified by the split operation (i.e., this is a descructive operation). Most of the time, the **split** verb is used to initially decompose a quad tree root down to a specific size level, using the size predicate. For instance, the command

split on size(1,1)

splits the current quad tree down to a 1×1 leaf size. This can be done when the quad tree is in any state whatsoever (e.g., after a merge operation has been performed). The command

split qt[3] using region[2] on grey(60) redefine (0,0,8,8)<0,0,16,16>

(a) makes `qt[3]` the current quad tree, (b) makes `region[2]` the current region, (c) redefines `region[2]` to have boundary specified by (0,0,8,8), (d) redefines `qt[3]` to have boundary specified by (0,0,16,16), and (e) splits the region of `qt[3]` which is inside `region[2]` so that each leaf of the resulting quad tree has pixels whose grey-scale levels differ by no more that 60.

4.7.2. Merge

The **merge** verb performs the `merge` operation as a function of the predicate specified. The conceptual input is a quad tree; the conceptual output is the same quad tree modified by the `merge` operation (i.e., this is a descructive operation). The **merge** verb is used to combine nodes whose parents meet the requirements of the predicate. For example the command

merge on grey(80)

would modify the current quad tree so that no non-leaf node of the modified quad tree has pixels all of which differ by less than 80 grey-scale levels.

4.7.3. Group

The **group** verb performs the `group` operation as a function of the predicate specified. The conceptual input is a quad tree; the conceptual output is a group set. The physical group set produced is placed in `gs[10]`. The **group** verb is used to combine leaf quad tree nodes into (potentially) non-rectangular groups (regions) which meet the requirements of the predicate. For instance, the command

group on grey(100) with smallest_first

takes the leaves of the current quad tree and combines them into groups all whose pixels differ by no more than 100 grey-scale levels. The **with** clause causes the leaf nodes of the neighbors of the current seed to be sorted in ascending order of size, so that the smallest leaf nodes will be considered for combination first. A candidate leaf node will be combined if its inclusion does not violate the predicate; it will not be combined if its inclusion does violate the predicate.

4.7.4. Grow

The **grow** verb is a variation of the **group** verb. However the **grow** verb applies the `group` operation to one and only one leaf of the quad tree, thus producing one and only one group. The conceptual input is a quad tree and a leaf of that quad tree; the conceptual output is a group. The purpose of this verb is to supply the user with a visual sense of how the `group` operation performs, given a specific predicate and quad tree. The seed quad tree leaf is specified by use of the track ball (or mouse) and the resulting group is displayed on the graphics device.

4.7.5. Join

The **join** verb performs the `join` operation as a function of the predicate specified. The conceptual input is a group set; the conceptual output is a group set. The physical group set produced is placed in `gs[10]`. The **join** verb is used to combine groups of the input group set into groups whose members satisfy the requirements of the predicate. The intent of the **join** verb is identical to that of the **group** verb except that the **join** verb takes a group set as input instead of a quad tree. For example, the command

join with biggest_first gs[0] on grey(120) with closest_grey

combines groups from the group set `gs[0]` so that all groups in the resulting group set have pixels whose grey-scale levels differ by no more than 120. In the process of selecting seeds for group combination, the largest unused group from the input group set is used first. Candidate

groups are selected on the basis of which are closest to the current seed group, as a function of the average grey-scale levels in the two groups. The resulting group set is placed in `gs[10]`.

4.7.6. Regrow

The **regrow** verb is a variation of the **join** verb. However the **regrow** verb applies the `join` operation to one and only one group of a group set, thus producing one and only one group. The conceptual input is a group set and a group in that group set; the conceptual output is a group. The purpose of this verb is to supply the user with a visual sense of how the `join` operation performs, given a specific predicate and group set. The seed group is specified by the track ball (or mouse) and the resulting group is displayed on the graphics device.

4.7.7. Strip

The **strip** verb allows the user to remove properties from a quad tree so that the memory associated with those properties can be deallocated for use in subsequent processing. Thus, the command

strip qt[3] of grey_prop

leaves the quad tree structure of `qt[3]` alone, but removes all grey-scale property nodes from the tree and deallocates the memory for subsequent use.

4.8. Reserved Words and Synonyms

There are a large number of reserved words in the interface language. Those that have synonyms are presented in Table 1. Other reserved words which have no synonyms are: **join, grow, regrow, strip, write, using, on, with, of, all, sorts, time input, image, glue, pause, sleep, bell, verbose, alloc_chunk, red, green, blue, turquoise, purple, white, yellow**, and **magenta**. None of the reserved words can be used in any other context. For instance, they

Table 1
Reserved Words and Their Synonyms

Word	Synonyms
quit	q, .
debug	db, dbg
define	df, dfn, def
redefine	redefining, redef
region	reg
qt	quadtree, tree, root
gs	groupset, group_set, groups, grpset, grps
split	splt
merge	mrg
group	grp
print	prnt
display	disp
clear	clr
read	rd
properties	props
predicates	preds
terminal	term
true	t, T
false	f, false
null	nil, nill, NULL, NIL, NILL
help	h, ?
set	sets
command	commands
utility	util, utilities
zoom	zr

cannot be used as predicate names, property name, sort names, or file names. However, they can be used as components of a UNIX system command escape.

5. Segmentation Paradigm

A *strategy* specifies a general approach to solving a problem. It indicates at the topmost level what kind of activities can occur during the solution of a problem. A *methodology* specifies a more concrete refinement of a strategy. It specifies a generalized flow of the activities that can occur, given a specific strategy as a guideline. An *algorithm* specifies an exact sequence of specific algorithmic components that can be applied during the solution of a problem, given a specific methodology. These three concepts constitute a heirarchy for making design decisions during the process of attempting to discover appropriate solutions to a given problem.

In the following subsections, different strategies, methodologies and algorithms will be discussed. We will attempt to show how the SMGJ Segmentation System can be used in different situation, and indicate how we have chosen to use it.

5.1. Strategies

One strategy for segmenting images is to attempt to produce the complete correct final segmentation by making one and only one "pass" over the image. The segmentation criteria might be a very simple one, such as grey-scale alone, or an extremely complex one, possibly combining grey-scale, texture and other special criteria. In general, it has been found that such an approach is not very successful, because, (a) if the criteria is simple, it is not sophisicated enough to do a good segmentation on all regions of the image, and (b) if the criteria is complex, it is dufficult to find an appropriate coordination of the component criteria that does a good segmentation job. Thus, the single pass strategy seems to be relatively unproductive.

The SMGJ Segmentation System is caple of addressing strategies like this one. Extremely complicated properties and predicates can be implemented and installed. However the complexity of producing, debugging and analyzing them is high compared to a more linearly distributed approach.

A second strategy is to make multiple passes over the image, refining the previous segmentation to produce a subsequent segmentation. In this strategy, each pass combines and/or separates regions present in the previous segmentation, based on a specific (simple) segmentation criteria applicable during the current pass. This strategy allows a layered approach in which intermediate results can be viewed and analyzed, thus allowing the selection of appropriate processing passes. This tends to reduce the conceptual complexity of searching for appropriate sequences of segmentation criteria.

The SMGJ Segmentation System is well suited for this kind of strategy and by design encourages its use. This is the strategy which we have used in the development of the properties and predicates which we have currently installed in the system.

5.2. Methodologies

Given a strategy in which different segmentation criteria will be applied over multiple passes, there are at least two major methodologies that might be employed. One is to undersegment the image originally, and continually separate regions during subsequent passes. Another is to oversegment the image originally and continually combine regions in subsequent passes. (Of course, there are other variations in the spectrum between these two approaches which might be employed.)

We have chosen to adopt the oversegmentation methodology in the development of the properties/predicates that we currently have installed. However, the system is capable of employing the undersegmentation methodology in the relm of quad trees (using only the **split** and **merge** verbs), and it would be possible to extend this into the group set relm with only minor modifications to the system. Specifically, the region of interest controls what portion of the image will be affected by verb commands. Currently the region of interest can only be rectangular and is defined by the user externally, independent of any other data structure (i.e.,

quad trees and group sets) available. However, if the region of interest were to be redefined to be any arbitrary set of groups in a group set (instead of the current rectangular region), then it would be possible to selectively process undersegmented portions of a segmentation. We have plans to incorporate this facility, but have refrained for performance reasons.

5.3. Algorithms

Given the oversegmentation methodology discussed above, the process of discovering appropriate segmentation algorithms becomes one of selecting specific separating/combining actions using properties/predicates (with appropriate parameters) which originally oversegment the image and then successively combine portions of the image without producing undersegmentation. Originally, the search is done on one specific image selected from a class of images. Once an appropriate sequence of actions and set of parameters has been found that works well for the selected image, the computational scenario (algorithm) can be applied to other images in the class. Small adjustments can be made to the parameters to enhance the overall segmentation quality over the entire class of images.

For instance, consider the simple algorithm presented in Figure 10. In this computational scenario, the display device is turned on and a 512×512 image (read from the file `wash.pds`) is read in and displayed. The root node for the quad tree is defined to cover the upper left hand

```
display = t
read wash
display image
define root(0,0,256,256)
split on size(2,2)
merge on grey(50)
group on grey(70)
join on grey(90)
join on small(50)
display(red) gs
```

Figure 10: A Simple Grey-Scale Segmentation Algorithm

quarter (256×256) of the image; this portion of the image is separated (split) into 2×2 squares in the quad tree. The nodes of the (split) quad tree are combined (merged) as much as possible under the constraint that no leaf quad tree node contains pixels differing in grey-scale by more that 50. Nodes from this (merged) quad tree are then grouped together as much as possible under the constraint that no group contains pixels differing in grey-scale by more that 70. This produces a set of groups whose union covers the entire 256×256 portion of the image. Groups from this group set are then combined (joined) as much as possible under the constraint that no group contains pixels differing in grey-scale by more that 90. Groups from this (new) group set are then combined (joined) so that no group containing 50 or less pixels is left uncombined.

5.4. Approaches

There are a number of approaches that might be used in searching for appropriate algorithms that are effective over a large class of images. For instance, assumming that the computational scenario (eg., that of Figure 10) performs well on a particular image in a class of images, it may be considered an algorithm for segmenting that class of images. However, it is possible that such a scenario works well on one image of a class, but does not work well on all members of the class. One way to enhance the effectiveness on the entire class of images is to adjust the parameters slightly to increase the average performance over the entire class of images. Another way may be to produce slightly more sophisticated scenarios using other (possibly combined) properties/predicates or using heuristics for selecting how groups in group sets are combined.

As an example of changing parameters, the algorithm in Figure 10 might be changed in several ways. For instance, splitting to a size of 1×1 can significantly affect the quality of the resulting segmentation. This is especially true for images which have narrow features, such as shadows and walkways. Another important parameter to consider adjusting is the grey-scale range associated with the **grey** predicate in the **merge**, **group**, and **join** verbs. Raising or

lowering the value of this parameter may not significantly affect the segmentation of the original image, but may significantly enhance the segmentation of the other images in the class.

A variation of the parameter adjustment approach can also be accomplished by applying a *staged* approach, in which multiple operations are sequencially performed varying the sensitivity parameter slowly (instead of "jumping" to the final value). For example, the three scenarios

```
split on size(2,2)
group on grey(90)
```

and

```
split on size(2,2)
merge on grey(50)
join on grey(70)
join on grey(90)
```

and

```
split on size(2,2)
merge on grey(40)
group on grey(50)
join on grey(60)
join on grey(70)
join on grey(80)
join on grey(90)
```

might be perceived to produce equivalent segmentations. They, in fact, do not. The difference has to do with "bleeding" of regions given "raw data". The first scenario tends to produce bad segmentations having odd shapes that do not correspond very well to features present in the image. The problem lies in the fact that there is no initial discrimination of basic components to be used for building blocks; all elementary regions (2×2 regions) have a very similar status. If, on the other hand, the processing is staged (as in the second and third scenarios), "bleeding" is reduced by giving operations more appropriate, well-developed building blocks. The higher the degree of staging, the better the final segmentation. The penalty, of course, is one of performance, since more operations must be performed.

```
split on size(1,1)
merge on grey(50)
group on grey(70)
join with biggest_first on grey(90) with closest_grey
join with biggest_first on mmagrey(110,55) with closest_grey
join with smallest_first on small(50) with closest_grey
display(blue) gs
```

Figure 11: A More Sophisticated Algorithm

As examples of using more sophisticated properties/predicates and heuristics, two scenarios will be presented. The first, shown in Figure 11, uses the sort option to select more appropriate seed and candidate groups. The **biggest_first** sort selects as initial seeds groups that are largest; the **smallest_first** sort has the reverse affect. The **closest_grey** sort selects as initial candidates groups that are closest in average grey-scale to the current seed group. A new compound predicate, **mmagrey** (min-max-average grey), is introduced in this scenario. In this case, groups are combined only if (a) the combination produces a group whose range of grey-scale levels differ by no more than 110, and (b) the average grey-scale of the individual groups differs by no more than 55. This tends to relax the constraint of 90 on the **grey** predicate while still requiring that the separate groups being combined have similar average grey-scale properties.

The scenario in Figure 12 introduces texture (instead of average grey-scale) as a secondary criteria for combining groups. Here, the predicate **greyltem** combines grey-scale information and the texture information associated with Laws' Texture Energy Measure. In this case, groups are combined only if (a) the combination produces a group whose range of grey-scale

```
split on size(1,1)
merge on grey(50)
group on grey(70)
join with biggest_first on grey(90) with closest_grey
join with biggest_first on greyltem(150,100) with closest_grey
join with smallest_first on small(50) with closest_grey
display(green) gs
```

Figure 12: Use of a Compound Predicate

levels differ by no more than 150, and (b) the length of the vector (in texture space) separating the two individual groups does not exceed 100.

6. Current Properties, Predicates and Sorts Installed

There are a number of properties, predicates and sorts currently installed in the SMGJ Segmentation System. This will, of course, increase as the system expands. In order to determine the installation status of a particular instanciation of the SMGJ Segmentation System, the following three commands can be used:

print properties
print predicates
print sorts

Properties and predicates are closely linked in the internal C code. There is a one to many mapping between the properties and predicates, i.e., a property may be associated with many predicates, but a predicate is associated with only one property. When a predicate is invoked to determine if a given region (quad tree node or group) satisfies the predicate, the data structure associated with that region is checked to determine whether or not the associated property has been calculated and attached to the data structure. If is is present, no calculation is performed; the property (which is already present and available) is accessed and used to determine the outcome of the predicate for that region. If the property is not present, the property is calculated, attached to the corresponding data structure and then used to determine the outcome of the predicate. Using this paradigm, properties of a region need be calculated no more than once, even though the region may be addressed multiple times by the processing of **split**, **merge**, **group**, and **join** verbs. This is a standard time/space tradeoff.

Property names are referenced only in the **strip** verb. Predicate names are reference only in the **split**, **merge**, **group**, and **join** verbs. Sorts are based on properties. They are reference

only in the **with** clauses of the **split**, **merge**, **group**, and **join** verbs.

6.1. Properties

The current properties installed are presented in Table 2. The names presented here are the formal names to be typed by the user through the interface language.

The **grey_prop** property contains four conceptual values: the minimum grey-scale value for all pixels found in the region, the maximum grey-scale value for all pixels found in the region, the number of pixels found in the region, and the sum of the grey-scale values for all pixels in the region. Thus, predicates based on min-max grey-scale (**grey**) and average grey-scale (**agrey**) can be efficiently evaluated. A fifth actual (Boolean) values is actually present; it is used with the **agrey** predicate.

The **size_prop** property contains two values: the x extent of the rectangular region (quad tree node), and the y extent of the rectangular region (quad tree node). This property is only used for the **size** predicate which is only available for the **split** and **merge** verbs.

The **ltem_prop** property contains 30 values. This property is based on a 15-dimensional texture space of the image. 15 of the values are the minimums in the separate dimensions; the other 15 values are the maximums in the separate dimensions.

Property Name
grey_prop
size_prop
ltem_prop
cooccur_prop
small_prop
greycooccur_prop
greyltem_prop

Table 2
Current Properties Installed

The **cooccur_prop** property contains the cooccurrence matrix for the particular region.

The **small_prop** property contains one Boolean value indicating whether or not the size of a region (group) is smaller than a given (parametric) value.

The **greycooccur_prop** property and **greyltem_prop** properties are actually pseudo-properties. They cause elementary properties (**grey_prop**, **cooccur_prop**, and **ltem_prop**) to be calculated and attached to the appropriate data structure.

6.2. Predicates

The current predicates installed are presented in Table 3. It is difficult to describe the exact semantics of these predicates in a few English statements. The descriptions presented here are intended to supply the major intent of the predicate. In order to understand the intricacies of the details, the C code must be read.

Predicate Name	Associated Property	Parameters
grey	grey_prop	delta_grey
agrey	grey_prop	delta_agrey
mmagrey	grey_prop	delta_grey delta_agrey
size	size_prop	delta_xsize delta_ysize
ltem_all	ltem_prop	delta_ltem
ltem_some	ltem_prop	delta_ltem
ltem_vect	ltem_prop	delta_ltem
small	small_prop	small_group
co_occur_pc	greycooccur_prop	co_occur threshold
co_occur_dbn	greycooccur_prop	co_occur threshold
greyltem	greyltem_prop	delta_grey delta_ltem
greycooccur	greycooccur_prop	delta_grey co_occur threshold

Table 3
Current Predicates Installed

The **grey** predicate determines whether or not the difference between the minimum and maximum grey-scale values are within **delta_grey** of one another. The **agrey** predicate is available only for the **join** verb. It determines whether or not the two component groups (potentially being **join**ed) have average grey-scales which differ by no more than **delta_agrey**. The **mmagrey** predicates is a compound predicate available only for the **join** verb. It determine whether or not the aggregate of the two component groups would have minimum and maximum grey-scale values within **delta_grey** of one another, and simultaneously whether or not the separate component groups have average grey-scales which differ by no more than **delta_agrey**.

The **size** predicate determines whether or not the x extend of the quad tree node exceeds **delta_xsize** and the y extent of the quad tree node exceeds **delta_ysize**. This predicate is available only for the **split** and **merge** verbs.

The **ltem_all** predicate determines whether or not *all* of the min-max differences in the 15-dimensional space differ by less than **delta_ltem**. The **ltem_most** predicate determines whether or not *most* (8 or more) of the min-max differences in the 15-dimensional space differ by less than **delta_ltem**. The **ltem_vect** predicate determines whether or not the min-max differences in the 15-dimensional space (when considered as a *vector*) has length less than **delta_ltem**.

The **small** predicate determines whether or not the number of pixels in the group exceeds **small_group**.

The **co_occur_pc** and **co_occur_dbn** predicates determine whether or not certain second order statistics of the cooccurrence matrix exceed **co_occur threshold**. These predicates address different second order statistics.

The **greyItem** predicate is a compound predicate which combines (*ands*) the **grey** predicated and the **Item_most** predicates. the **greycooccur** predicate is a compound predicate which combines (*ands*) the **grey** predicate and the **co_occur_pc** predicates.

6.3. Sorts

The currently installed sorts are presented in Table 4. The default sort is the **biggest_first** sort; if no **with** clause is present, this is the sort that is used.

The **biggest_first** sort sorts regions in descending order by number of pixels present in the region. The **smallest_first** sort sorts regions in ascending order by number of pixels present in the region.

The **closest_grey** sort sorts groups in ascending order by the absolute value of the difference in average grey-scale between the seed group and the candidate group.

Sort Name
biggest_first
smallest_first
closest_grey

Table 4
Currently Installed Sorts