Contents

Figures and Tables

Macintosh®

Virtual User Language Reference

DRAFT VERSION - 10/7/24

Chapter 1 Introduction

You are about to be introduced to a new language that has been designed particularly for specifying user interface test suites. Virtual User interprets scripts written in this language to run the tests.

Introduction

The Virtual User™ (VU) scripting language is at the core of the VU testing environment. VU can interpret scripts and execute the test described by the script remotely across AppleTalk™. This document contains a description of the language. Appendix A contains the grammar for the language.

A script is a set of statements that describes interaction between the Virtual User and the application under test (often referred to as the target application). Commands are the statements that specify the actual "virtual using". VU supports commands to do the following:

select menus	select menu items	select windows
close windows	zoom windows	size windows
drag windows	select buttons	select radio buttons
select check boxes	select controls	scroll scrollbars
type at the keyboard	press a key	release a key
move the mouse	press mouse button	release mouse button

An example of a script of two commands is shown below:

```
select [menuItem title:'Key Caps' menu:1];
type keystrokes: { "hello world", returnKey };
```

This script would direct the Virtual User to select the Key Caps item from the first menu and type the string, "hello world" followed by a strike of the return key. (This is assuming that the application under test includes the apple menu as the first one on the menubar and Key Caps is installed.) The components of this script will be described in the appropriate sections.

In addition to commands, high level programming constructs are provided to provide control over the execution of commands. At the present time, VU supports if-else conditionals, for loops, for each loops (to iterate over a list), and while loops. Scripts can be written without worrying about case. The interpreter is case insensitive. It is possible to make the matching of regular expressions to strings case sensitive using one of the command line options.

Chapter

2 Value Types and Expressions

Expressions are forms that combine variables and constants using operators. Expressions are evaluated to form values. This chapter first introduces you to the various value types available in this language and then describes the different kinds of expressions that can be formulated in your scripts.

Value Types

Before we discuss expressions, it is important to know the basic value types that can exist during a VU driven test. Statements contain expressions that are evaluated when a script is executed. The Virtual User reads an expression and determines what its value should be at that point during the test. Expressions will always evaluate to one of the following value types:

1. Numeric Value
2. String/Regular Expression
3. Descriptor
4. List of Values
5. Symbolic Value

Most of these are common to familiar programming languages with descriptors being an exception and symbolic values (or symbols) a less frequently seen value type. Descriptors will be described in detail in a later section. Basically they are descriptions of things that the Virtual User understands. The most common descriptors will contain information describing user interface items like windows, menu items, buttons, etc. Symbolic values have meaning in and of themselves. For example, each of the keys on a keyboard have a corresponding symbol such as **returnKey**, **enterKey**, **optionKey**, and **commandKey**.

In VU scripts, there are no type declarations. Values materialize from the evaluation of expressions or through unification. Expressions will be discussed in the next section. Unification, in short, is the binding of a variable to some value that exists in the system as a side effect of the evaluation of certain expressions. More often than not, unification will be used to find out information about a particular trait of a user interface element (e.g. the bounding rectangle of a window). Unifications are specified within descriptors and will be discussed in the section on descriptor expressions.

Boolean values exist implicitly. Any value can be truth tested. All descriptors are considered true except the null descriptor, [ ]. The empty list, { }, is considered false while all other lists are true. All numeric values are considered true except 0. All regular expressions are considered true except " (two single quotes), "" (two double quotes), and // (two slashes). The symbolic values, **false** and **undefined**, are false while all others are true. **true** is also a symbol which evaluates to itself.

Expressions

Statements contain expressions that are evaluated when the statement is interpreted. Expressions can be classified into the following categories:

- Variables
- Numbers
- Symbolic Identifiers
- Strings/Regular Expressions and those that manipulate them
- Arithmetic Operations
- Those that create, access, or manipulate lists when evaluated
- Those that evaluate to descriptors or access information from descriptors
- Relational Expressions
- Logical Expressions

Variables

Variables are named containers for values. Variable names must begin with a letter or underscore, after which any number of letters, digits and underscores may occur. Variables may get their values in three different ways. Variables can obtain values through assignment statements, parameter passing (a formal parameter to a task), or unification. Parameter passing is discussed in the section on tasks. Variables evaluate to whatever value they contain and variables that have not been given a value will evaluate to **undefined**.

Numbers

What's to say about numbers. Numbers permitted in this language are integers in the range from -32768 to 32767. When VU evaluates an out of range number, it flags an error and returns **undefined**.

Strings and Regular Expressions

Strings and regular expressions[1] (hereafter jointly referred to as regular expressions since strings are a particular case of regular expressions) use the same syntax as the MPW search tool. Meta-characters within a regular expression (e.g. ≈) are only applied while specifying traits in descriptor expressions. Descriptors will be described in their own section.

Within strings or regular expressions delimited by either " or /, a variable may be embedded if delimited by curly braces ({ }). When the regular expression is evaluated, the string representation of the variable's binding is embedded at that position in the string resulting from evaluation. This mechanism is currently only valid for variables bound to regular expressions and numbers. If the variable is unbound or bound to a value of the wrong type, no string embedding is done and a warning is issued. For example, the following script fragment illustrates the use of variables within regular expressions.

```
x := 4;
y := "VU 1.0b{x}";
z := /{x}{y}/;
```

Given that the above three statements have been executed, the following holds:

x would evaluate to 4
y would evaluate to "VU 1.0b4"
z would evaluate to /4VU 1.0b4/

The unary prefix operator, **card**, may be applied to a string/regular expression and the resulting expression will evaluate to the number of characters contained in the evaluated result. ∂n (newline) and ∂t (tab) count as 1 character within strings/regular expressions delimited by " or /. Knowing the length of a string is useful in cases such as when you want to know the length of some text that appears in an editable text box.

card "hello"	would evaluate to	5
card "hello world∂n"	would evaluate to	12
card /hello world≈/	would evaluate to	12

To reference a particular character within a string, the [ ] operator may be used. As with lists, an expression that evaluates to a numeric value (call it n) must be contained within the brackets. The expression will then evaluate to the nth-element of the string.

Example 1

"abcdef"[4] would evaluate to 'd'

Example 2

Assume the following relations are true:

[1] See the section on Regular Expression Matching for a complete definition of regular expressions

```
x = "window-1"
y = 5
```

then,

x[y] would evalute to 'o'

In addition to being able to concatenate strings using the MPW notation of embedding variables within strings using curly braces (e.g. “{x}{y}” as above), one can concatenate two strings using the binary operator, +. For example:

“first string” + “second string” would evaluate to “first stringsecond string”

Symbolic Identifiers

Symbolic identifiers are identifiers that have a reserved meaning for specific contexts of use. Symbolic identifiers are not to be confused with reserved words (e.g. **for**, **while**, **each**, **window**, **menu**, **in**) or variables which hold values. Examples of symbolic identifiers include **true**, **false**, and **returnKey**. There is a complete list of symbols in Appendix D. These special identifiers evaluate to themselves and should make sense in the context of use. The short script given at the beginning of this document illustrated a command that directs the Virtual User to do some typing. That command is shown below. This command shows the symbolic identifier, **returnKey**, being used to specify that the return key should be typed after the characters in the string, "hello world" have been typed.

```
Type k: { "hello world",returnKey };
```

Arithmetic Expressions

The following standard arithmetic binary arithmetic operators are supported:

+ - * / mod

along with the unary arithmetic operators:

+ -

Since only integer numbers are supported, / is naturally integer division. Please refer to the precedence chart in appendix C.

Lists

Lists are specified by enclosing any number of expressions, separated by commas in curly braces ({ }). Lists can be heterogeneous meaning that the expressions need not evaluate to values of the same type.

The list,

{ 1, "focus", true, 5 + 6 / 2, ima_variable, [window o:4 + 3] }

will evaluate to:

{ 1, "focus", true, 8, undefined, [window o:7] }

(assuming the variable, ima_variable, is unbound when the list is evaluated).

List Operators

Some list manipulation operations are supported through operators in the language. These operations are accessing individual elements of a list, counting the number of elements in a list, and concatenating two lists together.

An expression that evaluates to a list can be followed by a suffix of the form, [exp] where exp is some expression that evaluates to some positive numeric value call it n. The resulting expression will evaluate to the n-th element of the list. The following examples should illustrate the use of subscripting:

{"focus", 3, false, "vision", "passion"}[4] would evaluate to "vision"

```
x := {5,4,3,2,1}
```

x[2] would evaluate to 4

card is a unary prefix operator that, when applied to a list, evaluates to the number of elements in the list. For example:

card { a,b,c,d } would evaluate to 4

card { { 1,2,3 }, { 7,a,[window] }, false } would evaluate to 3

New lists can be built by combining existing lists. To specify the concatenation of two lists, the binary operator, +, should be used. Take the following script:

```
x := { 1,2,3 };
y := { 4,5,6,7,8 };
```

```
println x + y;
```

This script would produce the following output:

```
{ 1,2,3,4,5,6,7,8 }
```

Note that this list ceases to exist after println finishes with it. Both x and y retain their list values. The result of a + on two lists is to create a new list with copies of both component lists. Naturally, this result may be assigned to a different variable which would not destroy neither of x or y's bindings.

Descriptor Expressions

Descriptor expressions are the means by which a script author describes something that might exist during test execution. The most common use of descriptors will be to describe user interface items in the application under test such as windows, menus, and buttons. Using descriptors, the script writer can describe something to whatever degree of detail is needed to distinguish the object from others of the same type. For example, it may be enough to describe a window by giving only a title if the application names all of its windows uniquely.

User interface items are not the only things described by way of descriptors. VU, for example, allows time descriptors which can be used to specify a time to be used as a time of a day or duration of time depending on the context of use.

Descriptors are delimited by brackets ([ ]) and the descriptor type is located inside the left bracket. A descriptor type can be any of the following:

actor	application	button	checkBox
contentItem	control	editText	icon
keyboard	menu	menuItem	mouse
picture	popup	radioButton	screen
scrollBar	staticText	system	target
time	userItem	window	

After the descriptor type and before the closing right bracket, any number of traits may be given for a descriptor (including zero). The basic structure of a descriptor is shown below:

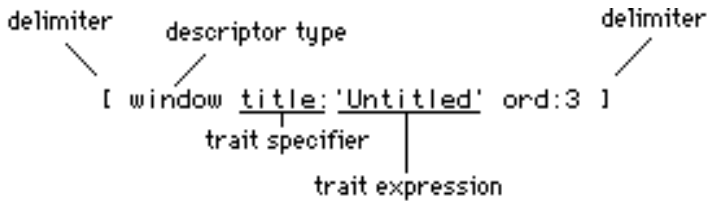
```
[ descriptor-type     trait1   trait2   etc… ]
```

Traits are things like the title of a window, a menu item's state of enablement, the thumb position of a scroll bar, etc…

An example of a window descriptor follows:

[window title:'Untitled' ord: 3]

The following diagram shows the various components of this descriptor:



This descriptor expression describes a window with "Untitled" as its title and is the third visible one in an application's window list. The trait specifiers **title:** and **ord:** are the first components of their respective traits. The interpreter is only concerned with the first letter of a trait specifier and the colon. All characters in between the first letter and the colon are ignored by the interpreter. Additional letters are allowed for script readability purposes. The following descriptors are equivalent to the one given above:

[window t:'Untitled' o:3]

[window the_title:'Untitled' ordinal_position_in_window_list:3]

Each of the descriptor types along with their allowable traits will be presented in a separate section. Some basic conventions are used to help script authors remember the various trait specifiers. In describing anything by its primary textual trait, **t:** for text is used. This includes things like window titles, button text, radio button text, etc… Most user interface items have a rectangle. For these traits, the trait specifier is **r:**. To specify the rank of something, **o:** (meaning ordinality) is used. Examples of **o:** traits are the rank of a window in the window list, the rank of a menu item within a menu, and the rank of a menu in a menubar. Here are some examples of descriptors:

```
[window]
[window t:'Untitled' o:1 style:document]
[menuItem]
[menuItem title:'New' m:'File']
[menu]
[menu title:'File' i: { [menuItem t:'New'],[menuItem t:'Open'],[menuItem t:'Close'],[menuItem t:'Quit'] }]
[menu title:'File' ord:2]
[menuItem  t:’hierarchical item’
            m:[menuItem t:’I have a hier menu attached m:[menu t:’menubar menu’]]]
[button t:'OK']
[button t:'OK' window_owner:[window style:dialog ord:1]]
[checkBox title:'Open at Startup']
[radioButton title:'Use All']
[time year:1989 month:4 day:15 hours:605 secs:34]
```

Descriptors are descriptions of things that might exist during a test. Matching is the process used by the Virtual User to find the thing being described by way of descriptor. Most often, matching involves finding windows, menus, menu items, and the like in a target application.

Matching is invoked in one of two ways. All descriptors that appear as operands to commands automatically get matched as part of command execution. For example, the command shown below directs the Virtual User to select a menu item. The menu item to be selected is described by way of descriptor as having "Quit" as its item name. Before the Virtual User may select this item, it must find where the item is located. This match as part of command execution is often referred to as an implicit match.

```
select [menuItem t:"Quit"];
```

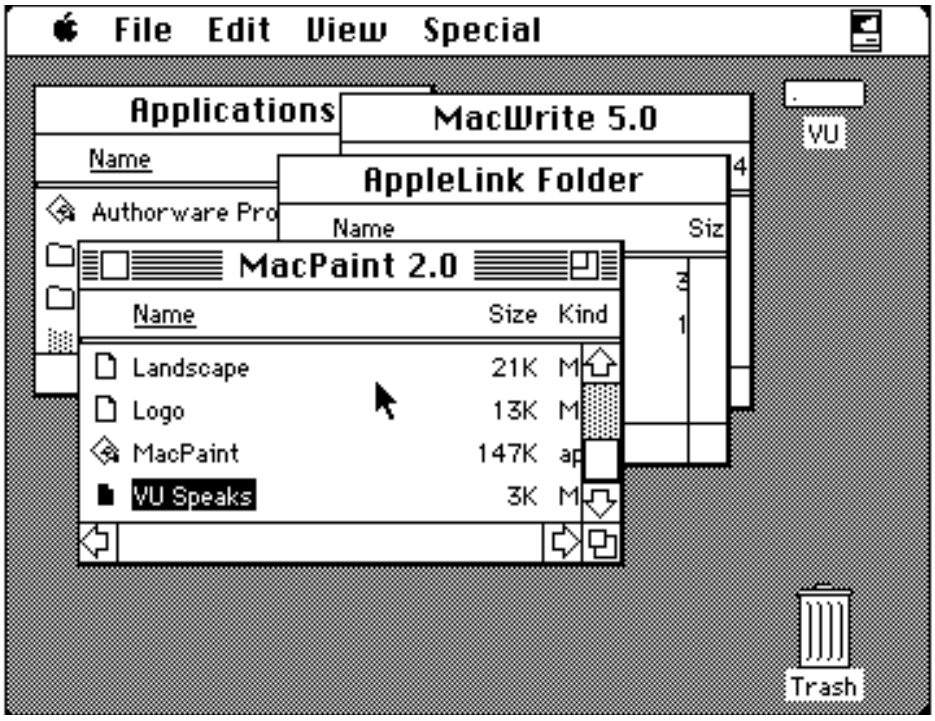
Sometimes, a script author will want to use the Virtual User's spying capabilities without actually doing anything like selecting menus or closing windows. An expression may be formed to cause matching to occur as part of that expression's evaluation process. There are two types of such expressions, match expressions and collect expressions.

Match Expressions

Match expressions are formed by applying the match operator (a unary prefix operator) to an expression that will evaluate to a descriptor when evaluated (the operand to the match). Evaluation of a match expression proceeds by first evaluating the operand and applying the match process to the resulting descriptor. The resulting value is a descriptor with all of its traits filled in. This "completed" or "fleshed out" descriptor is a description of what was actually found in the target application or wherever the matcher looked to find it. So there are two classes of descriptors, those that are placed in a script by a script author and those that are generated by the Virtual User as the result of matching a descriptor.

Let's assume we want to look for windows in the target application represented as a screen shot below. We'll use VU's script directed output facilities to show some examples of match expressions. In the following examples, the output has been formatted for this document.

Figure 2-1 Example on Match Expressions



The following statement says to print the result of matching a window whose rank is 1, namely the front window:

```
println match[window o:1];
```

The printed result of this statement is shown below. The result is a completed descriptor describing the actual window found in the target application. The various traits shown for this window descriptor are described in a separate section on window descriptors. Some of the basic traits include the window's title (t:'MacPaint 2.0') and its rank (o:1)

```
[window
    t:'MacPaint 2.0'
    s:document
    o:1
    c:true
    z:true
    g:true
    r: { 87, 116, 297, 254 }
    k: { [scrollBar t:'' r: { 87, 237, 281, 253 } s:{ 0, 0 } e:true ],
        [scrollBar t:'' r: { 280, 154, 296, 238 } s:{ 0, 0 } e:true ] }]
```

We could have matched the window by its title as the following example shows:

```
println match[window t:'MacPaint 2.0'];
```

The result of this statement is the same as the output from the previous example.

If more than one thing matches the descriptor, only one is arbitrarily chosen to be the result of the match as the following example shows. Here we are saying that we'd like to match a window whose title begins with "Mac" and ends with something we're not sure about (using a MPW style regular expression). Looking at the target application illustrated above, we see that there are two windows that could match this description. One has the title, "MacPaint 2.0", and the other has the title, "MacWrite 5.0". The Virtual User chose the one titled "MacPaint 2.0". The important thing to remember is that match expressions only generate one descriptor.

```
println match[window t:/Mac≈/];
```

```
[window  t:'MacPaint 2.0'
         s:document o:1 c:true z:true g:true
         r: { 87, 116, 297, 254 }
         k:{  [scrollBar t:'' r: { 87, 237, 281, 253 } s:{ 0, 0 } e:true ],
              [scrollBar t:'' r: { 280, 154, 296, 238 } s:{ 0, 0 } e:true ]  } ]
```

Collect Expressions

To generate all descriptors that match a certain descriptor, one can form a collect expression in the same manner as a match expression by substituting **collect** for **match**. A collect expression evaluates to a list of descriptors such that each descriptor in the list matches the descriptor given as the operand to the collect operator.

Collect expressions come in handy when you'd like to grab lists of things that change dynamically, like a Windows menu that many applications have.

The following specifies a collection of all windows. As a result of executing this statement against the target application we're using as an example, the list of 4 descriptors shown below would be printed.

```
println collect[window];
```

```
{    [window t:'MacPaint 2.0' s:document o:1 c:true z:true g:true
              r: { 87, 116, 297, 254 }
              k:{  [scrollBar t:'' r: { 87, 237, 281, 253 } s:{ 0, 0 } e:true ],
                   [scrollBar t:'' r: { 280, 154, 296, 238 } s:{ 0, 0 } e:true ] }],
     [window t:'AppleLink Folder' s:document o:2 c:true z:true g:true
              r: { 203, 93, 442, 246 }
              k:{  [scrollBar t:'' r: { 203, 229, 426, 245 } s:{ 0, 0 } e:true ],
                   [scrollBar t:'' r: { 425, 131, 441, 230 } s:{ 0, 800 } e:true] }],
```

```
    [window t:'MacWrite 5.0' s:document o:3 c:true z:true g:true
            r: { 155, 59, 431, 176 }
            k:{  [scrollBar t:'' r: { 155, 159, 415, 175 } s:{ 0, 0 } e:true ],
                 [scrollBar t:'' r: { 414, 97, 430, 160 } s:{ 0, 48 } e:true ] }],
    [window t:'Applications' s:document o:4 c:true z:true g:true
            r: { 29, 58, 188, 215 }
            k:{  [scrollBar t:'' r: { 29, 198, 172, 214 } s:{ 0, 0 } e:true ],
                 [scrollBar t:'' r: { 171, 96, 187, 199 } s:{ 0, 0 } e:true ] }] }
```

When we used /Mac≈/ , a regular expression, to describe a window in one of the match expression examples above, we only got one descriptor as a result. Using the same descriptor as an operand to a collect expression, we could get a list of all window descriptors that meet all the requirements.

```
println collect[window t:/Mac≈/];
```

```
{   [window t:'MacPaint 2.0' s:document o:1 c:true z:true g:true
            r: { 87, 116, 297, 254 }
            k:{  [scrollBar t:'' r: { 87, 237, 281, 253 } s:{ 0, 0 } e:true ],
                 [scrollBar t:'' r: { 280, 154, 296, 238 } s:{ 0, 0 } e:true ] }],
        [window t:'MacWrite 5.0' s:document o:3 c:true z:true g:true
            r: { 155, 59, 431, 176 }
            k:{  [scrollBar t:'' r: { 155, 159, 415, 175 } s:{ 0, 0 } e:true ],
                 [scrollBar t:'' r: { 414, 97, 430, 160 } s:{ 0, 48 } e:true ] }] }
```

If only one window satisfies the descriptor, a list of one descriptor will be returned. Collect expressions always evaluate to a list. Since only one window can have a specific rank, the following statement will return a list of one window descriptor with a rank of 1.

```
println collect[window o:1];
```

```
{   [window t:'MacPaint 2.0' s:document o:1 c:true z:true g:true
            r: { 87, 116, 297, 254 }
            k:{  [scrollBar t:'' r: { 87, 237, 281, 253 } s:{ 0, 0 } e:true ],
                 [scrollBar t:'' r: { 280, 154, 296, 238 } s:{ 0, 0 } e:true ] }] }
```

It is very useful to obtain specific trait values from a descriptor. Trait values include such things as a window title, a menu item rank within a menu, a button's bounding rectangle, etc… There are two ways to get at specific trait values. The first is by forming a dot expression, the second is called unification.

Dot Expressions

Dot expressions are the means by which a particular trait value can be obtained from a descriptor. These expressions are very similar in concept to expressions that access members of a structure in C. The dot operator (a period) is used to separate an expression and a trait specifier. The first operand must yield a descriptor. Using the same target application example, suppose we wanted to get the front window's title. That could be done as follows:

```
x := match [window ord:1];
println x.t;
```

This script fragment would produce the following output:

```
MacPaint 2.0
```

As with trait specifiers use inside of descriptors, the interpreter is only concerned with the first character of a trait specifier given as the second operand to a dot expression. So the above example could have been written as follows:

```
x := match [window ord:1];
println x.title_of_window;
```

Unification

Unification is a way to bind a variable to a trait value or a list of trait values as a side effect of the matching process. Unifications only mean something when a descriptor is matched, otherwise they are ignored with the unification variable's value binding unchanged. Unifications are expressed as part of a trait within a descriptor. There are two kinds of unification, simple unification and list unification.

Simple unification is specified by prefixing an identifier with ? after the trait specifier corresponding to the value you'd like unified. Simple unification specifies that the value for that trait in the descriptor resulting from the match is to be bound to the variable that follows the ?. By extending one of the previous examples, we can show how simple unification works.

```
println match[window t:?actual_title:/Mac≈/];
```

Here we've added a unification (?actual_title) to the title trait of the window descriptor. Execution of this statement would proceed by matching the descriptor first. It would try to match windows that begin with "Mac" followed by anything else. In the target application we've been using, there are two possible choices for this match so the matcher picks one of them arbitrarily. After the match, actual_title will hold the title of the window which the matcher chose ("MacPaint 2.0" in this case).

Remember, a match will evaluate to only one matched descriptor. It is possible, however, to grab traits from all things that matched the descriptor. List unification is used to specify this process. To specify list unification, a $ should precede the unification variable name. (List unifications should follow simple unifications which should be followed by the specified trait expression.) List unification specifies that the unification variable should be bound to a list containing the trait values from each of the things that equivalently matched the descriptor containing the list unification. Let's change the previous example by making it a list unification rather than a simple unification.

```
println match[window t:$all_titles:/Mac≈/];
```

The match expression still results in only one descriptor, but now the variable, all_titles, will be bound to a list containing the titles of the windows that could have been picked by the matcher. If there was only one window matching this descriptor, the list would have had only one element. Variables given as list unifications will always be bound to lists even if only one thing matched. In this example, there were two so all_titles would be bound to { 'MacPaint 2.0','MacWrite 5.0' }. The first element of the list will always contain the trait value of the descriptor chosen by the matcher as the result of match. It might be easy to be confused between collect expressions and list unification. Remember, collect expressions always evaluate to a list of descriptors that equally match a descriptor. List unification does correspond to this in that if you were to build the collection and build a list of the traits that you wanted to unify from all the descriptors in the collection, you'd have the same list as you would have using list unification.

If a simple unification is given along with the list unification, the simple unification variable will be bound to the same value as the first element contained in the list that will be bound to the list unification variable.

It is possible to use unification within a descriptor that is nested within another descriptor. For example:

```
match [menuItem t:'Open' m:[menu t:?menu_title]];
```

As a result of this match statement, the variable, menu_title, will be bound to the name of the menu containing the menu item with text “Open”.

Perfect Match Expressions

The matcher (that part of the Virtual User that does the matching) will always try to match something to a descriptor if there exists something to match that suitably matches the descriptor. It matches the thing that most closely matches the descriptor. If there isn't something to match it will return the null descriptor, **[ ]**. To specify that you only want to match something that explicitly matches the description you give, apply the postfix operator, ! (often referred to as "bang") to the expression that will evaluate to a descriptor. An expression formed by applying the bang operator to an expression is called a perfect match expression.

Relational Expressions

The following relational operators are supported:

= <> < > >=
<= ~=

Equality (=), inequality (<>), less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=) are all defined for numeric values.

For both regular expressions and strings, the equality (=) and inequality relations (<>) are defined to be straight character by character comparisons using lexicographic ordering rules (same rules as used in a dictionary).

It is also possible to determine whether a string is in the language defined by a given regular expression (i.e. does the string fit the pattern represented by the regular expression). The binary relational operator used to make this test is ~=. The first operand must be the string and the second operand must be the regular expression.

Examples:

```
'Virtual User' ~= /V≈U≈/          # would evaluate to true
'Virtual User' ~= /?irtual User/  # would evaluate to true
'Virtual User' ~= /Virtual Users/ # would evaluate to false
```

Three relations over descriptors are defined, equality (=), inequality (<>), and "twiddle equals" (~=). The operators '=' and '~=' are defined as follows. Inequality (<>) is the negation of the equality operator (=).

If A and B are descriptors, A = B if and only if all of the following hold:

1. Descriptor A is the exact same type as Descriptor B

   (e.g. a menu descriptor can never be equal to a window descriptor)

2. They have the same number of traits specified

3. For each of the traits specified in A, its value must be equal to the corresponding value specified in B

If A and B are descriptors, A ~= B if and only if all of the following hold:

1. Descriptor A is either the exact same type or a subtype of Descriptor B

2. The set of traits specified in B is a subset of the set of traits specified in A

3. For each of the traits specified in B, its value must be equal to the corresponding value specified in A

Twiddle equality is defined for descriptors so that a partially completed descriptor may be tested against a completed descriptor (one with all its traits given). It is of greatest use in comparing incomplete descriptors to fleshed-out descriptors resulting from match expressions (or contained within a list of descriptors resulting from a collect expression).

The following descriptor equality expression would evaluate to false because the descriptors are of different types:

[window t:'ABC' o:2] = [menu t:'ABC' o:2]

The following descriptor equality expression is true:

[window t:'ABC' o:2] = [window t:'ABC' o:2]

The following descriptor twiddle equality expression is true because i) they have the same descriptor type (window); ii) the set of traits given in the second descriptor (only t:) is a subset of the set of traits given in the first descriptor (t: and o:) and iii) the values given for the t: trait are equal.

[window t:'ABC' o:2] ~= [window t:'ABC']

The following descriptor twiddle equality expression is true because i) button descriptors are a subtype of contentItem descriptors; ii) the set of traits given in the second descriptor (only w:) is a subset of the set of traits given in the first descriptor (t: and w:) and iii) the values given for the w: trait are equal.

[button t:'Select' w:[window o:1]] ~= [contentItem w:[window o:1]]

Lists can be tested for equality. Two lists are equal if they both have the same number of elements and their corresponding elements are equal.

Symbolic values may be tested against each other for equality (=) or inequality (<>) only.

▲ **Warning** A relational expression will evaluate to undefined if any of its operands evaluate to undefined. ▲

Logical Expressions

Expressions can be connected by the logical operators **and**, **or**, and **not** to yield either **true** or **false**. Remember that any value can be truth tested. A precedence chart is given in Appendix C.

Examples:

`"foo" and "bar" or false`	would evaluate to	`true`
`true and []`	would evaluate to	`false`
`1 or 0`	would evaluate to	`true`
`1 <> 3 and [] = []`	would evaluate to	`true`

Chapter 3 Descriptor Traits

Descriptor expressions, as introduced in the previous chapter, are used to describe an object in the test environment. This chapter describes each type of descriptor allowed in this language in detail. For each descriptor type, a list of all its possible traits and their corresponding value types is mentioned.

Window Descriptors

Window descriptors are used to describe the state of a window on the target machine. Its traits are as follows:

t: used to specify or unify a window's title. The expression that follows the trait specifier and optional unification(s) must evaluate to a regular expression at runtime.

o: used to specify or unify the rank in the front-to-back ordering of the visible windows in the application's window list. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

s: used to specify or unify a window's style. The expression that follows the trait specifier and optional unification(s) must evaluate to one of the window style symbolic values. These values are as follows:

document dialog da shadow plain

c: used to specify or unify whether the window has a close box. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

z: used to specify or unify whether the window has a zoom box. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

g: used to specify or unify whether the window has a grow or size box. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

k: used to specify or unify the items that exist in the content region of a window. The expression that follows the trait specifier and optional unification(s) must evaluate to a list of descriptors at runtime. The descriptors must be one of the following types:

button	checkBox	contentItem	control
editText	icon	picture	popup
radioButton	scrollBar	staticText	userItem

r: used to specify or unify the dimensions of the window's bounding rectangle (in global coordinates). At runtime, the expression that follows the trait specifier and optional unification(s) must evaluate to a list of four numerical values that will be interpreted to be a rectangle of the following form:

{ left, top, right, bottom }

Content Item Descriptors

Content item is used here to refer to a class of descriptor types that can be used to describe something in the content region of a window. Descriptors that fall into this class include editText, icon, picture, staticText, userItem and the whole class of control descriptors defined later. These descriptor types basically have the same traits. One difference is that the non-controls like editText and staticText do not have control value (s:) and highlite state (h:). The traits for content items are as follows:

t: used to specify or unify the text (text enclosed inside a button, text attached to a radio button, editable text within an edit text box, text in a static text box, title of a popup etc…). The expression that follows the trait specifier and optional unification(s) must evaluate to a regular expression at runtime.

w: used to specify or unify the window in which this object resides. The expression that follows the trait specifier and optional unification(s) must evaluate to a window descriptor at runtime. If the expression evaluates to a regular expression, a window descriptor with that regular expression specified as the title will be assumed.

r: used to specify or unify the dimensions of its enclosing rectangle (in global coordinates). At runtime, the expression that follows the trait specifier and optional unification(s) must evaluate to a list of numerical values that will be interpreted to be a rectangle in global coordinates : { left,top,right,bottom }

e: used to specify or unify the enablement state of the item. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime.

Control Descriptors

Control descriptor is used here to refer to a class of descriptor types that form a subclass of content item descriptors. Descriptors that fall

h: (only valid for button, checkBox, control, radioButton, and scrollBar descriptors) used to specify or unify the highlight state of the control. The expression that follows the trait specifier and optional unification(s) must evaluate to a numeric value at runtime. The value can range from 0 to 255.

s: (only valid for button, checkBox, control, radioButton, and scrollBar descriptors) This trait specifier is used to state or unify the control setting. At runtime, the expression that follows the trait specifier and optional unification(s) must evaluate to a list of two numeric values. The first element will contain the current control value - the minimum value of the control. The second element will represent the difference between the maximum and minimum values (contrlMax minus contrlMin) for the control. Think of it as the control's value expressed as a fraction. Check boxes and radio buttons will typically have settings of { 0,1 } and { 1,1 }. For popups, the first element is the rank of the currently selected item and the second is the number of items in the popup menu. For scroll bars, the top of a scroll bar is { 0,X } and the bottom is { X,X } where X is the difference between the maximum and minimum values for the control.

Menu Descriptors

Menu descriptors are used to describe the state of a menu on the target machine. Its traits are as follows:

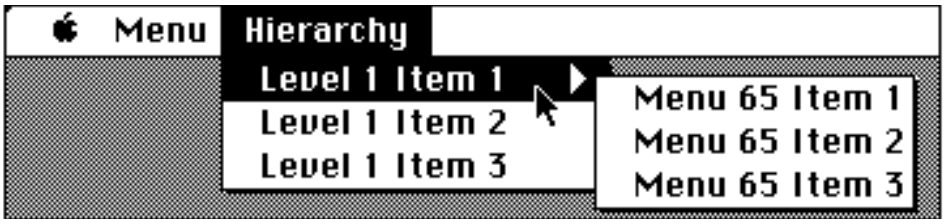
t: used to specify or unify the menu title as it appears in the menubar. The expression that follows the trait specifier and optional unification(s) must evaluate to a regular expression at runtime.

o: used to specify or unify the rank of the menu in the menubar. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

e: used to specify or unify the enablement of a menu (whether or not it is currently dimmed). The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation, but most often the symbolic identifiers, true and false will be used.

i: used to specify or unify the items contained in the menu. The expression that follows the trait specifier and optional unification(s) must evaluate to a list of menu item descriptors.

Figure 3-I Hierarchical Menus



Example:

Using the menubar shown in Figure 3-I, the following print statements will show how the matcher will interpret the given menu descriptors:

```
println match[menu t:'Hierarchy'];
```

The above statement would produce the following (formatted for this document)

```
[menu t:'Hierarchy' o:3 e:true
      i: { [menuItem t:'Level 1 Item 1' o:1 k:'' c:'' e:true
                     h: { [menuItem t:'Menu 65 Item 1' o:1 k:'' c:'' e:true],
                          [menuItem t:'Menu 65 Item 2' o:2 k:'' c:'' e:true],
                          [menuItem t:'Menu 65 Item 3' o:3 k:'' c:'' e:true] }
           [menuItem t:'Level 1 Item 2' o:2 k:'' c:'' e:true],
           [menuItem t:'Level 1 Item 3' o:3 k:'' c:'' e:true] } ]
```

Menu Item Descriptors

Menu item descriptors are used to describe the state of a menu item on the target machine. Its traits are as follows:

t: used to specify or unify the menu item text. The expression that follows the trait specifier and optional unification(s) must evaluate to a regular expression at runtime.

m: used to specify or unify the entity which owns the item. The expression that follows the trait specifier and optional unification(s) must evaluate to either a menu or menuItem or a popup descriptor at runtime. If a menu descriptor follows the m: specifier, the menuItem is a standard menuItem within a first level menu. If the menuItem's m: trait is another menuItem, then the item being specified is a hierarchical menuItem. If the expression evaluates to a regular expression, a menu descriptor with that regular expression specified as the title will be assumed. If the m: trait is a popup descriptor then the item being specified belongs to a popup menu in a window. An example of such an item is as follows:

[menuItem t:'9600' m:[popup t:'Baud Rate:' w:[window o:1]]]

o: used to specify or unify the rank of an item within the menu. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

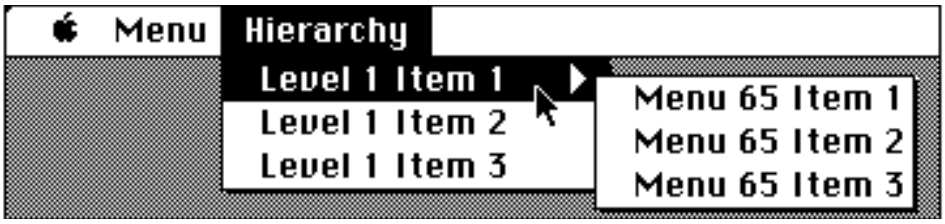
k: used to specify or unify the keyboard alias or keyboard equivalent. The expression that follows the trait specifier and optional unification(s) must evaluate to a string. The first character of this string will be used as the keyboard alias character specification. Most often, a character enclosed in single or double quotes should be specified.

c: used to specify or unify the mark character. The expression that follows the trait specifier and optional unification(s) must evaluate to a string. The first character of this string will be used as the mark character specification. Most often, a character enclosed in single or double quotes should be specified.

e: used to specify or unify the enablement of a menu item which is reflected in whether the item is dimmed/grayed out. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

h: used to specify or unify whether the menuItem has a submenu of items (hierarchical). The expression that follows the trait specifier and optional unification(s) must evaluate to a list of menuItem descriptors.

Figure 3-II Hierarchical Menu Items



Using the menubar as shown in Figure 3-II, the following print statements will show how the matcher will interpret the given menuItem descriptors:

Example1:

```
println match[menuItem t:'Level 1 Item 1'];
```

This statement would produce the following (formatted for this document)

```
[menuItem t:'Level 1 Item 1' m:[menu t:'Hierarchy' o:3 e:true] o:1 k:'' c:'' e:true]
```

Example 2:

```
println match[menuItem t:'Menu 65 Item 3'];
```

This statement would produce the following (formatted for this document)

```
[menuItem t:'Menu 65 Item 3'
          m:[menuItem t:'Level 1 Item 1'
                      m: [menu t:'Hierarchy' o:3 e:true] o:1 k:'' c:'' e:true]]
```

Keyboard Descriptors

Keyboard descriptors are used to describe the state of the keyboard attached to the target machine. Its traits are as follows:

t: used to specify or unify the type of keyboard. The expression that follows the trait specifier and optional unification(s) must evaluate to one of the keyboard type symbolic values. These values (derived from keyboardType constants in Inside Macintosh volume V, page 8) are as follows:

AExtendKbd ExtISOADBKbd MacAndPad MacPlusKbd
PortADBKbd PortISOADBKbd StandADBKbd StdISOADBKbd

UnknownKbd

s: used to specify or unify the keyScript or the script associated with the keyboard (as defined in Inside Macintosh volume V, page 298). The expression that follows the trait specifier and optional unification(s) must evaluate to a numeric value.

Mouse Descriptors

Mouse descriptors are used to describe the state of the mouse attached to the target machine. Its traits are as follows:

p: used to specify a mouse position. At runtime, the expression that follows the trait specifier and optional unification(s) must evaluate to a list of 2 numerical values that will be interpreted to be a point where the first element is the x-coordinate and the second is the y-coordinate.

b: used to specify or unify whether the mouse button is down. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime (true means the button is down). Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

Examples:

Suppose the mouse currently has x-coordinate 234 and y coordinate 187 and the mouse button is currently down. Then the statement,

```
current_mouse := match[mouse];
```

would cause current_mouse to be bound to:

```
[mouse p: { 234,187 } b:true]
```

Screen Descriptors

Screen descriptors are used to describe the state of a screen on the target machine. Its traits are as follows:

r: used to specify the rectangle surrounding the coordinate space that the screen is over. At runtime, the expression that follows the trait specifier and optional unification(s) must evaluate to a list of four numerical values that will be interpreted to be a rectangle of the following form:

{ left, top, right, bottom }

For a 9” screen it will be as follows:

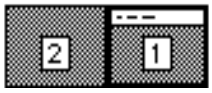
{ 0,0,512,342 }

m: used to specify or unify whether that particular screen is the main screen and, therefore, contains the menubar. The expression that follows the trait specifier and optional unification(s) must evaluate to a boolean value at runtime. Every expression has a boolean interpretation (for the context of use), but most often the symbolic identifiers, true and false will be used.

Example:

Suppose a MacII has two monitors and the monitors setting looks as follows in the control panel:

Figure 3-III Screen Configuration



Now execute the following assignment statement:

```
current_screen_config := collect[screen];
```

As a result, current_screen_config would be bound to the following list of screen descriptors:

```
{   [screen r:{ 0, 0, 640, 480 } m:true],
    [screen r:{ -640, 0, 0, 480 } m:false]  }
```

Application Descriptors

Application descriptors are used to describe an application on the target machine. Its traits are as follows:

t: used to specify or unify the name or title of the application. The expression that follows the trait specifier and optional unification(s) must evaluate to a regular expression at runtime.

Example:

If the foreground application running on the target under test was the MPW shell, the following statement that follows would produce the application descriptor shown below the statement as output:

```
println match[application];

[application t:"MPW Shell"]
```

Target Descriptors

Target descriptors are used to describe the state of the target machine. Its traits are as follows:

t: used to specify or unify a target name (the User name in the Chooser on a Macintosh running system software 6.0.x). The expression that follows the trait specifier and optional unification(s) must evaluate to a string or a regular expression at runtime.

z: used to specify or unify a target's network zone name. The expression that follows the trait specifier and optional unification(s) must evaluate to a string or a regular expression at runtime.

n: used to specify or unify a target's model name. The expression that follows the trait specifier and optional unification(s) must evaluate to one of the model name symbolic values. These values are as follows:

MachUnknown		MacII	MacIIci	MacIIcx	MacIIfx	MacIIx
MacPlus	Portable	SE	SE30	a512KE		

r: used to specify or unify the target's RAM in kilobytes. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

a: used to specify or unify the applications currently running in the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a list of application descriptors at runtime.

- *Note*: At present, unifying this trait will only return a list containing one application descriptor describing the front most application layer.

s: used to specify or unify the screens currently connected to the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a list of screen descriptors at runtime.

m: used to specify or unify the state of the mouse attached to the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a mouse descriptor at runtime.

k: used to specify or unify the state of the keyboard attached to the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a keyboard descriptor at runtime.

System Descriptors

System descriptors are used for describing the system software running on the target machine. They have the following traits:

v: used to specify or unify the version of the system software on the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a string or a regular expression value.

s: used to specify or unify the system script or the script associated with the system software on the target. The expression that follows the trait specifier and optional unification(s) must evaluate to a numeric value.

Actor Descriptors

Actor descriptors are used to describe the actors in the Virtual User system. They have the following traits:

t: used to specify or unify the name of the actor. The expression that follows the trait specifier and optional unification(s) must evaluate to a string or a regular expression value.

u: used to specify or unify the name of the target under the actor's control. The expression that follows the trait specifier and optional unification(s) must evaluate to a target descriptor.

Time Descriptors

Time descriptors are used to describe a certain time of day. The expression,

```
match [time]
```

will evaluate to the current time as set in the host system's clock. Time descriptors are handy for adding pauses to scripts, printing the current time at certain points in a script, keeping track of how long a certain operation is taking, etc… The possible traits for a time descriptor are listed below:

y: used to specify or unify a year (e.g. 1989). The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

m: used to specify or unify a month. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

d: used to specify or unify a day. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

h: used to specify or unify hours. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime. In specifying a time of day, this trait is specified using military time. For example, 1300 would be 1PM.

s: used to specify or unify seconds. The expression that follows the trait specifier and optional unification(s) must evaluate to a numerical value at runtime.

Descriptor Shortcuts

It's possible to give a string/regular expression for either a w: trait within a content descriptor or a m: trait within a menuItem descriptor and have VU assume a descriptor with a valid t: trait specified as that string/regular expression. It is also possible to give a numeric value for those two traits and have VU assume a window/menu descriptor with the rank specified as that numeric value. The following examples should clarify this shortcut mechanism:

[menuItem t: 'Open…' m: 'File']

is equivalent to:

[menuItem t: 'Open…' m:[menu t: 'File']]

~~~~~~

[menuItem t: 'Key Caps' m: 1]

is equivalent to:

[menuItem t: ' Key Caps' m:[menu o: 1]]

~~~~~~

[button t:'OK' w: 1]

is equivalent to:

[button t:'OK' w: [window o:1]]

~~~~~~

[editText t:'hello' w: 'untitled']
~~~~~~

is equivalent to:

```
[editText t: 'hello' w: [window t: 'untitled']]
```

Chapter 4 Statements

A scriptis made up of statements. A statement is terminated using a semicolon as in:

```
x := 1;
```

Statements can be grouped to form compound statements and blocks. Blocks are similar to those in Pascal. *Begin…End;* is used to group the statements forming a block. This chapter describes the simple statements namely the print statements, assignment statements and the commands. Compound statements are discussed in the following chapters.

Print Statements

When expressions are evaluated, runtime values materialize. To view the results of these evaluations, **print** statements are supported. This special statement should be formed by giving the keyword, **print**, followed by any number of expressions separated by commas. To format the output, use strings that contain tabs, spaces, and carriage returns.

Examples:

```
print "hello∂t", [window t:'hello'],"∂t", 34;  # yields the following output[3]

hello  [window t:'hello']  34
```

In the above example, the insertion point would be left at the end of the output line produced. By giving **println** rather than **print**, one can force a carriage return following the last printed expression evaluation result for that statement.

Assignment Statements

Assignment statements are statements which allow us to give values to variables. These statements have the variable on the left of the operator ':=' and an expression on the right of it. The variable on the left is assigned the value that the expression on the right evaluates to. Some examples of assignment statements follow:

```
str := "hello world";
desc := match[window o:1]!;
```

[3] ∂t denotes the tab character. See section on Regular Expression Matching for further information.

sign is used to start a comment. See section on Commenting a Script in Chapter 6 for details.

Commands

- *Note*: Since the subject of this section is not on descriptors, the descriptor examples will be given the form [descriptor-type …] where … is shorthand notation for some set of traits appropriate for the descriptor type specified (see the section on descriptors). The ellipses is not part of the language.

Selecting Menus

Dropping a menu can be done by way of script by giving the keyword, select, followed by an expression that will evaluate to a menu descriptor. There are no arguments that may be applied at this time.

```
select [menu t:"Tools"]; # momentarily drops a menu called tools
```

Selecting Menu Items

Selecting a menu item is done by specifying the keyword, select, followed by an expression that will evaluate to a descriptor of type, menuItem. One argument is allowed for menu item selections specifying whether the Virtual User should use the keyboard equivalent in making the selection. That argument's description follows:

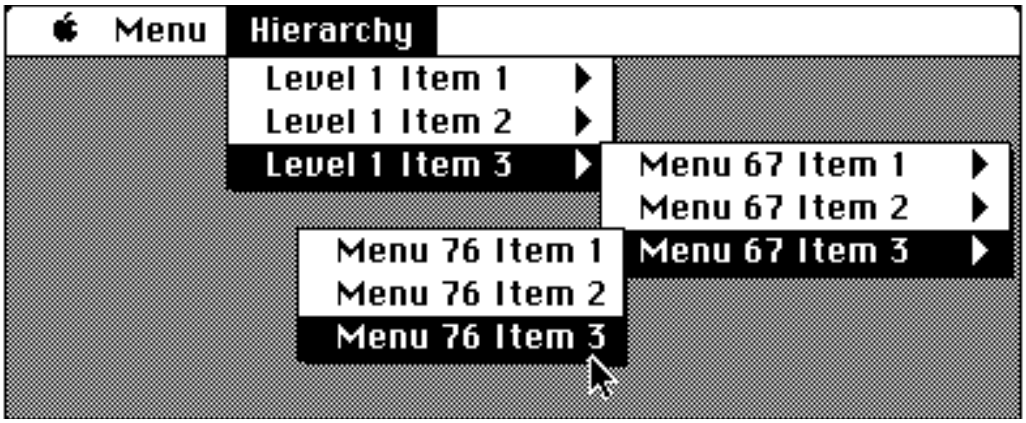
k: if the expression following this argument evaluates to true, the Virtual User will use the keyboard equivalent in making the selection. If it evaluates to false, the mouse will be used as in the default case.

Example:

```
select [menuItem t:/Plain≈/ m:'Style'] k: true;
```

The above selection will be done using the keyboard equivalent found for the menu item described. Note that no specific keyboard equivalent is specified. Whatever keyboard equivalent that is found in making the match will be used. If no equivalent is present, an error message will be generated and the statement is skipped.

Figure 4-I Selecting Hierarchical Menu Items



A number of command formulations could be used to specify the hierarchical menuitem selection shown in Figure 4-I. Here are just a few:

```
select [menuItem            t:'Menu 76 Item3'
                m:[menuItem t:'Menu 67 Item 3'
                            m:[menuItem t:’Level 1 Item 3’
                                        m:[menu t:’Hierarchy’]]]];

select [menuItem            t:'Menu 76 Item3'
                m:[menuItem t:'Menu 67 Item 3'
                            m:[menuItem t:’Level 1 Item 3’
                                        m:’Hierarchy’]]];

select [menuItem t:'Menu 76 Item3'];

select [menuItem t:'Menu 76 Item3' m:[menuItem t:’Menu 76 Item 3]];

select [menuItem t:'Menu 76 Item3' m:[menuItem m:[menuItem m:[menu]]]];
```

Selecting Windows

Window selections are specified by giving the keyword, select, followed by an expression that will evaluate to a window descriptor .

Examples:

```
select [window … ];
```

This command selects the window matching the descriptor by clicking in any exposed region.

Dragging Windows

Window dragging can be specified by giving the keyword, drag, followed by an expression that will evaluate to a descriptor of type window. Following the descriptor, one argument is needed to specify how the dragging should be done. The possibilities follow along with their semantics:

a: means you would like the window moved to an absolute position in global coordinates. A two member list of expressions that should yield numeric values is given where the first value will be interpreted to be the x-coordinate of the window's upper left corner and the second the y-coordinate.

- *Note*: The coordinates specified are to be the coordinates of the upper left corner of the entire window, not that of the window's content region.

r: means you would like to specify a relative change in position. Again, a two member list of expressions yielding numeric values is specified. The first is interpreted to mean "change in horizontal position" and the second to mean "change in vertical position".

Examples:

1.
```
drag [window … ] a: { 10,100};
```
would drag the window matching the descriptor such that after the drag operation the window's upper left corner would be at (10,100) in global coordinates.

2.
```
drag [window … ] r: { 20,-10 };
```
would drag the window matching the descriptor such that after the drag operation the window's left edge will have moved 20 pixels to the right and the windows top will have moved 10 pixels up.

Sizing Windows

Sizing windows involves specifying the keyword, size, followed by an expression that will evaluate to a descriptor of type window. Following the descriptor, one argument is needed to specify how the sizing should be done. The possibilities follow along with their semantics:

h: means you would like to specify what the window's height should be after the size operation. Following the h: should be an expression that evaluates to a numerical value which will be interpreted to be the window's resulting height.

w: means you would like to specify what the window's width should be after the size operation. Following the w: should be an expression that evaluates to a numerical value which will be interpreted to be the window's resulting width.

r: means you would like to specify a change in size relative to the current size. A two-member list of expressions that will yield numerical values should be given following the argument specifier (r:). The first value will be interpreted to mean a change in the width of the window and the second will specify how the window should change in height.

Examples:

1. `size [window … ] w: 34;`

   means that after the command is executed, the window matching the descriptor will be 34 pixels wide (if the application cooperates)

2. `size [window … ] h:114;`

   means that after the command is executed, the window matching the descriptor will be 114 pixels high. (if the application cooperates)

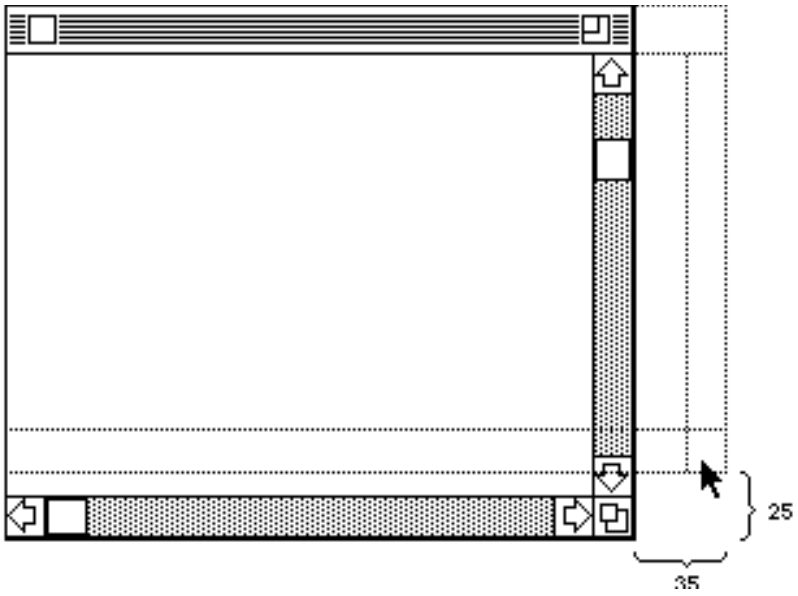
3. `size [window … ] w:223 h:110;`

   means that after the command is executed, the window matching the descriptor will be 223 pixels wide and 110 pixels high. (if the application cooperates)

4. `size [window … ] r: { 35,-25};`

   means that after the command is executed, the window matching the descriptor will be 35 pixels wider and 25 pixels shorter. This sizing action is depicted below:

Figure 4-II Sizing Windows



Closing Windows

Closing windows involves specifying the keyword, close, followed by an expression that will evaluate to a descriptor of type window. The matching window will receive a click in its close box.

Zooming Windows

Zooming windows involves specifying the keyword, zoom followed by an expression that will evaluate to a descriptor of type window. The matching window will receive a click in its zoom box.

Selecting Buttons

Button selection is specified by giving the keyword, select, followed by an expression that will evaluate to a button descriptor. (No arguments are needed or allowed)

Selecting Radio Buttons

Radio button selection is specified by giving the keyword, select, followed by an expression that will evaluate to a descriptor of type radioButton. (No arguments are needed or allowed.)

Selecting Check Boxes

Check box selection is specified by giving the keyword, select, followed by an expression that will evaluate to a descriptor of type checkBox. (No arguments are needed or allowed.)

Scrolling

Scrolling involves specifying the keyword, scroll, followed by an expression that will evaluate to a descriptor of type scrollBar. Following the descriptor, one argument is needed to specify how the scrolling should be done. The argument possibilities are listed below. To understand the arguments, it is helpful to review the s: trait for a scrollBar descriptor. This trait is used to specify or unify the current scrollbar setting expressed as a fraction. The denominator of the fraction represents the total number of values the scrollbar can range over. The numerator is the current value.

a: this argument must be followed by a fraction specified as a list of 2 numbers. The fractions can be thought of as a percentage denoting the position where the thumb should be left after the scrolling operation.

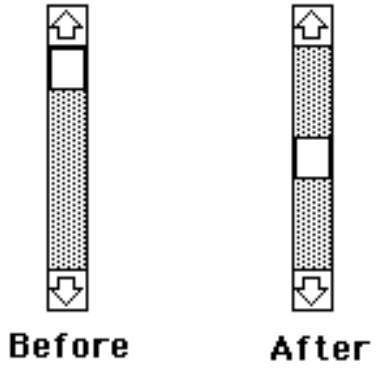
r: this argument again takes a list of two numbers which will be interpreted as a fraction (or percentage) change relative to the current position. If the fraction is positive the relative change will be in the positive direction (towards the downArrow) and if negative the change will be towards the upArrow.

Examples:

1. scroll [scrollBar …] absolute: { 1,2 };

   the diagram below shows the scrollbar state before and after the above example has been executed. The fraction { 1,2 } indicates that the scrollbar should be scrolled such the thumb is 50% of the total range of values.

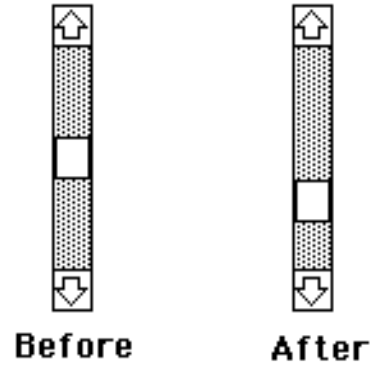
Figure 4-III Scrolling (Example 1)



2. scroll [scrollBar …] relative: { 1,2 };

   would drag the thumb of the scrollbar to a position halfway between where it currently is and the downArrow of the scrollbar. The scrollbars below show before and after states for the above scroll command.

Figure 4-IV Scrolling (Example 2)



Typing

To specify typing, the keyword, type, is given followed by a number of arguments. At present only one argument is allowed. A description of this argument follows:

k: can be used to specify the keystrokes to be typed by the Virtual User. It should be followed by some expression that will evaluate to a possibly heterogeneous list. The list may contain expressions that evaluate to strings or any of the special keystroke symbolic identifiers. A list of the special keystroke symbolic identifiers follows along with some command examples.

Keystroke symbolic identifiers:

backspaceKey	capslockKey	clearKey	commandKey	controlKey
delKey	downarrowKey	endKey	enterKey	escapeKey
f1Key	f2Key	f3Key	f4Key	f5Key
f6Key	f7Key	f8Key	f9Key	f10Keyf
f11Key	f12Key	f13Key	f14Key	f15Key
helpKey	homeKey	leftarrowKey	optionKey	pagedownKey
pageupKey	returnKey	rightarrowKey	shiftKey	tabKey
uparrowKey				

Example:

```
Type k: { "abc",tabKey,"def",tabKey,"ghi" };
```

The nine letters will be typed with a tab inserted after every three characters

Type causes the keystrokes to be both pressed and released in one statement. To press and release keys separately, the same mechanism is used as with the type command. The only difference is that one of **pressKey** or **releaseKey** should be substituted for the keyword, type .

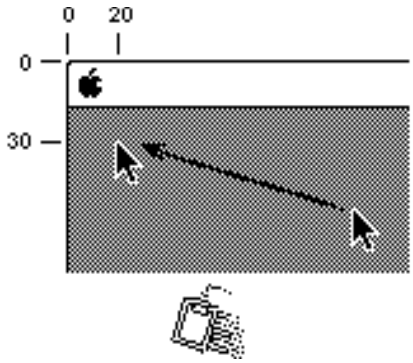
Controlling the Mouse Directly

It is possible to have VU directly control the mouse. Primitive operations such as moving the mouse position, clicking, double-clicking, pressing the mouse button, and releasing the mouse button are available.

To move the mouse directly, a move command may be used. **move** takes one argument which may either be a: (absolute move) or r: (relative move). Following the argument specifier chosen must be an expression that will evaluate to a list of two numeric values. If the argument specifier is a:, the list of two numeric values will be interpreted as a point in the coordinate system to which the mouse should be moved. The first element of the list will be the x-coordinate and the second, the y-coordinate. If the argument specifier is r:, the list of two numeric values will be interpreted to mean a relative change in the mouse position. The first element of the list will indicate the relative change in the x-direction and the second element will give the relative change in the y-direction. Both use the Mac coordinate system where values in the y-direction increase as you move downwards on the screen. The following example would move the mouse such that its hot spot would be at the point (20,30):

```
move absolute: { 20,30 }; # this action is depicted below
```

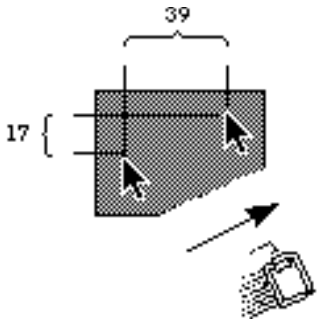
Figure 4-V Mouse Move (Absolute)



The next example would direct the mouse to be moved 39 pixels to the right and 17 pixels down:

```
move relative: { 39,17 }; # this action is depicted below
```

Figure 4-VI Mouse Move (Relative)



To click the mouse at any point in the script, the word, **click**, should be used followed by nothing but a semicolon. Similarly, to double click, the word, **doubleClick**, should be used. It is also possible to press the mouse without releasing. The word, **pressMouse**, should be used in this instance. To release the mouse, use **releaseMouse**. It is always best to use **click** and **doubleClick** statements rather than a combination of **pressMouse** and **releaseMouse** statements. The reason for this is that in a multiple target test, the time between a release of the mouse and the next click may be dependent on the network traffic. Both **click** and **doubleClick** do their pressing and releasing in one AppleTalk transaction.

Exit Statement

While the language provides constructs to promote structured programming of scripts, there are times when a situation occurs when one wishes to bail from a script. The exit statement may be used for these situations. Upon encountering an exit statement, the Virtual User will no longer continue reading that script for the target that took the path to the exit statement. Only the target which reached the exit in the script it is reading will stop executing. Other targets currently reading the script which have not encountered the exit statement will continue to run as will targets reading other scripts. Exit statements appear as shown below:

```
exit;
```

Chapter 5 Control Flow

This chapter describes the statements which decide the order in which computations are performed. The control flow statements in this language are *If-Else*, *For* loops, *For-Each* loops and *While* loops.

If-Else Statements

If-Else statements are supported to enable conditional statement executions. When an If-Else construct is encountered, the expression following **if** is evaluated and truth tested. If the resulting value is true, the statement or block of statements following the expression is executed. If the resulting value is false one of two things can happen. If there is an **else** associated with the **if** , the statement or block of statements following the **else** is executed. If there is no **else**, control skips to the next statement in the script.

If-Else constructs can be used to handle unforeseen situations like dialogs popping up. The following statement will cause the Virtual User to dismiss the front most window (if one exists) if it has an OK button:

```
if match [button text:'OK' w:[window ord:1]]!
       select [button text:'OK' w: [window ord:1]]!;
```

For Loops

Iterative For-loops are supported in the scripting language. The form is very similar to Pascal. All expressions must evaluate to numerical values. From the grammar in Appendix A, the form of a For loop looks as follows:

<for_loop> ::= **for** <variable> **:=** <expression> **to** <expression> <optional_step>

<optional_do> <executable_statement>

All expressions in the control portion of the For loop must evaluate to numerical values. The loop body can be either a single statement or a block of statements enclosed with **begin** and **end**. A For loop with no statements will simply be skipped. It is not possible to use a For loop with no body as a delay. To put delays into a script, use system tasks described in Appendix F.

When a For loop is encountered the loop control variable becomes bound to the first expression's resulting value. A For loop is controlled by whether the expression following **step** evaluates to a negative number or not. If it evaluates to a numerical value greater than or equal to 0, the loop body is entered if the loop control variable is not greater than the value resulting from the evaluation of the expression to the immediate right of the **to**. If the step expression evaluates to a numerical value less than zero, the loop body is entered if the loop control variable is not less than the value resulting from the evaluation of the expression to the immediate right of the **to**. If a step expression is not given, it is assumed to be 1. After each iteration of the loop, the result of the step expression evaluation is added to the loop control variable. The loop body is then repeatedly entered as long as the rule given above holds.

Examples:

1.
```
for i := 1 to 5 select [window o:i];
```

Assuming the target application currently has at least 5 selectable windows, this construct will direct the Virtual User to make 5 window selections with successively higher ranks beginning with 1. It should be noted that after selecting each of the windows the ranks of the windows change so selection may not proceed as one might expect.

2.
```
for font_item := 1 to 10 begin
      select [menuItem m: 'Font' o: font_item];
      for font_size_item := 1 to 5 begin
            select [menuItem m: 'FontSize' o:font_size_item];
      end;
end;
```

Assuming the target application has a "Font" menu with 10 fonts and a "FontSize" menu with 5 sizes, the construct above will direct the Virtual User to select each of the fonts from the font menu following each font selection with a selection of each of the font sizes in the order they exist in the "FontSize" menu.

3.
```
for i := 5 to 1 step -3 close [window];
```

Assuming there are at least two windows to be closed in the target application, this construct will direct the Virtual User to close two of them.

For Each Loops

For Each loops can be used to iterate over an entire list. The general form of a For Each loop is as follows:

for each <variable> **in** <expression> <optional_do> <executable_statement>

When a For Each loop is encountered during execution, the expression following the reserved word, **in**, is evaluated. The result of this evaluation must be a list (if it doesn't, the For Each loop is skipped). This evaluation is done only once each time the loop is initially entered, not for each iteration of the loop. The loop body is executed once for each element in this list. The loop control variable is bound to the first element in the list for the first iteration, the second element for the second iteration, and so on until it has iterated over the entire list.

Using For Each loops with collect expressions is a handy mechanism. Suppose an application maintains a Windows menu that contains the titles of all the selectable windows that are part of the current application state. The following script fragment will instruct the Virtual User to select each of the windows:

```
window_items := collect [menuItem m: [menu t:'Windows']];
for each item in window_items
       select [window t:item.title]!;
```

While Loops

The language also supports While loops. The standard While loop semantics are used. Each time a While loop is encountered, the expression immediately following the word, **while**, is evaluated. As long as the resulting value is true, the loop body is executed. The loop body will be executed until the expression evaluates to a value that is considered false (e.g. 0,false,[ ],{ })

While loops can be used to wait for a specific situation as the following example illustrates:

```
while match [window style:dialog]! ;
```

When this While loop is encountered, the script will pause until no dialog is present. As another example, the following loop could be used to close all the windows up to the first window without a close box:

```
while match [window]
	close [window];
```

Chapter 6 Making Scripts Modular

Function-like definitions are supported in VU's scripting language in the form of tasks. This chapter describes modularization of scripts using tasks.

Tasks

Tasks are specified by giving **task** followed by the task name followed by the task's formal parameter list. After the formal parameter list, the task body is given as a block (**begin** … **end**;). Formal parameters are separated by commas and are specified by name only. There are no type declarations in a formal parameter list. An optional default expression may be given for each formal parameter. If no corresponding value is passed for that parameter at runtime, the default expression is evaluated and the parameter takes the resulting value. A parameter with a default expression is specified using the same syntax as assignment. The parameter name is given followed by the assignment operator (:=) followed by the default expression.

Variable scoping is simple. Variables can be either local to a task or global to a script. All variables referenced within a task are local to that task (including the task's formal parameter list). A variable within a task may be considered global if it is preceded by the word, **global**, before or during its first use. Global declaration statements may be given within a task by giving **global** followed by any number of variable names separated by commas. The global declaration statement should also be terminated with a semi-colon. The same variable name may not be used as both a local variable and as a global variable. Its first use determines its scope. The task below declares X and Y to be global so after the invocation of this task X will persist and will have the value 1, as will Y with the value 3. Note that X is declared to be global using a global declaration statement while Y is declared at the point of its first use. A variable may be prefixed with the word **global**, anywhere within a task. All variables outside a task are global by default.

```
task A begin
      global X;
```

```
        X := 1;
        global Y := 3;
end;
```

Examples:

The following script illustrates the task definition and calling syntax. It consists of a task called printit that simply prints the value passed. If no value is passed, it will print the message, "no value was passed". In this example, the string, "no value was passed", is the default expression for the formal parameter variable called value_to_print.

```
Task printit(value_to_print := "no value was passed") begin
        println value_to_print;
end;
printit(4);
printit;
printit( {1,2,3,4} );
```

The above script would produce the following as output:

```
4
no value was passed
{ 1,2,3,4 }
```

As another example of a task definition, the following handles the "Do you want to save your changes" dialog that appears in most applications when the user closes a window before saving the file associated with that window.

```
Task handle_dirty_document(          should_save := true,
                           affirmative := "Yes",
                           negative := "No" )
begin
     if match [window o:1 s:dialog k: { [button title: affirmative],
                                        [button title: negative] } ]
     begin
          if should_save button_to_select := affirmative;
          else button_to_select := negative;
          select [button t:button_to_select]!;
     end;
end;
```

The above task can be called from any script that wishes to handle the "Do you want to save your changes" type dialog. The caller may may pass in three values. The first indicates which button the user should select to dismiss the dialog. If the parameter, should_save, is passed any true value the Virtual User will select the button that tells the application to save the document. Otherwise it will select the button indicating that no save is necessary. Following this first actual parameter, the caller may give the text used in the buttons for the application under test. If no button text strings are passed, the task assumes they are called "Yes" and "No". Assuming that you'd like to use this task for an application that presents the dialog with Save and Discard buttons, the following call could be used:

```
handle_dirty_document(true,"Save","Discard");
```

To force the use of a default parameter, simply leave that actual parameter blank in the call. For example, suppose we want to make the same call as above only we want the second parameter to take the default. That call would look as follows:

```
handle_dirty_document(true,,"Discard");
```

Within the task now, affirmative has the value "Yes" and negative has the value "Discard".

Returning Values from Tasks

Tasks can return values by way of return statements. **return** statements may only appear within a task definition, but can return any arbitrary expression. Therefore task calls can appear anywhere an expression can. If a task does not return a value and the result of a task is used, the resulting value is undefined. Following are some examples:

1. The following task returns a string and the task is called within a println statement:

```
task stupid_task() begin
       return "stupid";
end;
println stupid_task();
```

The output to the above task appears below:

```
stupid
```

2. The next example shows that any arbitrary expression may be returned from a task.

```
task send_list_of_garbage() begin
       x := 1;
       y := document;
       z := { "this","is","a","list" };
       a_window := [window t:'Doc A'];
       return { x,y,z,a_window,56,'abc' };
end;
println send_list_of_garbage();
```

The output to the above script is shown below:

```
{ 1,document,{ "this","is","a","list" },[window t:'Doc A'],56,'abc' }
```

3. As a final example, a recursive task to compute n-factorial is shown:

```
# computes n factorial
task factorial(n) begin
       if (n = 1) return 1;
       else return n * factorial(n - 1);
end;
```

System Supplied Tasks

A number of built-in tasks can be called like any other user-defined task only the task implementation is part of the environment. In other words, these tasks should be viewed as "black-boxes" that can be called, but not seen. See Appendix F for a list of system supplied tasks along with their semantics.

Libraries of Tasks

User defined tasks can be grouped to build task libraries. These libraries can then be shared by multiple scripts. A script that uses tasks defined in task libraries has to declare the names of the libraries. The following declarative statement is provided to do this:

Libraries <library name>, <library name>, … ;

Libraries is a keyword which is followed by library names separated by commas. The names of libraries that are required by the script are specified as strings.

Examples:

```
Libraries 'StringsLibrary',  'FinderOperationsLib';
```

This declares that the script is going to use tasks defined in the libraries StringsLibrary and FinderOperationsLib. The library name is basically the name of the file that contains the library.

Commenting a Script

Its generally a good idea to comment your scripts and task libraries for making maintenance easier. Commented scripts become more readable. Whenever the compiler sees the character, #, it ignores the rest of the characters on that line. A commented version of a script follows:

```
# this script opens the "Key Caps" DA types into it
select [menuItem title:'Key Caps' m: [menu ord:1]]; # select Key Caps
type keystrokes: { "hello world", returnKey }; # do some typing
```

For multiple line comments, the opening and closing delimiters are (* and *). It is legal to nest comments. Below is an example of a multiple line comment:

```
(* This is a
```

```
      multiple line comment with a (* nested
             multiple line comment *) contained within *)
```

Chapter 7 Message Passing

This chapter exemplifies the multitasking nature of the Virtual User environment. It describes how the different test processes can communicate amongst each other through the message passing constructs in the language.

Multitasking Environment

Virtual User provides you with a multitasking environment. It allows you to run multiple scripts simultaneously. Each script execution is treated as a separate process in the VU architecture. Each process has its own process handler called the **actor**. Currently each VU process handler (ie., **actor**) runs independent of the rest of the actors. The actors are created only to run a process. Currently, VU provides the user with only one way of creating a process, that is by means of a script.

In the current architecture, one actor cannot inadvertently interfere with another actor. This is generally a virtue. But it also prevents an actor from intentionally signalling the other actors. The signal can be used to enable the actors to synchronize before attempting a time critical operation. An example of such a situation is two machines attempting to copy files to a common destination on a file server.

Inter-process Communication

This section describes the protocol which dictates the process used by an actor to communicate with another actor. The phrase "actor communicating with another actor" here means one actor's process communicating with another actor's process. Of course, both the actors could be following the same script. To start a conversation between two actors, both the actors have to be aware of each other's intent. This intent is conveyed to them by a master entity in the VU architecture called the **Director**. The director plays the role of the dispatcher in our protocol. The director is transparent to the script author. It is mentioned here only for the sake of explaining the message passing process.

In the following pages, we have an example of a session between two actors in VU. First we go through the steps in a typical session in English. This is followed by a VU script example.

Example

Consider the situation when Actor1 wants to convey the message "hello there" to another actor Actor2. Actor1 is following Script1 and Actor2 is following Script2.

■ Step 1

Actor1 : Declares the intent of wanting to talk with Actor2.

Director : Checks if Actor2 wants to listen to Actor2.

(If Actor2 wants to listen to Actor1, it declares its intent)

Actor2 : Declares the intent of wanting to talk with Actor1.

Director : Conveys to both actors that the communication session is open.

■ Step 2

Actor1 : Sends the text ‘hello there’ to the recipient actor Actor2.

Director : Delivers the message to Actor2.

(Now, if Actor2 wishes to receive a message from Actor1 then)

Actor2 : Receives the message string from Actor1.

■ Step 3

(Actor1 can now either close the session or keep sending more messages and/or receive messages from Actor2. Actor2 also can now either close the session or keep receiving more messages and/or send messages to Actor1)

Actor2 : Sends the message ‘howdy’

Director : Delivers the message to Actor1.

(Now, if Actor1 wishes to receive a message from Actor2 then)

Actor1 : Receives the message string from Actor2.

…

■ Step 4

(Actor1 wishes to close session)

Actor1 : Declares intent to close the communication session.

Director : Closes the session for Actor1 (that is, Actor1 cannot receive any messages from Actor2 and Actor2 cannot send any messages to Actor1 anymore.)

(For the purpose of symmetry, Actor2 must also close the session once it's done receiving all messages from Actor1)

Actor2 : Declares intent to close the session with Actor1.

This example illustrates the use of the protocol in one session. An actor can have as many open sessions as needed at any time. But only one session can be set up between a given pair of actors. All the communication is done asynchronously. That is all the operations (openSession, send, receive and closeSession) terminate without any waiting. There is no need for completion routines. Instead, the script can repeat these operations any number of times. Each of these operations is provided as a system task in the scripting language. Hence, they all have return values which indicate the success of the call.

The next page illustrates the use of the protocol in a real VU script example. These scripts make use of the new system tasks which are provided to perform the session operations. The semantics of these system tasks are listed after the example.

VU Script Example

Following are 2 example VU scripts which talk to each other. An actor named 'Elvis' is supposed to run Script1 while another actor named 'Jimmy' is to run Script2.

Script1 -

```
status := '';
recipient := 'Jimmy';
while  not(status = 'open')
begin
       println "not open yet now trying…";
       status := openSession([actor t:recipient]);
       println "Status = ", status;
end;
if (send([actor t:recipient],{"hi", "how","are","you?"}))
       println "Send successful";
x := '';
for i:= 1 to 5 do
begin
       x := receive([actor t:recipient]);
       println "Message received = ", x;
end;
closeSession([actor t:recipient]);
```

Script2 -

```
status := '';
recipient := 'Elvis';
while  not(status = 'open')
begin
```

```
       println "not open yet now trying…";
       status := openSession([actor t:recipient]);
       println "Status = ", status;
end;
if (send([actor t:recipient],{"hello", "how","are","you?"}))
       println "Send successful";
x := '';
for i:= 1 to 5 do
begin
       x := receive([actor t:recipient]);
       println "Message received = ", x;
end;
closeSession([actor t:recipient]);
```

System Tasks that Enable Message Passing

The following is a list of system tasks that are provided in the language to facilitate scripts to pass messages between actors (see Appendix F for further description):

1. **openSession**(recipientActor) - This system task call takes in a single parameter, an actor descriptor. It returns either undefined or a string with one of the following values: ‘open’ or ‘wait’. A session becomes open only when the value returned is ‘open’. If the value returned is ‘wait’ it means that the recipient actor has not performed a corresponding openSession yet. In such a case, you should retry the openSession (maybe after a while). Undefined is returned in error cases such as when there exists no match for the actor descriptor passed as the recipient.
2. **send**(recipientActor, listOfMessages) - This task takes two parameters. The first one is the descriptor of the recipient actor. The second is a list of messages. A message can be a regular expression value or a number or a variable with such a value. For example, “this is a {object}”, ‘value is xyz’, /failed/, object and 1200 are all valid messages (where ‘object’ is a variable with a value such as “ball”). The message is converted into a string and sent in the form of a string. Send returns a boolean which is true if the send was successful otherwise it returns false. Send can fail if the recipient actor was not found or if there was no open session with the recipient actor.

3. **receive**(senderActor) - This task takes one parameter, the descriptor of the actor from which the message is to be received. If such an actor exists and an unreceived message sent by that actor exists then that message is returned in the form of a string. If no unreceived message exists then a null string will be returned. An undefined value will be returned in cases where no such actor is found.

4. **closeSession**(recipientActor) - This task takes one parameter, the actor descriptor. It returns the string value ‘done’ if there exists such an actor and a session with it is open at this end (it may already be closed from the other end). Otherwise, the task returns undefined.

Chapter 8 How the Matcher Works

Matching is the process of finding the intended object to act upon. This chapter describes in detail how Virtual User finds the objects described in the script statements.

Introduction

Matching was introduced in the language design mainly to allow the user to find the state of the target. This concept has been extended to provide a means to get at the state of the whole environment (which includes the target and the actors, time, etc.). In the current state of the system we support matching for time, target, window, menu, menuitem, controls (including button, radiobutton, checkbox and scrollbar), edit text and static text. In this description of the process of matching, we will concentrate on the target end, which will be used more often. The same mechanism, however, is applied to non-target elements too.

From here on in this chapter, matching has been treated as the process of comparing a description of a target element with the actual elements existing on the target. Target element here, refers to the various interface objects on the target like windows, menus, controls, etc. To perform matching VU builds up a model of the target in its knowledge base. This model is built by peeking into the target through a target resident agent.

There are two distinct causes that invoke the matcher. First, any command which requests an action at the target end invokes the matcher. Some examples of this are:

```
select [menuItem t:"Quit"];
drag [window t:"Untitled"] a:{100, 100};
```

Second, the script can directly invoke the matcher through a match or collect expression.

For example:

```
first_window := match[window o:1];
all_menus := collect[menu];
```

In both cases the input to the matcher is a description of a target element from the script (that is, a descriptor). The objective of the matcher is to find a (or all) corresponding element(s) in the target model which meets the specified description. Now we will see how the search is undertaken.

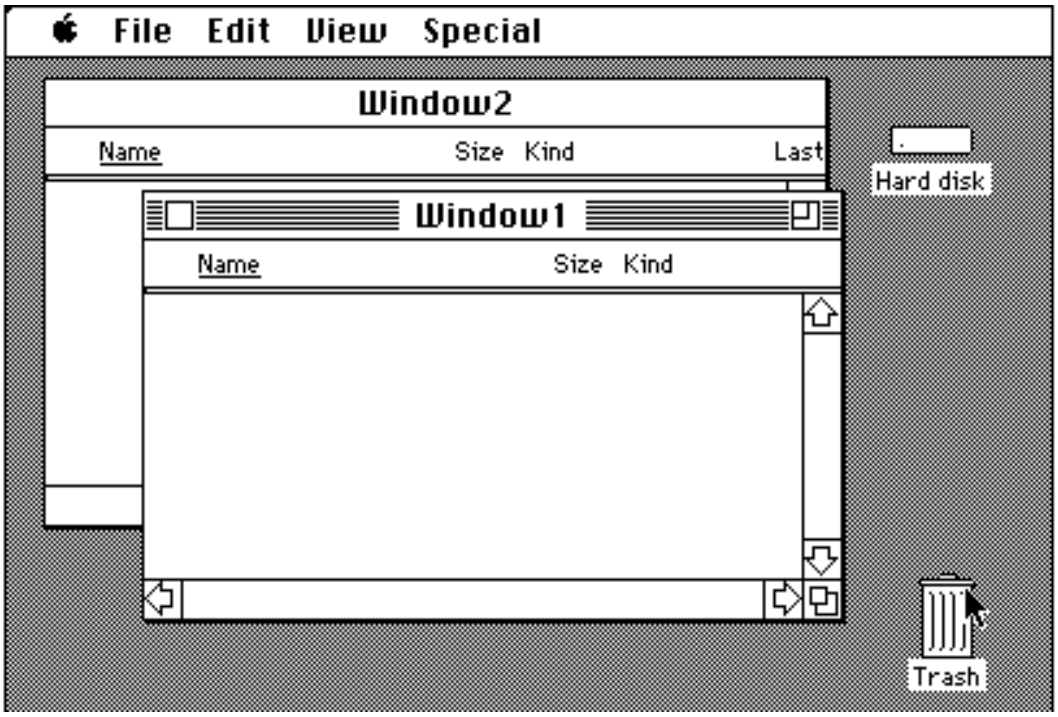
The Algorithm

As discussed earlier, a descriptor is made up of zero or more traits. A simple algorithm for finding the matching element is to look for an element of the specified type (say, window) whose traits are the same as the ones specified in the descriptor. This approach does not address many issues. Some issues which remain unsolved by this approach include:

1. how to handle situations where the descriptor is not complete, that is, it has fewer traits specified than those present in the candidate target elements
2. how to deal with partially specified trait values
3. are some traits more important for comparison than others; and if they are, how are their importance levels decided.

The matching scheme we use tries to solve these issues in the following way. First, if the descriptor is partially specified, the unspecified traits are not taken into consideration in the matching process. To elucidate the point, let us consider the situation where the target screen looks as shown in Figure 8-I.

Figure 8-I Matching Example



The target model has a window with the following description (in rough descriptor syntax):

[window title:"Window1" o:1 zoom:true closebox:true k: {[scrollbar…],[scrollbar…]}]

and the script provides a descriptor as follows:

```
select [window title:"Window1"];
```

It is expected that the window with the title "Window1" be chosen from the target model for selection (or for any other purpose that the context may demand). This expectation is justified despite the omission of several other characteristics of the window in the descriptor. The justification is that the script writer either did not consider the other traits important enough to be specified or that the writer did not know their values. By ignoring the unspecified traits about the window, in this example, the matcher will be able to pick the right window to select.

Second, the descriptor may contain a trait whose value has been partially specified. This feature is allowed only for certain traits like the ones which have a string value (titles etc.) and the ones which have a list value (content list of a window and menu items list of a menu). Continuing with our previous example, say the script includes the following statements:

```
select [window t:"Window"];
select [window t:"Window11"];
select [window t:"1Window"];
```

In the first case ("Window" Vs "Window1") the title string is not complete but 6 out of 7 characters are present and match correctly with the title in the model. The matcher recognizes the fact that there was a 86% (6/7) match for this trait.

The second case has a title ("Window11" Vs "Window1") with a 100% match and 1 extra non-matching character. The matcher treats this as a mismatch. The third case has exactly the same characters as the actual title but the ordering of characters are not correct. This one is also treated as a total mismatch. The order in which the characters occur is the most important factor. The second factor is that the there should not be any extraneous characters. A similar strategy is applied to a content list trait of a window and the menu items list of a menu. You can specify as many controls (menu items) as you know/care about and the matcher will get a percentage match from the corresponding model. But the order in which the list elements are specified is not a factor for these lists(since there is no well defined ordering for them).

For example,

```
select [window t:"Window1" k:{[scrollbar]}];
```

Here the 'k' trait gets a 50% match since the window has two controls (scroll bars are controls) but only one has been specified.

Finally, it remains to be decided if there is a difference in the significance of traits for the purpose of matching. Is the title of a window a more important trait than its having/not having a close box? The need for this decision arises when there are multiple candidates for the match which are very similar. For instance, in the figure the target has two windows with description as follows:

[window t: "Window1" o:1 …] (call it w1) and

[window t: "Window" o:2 …] (call it w2)

with most other traits (indicated here by …) being the same (all except the rectangle trait).

Consider the following statement in the script:

```
select [window t: "window1" o:2];
```

The matcher now faces the following choices:

1. say, 'sorry no match'
2. pick the window (w2) with rank 2, since its rank trait has a match (100%) and title has 86% match while for window w1 the title has a 100% match but rank has a 0% match. The cumulative match result of all the traits will give w2 more points than w1.
3. pick the window (w1) with title "Window1", since the title trait is more significant than the rank trait.

Choice one is correct, but serves little purpose. The second one seems to be more like what we would like to have, that is, accumulating the results of each trait comparison. On the other hand the third choice seems useful too, since the rank of a window is a volatile trait which will change after every selection/rotation action on windows. The script author may have made a mistake on that trait value. The author may have intended the window w1, since the title is exactly same and windows are more commonly referred to by name than rank. Alternatively, since the titles are very similar, the author may have spelled it wrong or may have made a plain mistake. The rank specification acts as confirmation to this. This would mean pick the second choice.

Our matcher tries to incorporate a combination of both the second and third strategies. Each trait of a descriptor is assigned a weight (significance level), based on common knowledge (very debatable, though). In our example, the title could have a weight 35 and the rank a weight 25. Before taking the cumulative match result we multiply each trait's result by its corresponding weight. So in this example, w1 gets a lower weighted cumulative result (100*35 + 0*25 = 3500) than w2 (86*35 + 100*25 = 5510). With this strategy the matcher would return window w2. The utility of this scheme will increase if the weights were not preassigned but user assignable. So in different contexts, the script writer may change the weights of the traits. Currently, VU doesn't provide this additional feature.

Another important mechanism of matching is the partitioning of traits into Significant and Non-significant groups. In our last example, consider if w1 had title "Window" and w2 had title "Folder". In this case, w1 will be the choice with our scheme discussed till now. But if the weights were different w2 might still be a viable candidate (because of its valid rank). To avoid such situations we mark some traits as Significant by giving them a weight higher than or equal to a Significance Threshold. In this case, if the Significance Threshold was 35 then the title falls in the significant group. The matcher always compares the traits which fall in this group first and if any of the traits in this group return a 0% match (no match) then that candidate is rejected with no consideration to the non-significant trait values. This feature provides a veto power to the Significant traits over the rest.

Appendix E gives a listing of the weights assigned to the traits of descriptors of various types.

Regular Expression Matching

Regular expressions are a shorthand language for specifying text patterns. Regular expressions can be used anywhere in place of strings. Its usefulness shows up mainly in specifying traits in descriptors, like the title of a window or the title of a menu. Regular expressions are always used within pattern delimiters "/" (example, /VU≈/). A special set of metacharacters, called regular expression operators, is used in regular expressions. The regular expression operators are listed in Table 8-I. The rest of this section describes the use of regular expressions.

Table 8-I Table of Regular Expression Operators

Operator	Meaning
c	Any character matches itself (unless it's one of the special characters listed below)
∂ c	Defeat the special meaning of the following character (c is taken literally) except ∂n = return and ∂t = tab
'…'	Literalize enclosed characters
"…"	Literalize enclosed characters, except ∂, and {
?	Any single character (other than a return)
≈	Any string of 0 or more characters that does not contain a return
[character…]	Any character in the list
[¬character…]	Any character not in the list (¬ is Option-L on the keyboard)
regularExpr*	Regular expression 0 or more
regularExpr+	Regular expression 1 or more regularExpr«n» Regular expression n times (« is Option-\ ; » is Option-Shift-\)
regularExpr«n,»	Regular expression n or more times
regularExpr«n1,n2»	Regular expression n1 to n2 times
(regularExpr)	Grouping(regularExpr)
regularExpr1regularExpr2	regularExpr1 followed by regularExpr2
•regularExpr	Regular expression at the beginning of a line
regularExpr ∞	Regular expression at the end of a line

These characters are considered special in the following circumstances:

∂	Special everywhere except within single quotation marks ('…')
? ≈ * + [« ( )	Special anywhere except within […], '…', and "…"
•	Special as the first character of an entire regular expression
∞	Special as the last character of an entire regular expression
/	Special if used to delimit a regular expression
¬	Special only after a left bracket, [
-	Special in brackets, except immediately following a left bracket, [

Their precedence (from highest to lowest) is as follows:

1.	( )
2.	? ≈ * + [ ] «
3.	• ∞

Character Expressions

In the simplest case, regular expressions consist of literal characters enclosed in slashes. For example:

```
/what the ?/
```

Notice one complication, however—if the literal character happens to be one of the regular expression operators (such as ?), it will be specially interpreted rather than taken as a literal character. If you want to specify a literal character that happens to have a special meaning within the context of regular expressions, you'll have to precede it with the escape character, ∂, or enclose it in quotation marks. The character ∂ has the effect of "literalizing" the character that follows it. For example, to find the literal expression given above, you would use one of the following forms:

/what the ∂?/ or /what the '?'/ or /'what the ?'/

You could also use double quotation marks, that is "what the ?".

Wildcard Operators

In addition to literal characters, regular expressions can include the operators ?, ≈ (Option-X), and [ ], which are used as follows:

?	Any character other than a return
≈	Any string not containing a return, including the null string (this is the same a ?*)
[characterList]	Any character in the character list (as defined below)
[¬ characterList]	Any character not in the list

A character list is an expression consisting of one or more characters enclosed in brackets ([…]). It matches any character found in the list. The case sensitivity of characters in the list is governed by the CaseSensitive (for Match) setting in the Commando dialog (ie., -cs option in the command line). A list may consist of individual characters or a range of characters, specified with the minus sign (–). For instance, the following two forms are equivalent:/[ABCDEF]/ /[A-F]/

You can also mix the two notations:/[0-9A-F$]/

This form specifies any of the characters 0 through 9, A through F, and $. To specify the] or - character, place it at the beginning of the list or literalize it with the escape character, ∂.

The negation symbol, ¬ (Option-L), lets you specify any character not in the list. For example: /[¬A-Z]/

This example specifies all characters except the letters A through Z. (To specify the ¬ character itself, place it anywhere in the list other than the beginning, or literalize it by preceding it with the escape character, ∂.)

Repeated Instances of Regular Expressions

The asterisk character (*) matches zero or more occurrences of the immediately preceding regular expression. The plus sign (+) matches one or more occurrences of an expression. For example, the form

```
/[0-9]+/
```

will find any string of one or more digits.

You can also specify an expression that occurs an explicit number of times by using the «n» notation:

regularExpr«n»	Regular expression n times
regularExpr«n,»	Regular expression at least n times
regularExpr«n1,n2»	Regular expression at least n1 times and at most n2 times

For example,

```
/' '«4,»/
```

This form specifies any string of four or more spaces.

Matching a Pattern at the Beginning or End of a Line

In the context of regular expressions, the • metacharacter (Option-8) means that the subsequent expression must be matched at the beginning of a line. For example, the regular expression:

```
/•main/
```

will match a string that begins with “main” but not a string that begins with “space main”. The beginning of a line is either the first character after a return or the first character of the file.

Likewise, the ∞ metacharacter (Option-5) means that the previous expression must be matched at the end of a string. The regular expression

```
/main∞/
```

will match a string that ends with “main” but not a string that ends with “main space ”. The end of a string is the last character of the string.

Inserting Invisible Characters

You can use the escape character, ∂ (Option-d), to insert the following special characters in text:

∂n	return
∂t	tab

Solving Matching Difficulties

What if a regular expression doesn't match what you intended? Ask yourself questions like these:

- Am I quoting special characters? For example, the (character is special. If you are matching for this character, then you must use "∂(".
- Do I remember the definitions of special characters? Review the special character definitions.
- Is my precedence and usage correct?
- Do the individual pieces match what I intended? Break the difficult expression down into small parts. Try each part separately to make sure that it does what you want. Then add each new, tested part to create more complicated expressions.

Appendix

A Grammar

Terminals that appear as they would in a script are shown in bold. Terminals whose token value appears in the script appear in outline. Nonterminals are bracketed between < and >. E is the empty string.

<script> ::=
    <statement_list>

<statement_list> ::=
    <statement_list> <statement> | E

<variable> ::=
    identifier |
    global identifier

<statement> ::=
    <executable_statement> |
    task identifier <formal_parameters> **begin** <task_statement_list> **end ;**

<formal_parameters> ::=
    (<parameter_list>) | E

<parameter_list> ::=
    <parameter> |
    <parameter_list> **,** <parameter> | E

<parameter> ::=
    identifier |
    identifier **:=** <expression>

<task_statement_list> ::=
    <task_statement_list> <task_statement>

<task_statement> ::=
    <executable_statement>
    return <expression> ; |
    global identifier ;

<executable_statement> ::=
    ; |
    <compound_statement> |
    <command> |
    identifier (<expression_list>) ; |
    <match_expression> ; |
    <variable> **:=** <expression> ;
    if <expression> <optional_do> <executable_statement> |
    if <expression> <optional_do> <executable_statement> **else** <optional_do> <executable_statement> |

while <expression> <optional_do> <executable_statement> |
for <variable> **:=** <expression> **to** <expression> <optional_step> <optional_do> <executable_statement> |
for each <variable> **in** <expression> <optional_do> <executable_statement> |
print <expression_list> **;** |
println <expression_list> **;** |
return <expression> **; |**
libraries <expresssion_list>**; |**
exit;

<optional_do> ::=
do | E

<optional_step> ::=
step <expression> | E

<compound_statement> ::=
begin <executable_statement_list> **end ;**

<executable_statement_list> ::=
<executable_statement_list> <executable_statement> | E

<command> ::=
<keyword> <expression> <argument_list> **;** |
<keyword> <argument_list> **;**

<keyword> ::=
click | **close** | **doubleClick** | **drag** | **move** | **pressKey** |
releaseKey | **pressMouse** | **releaseMouse** | **scroll** | **select** |
size | **type** | **zoom**

<argument_list> ::=
<argument_list> identifier **:** <expression> | E

<descriptor> ::=
[<descriptor_type> <trait_list> **]** |
[ ]

<descriptor_type> ::=
actor | application | **button** | **checkBox** | **contentItem | control** | **editText** |
icon | keyboard | menu | **menuItem** | **mouse | picture | popup** | **radioButton** |
scrollBar | **staticText** | **screen | system | target** | **time** | **userItem | window**

<trait_list> ::=
<trait_list> <trait> | E

<trait> ::=
identifier **:** <trait_expression>

<trait_expression> ::=

<expression> |
? <variable> |
$ <variable> |
? <variable> **:** <expression> |
$ <variable> **:** <expression> |
? <variable> **: $** <variable> |
? <variable> **: $** <variable> **:** <expression>

<match_expression> ::= **match** <expression>

<expression> ::=
integer |
<variable> |
<descriptor> |
regular_expression |
symbolic_identifier |
<expression> **.** identifier |
<expression> **[** <expression> **]** |
<match_expression> |
collect <expression> |
<expression> **!** |
(<expression> **)** |
<list_of_expressions> |
<expression> **+** <expression> |
<expression> **-** <expression> |
<expression> ***** <expression> |
<expression> **/** <expression> |
<expression> **mod** <expression> |
- <expression> |
+ <expression> |
not <expression> |
<expression> **=** <expression> |
<expression> **<>** <expression> |
<expression> **~=** <expression> |
<expression> **<** <expression> |
<expression> **>** <expression> |
<expression> **<=** <expression> |
<expression> **>=** <expression> |
<expression> **and** <expression> |
<expression> **or** <expression>

<list_of_expressions> ::=
{ <expression_list> **}**

<expression_list> ::=
<expression> |
<expression_list> **,** <expression> | E

B Reserved Words

The following is a list of VU reserved words. This list does not include symbolic values listed in Appendix D. Symbolic values (symbols) are also reserved. Reserved words will never contain an underscore. If your variable names or task names contain an underscore, it will never conflict with a reserved word.

acquireTarget	actor	actorName	and
application	assoc	begin	button
callpp	card	checkBox	click
close	closeBox	closesession	collect
contentItem	control	dialog	do
doubleClick	drag	each	editText
else	end	exit	for
getIndString	getString	global	icon
if	in	insert	isMember
isUndefined	keyboard	match	menu
menuItem	mod	mouse	mouseSpeed
move	not	openSession	or
pageDown	pageUp	patience	picture
popup	pressKey	print	println
radioButton	random	receive	releaseKey
releaseMouse	releaseTarget	remove	replace
return	screen	scroll	scrollBar
select	send	shadow	size
sizeBox	staticText	step	system
target	task	time	to
trace	type	typeOf	typeSpeed
upArrow	userItem	wait	while
window	zoom	zoomBox	

The following is a list of words reserved for future use. The compiler will warn of, but not prohibit, their use for now.

absolute	acquire	agent	beep
bitmap	break	case	continue
cursor	director	evaluate	event
execute	extern	external	feature
file	finderItem	font	generic
goto	onFailure	persist	play
playBack	record	recording	relative
release	script	static	tearoff
text	universal	view	windoid

C Operator Precedence

			( )			
		.		**[ ]**		
			!			
-	+	**not**		**match**	**collect**	**card**
		*****	**mod**	**/**		
		+		**-**		
=	<>	<	>	<=	>=	~=
		and		**or**		

D Symbolic Identifiers

The following is a list of symbolic values available in the language. Symbolic values (symbols) are also reserved words in the language. Appendix B lists the remaining reserved words.

aextendkbd	backspaceKey	capslockKey	clearKey
commandKey	controlKey	da	delKey
dialog	document	downarrowKey	endKey
enterKey	escapeKey	helpKey	homeKey
extisoadbkbd	macandpad	macpluskbd	f1Key
f2Key	f3Key	f4Key	f5Key
f6Key	f7Key	f8Key	f9Key
f10Key	f11Key	f12Key	f13Key
f14Key	f15Key	false	leftarrowKey
machunknown	macii	maciici	maciicx
maciifx	maciix	macplus	portable
se	se30	a512ke	optionKey
pagedownKey	pageupKey	plain	portadbkbd
portisoadbkbd	standadbkbd	stdisoadbkbd	returnKey
rightarrowKey	shadow	shiftKey	tabKey
true	undefined	uparrowKey	unknownkbd

E Trait Weights

Actor Descriptors

name	t	80
target	u	20

Application Descriptors

text	t	100

ContentItem Descriptors (EditText/StaticText/Icon/ Picture/UserItem)

text	t	35
enabled	e	25
rectangle	r	20
window owner	w	20

Control Descriptors

text	t	35
setting	s	15
enabled	e	15
highlite state	h	15
rectangle	r	10
window owner	w	10

Keyboard Descriptors

type	t	50
keyscript	s	50

Menu Descriptors

title	t	50
rank	o	30
enabled	e	10
items list	i	10

Menu Item Descriptors

text	t	35
rank	o	25
owner	m	12
keyboard equivalent	k	10
check (mark) character	c	9
enabled	e	8
submenu	h	1

Mouse Descriptors

position	p	50
button state	b	50

Popup Descriptors

text	t	30
window owner	w	30
rectangle	r	10
enabled	e	10
items list	i	10
setting	s	10

Screen Descriptors

main screen	m	70
rectangle	r	30

System Descriptors

version	v	50
system script	s	50

Target Descriptors

text	t	40
mouse	m	10
screen list	s	10
application list	a	10
name	n	10
keyboard	n	10
RAM(memory)	r	5
zone	n	5

Time Descriptors

year	y	20
month	m	20
day	d	20
hour	h	20
seconds	s	20

Window Descriptors

title	t	35
rank	o	25
style	s	10
close box	c	6
contents list	k	6
grow box	g	6
rectangle	r	6
zoom box	z	6

△ **Important** **At present the Significance Threshold has been set to 35. Hence all traits whose weight equal/exceed 35 fall in the category of Significant traits. To remind you, if in a match a significant trait fails to match, then the match will fail, with no consideration to other traits.** ▵

F System Supplied Tasks

The following is a list of interfaces to system supplied tasks available:

task acquireTarget(targetName, zone := '*') begin
 # black box
end;

- does a lookup for a machine with the specified targetname on the network in the given zone. If zone parameter is not supplied, the local zone is assumed. If such a machine is found with an Agent VU registered on the network then the current actor is given the control of that target. This task returns a number indicating the result of the acquire. Acquire is successful only if you get back a 0 as the return code.

return codes :

undefined	bad arguments passed to the task(an error message will accompany)
0	successful
1	Target failure (like target not responding, network failure etc.)
2	Actor already has a target under control; an actor is allowed only one target
3	Target with the specified name not registered on the network
4	Target has other host (one target cannot have two controlling hosts)

example usage:

`acquireTarget("Target1")` - this will acquire a target with chooser name 'Target1' in the current/local zone.

`acquireTarget('Target2', 'Grace Land')` - this will acquire a target with chooser name 'Target1' in the zone 'Grace Land'.

task actorName(newName :=(match[actor]).t) begin
 # black box
end;

- sets the name of the actor executing the script to the new name passed as a parameter. If no parameter is passed, the actor name remains unchanged. This task returns the current actor name (before the change). The following changes the actor name to "Elvis" and returns the actor name that existed before the change

```
prev_name := actorName("Elvis");
```

The following statement prints the current actor name:

```
println actorName();
```

task assoc(object, list) begin
 # black box
end;

This task is equivalent to the 'assoc' in Lisp. It gets the value associated with the 'object' in the association list provided in the 2nd argument 'list'.

Association List: A list of the form: {{<object1>,<value1>},{<object2>,<value2>},...}
where <object> and <value> are any expressions.

The <value> associated with the given 'object' is returned by the task. If there are multiple associations for the given 'object' then the <value> in the first association is returned. If there is no association is found then the value returned is false.

example usage:

```
bobs_age := assoc("bob", {{"art", 0}, {"bob", 5}, {"tom", 10}});
```

bobs_age is assigned to 5 (the value associated with the string "bob")

task closeSession(recipientActor) begin
 # black box
end;

- takes one parameter, the actor descriptor. It returns the string value 'done' if there exists such an actor and a session with it is open at this end (it may already be closed from the other end). Otherwise, the task returns undefined.

task descType(descriptor) begin
 # black box
end;

- takes one parameter, a descriptor. It returns the string value denoting the descriptor type if there exists such a descriptor in the language. Otherwise, the task returns undefined. The section on Descriptors enumerates all the descriptor types available in the language.

example usage:

```
d_type := descType([window t: 'Untitled-1' o:1]);
```

d_type is assigned to the string 'window'.

task getIndString(strListId, index, filename := (match[actor]).scriptfile) begin
 # black box
end;

- returns a string from the string list resource (resource type ‘STR#’) that has the resource ID as strListID. This task reads the string list resource from the resource file specified. It returns the string specified by the index parameter, which can range from 1 to the number of strings in the list. If the resource can't be read or the index is out of range, undefined is returned. If no filename is specified the resource is read from the resource fork of the script file.

task getString(strId, filename := (match[actor]).scriptfile) begin
 # black box
end;

- returns the string associated with the string resource (resource type ‘STR ’) with the given resource ID. It reads the string resource from the resource file. If the resource can't be read, getString returns undefined. If no filename is specified the resource is read from the resource fork of the script file.

◆ *Note*: If your application uses a large number of strings, storing them in a string list in the resource file will be more efficient. You can access strings in a string list with GetIndString, as described above.

task insert(element, position, list) begin
 # black box
end;

- this task inserts an 'element' at 'position' in the ‘list’ and returns a newly formed list. The cardinality (size) of the returned list is one more than that of ‘list’. The argument ‘list’ remains unchanged after the call.

◆ *Note* : 'position' must be an integer > 0, else undefined is returned.

example usage:

```
list := {‘a’, ‘b’, ‘d’, ‘e’};
newlist := insert(‘c’, 3, list);

# newlist now becomes {‘a’, ‘b’, ‘c’, ‘d’, ‘e’}
```

task isMember(element, list) begin
 # black box
end;

- checks if the 'element' is a member of the list contained in the 2nd argument 'list'. The comparison criteria for the membership test is same as that for the ‘=’ operator. This task returns true if the element is a member and false if it isn’t.

example usage:

```
list := {‘a’, ‘b’, ‘d’, ‘e’};
```

```
isMember('c', list); # returns false
isMember('d', list); # returns true
```

task isUndefined(expression) begin
 # black box
end;

- checks if the 'expression' evaluates to undefined. This task returns true if the expression evaluates to undefined and false if it doesn't.

example usage:

```
str := getIndString(1000, 128);
# getIndString returns undefined if no such string is found
# now:
isUndefined(str); # will return false if a string was found
isUndefined(str); # will return true if no string was found
```

task mouseSpeed(newSpeed := (match[actor]).mousespeed) begin
 # black box
end;

- sets the mouse speed to the speed passed as a parameter. If no parameter is passed, the mouse speed remains unchanged (do not worry about the default expression for now). This task returns the current mouse speed (before the change). The following changes the mouse speed to 10 and returns the mouse speed that existed before the change:

```
prev_speed := mouseSpeed(10);
```

The following statement prints the current mouse speed:

```
println mouseSpeed();
```

◆ *Note*: Mouse movements are made in distinct steps. The mouse speed is expressed as a positive number which represents pixels per step. Passing 0 tells the Virtual User that any mouse movements should be done in one step (as opposed to 0 pixels per step which would get you nowhere).

task openSession(recipientActor) begin
 # black box
end;

- this system task call takes in a single parameter, an actor descriptor. It returns either undefined or a string with one of the following values: ‘open’ or ‘wait’. A session becomes open only when the value returned is ‘open’. If the value returned is ‘wait’ it means that the recipient actor has not performed a corresponding openSession yet. In such a case, you should retry the openSession (maybe after a while). Undefined is returned in error cases such as when there exists no match for the actor descriptor passed as the recipient.

example usage:

```
status := '';
while not (status = 'open')
begin
     status := openSession([actor t:‘recipient’]);
end;
```

task patience(newSetting := (match[actor]).p) begin
 # black box
end;

- sets the patience to the setting passed as a parameter. The parameter may be any positive integer value. VU sets the patience to 1 at script startup time unless it is set differently from the command line for the particular target. If no parameter is passed, the patience remains unchanged (do not worry about the default expression for now). This task returns the current patience setting (before the change).

The patience of VU determines the duration of pauses that VU goes through while performing any action on the target. Since the performance (speed) of all target applications are not the same and its not currently possible for VU to assess the performance of the target, we want the script writer/tester to help VU. An example, of its utility is in dealing with systems and/or applications which are in early stages of development.

You can deal with these problems by increasing the patience of VU. How do you determine Patience for VU? The best way is experimentation. There is no well defined way to figure out the patience level you will need to set in your scripts. We suggest that you run the scripts with the default setting (1) and if that fails try increasing the patience by one and keep repeating this process until things work smoothly.

The following changes the patience to 3 and returns the patience setting that existed before the change:

```
save_patience := patience(3);
```

The following statement prints the current patience setting:

```
println patience();
```

task random(low_bound := 0,hi_bound := 32767) begin
 # black box
end;

- returns a random number in the range from low_bound to hi_bound. By default, random returns a number in the range from 0 to 32767. For example if you wanted to print a random number between 1 and 10, the following statement could be used:

```
println random(1,10);
```

task receive(senderActor) begin
 # black box
end;

- takes one parameter, the descriptor of the actor from which a message is to be received. If such an actor exists and an unreceived message sent by that actor exists then that message is returned in the form of a string. If no unreceived message exists then a null string will be returned. An undefined value will be returned in cases where no such actor is found.

example uasge:

```
msg := recieve(from_actor);
```

task releaseTarget() begin
 # black box
end;

- used to release a target during a script execution. This allows the script to either acquire a new target later or run without a target. The release also enables another host(another VU actor or an external application) to take control over the target. Note that a target can have only a single host at this time.

return codes :

undefined	bad arguments passed to the task(an error message will accompany)
0	successful
1	target failure (like target not responding, network failure etc.) , release failed
3	target with the specified name not registered on the network (released anyway)
4	target has other host (released anyway)
5	no target under control (so no release needed)

If releaseTarget returns 1 then VU has not released the target (Agent VU). This means you cannot do an acquireTarget on this failure. You may have to retry releasing till you get one of the other return codes.

example usage:

```
        if (releaseTarget() = 1)
           println "failed to release. try again";
# this will release the target which the actor has acquired earlier
# else if release fails print error message.
```

task remove(position, list) begin
 # black box
end;

- this task removes the element at 'position' in the 'list' and returns a newly formed list. The cardinality(size) of the returned list is one less than that of 'list'. The argument 'list' remains unchanged after the call.

example usage:

```
        list := {'a', 'b', 'c', 'd'};
        newlist := remove(3, list);
        # newlist now becomes {'a', 'b', 'd'}
```

◆ *Note* : 'position' must be an integer > 0, else undefined is returned.

task replace(element, position, list) begin
 # black box
end;

- replaces the element at 'position' in the 3rd argument 'list' with 'element' and returns a newly formed list. The cardinality (size) of the list remains unchanged. The argument 'list' remains unchanged after the call.

example usage:

```
        list := {'a', 'b', 'd', 'e'};
        newlist := replace('c', 3, list);
        # newlist now becomes {'a', 'b', 'c', 'e'}
```

◆ *Note*: 'position' must be an integer > 0 and <= card 'list', failing which undefined is returned.

task send(recipientActor, listOfMessages) begin
 # black box
end;

- this task takes two parameters. The first one is the descriptor of the recipient actor. The second is a list of messages. A message can be a regular expression value or a number or a variable with such a value. For example, "this is a {object}", 'value is xyz', /failed/, object and 1200 are all valid messages (where 'object' is a variable with a value such as "ball"). The message is converted into a string and sent in the form of a string. Send returns a boolean which is true if send was successful, otherwise it returns false. Send can fail if the recipient actor was not found or if there was no open session with the recipient actor.

example usage:

```
recipient_actor := [actor t:'joe'];
msg_list := {"hi john?", "how are you?"};
status := send(recipient_actor, msg_list);
```

task trace(traceSetting := (match[actor]).t) begin
 # black box
end;

- turns debug trace on/off. If the parameter passed evaluates to true, trace is turned on. If it evalutes to false, trace is turned off. If no parameter is passed, the trace setting remains unchanged (Do not worry about the default expression for now. It specifies that the speed should remain unchanged). This task returns the current setting (before the change).

◆ *Note*: tracing is only visible within the log file (if specified). You can watch the trace (or traces for multiple targets) as the script(s) execute if you have a window open for the log file(s).

task typeOf(expression) begin
 # black box
end;

- returns the type of value that the 'expression' evaluates to. This task returns a string denoting the type.

example usages:

```
type := typeOf(/Untitled≈/);
# type is now assigned the value 'regularExpression'

type := typeOf("Untitled-1");
# type is now assigned the value 'string'

type := typeOf([menuItem t: 'open' m:[menu t: 'File']]);
# type is now assigned the value 'descriptor'
```

The various possible return values are:

'descriptor'
'integer'
'list'
'regularExpression'
'string'
'symbol'
'undefined'

task typeSpeed(newSpeed := (match[actor]).k) begin
 # black box
end;

- sets the rate at which the Virtual User types at the keyboard. If no parameter is passed, the speed remains unchanged (Do not worry about the default expression for now. It specifies that the speed should remain unchanged). This task returns the current type speed (before the change). The following changes the mouse speed to 5 and returns the type speed that existed before the change:

```
prev_speed := typeSpeed(5);
```

The following statement prints the current type speed:

```
println typeSpeed();
```

- *Note*: the type speed is to be interpreted as the maximum number of characters the Virtual User will type in a given second. The number must be positive and represents a maximum number of characters to be typed because the Virtual User cannot maintain a constant rate of keystroking due to uncertainty in network traffic.

task wait(seconds := 0,minutes := 0,hours := 0) begin
 # black box
end;

- waits/pauses for the specified amount of time.

Index