Rational. System Architect

Usrprops Extensibility Guide

IBM Rational

System Architect USRPROPS Extensibility Guide

Release 11.3.1.2

Before using this information, read the "Notices" in the Appendix, on page 5-1.

This edition applies to IBM® Rational® System Architect®, version 11.3.1.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

© Copyright IBM Corporation 1986, 2010
US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Table of Contents

Table of Contents i

Extending a Rational System Architect Encyclopedia's Metamodel **1-1**
Extending *Rational System Architect* 1-2
Rational System Architect's Encyclopedia Metamodel 1-3
How to Modify the Metamodel 1-8
Selecting the Diagram and Property Sets for an Encyclopedia 1-10

Modifying the Metamodel with USRPROPS.TXT **2-1**
Accessing and Editing the USRPROPS.TXT File 2-3
Composition and Syntax 2-7
Grouping Commands to Create Modeling Elements 2-10
Dialog Controls 2-14
Ordering and Laying Out USRPROPS.TXT Changes 2-19
Example of Making Changes to USRPROPS.TXT 2-23
Defining a LIST of Values 2-29
Renaming Existing Diagram, Symbol, or Definition Types 2-31
Creating New Diagram, Symbol, or Definition Types 2-37
Assigning a Symbol Type to a Diagram Type 2-39
Assigning a Line Symbol Type to a Diagram Type 2-40
Limitations of Assigning a Symbol Type to a Diagram Type 2-41
Assigning a Definition Type to a Symbol Type 2-43
Depicting a Symbol with a Bitmap or Metafile 2-44
Specifying Depiction Files for New Encyclopedias 2-48
User-Defined Symbol Presentation Based on Property Value 2-50
Specifying Properties for Diagrams, Symbols, and Definitions 2-54
Specifying Properties for Diagram Types 2-56
Specifying Properties for Symbol Types 2-58
Specifying Properties for Definition Types 2-62
Property Statements 2-65
Using ListOf, OneOf, and ExpressionOf 2-69
ListOf 2-70
OneOf 2-73
ExpressionOf 2-74
ZOOMABLE Command 2-76
Modifying the Aesthetic Look of Dialogs 2-78
LAYOUT Command 2-79

Creating Tabs with the CHAPTER Command 2-89
GROUP Command 2-91
Positioning Controls and Labels 2-94
Specifying the Display of Values on Symbols 2-100
Syntax of the DISPLAY Command 2-103
Specifying Key and Keyed By Properties 2-108
Examples of Key and Keyed By 2-115
Hiding Standard Entries in the SAPROPS.CFG File 2-124
Error Messages 2-126
Runtime Edits 2-129

USRPROPS.TXT Keywords 3-1
USRPROPS Keywords 3-2

IBM Support 4-1
Contacting IBM Rational Software Support 4-2

Appendix 5-1
Notices 5-2
Trademarks 5-5
Index iii

1

Extending a System Architect Encyclopedia's Metamodel

Introduction

This chapter introduces the mechanisms to extend a IBM® Rational® System Architect® encyclopedia's metamodel through USRPROPS.TXT.

Topics in this Chapter	Page
Extending Rational System Architect	1-2
Rational System Architect’s Encyclopedia Metamodel	1-3
How to Modify the Metamodel	1-8

Extending Rational*System Architect*

Rational System Architect can be extended and customized in many ways. Its drawing behavior can be customized through a variety of selections made in the tool and the sa2001.ini file. It's toolbars may be customized, its Matrix Editors may be customized, its reports may be customized, and so forth. Rational System Architect also has built-in support for Microsoft Visual Basic for Applications, which enables the user to write native macros that can run inside Rational System Architect to do all sorts of things, such as adding useful utilities, or even effecting the behavior of the tool.

Extending the Metamodel through USRPROPS.TXT

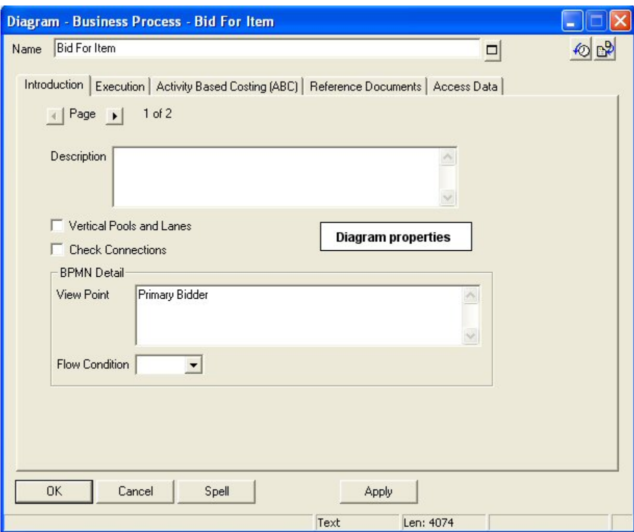
In addition to all of that, one of the most powerful features of Rational System Architect is that users may also tailor and extend the underlying metamodel of how information is stored in an encyclopedia. The default metamodel of a Rational System Architect encyclopedia is specified in a file called SAPROPS.CFG (the main **S**ystem **A**rchitect **prop**erties file), which controls things like the symbols that are on diagrams, the relationshiip between symbols and their definitions, and the properties of symbols, definitions, and diagrams. User modifications to the metamodel are specified in a text file called USRPROPS.TXT, which, when an encyclopedia is loaded, is parsed along with SAPROPS.CFG to create a SAPROPS.BIN file. USRPROPS.TXT overrides SAPROPS.CFG. You may edit the USRPROPS.TXT file to customize or extend the metamodel of an encyclopedia using a scripting language native to Rational System Architect.

Rational System Architect's Encyclopedia Metamodel

What the Metamodel Provides

The metamodel is a model of the way Rational System Architect stores the diagrams, symbols, and definitions that you create while you're doing your work. Rational System Architect's metamodel includes all diagram types, symbol types, and definition types, the properties that each of those types contains, and various relationships between these modeling elements.

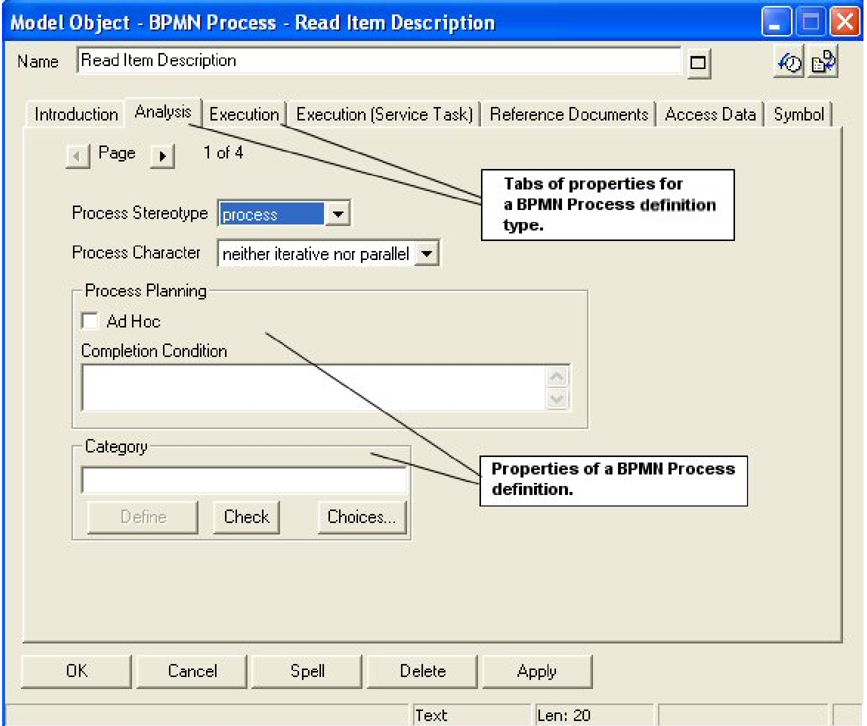
An example of a *diagram type* is a Business Process diagram; it has *properties* such as options whether or not to show pools and lanes horiztonally or vertically (Vertical Pools and Lanes), whether or not to automatically check line-symbol connections on the diagram as you draw (Check Connections), etc.



An example of a *symbol type* is a BPMN Process symbol. A BPMN Process symbol is drawn on a Business Process diagram – the diagram *contains* symbols, and symbols are *contained in* a diagram – an example of two of the many relationships in the encyclopedia metamodel. An example of

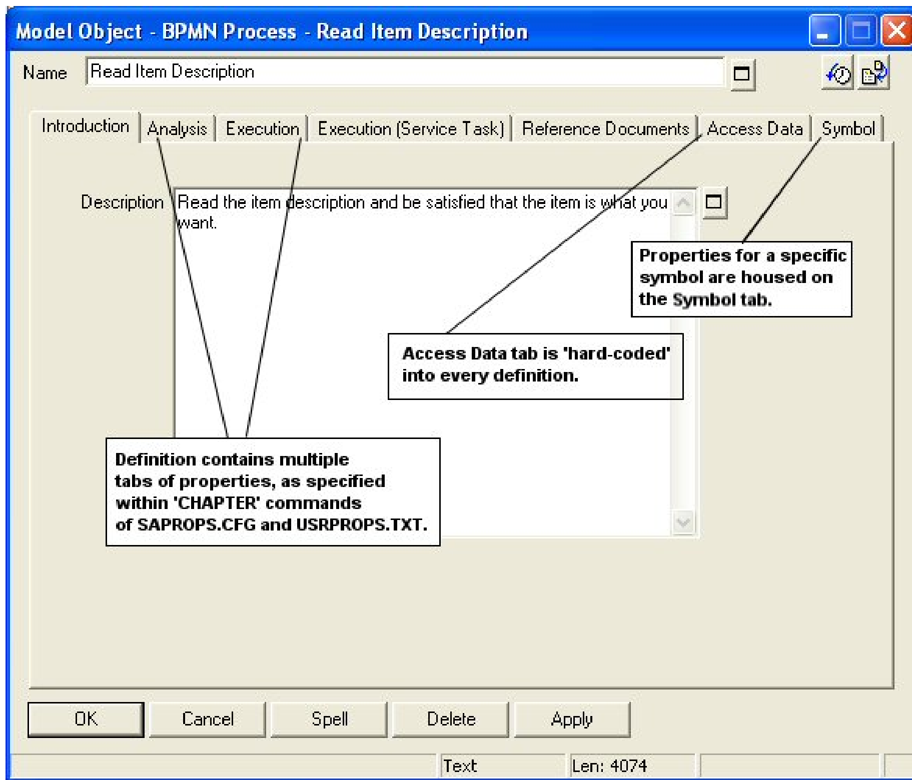
a *definition type* is a BPMN Process definition. A BPMN Process symbol graphically represents a BPMN Process definition. Most definitions are represented by a symbol; some are not – attribute or method definition types, for example, are not represented by any symbol on any diagram. They are both *included in* (another relationship) a class definition type.

Similar to a diagram type, each definition type contains properties. If you open a definition from Rational System Architect’s explorer, you will see those properties in the definition’s dialog, categorized into appropriate tabs and groups.

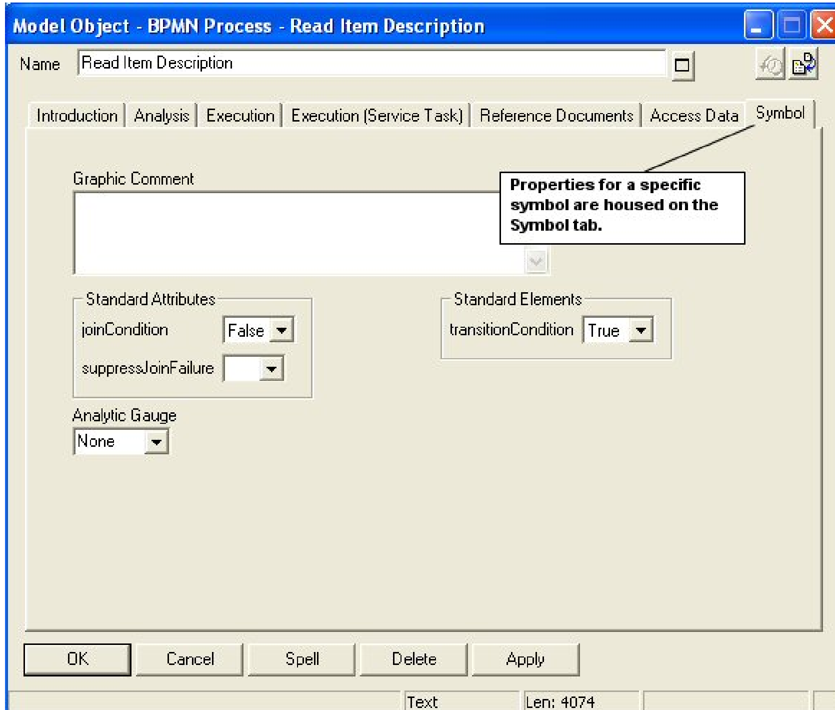


Similar to a diagram type and a definition type, each symbol type contains properties. If you open a symbol’s underlying dialog (double click on the symbol on a diagram workspace, or right-mouse click on it and choose Edit, or select it and choose Edit, symbol-type), you will see the properties of the

underlying definition that the symbol represents (the same as those presented if you opened the definition from the explorer), **and** you will also see an additional Symbol tab.



The Symbol tab provides properties specific to the symbol.



Each symbol that you draw on a diagram is a separate instance that points to the same definition. So you may draw a Process symbol named Read Item Description on one Business Process diagram and color it red, and draw another Process symbol named Read Item Description on another Business Process diagram and color it green. If you make a change to the Read Item Description definition (add a word to its Description property, for example), that change will be reflected when you open the definition of either the red or the green Read Item Description symbol. Two separate symbols – one underlying definition.

Relationships between and amongst diagram types, symbol types, and definition types can be complex. For instance, in Rational System Architect, a class diagram belongs to the package that it is created in. There is a ‘belongs to’ relationship between a class and a package. What’s more, a class is ‘keyed to’ the package it belongs to. The ‘keyed to’ relationship provides uniqueness to a class’s namespace –

you can have a class Person in a Human_Resources package that has completely different contents than a class Person in a Hotel_Reservation package. So 'keyed by' is another relationship that exists between a class and a package. Similarly, a method belongs to a class which belongs to a package. A method is also keyed by its class which is keyed by its package. Moreover, the user may create child diagrams (such as a State diagram) for class symbols on that class diagram. In this case a State diagram 'is child of' a class symbol – yet another relationship.

How to Modify the Metamodel

Rational System Architect has been delivered to you with a preset metamodel of diagrams, symbols, definitions, properties, and relationships. You may accept this metamodel as is, or extend or tailor it to suit your modeling needs. Tailoring includes changing what is already provided, or adding your own new diagram types, symbol types, and definition types.

Physical Makeup of an Encyclopedia's Metamodel – SAPROPS.CFG and USRPROPS.TXT

Each Rational System Architect encyclopedia has its metamodel specified by two files: SAPROPS.CFG (the System Architect Property configuration file) and USRPROPS.TXT (the User Properties file). These two files reside in the FILES table of each encyclopedia.

The SAPROPS.CFG file contains the default metamodel specified by IBM for each encyclopedia used with a particular version of the product. The USRPROPS.TXT file by default is an empty file, except for some comment (REM, or reminder) statements. Users add code to the USRPROPS.TXT file to modify the metamodel.

When Rational System Architect opens an encyclopedia, it parses the SAPROPS.CFG file, and then parses the USRPROPS.TXT file to create an SAPROPS.BIN file. Whatever is specified in USRPROPS.TXT overrides or is added to the SAPROPS.CFG specification in creating the SAPROPS.BIN file. It is the SAPROPS.BIN file that is used to present the metamodel to the user.

There are a few important items of the metamodel that you cannot override in SAPROPS.CFG using USRPROPS.TXT:

- You cannot remove a LIST or LISTONLY reference which has been defined in SAPROPS. However you can modify the text to be displayed in the list or listbox.
- You cannot remove a label which has been defined in SAPROPS. However you can modify the text to be displayed in the label.

The 'Master' SAPROPS.CFG & USRPROPS.TXT Files

In addition to residing in the FILES table of each encyclopedia, a 'master' copy of the SAPROPS.CFG file and the USRPROPS.TXT file are also provided within Rational System Architect's main executable directory (usually <C>:\Program Files\IBM\Rational\System Architect Suite\11.3.1\System Architect). When an encyclopedia is created for the first time, the 'master' SAPROPS.CFG and USRPROPS.TXT files residing in Rational System Architect's executable directory are automatically placed in its Files table. Therefore, if you change the contents of the USRPROPS.TXT file in the main Rational System Architect directory, you will be changing the metamodel of all new encyclopedias that you create. As a result, many people make sure the 'master' USRPROPS.TXT in the main Rational System Architect executable directory has all properties required for their company and project standards.

Initially, the 'master' USRPROPS.TXT file is essentially empty – it contains only some remarks at the head of the file, prefaced by a **REM** (reminder, or comment) command.

CONFIG.PRP File

Rational System Architect provides you with a third file, called CONFIG.PRP, which is an exact copy of SAPROPS.CFG. CONFIG.PRP is located in Rational System Architect's executable directory (usually <C>:\Program Files\IBM\Rational\System Architect Suite\11.3.1\System Architect). CONFIG.PRP is provided so that you can view, cut and copy the commands and properties that are also in SAPROPS.CFG without having to worry about accidentally disturbing SAPROPS.CFG itself. You can cut or copy commands from the CONFIG.PRP file and paste them into the USRPROPS.TXT file, and then make modifications.

Selecting the Diagram and Property Sets for an Encyclopedia

Besides modifying the metamodel via USRPROPS.TXT, you may also select what diagram and property sets are turned on for an encyclopedia at any given time via the Rational System Architect Property Configuration dialog (accessed by selecting Tools, Customize Method Support, Encyclopedia Configuration).

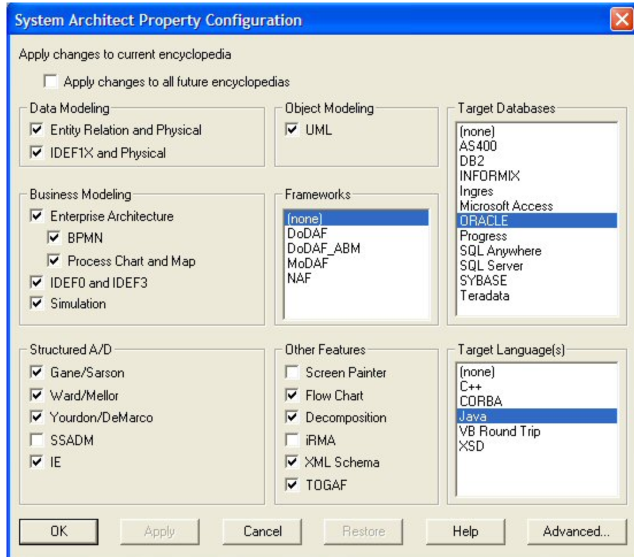


Figure 1-1. Project Configuration Dialog: choose the diagram types, and the other useful diagrams for this encyclopedia.

You may toggle on or off diagram sets and property sets, and click on the Advanced button in this dialog to make further refinements of what diagram and property sets are active in an encyclopedia.

SADECLAR.CFG

The selections you make in the Property Configuration dialog directly affect the contents of the SADECLAR.CFG file, which is housed in the Files table of every encyclopedia. This file, in turn, is referenced by #IFDEF (note: there is no space between the ‘#’ and the ‘IF’) statements in the SAPROPS.CFG and USRPROPS.TXT files. For example,

the SAPROPS.CFG file contains # IFDEF's for UML – if the UML modeling method is toggled on (in the above dialog and therefore in SADECLAR.CFG), then # IFDEF's in SAPROPS.CFG will turn on or off appropriate properties for UML diagrams.

As the picture below shows, selections for diagram and property sets that you make in the Property Configuration dialog (which toggle on or off choices in SADECLAR.CFG), in effect, filter the SAPROPS.CFG file properties in use for the encyclopedia. User modifications that you make to USRPROPS.TXT are parsed on top of the filtered SAPROPS.CFG file, to produce an SAPROPS.BIN file that provides the metamodel for an encyclopedia as Rational System Architect is running. Whenever you bring up a property or definition dialog, or run a report, Rational System Architect goes to SAPROPS.BIN to find the relevant properties of the model element you are defining.

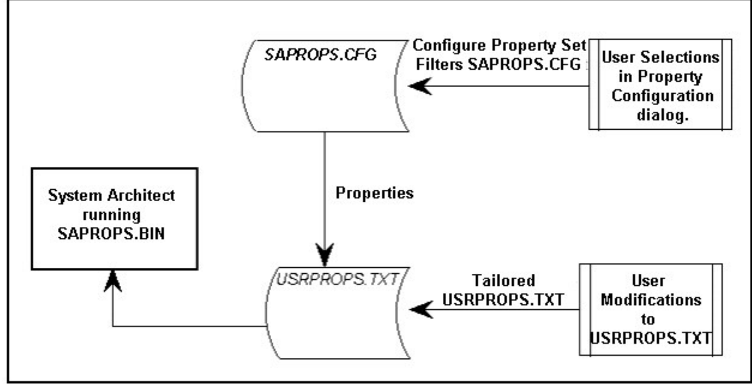


Figure 1-2. The relationship between SAPROPS.CFG, USRPROPS.TXT, and the user's choice of diagrams, properties, and modeling technique.

You may export the SADECLAR.CFG file from the Files table of an encyclopedia, and open it using any text editor to see the specific property sets that are available to use as switches for # IFDEF statements in USRPROPS.TXT.

Some do not totally match the words/labels used in the Property Configuration dialog. For example, the Enterprise Architecture choice is actually called Business Enterprise in SADECLAR.CFG. So a #IFDEF "Enterprise Architecture"

statement in USRPROPS.TXT would be meaningless and cause a parsing error; the correct statement should be #IFDEF “Business Enterprise”.

Figure 1-3. Contents of SADECLAR are used for # IFDEF switches in USRPROPS.TXT.

```
DECLARE "Business Enterprise" UNDEFINED
DECLARE " UML Class" DEFINED
DECLARE " UML State" DEFINED
DECLARE " UML Sequence" DEFINED
DECLARE " UML Collaboration" DEFINED
DECLARE " UML Component" DEFINED
DECLARE " UML Deployment" DEFINED
DECLARE " UML Use Case" DEFINED
DECLARE " UML Activity" DEFINED
DECLARE "UML Object-oriented" DEFINED
DECLARE " System Architecture" UNDEFINED
DECLARE " System Area Map" UNDEFINED
```

2

Modifying the Metamodel with USRPROPS.TXT

Introduction

This chapter describes the theory and mechanisms behind Rational System Architect's extensible metamodel.

Topics in this chapter	Page
Accessing and Editing the USRPROPS.TXT File	2-3
Composition and Syntax	2-7
Ordering and Laying Out USPROPS.TXT Changes	2-19
Defining a LIST of Values	2-29
Renaming Diagram, Symbol, or Definition Types	2-37
Creating New Diagram, Symbol, or Definition Types	2-37
Depicting a Symbol with a Bitmap or Metafile	2-44
Specifying Properties for Diagrams, Symbols, & Definitions	2-54
Using ListOf, one of, and ExpressionOf	2-69
Modifying the Aethetic Look of the Dialogs	2-78
Specifying the Display of Property Values on Symbols	2-101

Specifying Key and Keyed By Properties 2-109

Accessing and Editing the USRPROPS.TXT File

The USRPROPS.TXT file can be edited in any text editor. The one requirement is that it must be saved as a TEXT file.

Accessing the Master USRPROPS.TXT File

As mentioned earlier in this chapter, the master USRPROPS.TXT file is automatically placed in any new encyclopedia you create. Many organizations modify the master USRPROPS.TXT file so that all new encyclopedias contain the same metamodel extensions. To edit the master USRPROPS.TXT file:

- Select Tools, Customize User Properties, Edit USRPROPS.TXT (Master), or
- Simply navigate to the <C>:\Program Files\IBM\Rational\System Architect Suite\11.3.1\System Architect directory, and open the USRPROPS.TXT file found there.

Accessing an Encyclopedia's USRPROPS.TXT File

An encyclopedia's USRPROPS.TXT file is located in the Files table within the encyclopedia's SQL Server database. To edit it, you must first export it out of the Files table of the database. Then, after editing it, you must import it back into the Files table of the database, and reopen your encyclopedia (so the SAPROPS.CFG and USRPROPS.TXT files can be parsed).

There are a number of ways to access an encyclopedia's USRPROPS.TXT file.

- You may use Rational System Architect's native USRPROPS.TXT export/import facility (select Tools, Customize User Properties, Export USRPROPS.TXT (Encyclopedia), or
- You may use Rational System Architect's **Encyclopedia File Manager** utility (select Tools, Encyclopedia File Manager), or
- You may use **SAEM** (from outside of Rational System Architect, select Start, Programs, IBM Rational, IBM Rational Lifecycle Solutions Tools, IBM

Rational System Architect 11.3.1, SAEM, and refer to SAEM’s help).

Using Rational System Architect’s Native USRPROPS.TXT Export/Import Facility:

To edit the USRPROPS.TXT file using Rational System Architect’s native USRPROPS.TXT Export/Import facility, perform the following steps:

1. Select Tools, Customize User Properties, Export USRPROPS.TXT (Encyclopedia).
2. In the **Export User Properties** dialog that opens, select a directory to export the USRPROPS.TXT file to. Click the Save button; the USRPROPS.TXT file will be saved to the selected directory, and open automatically in Notepad.
3. Once you edit the file, select Tools, Customize User Properties, Import USRPROPS.TXT (Encyclopedia) to reimport the modified USRPROPS.TXT file into the Files table of the encyclopedia’s database.
4. Reopen the encyclopedia for it to parse its SAPROPS.CFG file and its modified USRPROPS.TXT file.

Using Encyclopedia File Manager:

To edit the USRPROPS.TXT file using Encyclopedia File Manager, perform the following steps:

1. Select Tools, Encyclopedia File Manager.
2. In the Encyclopedia File Manager dialog, make sure that the Export choice is toggled on in the lower left-hand corner. Select the USRPROPS.TXT file in the Select a file to export list, and select a directory to export the file to using the ‘…’ button of the Export selected file to property.
3. Modify the file in a text editor, and use Encyclopedia File Manager to import the file back into the Files table of the encyclopedia’s database.

4. Reopen the encyclopedia for it to parse its SAPROPS.CFG file and its modified USRPROPS.TXT file.

Using SAEM:

You may also access and edit an encyclopedia's USRPROPS.TXT file using SAEM. Refer to SAEM's help for instructions on how to connect to a server, select a database, and export/import files out of/into the database.

Reloading the Properties Files

As mentioned in the steps above, everytime you reimport a modified USRPROPS.TXT into an encyclopedia, you need to reopen the encyclopedia using Rational System Architect. Reopening the encyclopedia causes the SAPROPS.CFG and USRPROP.TXT files of the encyclopedia to be parsed, creating an SAPROPS.BIN (binary) file, which is what is used to present the metamodel. If error free, the changes to the metamodel take effect immediately.

If, upon parsing the USRPROPS.TXT file, Rational System Architect encounters errors in the USRPROPS.TXT code, it issues either a warning or error message. Rational System Architect will open the encyclopedia after a warning, but will **not** open the encyclopedia if an error is encountered. A message such as that shown below is displayed:

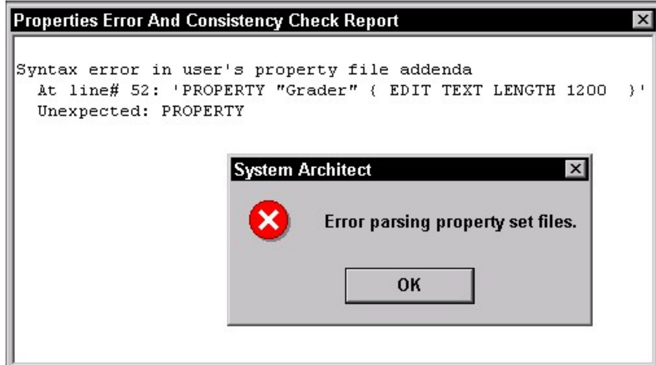


Figure 2-1. Properties Files Error Dialog

Once an error is encountered, you will not be able to access the offending USRPROPS.TXT file using Rational System Architect's Native USRPROPS.TXT Export/Import Facility (if you select Tools, Customize User Properties, the Export USRPROPS.TXT (Encyclopedia) choice will be greyed out).

To access and edit the USRPROPS.TXT file after an error occurs, you must use Rational System Architect's Encyclopedia File Manager (select Tools, Encyclopedia File Manager) or SAEM (from outside of Rational System Architect, select Start, Programs, IBM Rational, IBM Rational Lifecycle Solutions Tools, IBM Rational System Architect 11.3.1, SAEM, and refer to SAEM's help).

Composition and Syntax

Like most programming languages, the language syntax of USRPROPS.TXT is composed of a series of strings. At least one *white space* character is required to separate strings from each other (white space includes spaces, tabs, commas, carriage return/line feeds, and some others). When there are several white space characters one after the other, such as a carriage return followed by a tab, they are grouped together and treated as one.

If a string includes one or more embedded spaces, be sure to enclose the string within double quotes, for example, use **"Data Flow"**, not **Data Flow**.

Keywords

The USRPROPS.TXT language has a certain set of ***keywords***. Depending on its placement, a keyword is considered to be either a ***command*** or an ***argument***. All keywords allowed in USRPROPS.TXT are listed in **Chapter 3, USRPROPS.TXT Keywords**.

Case Insensitivity of Keywords

Keywords in USRPROPS.TXT are ***not*** case sensitive, and you may use capital letters, or small letters, or mixed. In this manual and in the sample USRPROPS.TXT file, commands and all other keywords are all caps for readability only. Examples of commands are:

```
BEGIN or Begin or BegiN
EDIT or Edit
LIST or List or LiST
LISTONLY or Listonly or ListOnly
RENAME or Rename or ReName, etc
```

Commands

Commands are *always* keywords and they *always* start a new phrase. When Rational System Architect parses USRPROPS.TXT, it knows that the first string in the file must be a valid keyword command. Each command must be followed by a known number of argument strings (zero or one or more) and then another command must be found.

Arguments

Strings that follow commands are arguments. Some arguments may be keywords. Other arguments consist of textual strings that provide the names of Diagrams, Symbols, Definitions, Properties, List Values, Labels, Help Strings, etc,

that are found in subsequent dialogs. Here are some examples:

- LIST **"Processor Scheduling**"

  "Processor Scheduling" is not a keyword. It is used as an argument in the expression above.

- DISPLAY { FORMAT **KEY** LEGEND "Key data" }

  "KEY" is a keyword. It is used as an argument in the expression above.

Case Sensitivity of Arguments that Are Text Strings

As mentioned previously, keywords are ***not*** case sensitive. However, arguments that are text strings ***are*** case sensitive. For example, using the LIST "Processor Scheduling" argument above, any references to that list in either SAPROPS.CFG or USRPROPS.TXT must be spelled exactly the same way, with the same case sensitivity. For example, if we specify the following list:

```
LIST "Processor Scheduling"
{
VALUE"preemptive"
VALUE"nonpreemptive"
}
```

Then a valid reference to that list should have the same exact spelling.

```
DEFINITION "Hardware Processor"
PROPERTY "Scheduling"
{ EDIT text LIST "Processor Scheduling" LENGTH 20
DISPLAY { LEGEND "Sched" } }
```

However, the following syntax will give you an error message stating 'List "PROCESSOR SCHEDULING" not Found.'

```
DEFINITION "Hardware Processor"
PROPERTY "Scheduling"
{ EDIT text LIST "PROCESSOR SCHEDULING" LENGTH 20
DISPLAY { LEGEND "Sched" } }
```

Similarly, any properties referenced in reports must use the spelling and case of the entry in SAPROPS.CFG and/or USRPROPS.TXT file.

Grouping Commands to Create Modeling Elements

Opening and closing braces, **{ }**, or, alternatively, BEGIN...END commands, are used to group commands in order to form modeling elements.

Diagrams, Symbols, and Definitions

Rational System Architect's repository supports three main modeling elements – sometimes referred to as *dictionary classes* – ***diagrams***, ***symbols*** (which are drawn on diagrams), and ***definitions*** (which may or may not be represented by symbols). The BEGIN .. END or { } structure is used to specify the contents of these modeling elements, as follows:

Diagram "Name of Diagram Type**"**
{
[contents]
}

Symbol "Name of Symbol Type**"**
{
[contents]
}

Definition "Name of Definition Type**"**
{
[contents]
}

or

Diagram "Name of Diagram Type**"**
BEGIN
[contents]
END

Etc

Properties

The contents of these modeling elements consist of properties and layout commands. The BEGIN .. END or { } structure is used to group property commands, thusly:

Definition **"**Name of Definition Type**"**
{

```
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
…
}
```

Certain keywords that create clauses within a property also require opening and closing braces to delineate the command's arguments, such as the KEYED BY command.

```
Definition "Name of Definition Type"
{
PROPERTY { [specification of property] KEYED BY {
[clause] }  }
…
}
```

Layout

The LAYOUT command also requires opening and closing braces or a BEGIN .. END statement.

```
Definition "Name of Definition Type"
{
LAYOUT { [specification of layout] }
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
…
}
```

Chapter

Properties in a dialog may be further grouped into tabs and groups. Tabs are specified by a CHAPTER command – the CHAPTER command does not require – and in fact ***must not have*** – opening and closing braces or BEGIN .. END statements. It simply groups all properties below it in a specification into a tab (within the ensuing dialog), until the next CHAPTER command is encountered in the specification.

```
Definition "Name of Definition Type"
{
LAYOUT { [specification of layout] }
CHAPTER "First Tab"
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
…
CHAPTER "Second Tab"
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
…
}
```

Groups

Unlike the CHAPTER command, GROUPS do ***require*** the opening and closing braces or a BEGIN .. END statement.

```
Definition "Name of Definition Type"
{
LAYOUT { [specification of layout] }
CHAPTER "First Tab"
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
GROUP "Things That Go Together"
{
PROPERTY { [specification of property]  }
PROPERTY { [specification of property]  }
}
…
}
```

Lists

You may also specify lists in an encyclopedia – either preset lists containing textual values or lists of definitions that you create while modeling. Lists of definitions that you create while modeling are built using the ONE OF, LISTOF, and EXPRESSIONOF commands within a property statement, and are discussed later. Textual lists are built by specifying the values of the list in a separate list statement, with the values enclosed within opening/closing braces or a BEGIN.. END structure. The LIST statement is normally placed near the top of the USRPROPS.TXT file, and referenced within the appropriate property specification of a Diagram, Symbol, or Definition.

```
LIST "List of Things"
{
VALUE One
VALUE Two
VALUE "Two and a Half"
}
```

A Note on Syntax

Indentations and new lines are used solely to enhance readability, and have no meaning to the USRPROPS.TXT processor other than to act as whitespace separators between strings. The above example could be written like this:

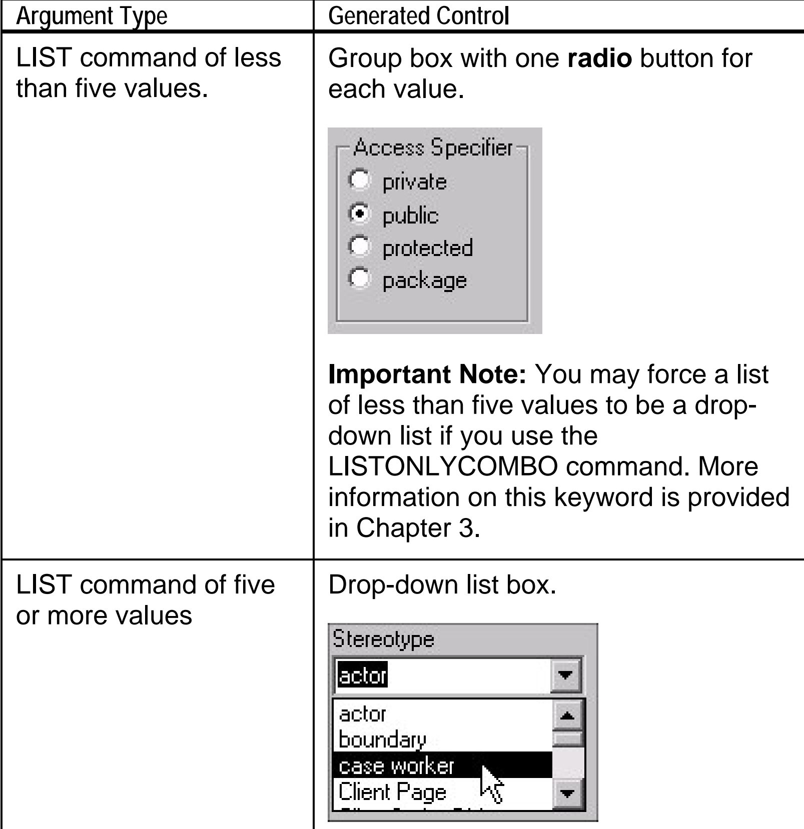
```
LIST "List of Things"   {    VALUE One    VALUE
Two    VALUE “Two and a Half”    VALUE }
```

While this format is perfectly acceptable to Rational System Architect, it probably makes maintenance of the USRPROPS.TXT file more difficult, and should therefore be avoided.

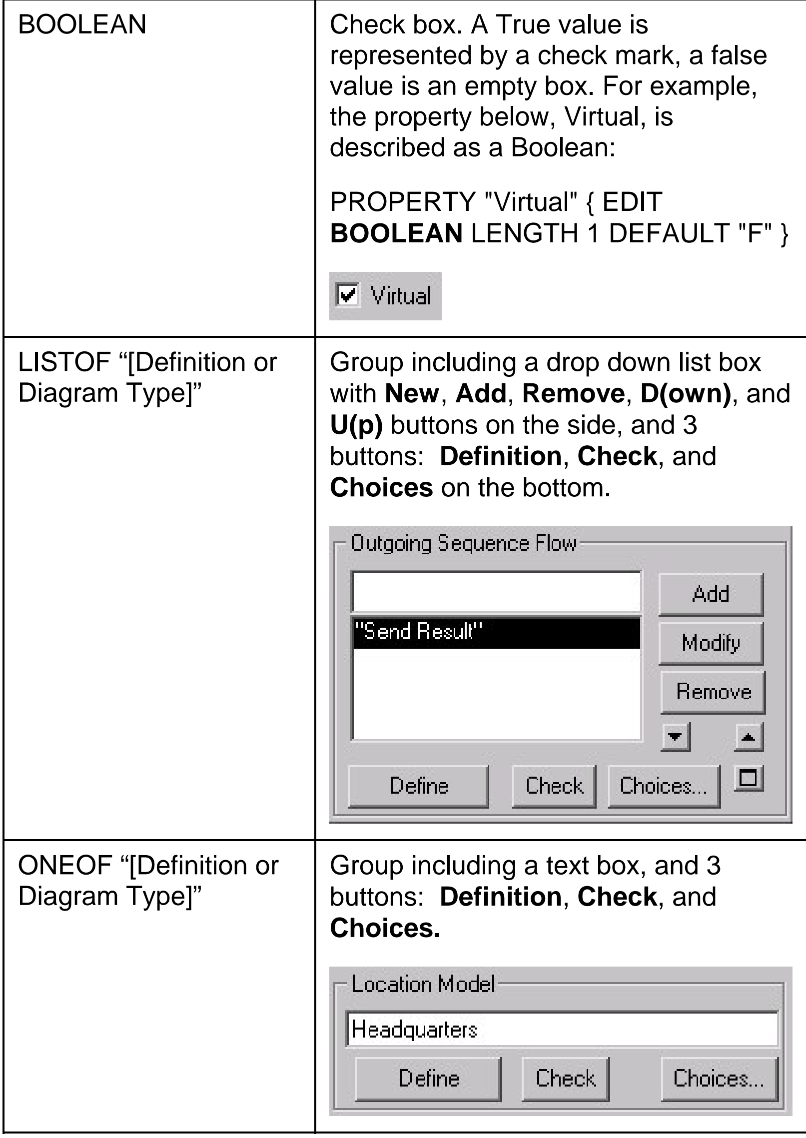
Dialog Controls

The table below describes dialog controls that can be created via appropriate commands in USRPROPS.TXT.

Table 2-2. Controls Generated from Property Expressions



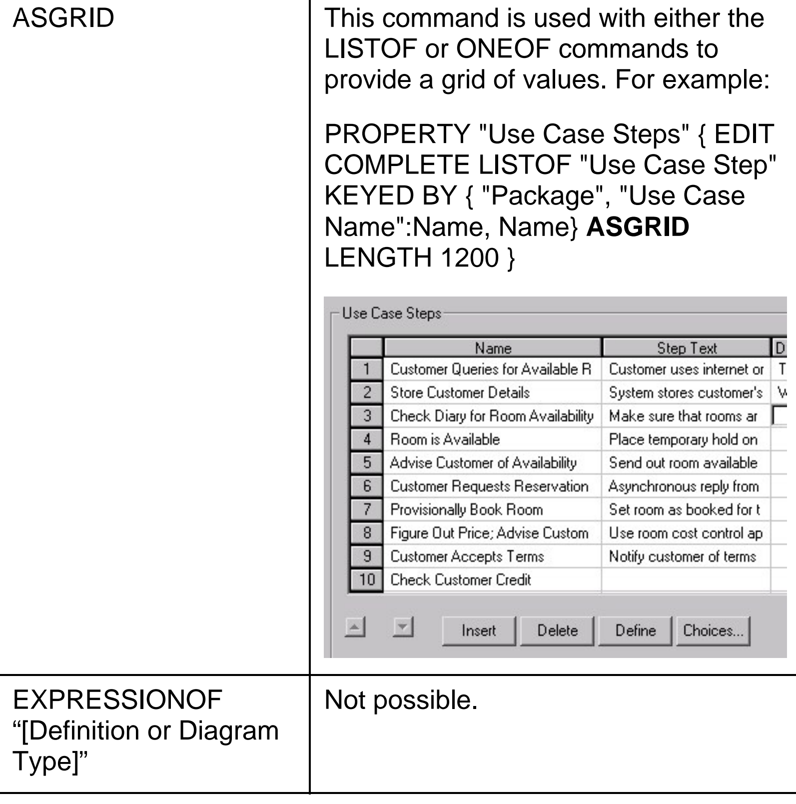
Argument Type	Generated Control
LIST command of less than five values.	Group box with one **radio** button for each value. **Important Note:** You may force a list of less than five values to be a drop-down list if you use the LISTONLYCOMBO command. More information on this keyword is provided in Chapter 3.
LIST command of five or more values	Drop-down list box.

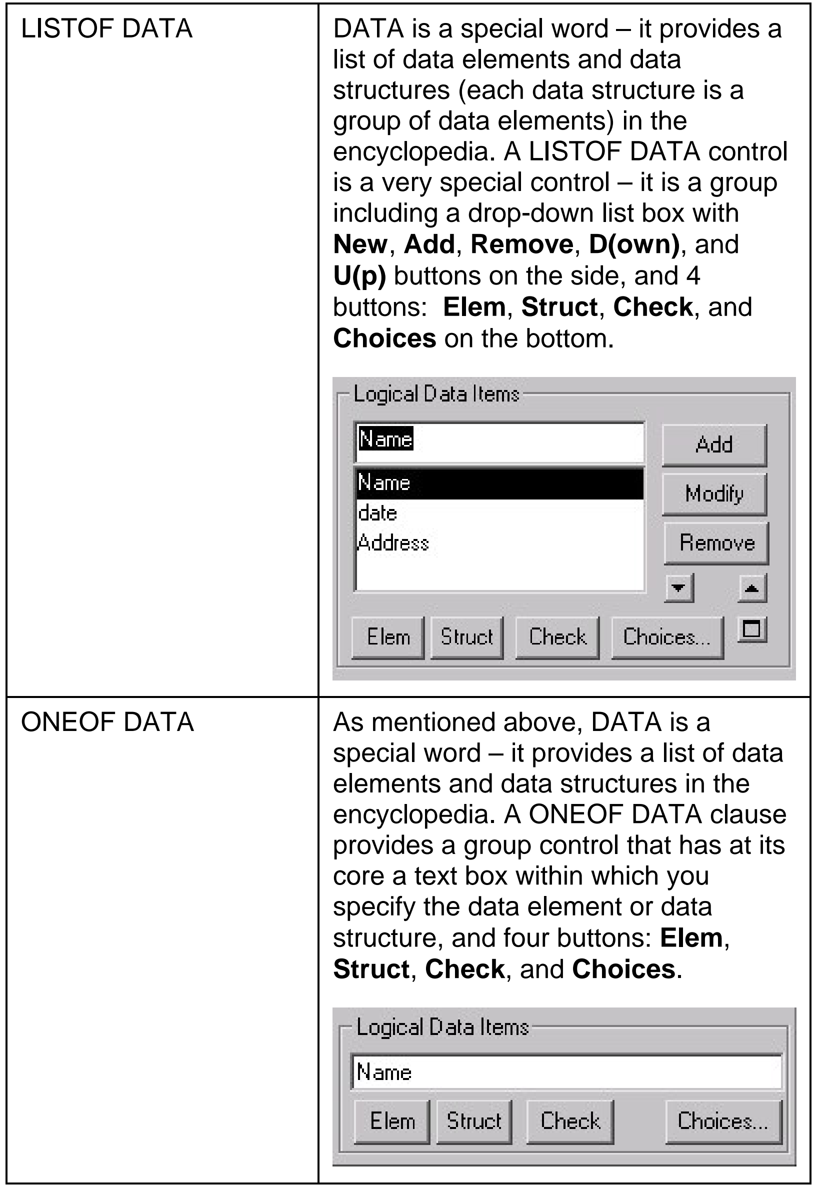


BOOLEAN	Check box. A True value is represented by a check mark, a false value is an empty box. For example, the property below, Virtual, is described as a Boolean: PROPERTY "Virtual" { EDIT **BOOLEAN** LENGTH 1 DEFAULT "F" } Virtual
LISTOF “[Definition or Diagram Type]”	Group including a drop down list box with **New**, **Add**, **Remove**, **D(own)**, and **U(p)** buttons on the side, and 3 buttons: **Definition**, **Check**, and **Choices** on the bottom. Outgoing Sequence Flow "Send Result" Add Modify Remove Define Check Choices...
ONEOF “[Definition or Diagram Type]”	Group including a text box, and 3 buttons: **Definition**, **Check**, and **Choices.** Location Model Headquarters Define Check Choices...

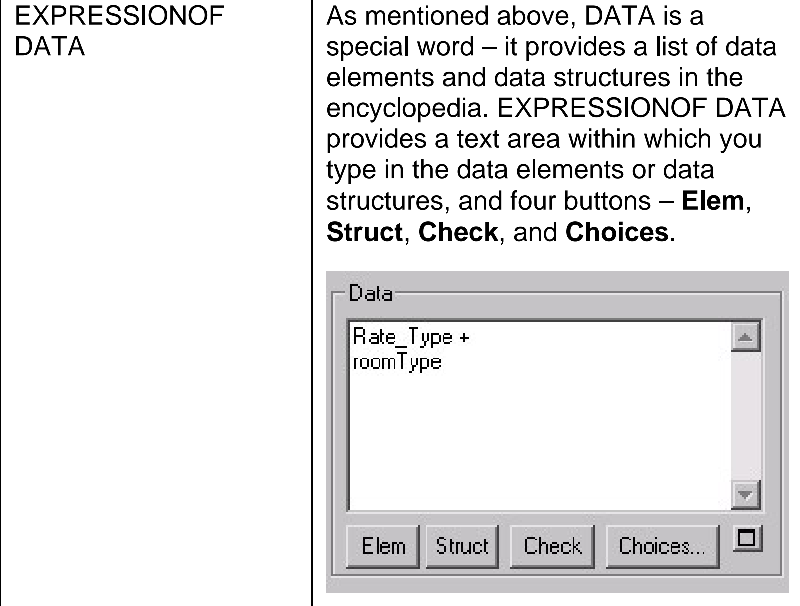
<table>
<tr>
<td>ASGRID</td>
<td>This command is used with either the LISTOF or ONEOF commands to provide a grid of values. For example:

PROPERTY "Use Case Steps" { EDIT COMPLETE LISTOF "Use Case Step" KEYED BY { "Package", "Use Case Name":Name, Name} ASGRID LENGTH 1200 }


</td>
</tr>
<tr>
<td>EXPRESSIONOF “[Definition or Diagram Type]”</td>
<td>Not possible.</td>
</tr>
</table>



LISTOF DATA	DATA is a special word – it provides a list of data elements and data structures (each data structure is a group of data elements) in the encyclopedia. A LISTOF DATA control is a very special control – it is a group including a drop-down list box with **New**, **Add**, **Remove**, **D(own)**, and **U(p)** buttons on the side, and 4 buttons: **Elem**, **Struct**, **Check**, and **Choices** on the bottom.
ONEOF DATA	As mentioned above, DATA is a special word – it provides a list of data elements and data structures in the encyclopedia. A ONEOF DATA clause provides a group control that has at its core a text box within which you specify the data element or data structure, and four buttons: **Elem**, **Struct**, **Check**, and **Choices**.

EXPRESSIONOF DATA	As mentioned above, DATA is a special word – it provides a list of data elements and data structures in the encyclopedia. EXPRESSIONOF DATA provides a text area within which you type in the data elements or data structures, and four buttons – **Elem**, **Struct**, **Check**, and **Choices**. 

Ordering and Laying Out USRPROPS.TXT Changes

The general ordering of sections of SAPROPS.CFG is as follows:

- **LIST** command section
- **DIAGRAM** command section
- **SYMBOL** command section
- **DEFINITION** command section
  - **LAYOUT** command subsection (default for entire definition dialog)
  - **CHAPTER** command subsection
  - **GROUP** command subsection
  - **LAYOUT** command subsection
  - **PROPERTY** command subsection

Although all entries in USRPROPS.TXT are optional, you should follow a similar layout as SAPROPS.CFG, adding a **RENAME** command section, if used, to the top of the file. The general ordering of sections for USRPROPS.TXT should be as follows:

- **RENAME** command section (in this section you rename USER DIAGRAMS, USER SYMBOLS, and USER DEFINITIONS to create your own diagram, symbol, or definition types (see page 2-31)
- **LIST** command section (see page 2-29)
- **DIAGRAM** command section (see page 2-56)
- **SYMBOL** command section (see page 2-58)
- **DEFINITION** command section (see page 2-62)
  - **CHAPTER** command subsection (see page 2-90)
  - **GROUP** command subsection (see page 2-92)

LAYOUT command subsection (see page 2-92)

PROPERTY command subsection (see page 2-65)

.

Rules for Modifying USRPROPS.TXT

The following rules should be kept in mind when creating USRPROPS.TXT:

1. USRPROPS.TXT entries are additions to or replacements for entries in SAPROPS.CFG.
2. The USRPROPS.TXT entry must begin with the relevant LIST, RENAME, DIAGRAM, SYMBOL or DEFINITION statement.
3. USRPROPS.TXT entries that are *additions* to SAPROPS.CFG go to the end of the relevant section. For example, a LIST block not in SAPROPS.CFG is essentially added after all other LIST blocks in SAPROPS.CFG.
4. Unless the **CHAPTER** command is included, USRPROPS.TXT entries go to the end of the relevant dialog. For example, a new property for a Class definition is added after all other properties in the Class's definition dialog.
5. If a **CHAPTER** command already in SAPROPS.CFG is included in USRPROPS.TXT, the USRPROPS.TXT entries go to the end of the existing chapter (or tab).
6. If a **GROUP** command already in SAPROPS.CFG is included in USRPROPS.TXT, the USRPROPS.TXT entries go to the end of the existing group.
7. The **GROUP** command produces a group box, a standard Windows control, into which all subsequent controls must be placed. If there are too many entries, so that the size of the group is larger than the size of the

monitor, extraneous properties are not included, and not displayed. A warning message to that affect is displayed when the encyclopedia is opened.

8. If a property is added to a group that has **PLACEMENT** commands on its properties in SAPROPS.CFG, the **PLACEMENT** command must also be used for the new property(ies) added in USRPROPS.TXT.

Layout Of USRPROPS.TXT Code

If you have neither a USRPROPS.TXT nor a SAPROPS.CFG file, however, every diagram, symbol and definition still has a *name* and the property *description*. The default values for *description* are included later in this section. As mentioned previously, the complete text of SAPROPS.CFG is included in the file called CONFIG.PRP. This is a standard ASCII text file; the entries can be used as models for changes and additions to USRPROPS.TXT.

How you lay out the actual code in the USRPROPS.TXT file itself is up to you. We recommend providing a tab structure so that it is easier to see the beginnings of List, Diagram, Symbol, and Definition statements. However, different text editors may represent tabs different ways – for example, if you use Microsoft Word as your text editor, and then open up the USRPROPS.TXT later in a different text editor, the tabs you set in Word may be spaced completely differently.

Figure 2-2. Example Code Layout for USRPROPS.TXT

```
REM "USRPROPS.TXT"
REM "Copyright Telelogic.  All rights reserved."

REM "Instructions for modifying this file are in the on-line help."


RENAME DIAGRAM    "User 1" to "Zoo"
RENAME SYMBOL     "User 1" to "Mammals"
RENAME SYMBOL     "User 2" to "Reptiles"
RENAME DEFINITION "User 1" to "Mammal"
RENAME DEFINITION "User 2" to "Reptile"


LIST "Importance"
{
        VALUE "Should Have"
        VALUE "Must Have"
        VALUE "Icing on the Cake"
}


DIAGRAM "Zoo"
{
        HIERARCHICAL
        PROPERTY "Hierarchical Numbering"{ EDIT Boolean LENGTH 1 DEFAULT "T" }
        PROPERTY "First Node Number" { EDIT Text  Length 20 DEFAULT "1" }
}

SYMBOL "Mammals"
{
        DEFINED by "Mammal"
        ASSIGN TO "Zoo"
}

SYMBOL "Reptiles"
{
        DEFINED by "Reptile"
        ASSIGN TO "Zoo"
}


Definition "Reptile"
{
    Chapter "My Properties"
    LAYOUT { ALIGN OVER }

    PROPERTY "Tail" { Edit Boolean Default "T" }
    PROPERTY "Number of Legs" { EDIT Numeric LENGTH 2 }

}
```

'Comment' Section

'RENAME' Section

'LIST' Section

'DIAGRAM' Section

'SYMBOL' Section

'DEFINITION' Section

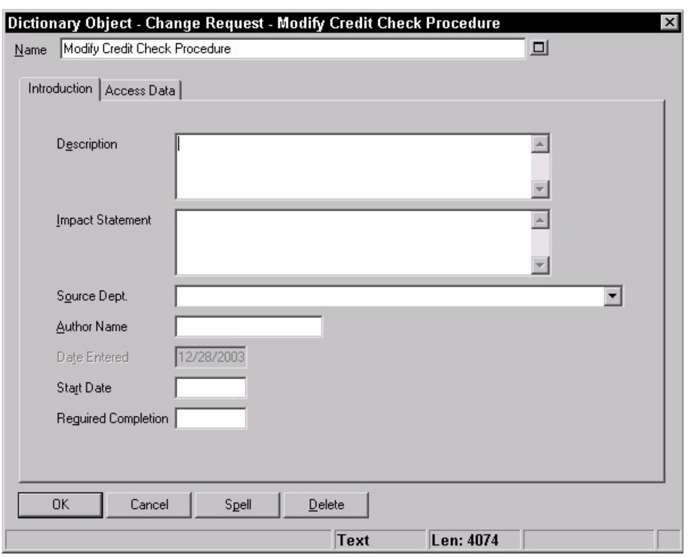
Example of Making Changes to USRPROPS.TXT

In this section, we will make changes to a definition that already exists in SAPROPS.CFG. The following code can be found in SAPROPS.CFG:

```
DEFINITION "Change Request"
{
ADDRESSABLE
LAYOUT { COLS 2 TAB ALIGN LABEL }
PROPERTY "Impact Statement" { EDIT Text LENGTH 1000 }
PROPERTY "Original Source" { EDIT Text LIST "Business Unit"
    LENGTH 80   LABEL "Source Dept." }
PROPERTY "Author Name" { EDIT Text LENGTH 25 }
PROPERTY "Date Entered" { EDIT date INITIAL date READONLY
    LENGTH 10 }
PROPERTY "Start Date" { EDIT date LENGTH 10 }
PROPERTY "Required Completion Date"
  { EDIT date LENGTH 10  LABEL "Required Completion" }
```

The picture below shows the **Dictionary Object** dialog for the above definition block (note that we informally call this the **definition** dialog throughout most of this manual).

Figure 2-3. Change Request Definition Dialog as Defined in the Master Configuration Property Set File



Note that there is an **Introduction** tab even though our SAPROPS.CFG has not called this out specifically with a CHAPTER command. If no CHAPTER command is specified, Rational System Architect automatically provides a default **Introduction** tab. The **Access Data** tab is hard-coded in the software and not specified in SAPROPS.CFG.

Making a Change With USRPROPS.TXT

We make changes to the Change Request definition by adding the following code to USRPROPS.TXT and reopening the encyclopedia:

```
DEFINITION "Change Request"
{
LAYOUT { COLS 2 TAB ALIGN LABEL }
PROPERTY "Author Name" { LABEL "Client Division" }
PROPERTY "Supervising Manager" { EDIT text LENGTH 45 }
PROPERTY "On time" { Edit Boolean Length 1 DEFAULT "T" }
}
```

The table explains each line of the USRPROPS.TXT code above, and the effect it has.

Table 2-1. Effect of USRPROPS.TXT Entries

USRPROPS.TXT Entry	Effect
DEFINITION "Change Request" {	Specifies a change to the Definition "Change Request"
LAYOUT { COLS 2 TAB ALIGN LABEL }	Sets up a two-column layout for the properties below the LAYOUT command (until the end of the definition is reached or another LAYOUT command is encountered)..
PROPERTY "Author Name" { LABEL "Client Division" }	This modifies an existing property – it changes the label on the field from *Author Name* to *Client Division*
PROPERTY "Supervising Manager" { EDIT TEXT LENGTH 45 }	This adds a new property, which is a text field, *Supervising Manager*, to the dialog box.

PROPERTY "On time" { EDIT BOOLEAN LENGTH 1 DEFAULT "T" } }	This adds a new property, which is a check box, to the dialog box to indicate whether the change request is meeting the deadline.

We import our changed USRPROPS.TXT file into our Rational System Architect encyclopedia, and reopen the definition of a change request, to see the changes to its dialog – note that the information on the **Introduction** tab has now spilled onto two pages.

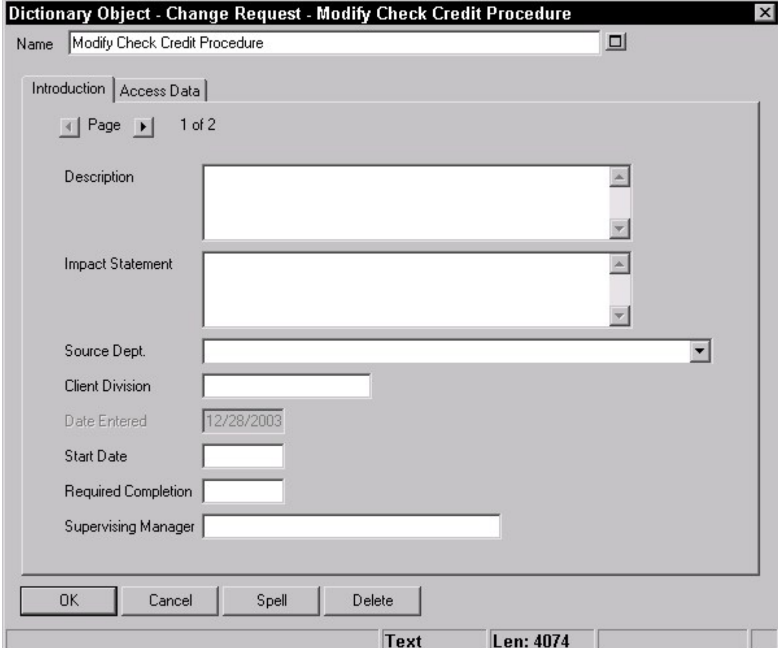
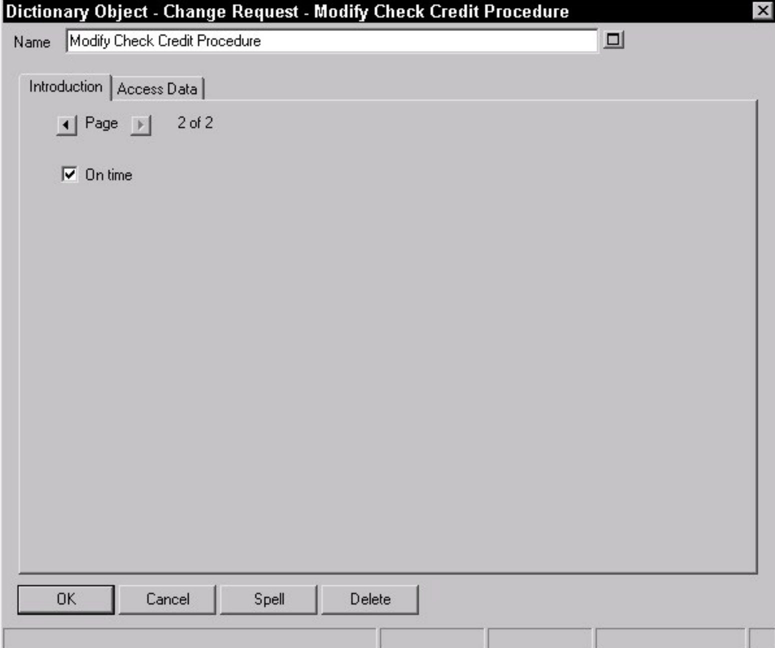


Figure 2- 4. Change Request Definition Dialog as Modified by Entries in USRPROPS.TXT



The information spilled out onto two pages of the **Introduction** tab because of the two new properties we added. They get added to the end of the definition (they do not get added to the end of the **Access Data** tab because this tab doesn't count – it is hard coded and not part of SAPROPS.CFG).

Only Change What Needs to Be Changed

Notice that we did not re-enter the entire PROPERTY statement that exists in SAPROPS.CFG into our USRPROPS.TXT file. We simply need to enter specific statements that need to be changed, besides any new statements that we are adding. And even for the statements that we are changing that we are re-entering, we only need to add the part of the statement that is changing. In our example, the one statement from SAPROPS.CFG that we re-entered and changed was:

```
PROPERTY "Author Name" { EDIT TEXT LENGTH 25 }
```

In our USRPROPS.TXT file, we only wanted to change the label on this property, so we simply entered:

```
PROPERTY "Author Name" { LABEL "Client Division" }
```

The length of the property and the fact that it is text (rather than numeric or Boolean) remain unchanged; only the label to the left of the control in the dialog has been changed.

One More Change and a Warning

Let's try another change – we add the text in bold below to our USRPROPS.TXT code:

```
DEFINITION "Change Request"
{
LAYOUT { COLS 2 TAB ALIGN LABEL }
PROPERTY "Impact Statement"  { EDIT text LENGTH 100 }
PROPERTY "Author Name" { LABEL "Client Division" }
PROPERTY "Supervising Manager" { EDIT text LENGTH 45 }
PROPERTY "On time" { Edit Boolean Length 1 DEFAULT "T" }
}
```

The explanation of this change is explained below:

USRPROPS.TXT Entry	Effect
PROPERTY "Impact Statement" { EDIT TEXT LENGTH **100** }	Attempts to modify an existing property, reducing the space for the *Impact Statement* from 1000 characters to 100.

We import this USRPROPS.TXT back into our encyclopedia, and reopen the encyclopedia, and receive a warning message from Rational System Architect:

```
Warning: In user's property file
addenda between line number 70 and line
number 72. Illegal attempt to shorten
the length of a property. Original
length retained.
```

Rational System Architect does not allow you to decrease the length of a field – you can only increase it. The reason for this is that users may have already entered information into a text field that will be lost if you decrease the length of this field, and thereby decrease the amount of information that the encyclopedia can hold for this property, at a later time.

Rational System Architect issues the warning, ignores the faulty code, and opens the encyclopedia. As mentioned previously, if this had been an error message, the encyclopedia would not open until you fixed the USRPROPS.TXT.

If we were instead attempting to increase the length of the Impact Statement field, Rational System Architect would accept the change gladly.

```
PROPERTY "Impact Statement"  { EDIT text LENGTH 1200 }
```

Defining a LIST of Values

You may specify a list of items that is provided to the user as a drop-down list or check-box list in dialogs. The values of the list must be specified in a List definition. The List definition is then referenced in the Diagram, Symbol, or Definition where it is being used. Lists must be placed in USRPROPS.TXT before any Diagram, Symbol, or Definition entries that reference them.

Management of the USRPROPS.TXT file is easier if all List definitions are at the top of the file, following any Rename commands.

Syntax of the LIST Definition

A list definition starts with the keyword **LIST** followed by a string (the argument) that is the name of the list. Names with embedded spaces must be bounded by double quotes. The **LIST** definition is bracketed by opening and closing braces **{ }** or, alternatively, with the **BEGIN...END** structure. Within the brackets you specify the values of the list, each called out by the command keyword, **VALUE**. If a value has one or more embedded spaces, it must be enclosed within double quotes.

```
LIST  list_name
{
VALUE value_name_1
VALUE value_name_2

…
}
```

Example:

```
List "Method Stereotypes"
{
VALUE Get
VALUE Let
VALUE Set
VALUE “Stereotype with embedded spaces”
}

DEFINITION “Method”  {..PROPERTY "Stereotype"{
EDIT Text LIST "Method Stereotypes" Default ""
LENGTH 30  }   …}
```

Indentations and new lines are used solely to enhance readability, and have no meaning to the USRPROPS.TXT processor other than to act as white space separators between strings. The above example could be written like this:

```
LIST "Method Stereotypes"   {    VALUE get    VALUE let
VALUE set  VALUE “Stereotype with embedded spaces”
}
```

While this format is perfectly acceptable to Rational System Architect, it makes maintenance of the USRPROPS.TXT file more difficult, and should therefore be avoided.

Check-Boxes Versus Drop-Down List

Rational System Architect automatically displays a list as a list of checkbox choices if the number of values in the LIST statement is four or less. If the number of values is five or more, the list is automatically displayed as a drop-down list box. Users may type in their own value in a drop-down list box. If you wish to have a drop-down list box but only have four or less LIST values, use the LISTONLYCOMBO keyword.

Entering Your Own Values

If the list is provided as a drop-down list, then the user can select one of the values from the list, or type in their own value (unless the LISTONLY or LISTONLYCOMBO command has been used – see Chapter 3, LISTONLY or LISTONLYCOMBO command).

Renaming Existing Diagram, Symbol, or Definition Types

Each DIAGRAM, SYMBOL, and DEFINITION statement must refer to an object known to Rational System Architect.

However, in case any of the names in the provided SAPROPS.CFG file are not appropriate, you have the ability to change them to meet the requirements of your individual project or company standards. The RENAME statements should be entered at the top of USRPROPS.TXT, prior to all other commands and statements. The general syntax of the **RENAME** command is:

```
RENAME class_name from_type_name TO to_type_name
```

The following three statements rename a diagram, symbol, and definition:

```
RENAME DIAGRAM "Data Flow Gane & Sarson"
TO "Data Flow Chris & Trish"

RENAME SYMBOL "Data Transform" in
"Data Flow Ward & Mellor"
TO "Process A"

RENAME DEFINITION "Process"
TO "Process A"
```

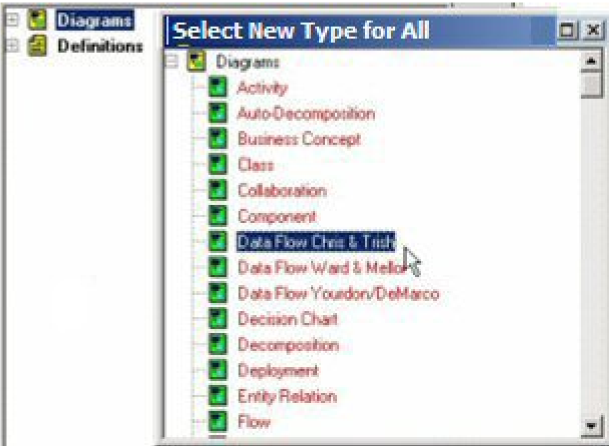


Figure 2- 5. The Type pull-down menu does not display *Data Flow Gane & Sarson*, but *Data Flow Chris & Trish*

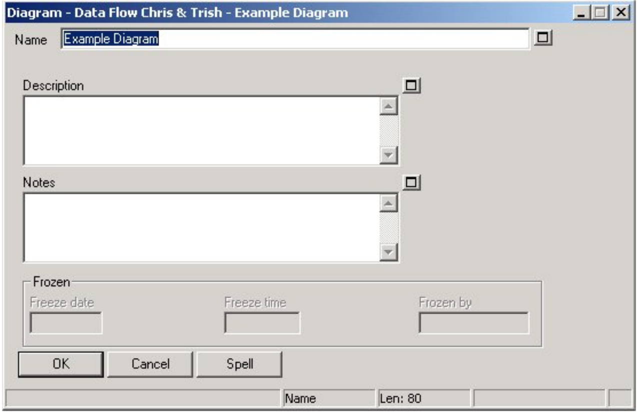


Figure 2- 6. The Diagram Properties Modify Dialog also displays in the title *DFD Chris & Trish,* not *DFD Gane & Sarson.*

The **RENAME SYMBOL** command could be used by designers working with Ward & Mellor DFD's who prefer the name *Process* to *Transform*: Click on a *Control Transform* on a DFD Ward & Mellor. Then double-click the symbol in the diagram to display the **Diagram <Type> <Name>** dialog. The symbol's type is *Control Transform*.

In order to rename the symbol, the following command must be entered in USRPROPS.TXT:

```
RENAME class_name from_type_name IN
from_diagram_name TO to_type_name
```

For example,

RENAME SYMBOL "Control Transform" IN "DFD Ward & Mellor" TO "Process" ACCELERATOR "r"

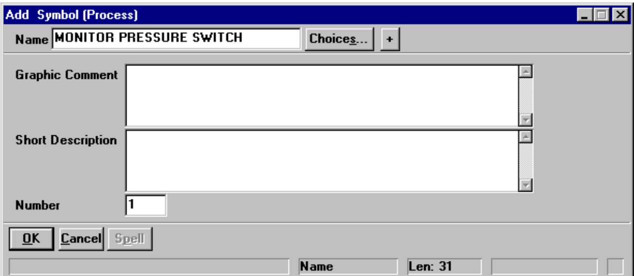


Figure 2-7. Symbol Properties Dialog Before RENAME

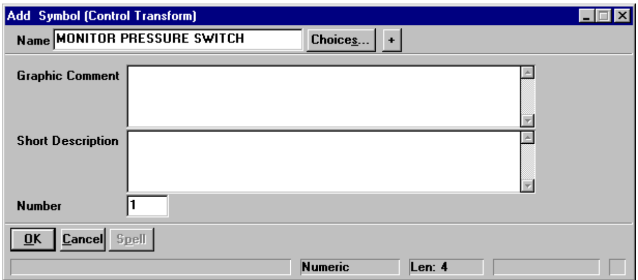


Figure 2-8. Symbol Properties Dialog After RENAME

Click on the **Edit** menu and select **Edit <Symbol Type>** again while the *Control Transform* is selected. Note the title of the **Definition Modify** dialog: the definition is *Process,* not *Control Transform*. That is, the definition of the symbol *Control Transform* maps to the definition *Process*. Let us assume that you use DFD Ward & Mellor, rather than DFD Gane & Sarson, and prefer that the definition name match the symbol name.

The syntax of the **RENAME** command for a definition is:

RENAME class_name from_type_name TO to_type_name

RENAME DEFINITION "Process" TO "Control Transform"

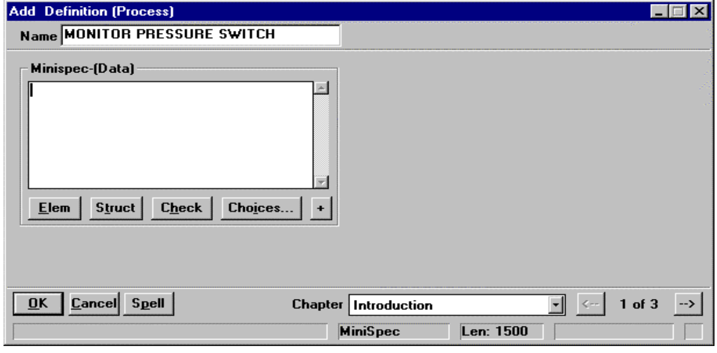


Figure 2-9. Symbol Definition Dialog Before RENAME

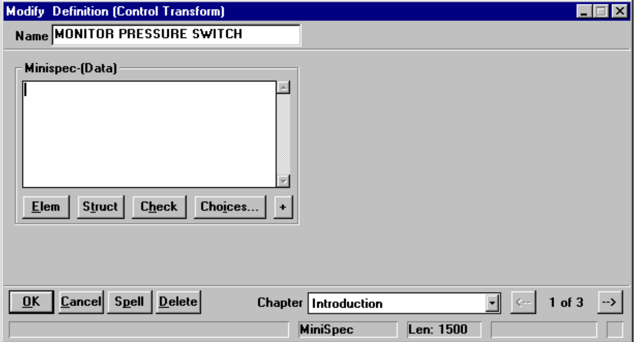


Figure 2-10. Symbol Definition Dialog After RENAME

On the other hand, if you use both DFD Gane & Sarson (whose *Process* symbols map to definition *Process*, and DFD Ward & Mellor (whose *Control Transform* symbols map to definition *Process*), you may wish to rename only the definitions of the *Control Transforms*, not the definitions of all processes. The following entries in USRPROPS.TXT would perform that rename:

RENAME DEFINITION "User 2"[1] to "Control Transform"

SYMBOL "Control Transform" in <diagram name>
{ DEFINED BY "Control Transform" }

If the Definition "Control Transform" does not include a set of properties, it has only the property *Description*. In the example we are working on, the following definition block,

---

[1] There are 150 "User n" definitions available for your use, starting with User 1.

identical to that of *Process*, was added to USRPROPS.TXT. You could, of course, have any properties you feel appropriate; you need not copy those of an existing definition.

```
DEFINITION "Control Transform"
{
PROPERTY "Description"
  { EDIT Minispec LENGTH 750 }
PROPERTY "Complexity"
  { EDIT numeric LENGTH 10 }
PROPERTY "Memory Allocation (KB)"
   { EDIT numeric LENGTH 7 }
PROPERTY "Priority"
  { EDIT numeric LENGTH 3 MINIMUM 0 MAXIMUM 999 }
PROPERTY "Process Class"
  { EDIT text LISTONLY LIST "Process Class" LENGTH 20 }
PROPERTY "Processing Time Allocation"
  { EDIT numeric LENGTH 3 MINIMUM 0 MAXIMUM 100 }
PROPERTY "Purpose"
  { EDIT text LENGTH 4095 }
PROPERTY "Transaction Rate"
  { EDIT numeric LENGTH 10 MINIMUM 1 MAXIMUM 10 }
}
```

RENAME and Reporting

The **RENAME** command also affects the way you write reports. Any place where the old name was used, the new name must be used instead. In the GUI reporting system, you'll have to re-select the diagram, symbol, or the definition property names after they've been changed in SAPROPS.

Before:

```
REPORT "List of Processes"
{
TABULAR 1 {
SELECT Name, "Update Date", Description
WHERE Class = Definition
WHERE Type = "Process"
ORDERBY Name
}
}
```

If you look at the report using the Text Editor (**Reports** dialog, **EDIT** command) what you'll see will be the following.

After:

```
REPORT "List Of Transforms"
{
TABULAR 1 {
SELECT Name, "Update Date", Description
WHERE Class = Definition
WHERE Type = "Control Transform"
ORDERBY Name
}
}
```

Creating New Diagram, Symbol, or Definition Types

You may create new diagram, symbol, or definition types in a Rational System Architect encyclopedia. You do this by using the RENAME command to rename pre-existing diagram, symbol, or definition types provided for this purpose. Again, RENAME commands should be placed at the top of the USRPROPS.TXT file, just below the opening REM (Reminder, or Comment) statements.

Creating New Diagrams

You can create up to 50 new diagram types in a Rational System Architect encyclopedia. To add a new diagram type, in USRPROPS.TXT you rename one of 50 generic diagram types available – User 1 through User 50. The syntax is as follows:

```
RENAME DIAGRAM “User 1” TO My_Diagram
```

Note that if you wish to have embedded spaces in a new diagram, symbol, or definition type that you are creating, you must place the name in quotation marks. For example:

```
RENAME DIAGRAM “User 1” TO “My Diagram”
```

Once you create the new diagram type, you'll want to specify what type of symbols can be drawn on it. You can create new symbol types, or assign symbols that already exist on other diagrams to the new diagram type. This is covered in the next section, Assigning a Symbol Type to a Diagram Type.

By default, user diagrams are networks (of symbols), but you may also specify that a user diagram is of type **Hierarchical**. A Hierarchical diagram in Rational System Architect has special drawing rules imposed on it, enabling you to connect symbols in a hierarchy and have line symbols automatically drawn. Other related hierarchical functionality (such as hierarchical numbering) is supported. To specify that a diagram is of type Hierarchical, use the HIERARCHICAL keyword, for example:

```
DIAGRAM "Zoo" {HIERARCHICAL}
```

Creating New Symbols

You can create up to 150 new symbol types in a Rational System Architect encyclopedia. To add a new symbol type, you rename one of 150 generic symbol types provided – User 1 through User 150. The syntax is as follows:

```
RENAME SYMBOL "User 3" to "whatever"
```

Specifying New Line Symbols

A line symbol is a line that can be drawn between two symbols, such as a relationship line, an inherits line, an association, a flow line, etc. You can create a new line symbol type in an encyclopedia. You must specify that it looks and behaves like an existing line symbol type on another diagram. You use the same RENAME SYMBOL command as for a regular (‘node’) symbol, but later in the USRPROPS.TXT, you must also specify how the line symbol is drawn, using the DEPICT LIKE command.

```
RENAME SYMBOL "User 4" to "My Line Symbol"

SYMBOL "My Line Symbol"
{ DEPICT LIKE "Dependency" IN "UML Class"
ASSIGN To "Wireless Network" }
```

Creating New Defintions

You can create up to 150 new definition types in a Rational System Architect encyclopedia. To add a new definition type, you rename one of 150 generic definition types provided – User 1 through User 150. The syntax is as follows:

```
RENAME DEFINITION "User 3" to "whatever"
```

A symbol typically represents a definition type. For information on this, see the section that follows, Assigning a Definition Type to a Symbol Type.

Assigning a Symbol Type to a Diagram Type

You may assign new symbol types or existing symbol types (symbols that already exist in another diagram) to new or existing diagram types. Symbol types may be added to diagram types using the following syntax:

ASSIGN <symbol-type-name> [IN <diagram-type-name1>] **TO** <diagram-type-name2>

Symbol types may also be added to diagram types within the SYMBOL specification using the **ASSIGN .. TO** keyword combination, as follows:

SYMBOL <symbol-type-name> [IN <diagram-type-name1>] **{ASSIGN TO** <diagram-type-name1>**}**

Example

For example, the USRPROPS.TXT below creates a new diagram type called a Wireless Network diagram, which provides of three new symbol types to be drawn on it – a Satellite, a Computer, and a Server, and one existing symbol type to be drawn on it – a state symbol from a “State Transition Ward & Mellor” diagram (as compared to a State symbol from a UML State diagram or an IDEF3 State diagram, etc:

```
RENAME DIAGRAM "User 1" To "Wireless Network"

RENAME SYMBOL "User 1" TO "Satellite"
RENAME SYMBOL "User 2" TO "Computer"
RENAME SYMBOL "User 3" TO "Server"

ASSIGN "State" IN "State Transition Ward & Mellor" TO
“Wireless Network”

SYMBOL "Satellite" {ASSIGN TO "Wireless Network"}
SYMBOL "Computer" {ASSIGN TO "Wireless Network"}
SYMBOL "Server" {ASSIGN TO "Wireless Network"}
```

Note: Also see section *Limitations on Assigning a Symbol Type to a Diagram Type*.

Assigning a Line Symbol Type to a Diagram Type

Again, as mentioned previously in this section, a line symbol is a line that can be drawn between two symbols, such as a relationship line, an inherits line, an association, a flow line, etc. You can create a new line symbol type in an encyclopedia. You must specify that it looks and behaves like an existing line symbol type on another diagram. You use the same RENAME SYMBOL command as for a regular ('node') symbol, but later in the USRPROPS.TXT, you must also specify how the line symbol is drawn, using the DEPICT LIKE command.

User defined symbols (User 1 through User 150) are provided for both regular ('node') symbols and line symbols, so be careful that you don't use the same User number for two different symbols.

Example

In the example below, we add a new line drawing symbol to our USRPROPS.TXT, in bold:

RENAME DIAGRAM "User 1" To "Wireless Network"

RENAME SYMBOL "User 1" TO "Satellite"
RENAME SYMBOL "User 2" TO "Computer"
RENAME SYMBOL "User 3" TO "Server"
RENAME SYMBOL "User 4" To "Relates To"

ASSIGN "State" IN "State Transition Ward & Mellor" TO "Wireless Network"
SYMBOL "Satellite" {ASSIGN TO "Wireless Network"}
SYMBOL "Computer" {ASSIGN TO "Wireless Network"}
SYMBOL "Server" {ASSIGN TO "Wireless Network"}

SYMBOL "Relates To"
{ **DEPICT LIKE** "Dependency" IN "UML Class"
ASSIGN To "Wireless Network" }

Note: Also see section *Limitations on Assigning a Symbol Type to a Diagram Type*.

Limitations of Assigning a Symbol Type to a Diagram Type

The following limitations exist for assigning symbol types to diagram types:

No assignment may be made to any of the following diagram types:

- DB2 Physical
- Entity Relation
- Logical Data Model
- Logical View
- Physical Data Model

None of the following symbols may be assigned because of special code in Rational System Architect:

- "Associative Entity" in diagram "Entity Relation"
- "Entity" in diagram "Entity Relation"
- "Identifying Relation" in diagram "Entity Relation"
- "Inconsistent Relation" in diagram "Entity Relation"
- "Nonidentifying Relation" in diagram "Entity Relation"
- "Non-specific Relation" in diagram "Entity Relation"
- "Super-sub Relation" in diagram "Entity Relation"
- "Weak Entity" in diagram "Entity Relation"
- "Association" in diagram "OMT Object Model"
- "Class" in diagram "OMT Object Model"
- "Identifying Constraint" in diagram "Physical Data Model"
- "Nonidentifying Constraint" in diagram "Physical Data Model"
- "Table" in diagram "Physical Data Model"
- "Class" in diagram "UML Class"
- "Interface" in diagram "UML Class"
- "Actor" in diagram "UML Use Case"
- "Boundary" in diagram "UML Use Case"
- "Case Worker" in diagram "UML Use Case"
- "Control" in diagram "UML Use Case"

- "Entity" in diagram "UML Use Case"
- "Worker" in diagram "UML Use Case"
- Additionally, none of the following symbols may be assigned to another diagram because their definitions are keyed by Model:
- "Access Path" in diagram "Entity Relation"
- "Relation" in diagram "Entity Relation"
- "Relation Diamond" in diagram "Entity Relation"
- "Individu" in diagram "Modèle Conceptuel des Données"
- "Relation Ligne" in diagram "Modèle Conceptuel des Données"
- "Keyed Entry Point" in diagram "SSADM Data Structure"
- "Non-Keyed Entry Point" in diagram "SSADM Data Structure"
- "Relation" in diagram "SSADM Data Structure"
- Additionally, the following BPMN symbols can’t be assigned to another diagram type:
- “Pool” in diagram Business Process
- “Lane” in diagram Busines Process

Note: Some symbols both have special code and also are keyed by Model, they are shown only in the first list of symbols.

Many symbols can normally reside in more than one diagram type. Only one diagram type is shown for any symbol in the above lists.

Assigning a Definition Type to a Symbol Type

If you add new symbols to an encyclopedia in USRPROPS.TXT, you must specify what definition type they are associated with using this keyword. If a new symbol specified in USRPROPS.TXT is missing this clause, Rational System Architect will give a parsing warning when opening the encyclopedia, and default to the null definition for the symbol, which consists simply of the Description property.

```
SYMBOL "My Symbol"
{
DEFINED BY " My Definition"
ASSIGN TO "My Diagram"
}
```

Example

In the example below, the symbol type “Satellite” is specified to be defined by the definition type “Satellite” (the fact that they happen to share the same name is not enough).

```
Rename Diagram "User 1" TO "Wireless Network"
Rename Symbol "User 1" TO "Satellite"
Rename Definition "User 1" TO "Satellite"

SYMBOL "Satellite"
{ DEFINED BY “Satellite” ASSIGN To "Wireless Network" }
```

Depicting a Symbol with a Bitmap or Metafile

You may depict a symbol with a bitmap (.bmp) or Windows Metafile (.wmf) that you supply. You may specify how a symbol is depicted on the diagram workspace and also how it is depicted in the toolbox and Draw menu, by adding a depictions clause to the symbol's declaration, as follows:

```
SYMBOL <symbol-type-name>

        { ...
        DEPICTIONS { DIAGRAM <depiction-file> }
        DEPICTIONS { MENU <depiction-file> }
        ...}
```

The DIAGRAM command specifies the depiction file to be ***drawn*** on the diagram workspace. You should use a ***Windows Metafile (.WMF)*** for the DIAGRAM command because it is a vector image that will scale properly if you drag on its handlebars to increase or decrease it in size.You can also use .BMP’s for the DIAGRAM command, but they do not scale well.

WMFs are vector files, which means that they store mathematical formulas about how an image should be displayed on a screen. One major benefit of this format is that it provides scalability without the loss of image quality. WMF files do not become jumbled or jagged as you zoom in or out on them.

The **MENU** command specifies the depiction file to appear on the ***toolbars***, ***menus***, and other areas. It is this graphic that you click on to select a symbol to draw. For the toolbar, using **bitmap** images is best, since there is no need for them to scale. Usually, it is best to create a ***16x16 pixel*** bitmap for each symbol that you want to represent in the toolbar.

> BMPs are raster files, which means that they store information about each pixel on an image. Although bitmaps can render rich, photo-quality images, they become jumbled when you zoom in or jagged when we zoom out.

The **<depiction-file>** is the name and full path of a bitmap or a metafile. You may specify a directory outside your encyclopedia's path, but it is advised to add the bitmaps and metafiles directly to the Files table of an encyclopedia's database.

To add your own depiction files to an encyclopedia, follow these steps:

1. **Make the necessary changes to USRPROPS.TXT.** An example for such code is:

   ```
   RENAME DIAGRAM “User 1” TO “Wireless Communications”
   RENAME SYMBOL “User 1” TO “Satellite”
   SYMBOL "Satellite"
   {ASSIGN To "Wireless Network"
   DEPICTIONS { DIAGRAM satellite.wmf }
   DEPICTIONS { MENU satellite_toolbar.bmp }
   }
   ```

2. **Import your .BMP and .WMF files into the encyclopedia's FILES table**. You may either use Rational System Architect's Encyclopedia File Manager (Tools, Encyclopedia File Manager), or SAEM (Start, Programs, IBM Rational, IBM Rational Lifecycle Solutions Tools, IBM Rational System Architect 11.3.1, SAEM – see its help on how to use), or Microsoft's Enterprise Manager to import your user-defined graphics files into the FILES table of the encyclopedia database. Encyclopedia File Manager can only import one file at a time. If you have multiple graphics files, we recommend you use SAEM to import the files into the FILES table.

   The names of the files that you import should be consistent with your Usrprops.txt code. In the above example, we have used a relative path by not specifying any path at all – simply listing satellite.wmf

and satellite.bmp. This means that the depiction files should be imported directly into the Files table of the database.

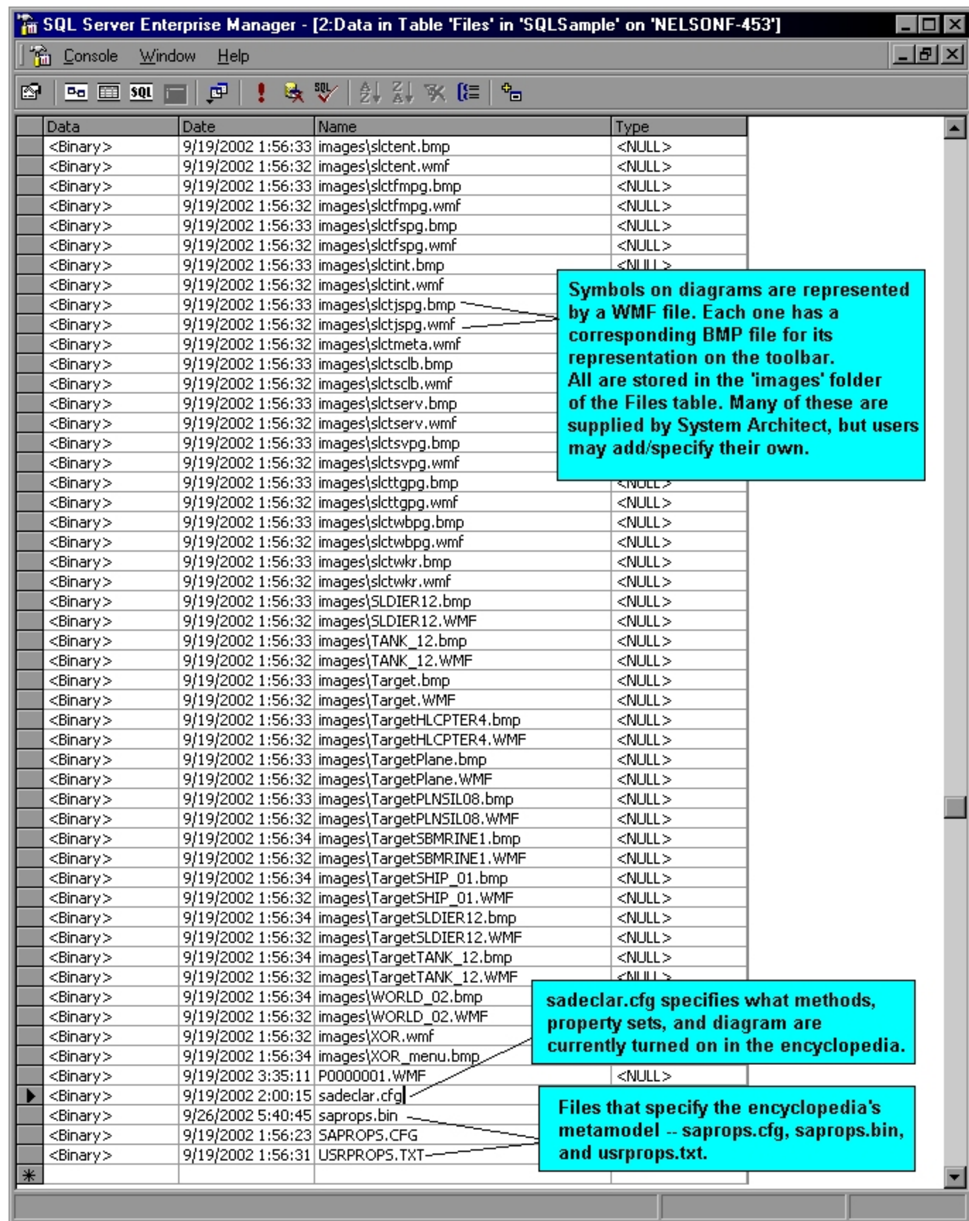
Recommendation: We recommend that you follow an established convention in Rational System Architect, and append the name of your depiction files with 'images/' to simulate that each depiction file is in an 'images' subdirectory of the FILES table. If you use SAEM to import multiple files at a time, make sure that they are in a directory that is named 'images', located anywhere on your computer. SAEM will automatically append the name of all files imported from a directory named images with 'images/' at the front of each graphic's file name. You should, likewise, specify the 'images\' before the name of each depiction file in your USRPROPS.TXT, which would make the above example:

```
RENAME DIAGRAM “User 1” TO “Wireless Communications”
RENAME SYMBOL “User 1” TO “Satellite”
SYMBOL "Satellite"
{ASSIGN To "Wireless Network"
DEPICTIONS { DIAGRAM images\satellite.wmf }
DEPICTIONS { MENU images\satellite_toolbar.bmp }
}
```

There are two advantages to using this strategy. First, it provides a sort of name independence and logical grouping strategy for user-specified images. Second, it is consistent with the way that images are handled when new encyclopedias are created – Rational System Architect takes all graphics in the ..\System Architect\images directory, places them in the FILES table of the new encyclopedia, and gives them a name that is appended with 'images\'.

See the figure below for a look inside the Files table of an encyclopedia's database, on how the 'images\' prefix to depiction files provides a logical grouping of images.

Figure 2-11. 'Files' Table of Encyclopedia Database.



3. **Reopen the Encyclopedia for the changes to take effect.**

Specifying Depiction Files for New Encyclopedias

If you are creating a new encyclopedia, you have an option – you can create the encyclopedia first and then import one or more user-provided graphics files into it via SAEM, Encyclopedia Manager, or SQL Server's Enterprise Manager as described above, or you may place your user-provided images into Rational System Architect's main ***images*** directory (under the main software directory – <C>:\Program Files\IBM\Rational\11.3.1\System Architect Suite\System Architect\images) before creating the encyclopedia. Rational System Architect takes all graphics in its main images directory and places them in the Files table of all new encyclopedias created.

If you wish the same user-specified graphics files to go into all new encyclopedias that you or other team members create, perform the following steps:

1. **Copy and Paste Your .BMP and .WMF files into Rational System Architect's 'Images' Subdirectory**. Before creating new encyclopedias, place your .BMP and .WMF files into the Images directory within the Rational System Architect main program directory. All team users that will be creating new encyclopedias at any time in the future should do this. These files will automatically be placed in the FILES table of the encyclopedia that are later created. Rational System Architect will append each file name with 'images\', so a figure called Fred.bmp will be created in the new encyclopedia's FILES table with the name images\Fred.bmp. This is a shortcut to creating the encyclopedia, and then importing the user-provided graphic files into the encyclopedia afterwards.

2. **Make the necessary changes to USRPROPS.TXT.** You use the DEPICTIONS command (and, optionally, the RETAIN STYLE command). Information on how to make the necessary code changes are provided in Rational System Architect's help. An example for such code is:

```
Rename Symbol "User 3" To "Radar
SYMBOL "Radar"
{ASSIGN To "Wireless Network"
DEPICTIONS { DIAGRAM RETAIN STYLE "C:\Program
Files\IBM\pictures\radar.bmp" }
DEPICTIONS { MENU "C:\Program
Files\IBM\pictures\radartoolbar.bmp" }}
```

3. **Reopen the Encyclopedia for the changes to take effect**

User-Defined Symbol Presentation Based on Property Value

You may specify how a symbol gets drawn *based on the value of a property* of the symbol's definition. In UML, this property is generally a stereotype. However, this functionality applies across the board to all symbol types, not just UML symbols, and not just to the stereotype property.

To enable this function, the **DEPICTIONS** clause is used directly within a **LIST** statement in USRPROPS.TXT.

```
LIST “New List Type”
{
VALUE “List Item One” DEPICTIONS {DIAGRAM
imageone.wmf MENU imageone_toolbar.bmp}
…
}
```

Example

In the following example, a new list is specified for Node Stereotypes. These stereotypes are applied to a Node symbol on a UML deployment diagram, so that a user may draw a node symbol using his or her own graphic files that he/she has imported into the FILES table of the encyclopedia database.

```
List "Node Stereotypes"
{
Value "Firewall" DEPICTIONS {DIAGRAM images\firewall.wmf
MENU images\firewall.bmp}
Value "Cell_Phone"  DEPICTIONS {DIAGRAM
images\cell_phone.wmf MENU images\cell_phone.bmp}
Value "Database"  DEPICTIONS {DIAGRAM images\data.wmf
MENU images\data.bmp}
Value "Hub"   DEPICTIONS {DIAGRAM images\hub.wmf
MENU images\hub.bmp}
Value "Modem"  DEPICTIONS {DIAGRAM images\modem.wmf
MENU images\modem.bmp}
Value "Multiplexer"  DEPICTIONS {DIAGRAM
images\multiplexer.wmf MENU images\multiplexer.bmp}
Value "PDA"  DEPICTIONS {DIAGRAM images\pda.wmf MENU
images\pda.bmp}
```

Value "Printer" **DEPICTIONS** {**DIAGRAM** images\printer.wmf
MENU images\printer.bmp}
Value "Projector" **DEPICTIONS** {**DIAGRAM**
images\projector.wmf **MENU** images\projector.bmp}
Value "Radio Tower" **DEPICTIONS** { **DIAGRAM**
images\radio_tower.wmf **MENU** images\radio_tower.bmp}
Value "Router" **DEPICTIONS** { **DIAGRAM** images\router.wmf
MENU images\router.bmp}
Value "Satellite" **DEPICTIONS** { **DIAGRAM** images\satellite.wmf
MENU images\satellite.bmp}
Value "Satellite Dish" **DEPICTIONS** { **DIAGRAM**
images\dish.wmf **MENU** images\dish.bmp}
Value "Scanner" **DEPICTIONS** { **DIAGRAM**
images\scanner.wmf **MENU** images\scanner.bmp}
Value "Server" **DEPICTIONS** { **DIAGRAM** images\server.wmf
MENU images\server.bmp}
Value "Switch" **DEPICTIONS** { **DIAGRAM**
images\kvm_switch.wmf **MENU** images\kvm_switch.bmp}
Value "Tablet_PC" **DEPICTIONS** { **DIAGRAM**
images\tablet_pc.wmf **MENU** images\tablet_pc.bmp}
Value "Terminal" **DEPICTIONS** { **DIAGRAM**
images\terminal.wmf **MENU** images\terminal.bmp}
}

SYMBOL "Node" in "Deployment"
{
PROPERTY "Stereotype" { **INVISIBLE** EDIT Text ListOnly
List "Node Stereotypes" DEFAULT "" LENGTH 32}
}

DEFINITION "Node"
{
PROPERTY "Stereotype"
{ EDIT Text **LIST "Node Stereotypes"** Default "" LENGTH 32 }
}

In the example USRPROPS.TXT above, note that the LIST of “Node Stereotypes” is referenced in both the SYMBOL and the DEFINITION of a node. In the SYMBOL, the property is made INVISIBLE. The SYMBOL maintains a reference to the stereotype that is specified for its underlying definition.

The USRPROPS.TXT code above changes the toolbar of a Deployment diagram, providing a drop-down list of

available stereotypes (and the corresponding bitmap for each).

Figure 2-12. User-Provided Depiction Files.

The user may select a stereotype and draw it on the diagram, where it is represented by the corresponding .WMF file.

After drawing the symbol, you may right-mouse click on each symbol and choose to:

- Display as <Node>, or
- Adorn with Stereotype (in which a thumbnail of the metafile is placed to the right of the symbol's name), or
- Display According to Stereotype.

Once drawn, you may specify the colors of a metafile as you would any other symbol in Rational System Architect, using the Symbol Style toolbar (or by selecting the symbol and choosing Format, Symbol Format,Symbol Style). This includes line coloring, fill coloring, font coloring, etc.

Retain Style

You may specify that metafiles that you provide retain their original graphical style and coloring when used in Rational

System Architect. You use the RETAIN STYLE keyword to specify this. For example:

```
LIST "Node Stereotypes"
{
VALUE "Firewall" DEPICTIONS {DIAGRAM RETAIN
STYLE images\firewall.wmf MENU images\firewall.bmp}
..
}
```

When drawn on the diagram, the user-provided metafile, firewall.wmf, is drawn with exactly the same colors as it is outside of Rational System Architect, and cannot be changed by Rational System Architect's color tools.

Displayable Properties on Depicted Symbols

Rational System Architect enables you to specify up to 37 properties to display on a symbol using the DISPLAY keyword. This is also true with symbols depicted by user-provided depiction files.

Please reference the section titled *Specifying the Display of Values on Symbols*, later in this chapter, for more information, or see the DISPLAY keyword in Chapter 3.

Specifying Properties for Diagrams, Symbols, and Definitions

There are three classes in every **Rational System Architect** encyclopedia: *diagram*, *symbol* and *definition*. Each can be defined with its own set of properties.

The following table includes all mandatory and optional entries found outside a Diagram, Symbol, or Definition statement in USRPROPS.TXT.

Table 2-3. Mandatory and Optional Entries for a Diagram, Symbol, or Definition Specifications

Entry	Mandatory Optional	Note
DIAGRAM { } or DIAGRAM BEGIN END or SYMBOL { } or SYMBOL BEGIN END or DEFINTION { } or DEFINITION BEGIN END	Mandatory	Begins and ends the declaration.
CHAPTER chapter_name	Optional	Includes subsequent properties in existing chapter, or adds new chapter

GROUP group_name
{
PROPERTY prop_name
PROPERTY prop_name
}
	Optional	Places all subsequent properties within one group for layout control

Entry	Mandatory Optional	Note
LAYOUT { alignment_criteria PACK_TAB_criteria COLS no_of_columns JUSTIFY }	Optional	[Align Body | Align Label | Align Over] [Pack | Tab] COLS <number>

PROPERTY property_name
{
}
	Mandatory	You may use { or BEGIN, and } or END

Specifying Properties for Diagram Types

The default property of all diagrams is *Description*. Description is defined as a text field 4074 characters. Diagram properties are those that a user may want to set for an entire diagram, such as whether to display swimlanes (or pools) vertically or horizontally. A typical **Diagram Properties** dialog is shown below.

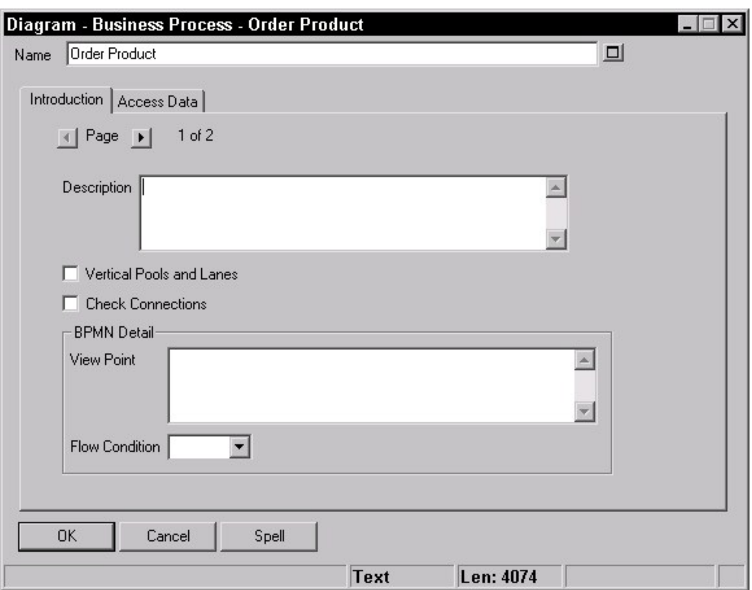


Figure 2- 13. Diagram Properties Dialog

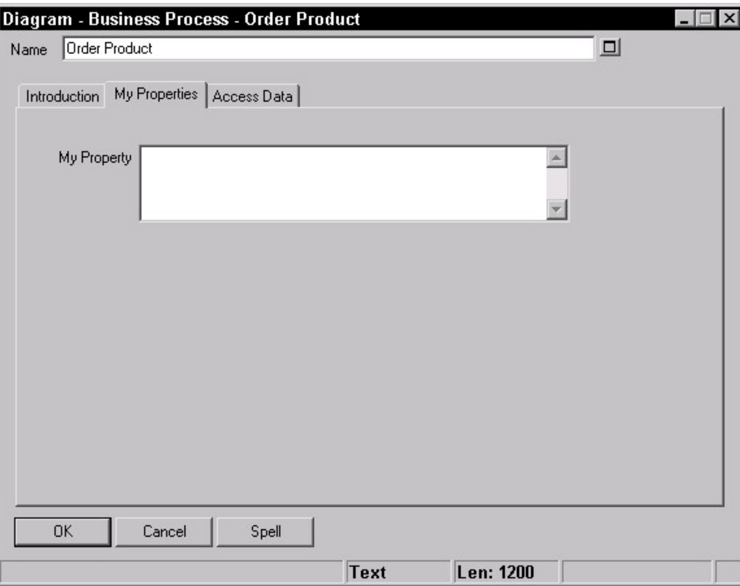
To add more properties for a diagram, use the following syntax:

```
DIAGRAM diagram_type
{
PROPERTY-1 <property_name>
  { <property_value> }
PROPERTY-2 <property_name>
  { <property_value> }
PROPERTY-3 <property_name>
  { <property_value> }
}
```

For example, adding the following statements to USRPROPS.TXT modifies the **Diagram Properties** dialog box for the Business Process diagram type, as shown in the picture that follows:

```
DIAGRAM "Business Process"
{
CHAPTER “My Properties”
PROPERTY "My Property" { EDIT Text LENGTH 1200 }
}
```

Figure 2-14. Revised Diagram Properties Dialog



Specifying Properties for Symbol Types

Symbol Properties are provided in the Symbol tab of a symbol's definition dialog.

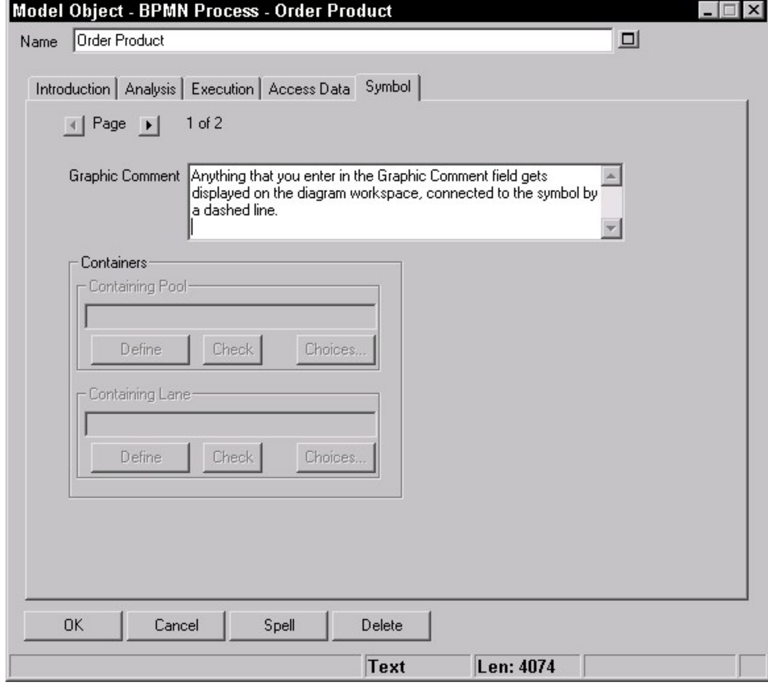


Figure 2-15. Symbol Properties Dialog Where the Default Property is *Graphic Comment*

Graphic Comment

The default property of all symbols is *Graphic Comment*. *Graphic Comment* is defined as a text field of 4074 characters. Anything that you enter in the Graphic Comment field is displayed as a comment on the diagram workspace, connected to the symbol by a line. The line is only drawn if the graphic comment is a certain distance from the symbol. You may adjust this distance by selecting the symbol and choosing Format, Diagram Format, Notation and adjusting the Line to Remote Text options.

You may also choose to have the graphic comment displayed inside the symbol (select the symbol and choose Format,

Symbol Format, Text Position, and toggle off the Place Graphic Comment Outside selection). You may also turn on/off display of the Graphic Comment completely (right-mouse click on the symbol and choose Display Mode, then toggle off Graphic Comment).

Adding More Properties for a Symbol

To add more properties for a symbol, use the following syntax:

```
SYMBOL symbol_type IN diagram_type
{
PROPERTY-1 <property_name>
  { <property_value> }
PROPERTY-2 <property_name>
  { <property_value> }
PROPERTY-3 <property_name>
  { <property_value> }
}
```

It is important that you specify the diagram type that the symbol you are referring to is contained in. A symbol may appear on many different diagram types.

Example

For example, we can make the following changes to USRPROPS.TXT:

```
SYMBOL "State" IN "State"
{
PROPERTY "Short Description"
  { EDIT Text LENGTH 1500 }
PROPERTY "Number" { EDIT Numeric LENGTH 4 }
}
```

These changes add *Short Description* and *Number* to the properties of a state symbol on a UML State diagram; *Graphic Comment* is always available. The modified dialog box is shown below:

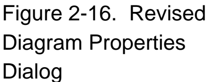
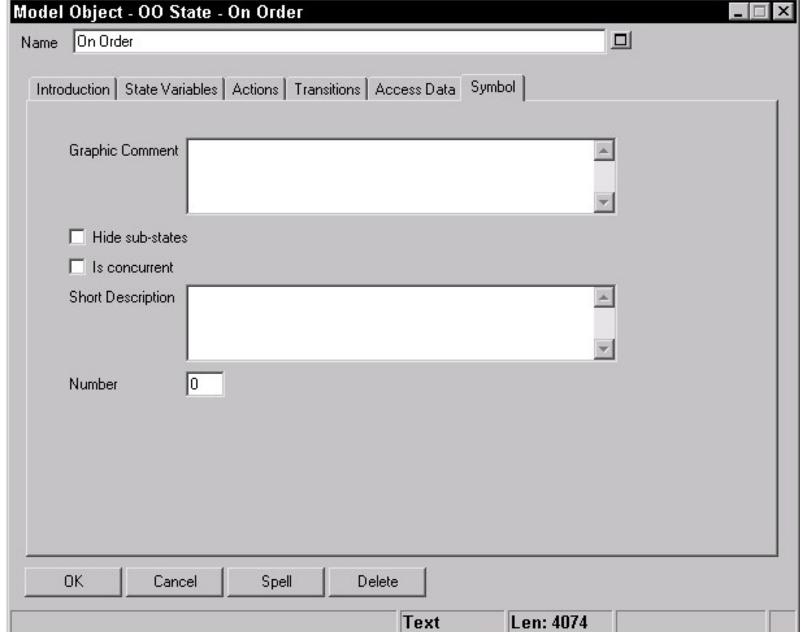


Figure 2-16. Revised Diagram Properties Dialog



Some symbol types occur on many different diagrams. Continuing with the example above, there are other types of state diagrams within Rational System Architect that have state symbols, such as the IDEF3 Object State Transition diagram, the OV-06b Op State Transition diagram, and the State Transition Ward & Mellor diagram. If we want the *Short Description* and *Number* properties to occur on these three types, we must include the property block three times: once for each diagram type.

```
SYMBOL "State" IN "IDEF3 Object State Transition"
{
PROPERTY "Short Description"
{ EDIT Text LENGTH 1500 }
PROPERTY "Number" { EDIT Numeric LENGTH 4 }
}
```

```
SYMBOL "State" IN "OV-06b Op State Transition"
{
PROPERTY "Short Description"
{ EDIT Text LENGTH 1500 }
PROPERTY "Number" { EDIT Numeric LENGTH 4 }
}

SYMBOL "State" IN "State Transition Ward & Mellor"
{
PROPERTY "Short Description"
{ EDIT Text LENGTH 1500 }
PROPERTY "Number" { EDIT Numeric LENGTH 4 }
}
```

Specifying Properties for Definition Types

All definitions have a *Name*. In addition, every definition has the default property *Description*, which we provide more details on later in this section. There really is no typical **Definition** dialog, since definitions tend to be unique within methodology and type. Below is a dialog for a Class definition.

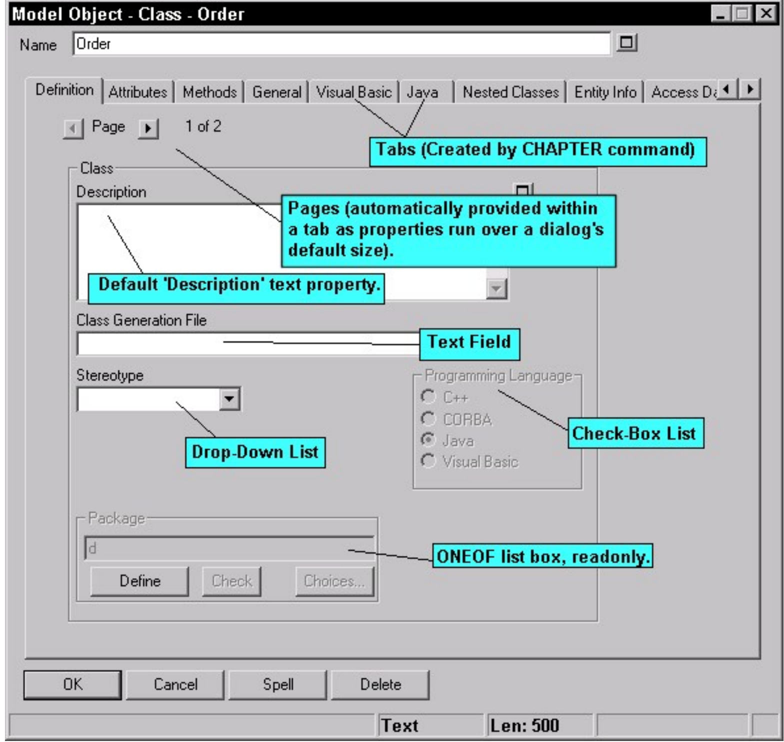


Figure 2-17. Model Object Dialog (Definition type Class)

Syntax

A definition block starts with the keyword *Definition* followed by a string (the argument) that is the name of the definition type. The name must be one of those known to Rational System Architect – either one found in SAPROPS.CFG, if

you are modifying or adding to an existing definition, or one that you have created by using the RENAME “User 1” through RENAME “User 150” commands (to create a new definition type, you rename one of these 150 user-provided definition types). Definition type names that have embedded spaces must be enclosed within double quotes (so for example, “User 1”).

The dictionary definition is bracketed by the `BEGIN...END` (or open/close braces **{ }**) keywords. Within the brackets are a set of defining commands, each consisting of the command keyword, `PROPERTY`, followed by its arguments. When you invoke a **Dictionary Object** dialog, its pages are populated by the properties named in the definition block.

Each PROPERTY entry has its own sub-set definition, again bracketed by the **{EDIT... }** braces and keyword. Each definition consists of phrases made up of keywords such as *Boolean, Date, Expression, ExpressionOf, ListOf, Minispec, Numeric, OneOf, Text*, and *Time*. Details of property definitions are given later in this section.

In summary, to add more properties for a definition, use the following syntax:

```
DEFINITION definition_type
{
PROPERTY-1 <property_name>  { <property_value> }
PROPERTY-2 <property_name>  { <property_value> }
PROPERTY-3 <property_name>  { <property_value> }
}
```

For example:

```
DEFINITION "Class"
{
CHAPTER "Definition"
GROUP "Class"
{
LAYOUT { COLS 2 ALIGN OVER TAB }
PROPERTY "Description"{ ZOOMABLE EDIT Text LENGTH
500 }
PROPERTY "Class Header File" { EDIT Text LABEL "Class
Generation File" LENGTH 80 }
PROPERTY "Stereotype" { EDIT Text LIST "Class
Stereotypes" INIT_FROM_SYMBOL Default "" LENGTH 20 }
..}
```

In the example above, the default first tab of the definition, which unless otherwise specified is “Introduction”, has been changed to “Definition” – that is what the CHAPTER “Definition” command does.

Description

As mentioned at the start of this section, every definition has the default property *Description*. Unless otherwise specified in SAPROPS.CFG, *Description* is defined as a text field of 4074 characters. You may increase a Description’s field size in USRPROPS.TXT by simply respecifying the Description property and increasing the number of characters. For example:

```
DEFINITION "Class"
{
PROPERTY "Description"  { EDIT LENGTH  16000 LINES
5 }
}
```

The above example specifies that the Description property of a class can hold 16,000 characters but only the first 5 lines are displayed in the class definition’s dialog.

Important Note: There are a few definition types within Rational System Architect that use the Description property for special purposes. For example, the definition of an Entity has been redefined as a LISTOF "Attribute" FROM "Data. The *Description* property for Process definitions have been redefined as *Minispec,* since the data contents of processes are generally minispecs, structured English, pseudo-code, and the like. In each of these cases, the Description property has been relabeled as well, to Attribute or Minispec, respectively. For example, here is the specification for a Process definition in SAPROPS.CFG:

```
DEFINITION "Process"
{
PROPERTY "Description"
{ EDIT Minispec LENGTH 750 LABEL "Minispec" }
..}
```

It is important to note that when writing reports, the name of the property is *Description*, and must be referred to as such.

Property Statements

You specify Property statements within Diagram, Symbol, or Definition specifications. The syntax of a Property statement is as follows:

```
PROPERTY property-name
{ EDIT edit-type
}
```

The following table includes all mandatory and optional entries for a property statement in USRPROPS.TXT.

Table 2-3. Mandatory and Optional Entries for a Property Statement

Entry	Mandatory Optional	Note

PROPERTY property_name
{
}
	Mandatory	You may use opening/closing braces, {..}, or BEGIN .. END statements.

EDIT edit-type	Optional	[Boolean | Date | ExpressionOf DATA | ListOf "dictionary-type" | Minispec | Numeric | OneOf "dictionary-type" | Text | Time]
LABEL label_string	Optional	The name of the control in the dialog; replaces the property name, which is the default

Entry	Mandatory Optional	Note
LENGTH length-argument	Optional	The maximum length of the field in characters 0 < numeric < 4095
LIST list-name	Optional	Indicates that a list of the list-name be displayed when the property is selected for input by the user; the user may select from the list or type in another value
LISTONLY LIST list-name	Optional	Indicates that the only input allowed is from an optionally displayed list
MINIMUM numeric MAXIMUM numeric	Optional	The minimum/maximum numeric value the field can have
DISPLAY { FORMAT format-type LEGEND legend-name }	Optional	Defines one of 37 possible displayable properties for symbol
DEFAULT default_string	Optional	If no user entry, entry in string is used

Table 2-3. Mandatory and Optional Entries for a Property Statement (Continued)

Entry	Mandatory Optional	Note
READONLY	Optional	Stops all input from either the keyboard or any displayed list
INVISIBLE VISIBLE	Optional	Makes the property invisible or visible. Use this entry to reverse the value in SAPROPS.
CHECKOUT initial-type	Optional	The value which is automatically completed when the dictionary entry is checked out. [DATE | TIME | AUDITID]
FREEZE initial-type	Optional	The value which is automatically completed when the dictionary entry is frozen [DATE | TIME | AUDITID]

Entry	Mandatory Optional	Note
INITIAL initial-type	Optional	The value which is automatically completed the first time the dictionary entry is accessed and saved [DATE | TIME | AUDITID]
UPDATE update-type	Optional	That value which is automatically completed the first and each subsequent time the dictionary entry is accessed and saved [DATE | TIME | AUDITID]
HELP	Optional	35-40 characters displayed in the dialog status bar when the control is in Focus.

Using ListOf, OneOf, and ExpressionOf

The ListOf, OneOf, and ExpressionOf keywords provide a very powerful concept that is used throughout the Rational System Architect's metamodel. Each provides you with the ability to say that a property of a definition references another object type – either Diagrams, Symbols, or Definitions.

So you are enabled to say that a class contains a list of methods (ListOf command), or a message between two objects references a method in the calling object (OneOf command), or that a process expresses procedures performed on data (ExpressionOf). We will look at these three expressions in turn in the sections to follow.

Note: As with any keyword specified in USRPROPS.TXT, the case of the ListOf, OneOf, or ExpressionOf keywords is unimportant. We use all capitals for all keywords throughout this manual, except in this section, since ListOf, OneOf, or ExpressionOf are more descriptive of these keywords than the 'all capitals' versions, LISTOF, ONEOF, and EXPRESSIONOF.

ListOf

The ListOf command enables you to specify that a property contains a list of other objects – diagrams, symbols, or definitions. For example, a Class contains a property called Attributes, which is a list of class attributes. Class attribute is a definition type in of itself, which has its own set of properties. The object type referenced must have been defined already in SAPROPS.CFG or at the top of the USRPROPS.TXT file.

Contrast the ListOf property to simple textual list. Elements in the ListOf list increase as users add definitions to the repository; for a simple list the number of elements in the list presented to the user is static (based on the LIST statement at the head of the USRPROPS.TXT file).

The syntax for the ListOf command is as follows:

> PROPERTY "Your Property" { EDIT **LISTOF** <”Referenced Definition Type"> LENGTH 1200}

Filtering the List of Items

The list of items provided in the list for a ListOf command can be filtered. Filter keywords are available such as OF DEFINITION REFERENCED IN and OF DEFINITION AND SUPERS REFERENCED IN. Please see Chapter 3 for more information on these keywords.

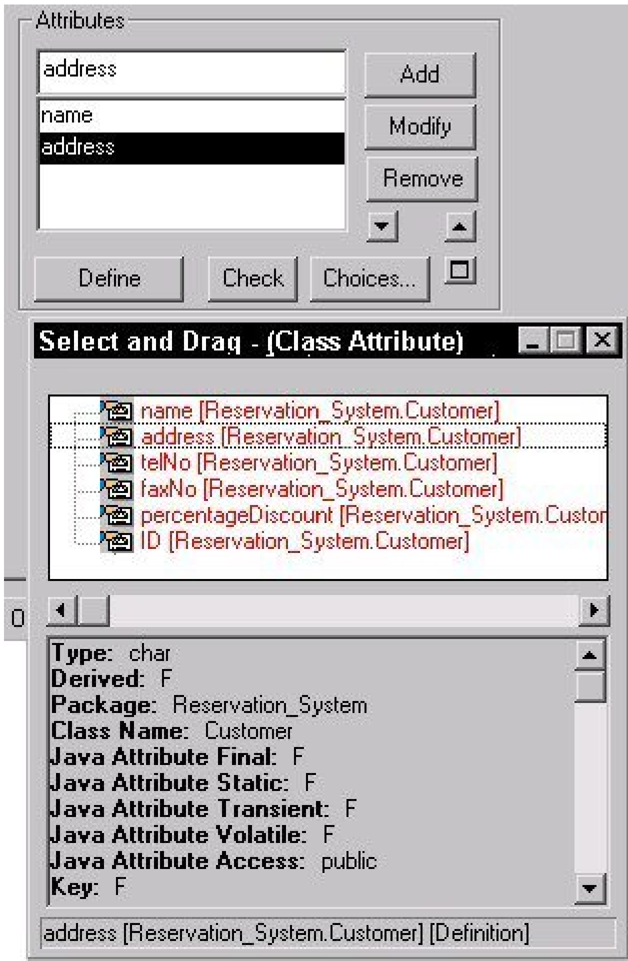
Example:

The example below shows the code for an object definition, which includes the property “Attributes”, which is a ListOf “Class Attributes”. “Class Attributes” is another definition type, defined in SAPROPS.CFG.

Definition “Object”
{ ..
PROPERTY "Attributes" { ZOOMABLE EDIT **LISTOF "Class Attribute"** OF DEFINITION REFERENCED IN "Class" KEYED BY {"Package", "Class Name":"Class", Name} LENGTH 4096 DISPLAY {FORMAT COMPONENT_SCRIPT _FmtNewUMLObjInstAttr LEGEND "$$FORCE$$"} }
..}

The figure below shows the default ListOf dialog created by the LISTOF command. It includes a Choices button, which, when pressed, presents a list of all definitions of the referenced type ("Class Attribute" in this example) in the encyclopedia. The OF DEFINITION REFERENCED IN command in the above code specifies that only those class attributes contained in the object's class are listed.

Figure 2-18. Example of LISTOF List.



Creating Grids for ListOf

You may present the items in a ListOf property as a grid by adding the ASGRID keyword.

Example:

```
Definition "Use Case"
{
CHAPTER "Steps"
PROPERTY "Use Case Steps" { EDIT COMPLETE
LISTOF "Use Case Step" KEYED BY { "Package", "Use
Case Name":Name, Name} ASGRID LENGTH 1200 }
 }
```

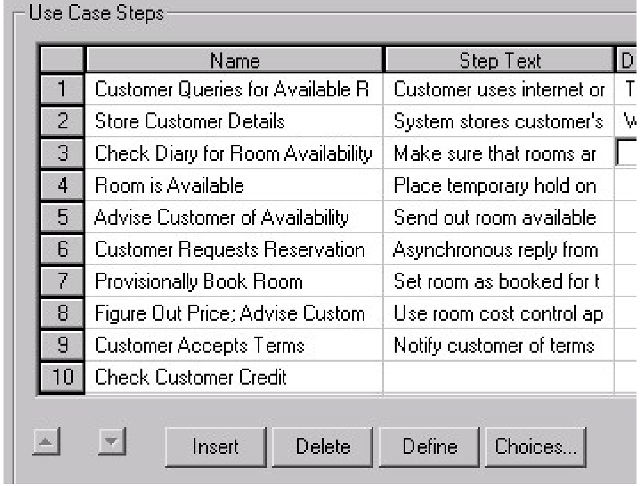


Figure 2-19. Example of LISTOF ASGRID List.

Heterogeneous Lists for ListOf

A typical ListOf statement provides a list of one object type. You may also create a list that references more than one object type using the HETEROGENEOUSLISTOF keyword.

Example:

```
Definition " Procedure"
{
PROPERTY "Underlying Procedure" { EDIT
HETEROGENEOUSLISTOF " Use Case",
"Class", "Method", "Use Case Step" READONLY}
..}
```

In the example above, the “Underlying Procedure” property of the “Procedure” definition can be populated with definitions of the type Use Case, and/or Class, and/or Method, and/or Use Case Step.

For more information on the HETEROGENEOUSLISTOF command, see Chapter 3.

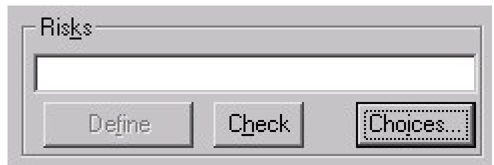
OneOf

A OneOf list box provides a list box that enables the user to select one, and only one, of a list of objects (Diagrams, Symbols, or Definitions) of a certain type. The object type referenced must have been defined already in SAPROPS.CFG or at the top of the USRPROPS.TXT file.

Example:

```
DEFINITION "Issue"
{
PROPERTY "Assigned To"{EDIT ONEOF "Risk" LENGTH
100}
..}
```

Figure 2-20. Example of ONEOF Listbox.



Filtering the List of Items

Similar to the ListOf list, the list of items provided in the list for a OneOf command can be filtered. Filter keywords are available such as OF DEFINITION REFERENCED IN and OF DEFINITION AND SUPERS REFERENCED IN. Please see Chapter 3 for more information on these keywords.

Hetereogeneous OneOf List

A typical OneOf statement provides a list of one object type. Similar to the ListOf list, you may also create a OneOf list that references more than one object type using the HETEROGENEOUS**ONE**OF keyword.

See Chapter 3 for more information on the HETEROGENEOUSONEOF command.

ExpressionOf

ExpressionOf allows you to express references to objects using complex operators and delimiters. While Rational System Architect expects you to use ExpressionOf to refer to data elements and data structures (DATA), it is not restricted to that use.

References defined with ExpressionOf are entered in a dialog box using the syntax

```
A + B + C

or

A +
B +
C

or

A
B
C
```

The elements may be written on one line or more than one line; the division between one element and the next is determined by white space. By convention, a + sign is used to divide the individual data items, but it is not required.

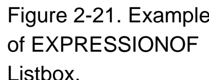
The following special operators and delimiters can be used in specifying expressions:

Table 2-5. Special Operators and Delimiters Used in Specifying Expressions

+	And (optional)
[...|...]	Either-or
{...}	Iterations of
i{...}j	Allow from 'i' to 'j' iterations of
(...)	The enclosed component is optional
@	The component is a key field
@n	The component is the nth element of the elements making up a compound key
...	The enclosed text is a comment
/.../	The enclosed text is a comment but has significance to the Schema Generator

Sub-expressions can be nested within other expressions. For example, ITERATIONS OF can be included within EITHER OR brackets.

[n1{...}n2 | n3{...}n4]



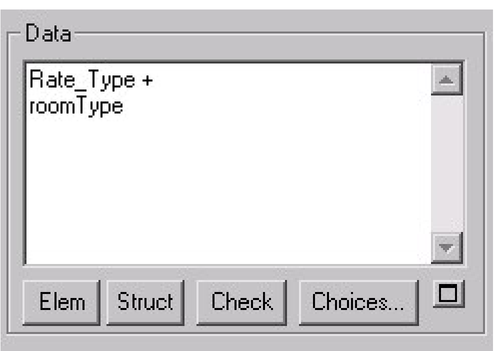


Figure 2-21. Example of EXPRESSIONOF Listbox.

ZOOMABLE Command

The **ZOOMABLE** command enables the user to temporarily expand the size of a list box in order to more easily enter or see large blocks of text. The most common places to add this command would be in a process definition, where minispecs are usually entered, or the description property of an entity, where foreign key information tends to be fairly long.

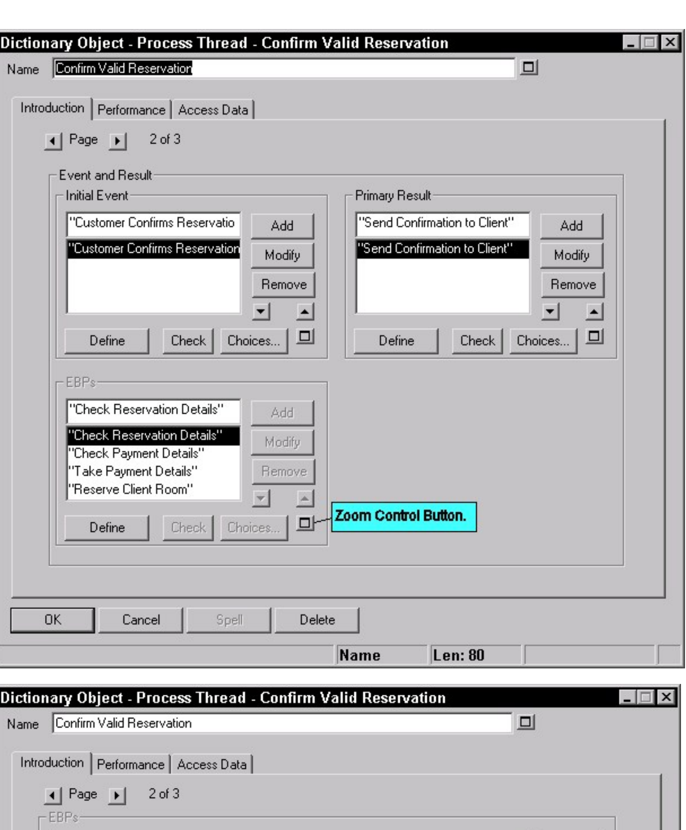
The command in USRPROPS.TXT is written:

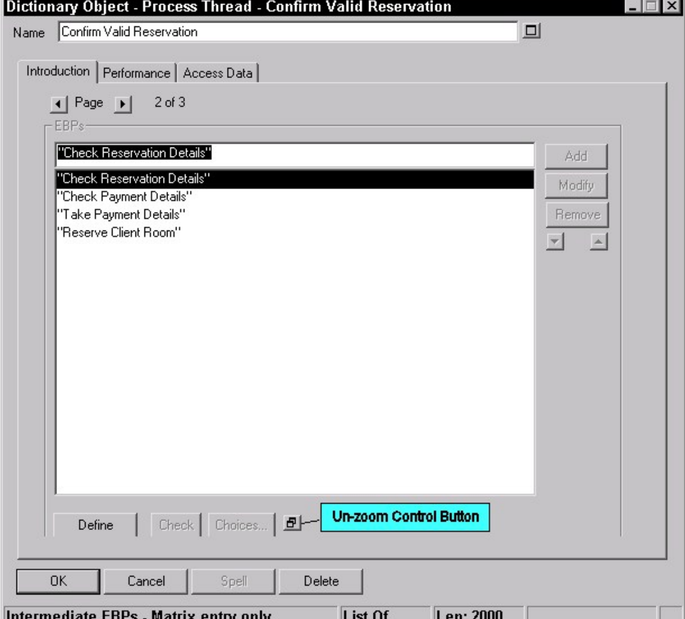
```
DEFINITION "Process"
{
PROPERTY "Description"
{ ZOOMABLE }
}
```

The **ZOOMABLE** command adds a small button to the right-hand corner of the list box. The button has a plus sign in it when the box is not zoomed, and a minus sign when it is zoomed.

The effect of the command is demonstrated by the two pictures in Figure 2-22. The top picture shows the minispec area in its usual, non-expanded state; on the bottom picture, the list box has been expanded to cover the entire dialog page.

Figure 2-22. A list box in a "non-zoomed" and "zoomed" state.





Modifying the Aesthetic Look of Dialogs

A certain amount of control can be obtained over the display of controls and their labels within the dialogs. For example, labels can be displayed over the control, directly next to the control, or separated by an amount of space determined by the longest label within a group.

It is not necessary to figure out how many controls can fit on one dialog page. Rational System Architect automatically computes the number of controls that fit on a dialog page based on the amount of space available for display, and breaks up a dialog into pages that you may flip through via a Page arrow in the upper left of a dialog.

You may specify your own arrangement of controls in a dialog through use of the **LAYOUT** command to specify columns, positioning of control labels, and justifications, the **CHAPTER** command to create tabs, and the **GROUP** command to create groups of property controls. In addition, you can specify the exact placement of each control and label for any given page of a dialog using positioning controls (see page 2-95, *Positioning Controls and Labels*, for instructions).

LAYOUT Command

The LAYOUT command enables you to specify how many columns property controls are laid out into in a dialog, and how the titles (or labels) of the control is positioned (to the left of the control or over it), etc.

Using the LAYOUT command is optional. If you do not use it, Rational System Architect deploys the default layout scheme. The default layout scheme is to have all controls laid out in one column, with the name (or label) of each control placed to the left of the control (the ALIGN LABEL command).

You may specify a LAYOUT command within CHAPTERS (which corresponds to a tab in the ensuing dialog) and GROUPS of a Diagram, Symbol, or Definition specification. The LAYOUT command has the following effects in a CHAPTER and GROUP:

Within a Chapter: You may specify a unique LAYOUT command for each Chapter of a Diagram, Symbol, or Definition specification. All property controls, including entire groups, are laid out according to the LAYOUT command of the Chapter. If you specify more than one LAYOUT command within a Chapter, all LAYOUT commands within that Chapter are ignored and the default layout is used instead.

Within a GROUP: You may specify a LAYOUT command within a Group, so that properties in the Group are laid out according to the group's LAYOUT specification.If you specify more than one LAYOUT command within a Group, all LAYOUT commands within that Group are ignored and the default layout is used instead.

An example ordering of LAYOUT commands and their effects is as follows:

```
DIAGRAM (or SYMBOL or DEFINITION)
  CHAPTER 1
    LAYOUT 1
      PROPERTY – laid out (in chapter) according to LAYOUT 1
      PROPERTY – laid out (in chapter) according to LAYOUT 1
      GROUP  – laid out (in chapter) according to LAYOUT 1
        LAYOUT 2
```

```
            PROPERTY – laid out (in group) according to LAYOUT 2
            PROPERTY – laid out (in group) according to LAYOUT 2
        GROUP  – laid out (in chapter) according to LAYOUT 1
          LAYOUT 3
             PROPERTY – laid out (in group) according to LAYOUT 3
             PROPERTY – laid out (in group) according to LAYOUT 3
    CHAPTER 2
     LAYOUT 4
         PROPERTY – laid out (in chapter) according to LAYOUT 4
    CHAPTER 3
     LAYOUT 5
         PROPERTY – laid out by default scheme because of 2
LAYOUT commands (5 and 6) in this Chapter
     LAYOUT 6
```

Layout of the "Introduction" Tab

Note that the first Chapter of a Diagram, Symbol, or Definition dialog, which includes the Description property, is always laid out by the default layout scheme – which is a one-column layout and the 'Description' label is to the left of the text box.

Default Layout Behavior

If a property control is too wide to fit within the specified column structure of a LAYOUT command, then that control is laid out by one column so that it fits in the dialog or Group; the other controls that are of sufficient width to be laid out according to the LAYOUT command are laid out accordingly.

For example, if you have specified a 4-column layout for a Group that itself is itself located in a Chapter (tab) that has a two-column layout specified, and one of the properties in the group is too wide to fit in the space available but the others are small enough to fit within a 4-column layout, the property that is too wide is laid out by itself, and the other properties are laid out to conform to the 4-column layout within the Group.

Example

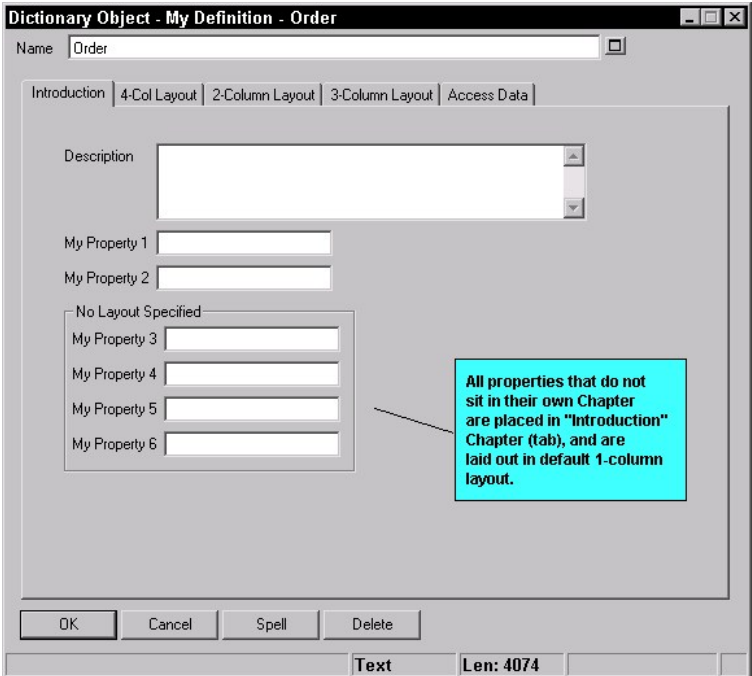
In the following example, we examine the LAYOUT command's effect inside CHAPTER and GROUP statements within a newly definied, user-specified definition.

```
RENAME DEFINITION “User 1” TO “My Definition”
DEFINITION "My Definition"
{
LAYOUT { COLS 3 ALIGN OVER TAB }
    PROPERTY "My Property 1"{ EDIT Text Length 10}
    PROPERTY "My Property 2"{ EDIT Text Length 10}
    GROUP "No Layout Specified" {
      PROPERTY "My Property 3"{ EDIT Text Length 10}
      PROPERTY "My Property 4"{ EDIT Text Length 10}
      PROPERTY "My Property 5"{ EDIT Text Length 10}
      PROPERTY "My Property 6"{ EDIT Text Length 10}
}
   CHAPTER "4-Col Layout"
    LAYOUT { COLS 4 ALIGN OVER TAB }
    GROUP "2-Column Group" {
        LAYOUT { COLS 2 ALIGN OVER TAB }
        PROPERTY "G1"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
        PROPERTY "G2"{EDIT Boolean LENGTH 1 DEFAULT "F"}
        PROPERTY "G3"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
        PROPERTY "G4"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
        }
    GROUP "1-Column Group" {
        LAYOUT { COLS 1 ALIGN OVER TAB }
        PROPERTY "Group Property 5"{ EDIT Text Length 10}
        PROPERTY "Group Property 6"{ EDIT Text Length 10}
        }
     PROPERTY "My Property 7"{ EDIT Text Length 5}
     PROPERTY "My Property 8"{ EDIT Text Length 5}
     PROPERTY "My Property 9"{ EDIT Text Length 5}
     PROPERTY "My Property 10"{ EDIT Text Length 5}
     PROPERTY "My Property 11"{ EDIT Text Length 1200}
     PROPERTY "My Property 12"{ EDIT Text Length 1200}

   CHAPTER "2-Column Layout"
     LAYOUT { COLS 2 ALIGN LABEL TAB }
      PROPERTY "My Property 13"{ EDIT Text Length 10}
      PROPERTY "My Property 14"{ EDIT Text Length 10}
      PROPERTY "My Property 15"{ EDIT Text Length 10}
      PROPERTY "My Property 16"{ EDIT Text Length 10}
      PROPERTY "My Property 17"{ EDIT Text Length 10}
      GROUP "3-Column Group" {
        LAYOUT { COLS 3 ALIGN OVER TAB }
```

```
        PROPERTY "G5"{ EDIT Boolean LENGTH 1 DEFAULT "T"}

    PROPERTY "G6"{EDIT Boolean LENGTH 1 DEFAULT "T"}
    PROPERTY "G7"{ EDIT Boolean LENGTH 1 DEFAULT "T"}
    PROPERTY "G8"{ EDIT Boolean LENGTH 1 DEFAULT "T"}
  }
CHAPTER "3-Column Layout"
LAYOUT { COLS 3 ALIGN OVER TAB }
PROPERTY "My Property 18"{ EDIT Text Length 10}
PROPERTY "My Property 19"{ EDIT Text Length 10}
PROPERTY "My Property 20"{ EDIT Text Length 10}
PROPERTY "My Property 21"{ EDIT Text Length 10}
PROPERTY "My Property 22"{ EDIT Text Length 10}
PROPERTY "My Property 23"{ EDIT Text Length 10}
}
```

We examine this USRPROPS.TXT code in the figures below. The first figure shows that the top-most layout command in the Definition, LAYOUT { COLS 3 ALIGN OVER TAB }, is ignored, since it is not assigned to any CHAPTER and it cannot override the layout of the first “Introduction” tab, which is set to default to a 1-column layout.

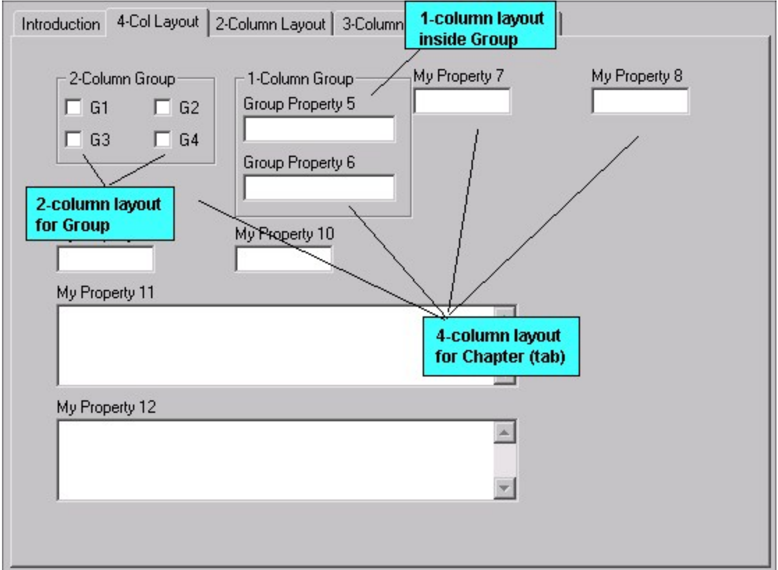
Figure 2-23. First LAYOUT command is ignored since it is not assigned to a CHAPTER (tab).



The second tab in the dialog is specified by the CHAPTER "4-Col Layout" command. Its layout is specifed as being 4-columns, with the title or label of each control placed over the control (CHAPTER "4-Col Layout" LAYOUT { COLS 4 ALIGN OVER TAB }.

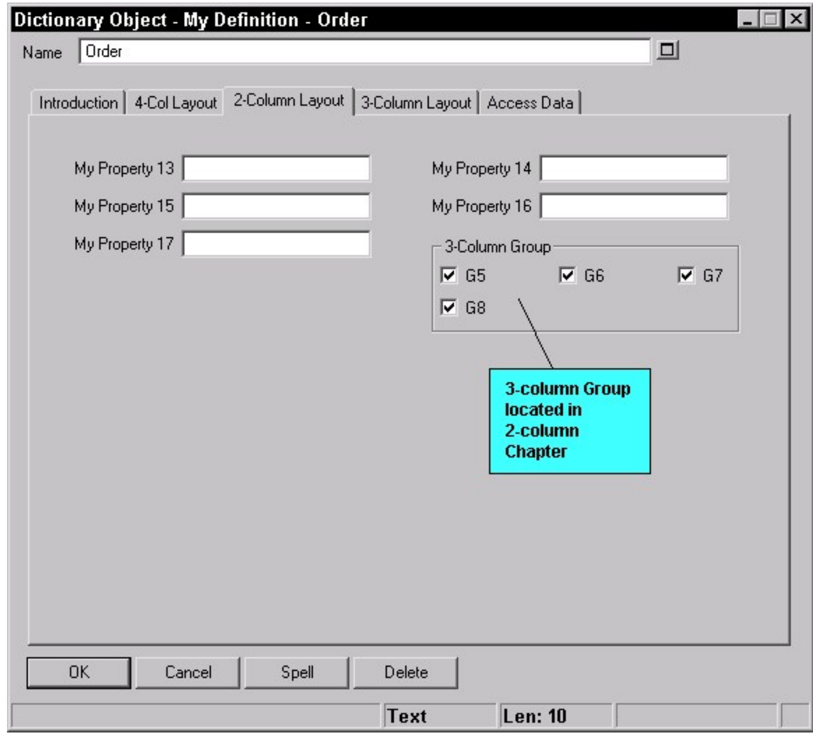
You can see from the figure below that even entire groups (such as "2-Column Group" and "1-Column Group" are laid out in the Chapter within a 4-column layout, as are properties (such as "My Property 7" through "My Property 10". Properties that are too wide to fit within the 4-column layout scheme are laid out by 1 column (such as "My Property 11" and "My Property 12", which are both Length 1200).

Figure 2-24. 4-Column Chapter Containing 2-Column and 1-Column Groups.



The 2-column Chapter, similarly, contains properties laid out in 2 columns, including a Group, within which properties are laid out in 3 columns.

Figure 2-25. 2-Column Chapter Containing 3-Column Group



The final Chapter contains properties in a 3-column layout. Note that these properties are narrow enough (Length 10) to fit in 3 columns.

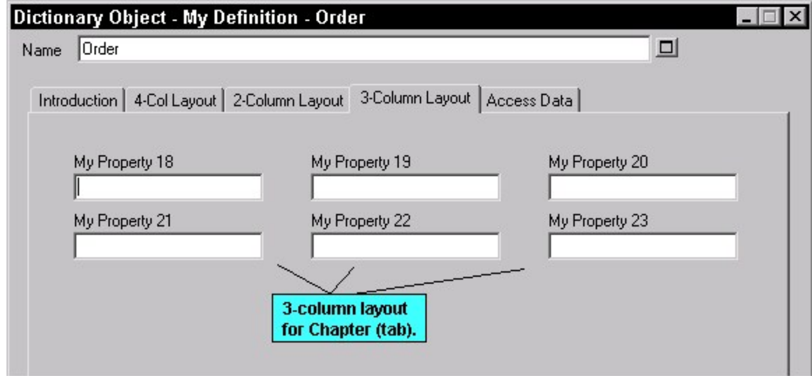


Figure 2-26. 2-Column Chapter Containing 3-Column Group

If any of these properties were too wide to fit in a 3-column layout, then that property would be laid out independently of the others, in a 1-column format. Changing “My Property 20” from LENGTH 10 to LENGTH 100 causes its control to be displayed as shown in the figure below. All other properties in the dialog remain laid out in 3-columns.

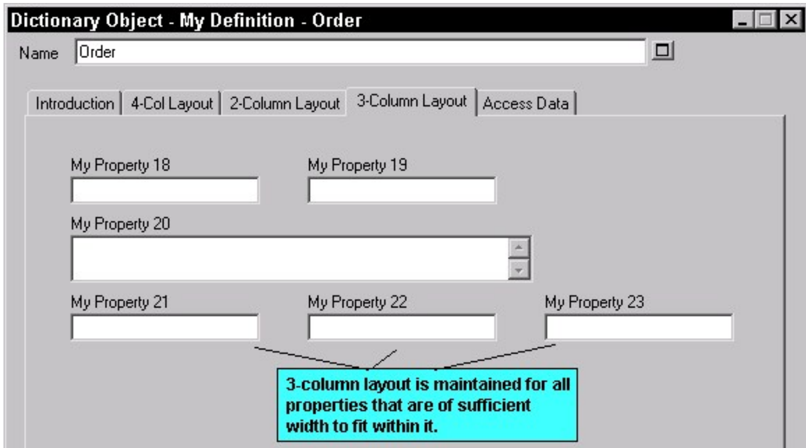


Figure 2-27. 2-Column Chapter Containing 3-Column Group, with one wide property.

LAYOUT Command Arguments

The valid values of the sub-commands used in the **LAYOUT** command are as follows:

LAYOUT { [ALIGN BODY | ALIGN LABEL | ALIGN OVER] [PACK | TAB] COLS <number> **}**

The sequence of the sub-commands is not important.

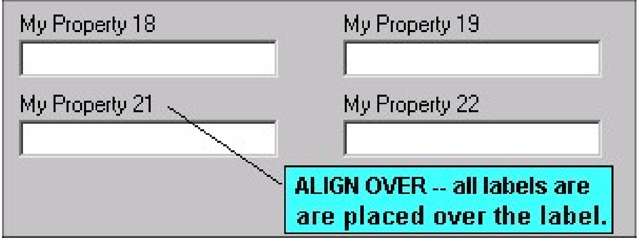
<u>Align Property Titles (or Labels) To Their Controls</u>

Every property has a title, or name. Remember you can relabel a property using the LABEL command. The ALIGN command takes the title of a property, or its label if it has been relabeled, and places it in a certain position next to the control itself, as follows:

1. **ALIGN BODY** and **ALIGN LABEL**: all controls are aligned one space to the right of the widest label in that column.

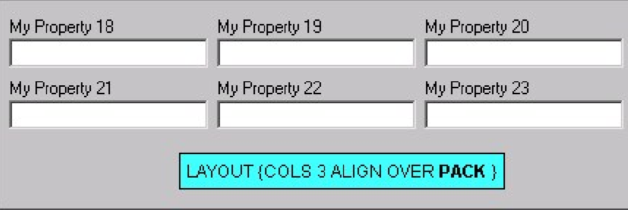
(Note – ALIGN BODY used to put all controls one space to the right of the label, but it was subsequently changed to be the same as ALIGN LABEL).

2. **ALIGN OVER**: label is over the control.

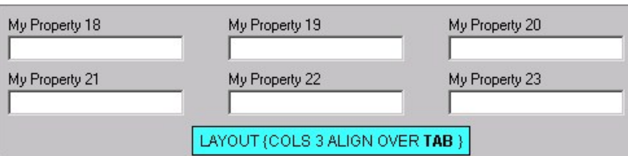


Vertical Positioning

1. **PACK***:* Sets of controls and labels in multiple columns are separated from the next set to the right by the minimum amount of space.



2. **TAB**: Controls and labels in multiple columns are separated by tabs so the entries in each row line up directly below the entries in the row above.



Columns

COLS <number_of_columns>: Controls the number of columns into which the properties are divided

Justify

JUSTIFY: This command is no longer used in SAPROPS.CFG or USRPROPS.TXT. It is ignored by the USRPROPS.TXT parser. It used to line up all controls to the right and left margin of the dialog page.

Creating Tabs with the CHAPTER Command

The **CHAPTER** command can be used to control the contents of a dialog page, and to produce a *tab*. If there is more information than will display within a tab, multiple pages within the tab are automatically created.

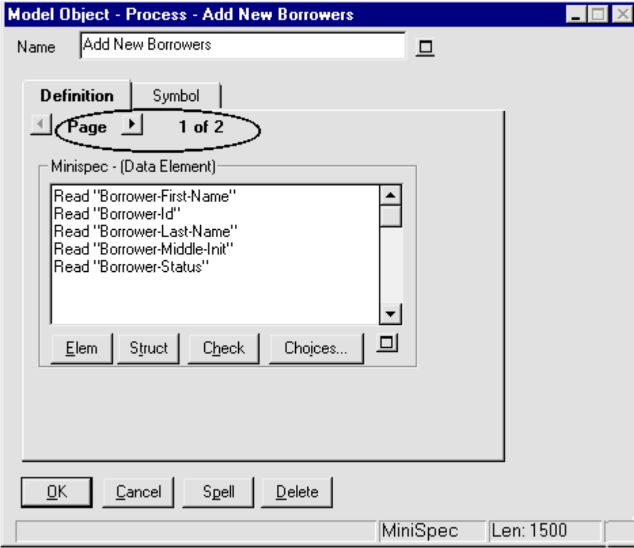


Figure 2-28. The Model Object dialog showing two pages for the Definition tab.

Syntax and Positioning of CHAPTER Command

To create a tab, you use the CHAPTER command and specify the name of the tab as an argument (using double quotes around the name if there are embedded spaces). The CHAPTER command does not require opening and closing braces, { }, or a BEGIN .. END statement block.

CHAPTER Name_of_Tab

or

CHAPTER “Name of Tab”

All properties listed in the specification after the CHAPTER command fall in that CHAPTER (or tab), until the next CHAPTER command is encountered. The **CHAPTER**

command may be placed at any point within a diagram, symbol, or definition specification of USRPROPS.TXT, except within a GROUP block.

> **Note:** The word “CHAPTER” is used for this command instead of the more obvious word “TAB” because “TAB” has always had a different meaning in USRPROPS.TXT (it is used in the LAYOUT command).

The following rules are in effect for the CHAPTER command:

- A property added via USRPROPS.TXT without a **CHAPTER** command is placed at the end of all definition property statements, and, therefore, on the last page of the last 'Definition' tab making up the definition dialog. (Please note that a definition may contain one or more tabs of information tied to the symbol, which come at the very end of the definition dialog. The first of these 'symbol' tabs is often named 'Symbol', but may be renamed to another name. The symbol tabs only appear if you open the definition dialog of a symbol on a diagram; if you open a definition from the explorer, they do not appear).
- You may add a property to an existing tab called out in SAPROPS.CFG by respecifying the tab with a **CHAPTER** command in USRPROPS.TXT. Properties that you add are placed at the end of that tab in the dialog.
- Tabs for new **CHAPTER** commands that you add to existing Diagram, Symbol, or Definition specifications are placed at the end of the dialog, after all pre-existing tabs.
- If a **GROUP** command is desired, it must be nested inside a **CHAPTER** command.

Using the LAYOUT Command Within a CHAPTER

The LAYOUT command may occur anywhere under a CHAPTER command, and it will have effect on all of the controls for properties within the CHAPTER. If you specify more than one LAYOUT command in a CHAPTER, the USRPROPS.TXT parser rejects them all and provides the default layout (COLS 1 ALIGN LABEL).

GROUP Command

The GROUP command is used to place a set of property controls in a Group box.

Group Command Syntax

The syntax of the command is as follows:

GROUP Name_of_Group

{ <properties to be enclosed in Group>

}

or

GROUP "Name of Group"

{ <properties to be enclosed in Group>

}

As shown above, the GROUP command requires that the properties in the group be specified within open and closed brackets, { }. If the name of the group contains embedded spaces, you must enclose the name in double quotes. You may also specify no name to the GROUP by specifying opening and closing double quote marks with no typing within them, for example:

GROUP ""

{ <properties to be enclosed in Group>

}

Important Note: All Group names within a Diagram, Symbol, or Definition specification must be unique, even if they are located in different Chapters. So, for example, if you create a Group "x" in one Chapter of a definition, and create a second Group "x" containing different properties, all properties of both Group "x"'s will be contained within one Group "x" that will be located in the first Group's Chapter. If you wish to have two or more Groups with similar names in the same specification, add a blank space(s) to subsequent occurances of the Group, for the above example, Group "x ".

Using the LAYOUT Command Within a GROUP

You may specify a LAYOUT command within a GROUP. If you do, it overrides the LAYOUT command of the definition or CHAPTER (tab) that the Group is in, only for the properties in the group.

Example

The following USRPROPS.TXT code is used to create the Groups shown in the figure below.

```
RENAME DEFINTION “User 4” To “My Group Definition”

DEFINITION “My Group Definition”
{
CHAPTER "4-Col Layout"
LAYOUT { COLS 4 ALIGN OVER TAB }
       GROUP "2-Column Group"
       {
       LAYOUT { COLS 2 ALIGN OVER TAB }
       PROPERTY "G1"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
       PROPERTY "G2"{EDIT Boolean LENGTH 1 DEFAULT "F"}
       PROPERTY "G3"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
       PROPERTY "G4"{ EDIT Boolean LENGTH 1 DEFAULT "F"}
        }
       GROUP ""
        {
        LAYOUT { COLS 1 ALIGN OVER TAB }
        PROPERTY "Group Property 5"{ EDIT Text Length 10}
        PROPERTY "Group Property 6"{ EDIT Text Length 10}
        }
}
```

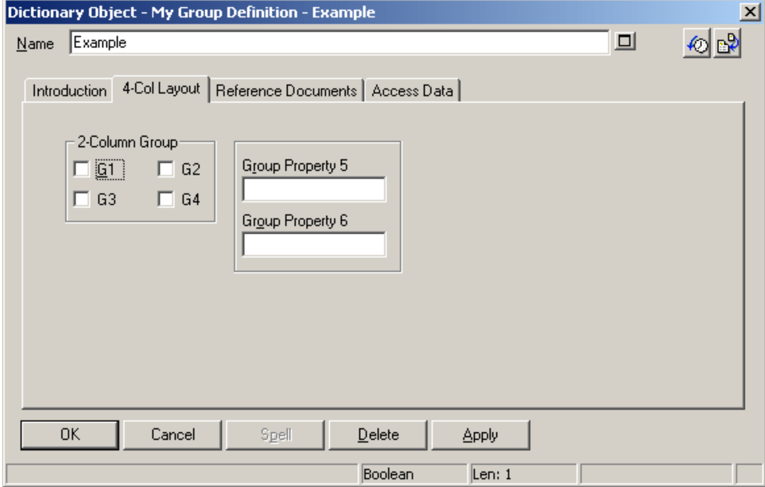


Figure 2-29. Use of the GROUP and LAYOUT Commands.

Positioning Controls and Labels

The syntax for exact placement is:

```
PLACEMENT { PROPPOS(n,n)
            PROPSIZE(n,n) }
```

Example 1:

```
PLACEMENT { PROPPOS(4,12)
            PROPSIZE(150,40) }
```

Additionally, you may specify label positioning. The syntax for exact label placement is:

```
PLACEMENT { LABELPOS (n,n)
            PROPPOS(n,n)
            PROPSIZE(n,n) }
```

Example 2:

```
PLACEMENT { LABELPOS (4,2)
            PROPPOS(4,12)
            PROPSIZE(150,40) }
```

In Example 1 above, PROPPOS (4,12) places the control 4 windows units horizontally, and 12 windows units vertically from the upper left-hand corner of the dialog. Or, to put it another way, 4 units to the left of the upper left-hand corner of the dialog, and 12 units down from that corner. PROPSIZE (150,40) makes the control 150 windows units wide and 40 windows units long.

In Example 2 above, LABELPOS places the label for the control 4 windows units horizontally, and 2 windows units vertically from the upper left-hand corner of the dialog. The label, therefore, is the same distance from the edge of the dialog, but 10 windows units above the control.

Important Note: You should **not mix** PLACEMENT with default positioning commands within a CHAPTER. Doing so will cause odd positioning results.

A portion of the syntax from SAPROPS.CFG for the **Entity Definition** dialog follows.

```
CHAPTER "SQL Server Triggers & Table Segment"
GROUP "Default Referential Integrity Triggers"
        LABEL "Default Referential Integrity" {
        LAYOUT { COLS 1 ALIGN LABEL TAB }

PROPERTY "Insert Trigger Name"
        { EDIT Text LENGTH 31
        LABEL "Insert Trigger"
        PLACEMENT {LABELPOS(4, 24)
            PROPPOS(50, 24) PROPSIZE(110, 12)} }

PROPERTY "Update Trigger Name"
        { EDIT Text LENGTH 31
        LABEL "Update Trigger"
        PLACEMENT {LABELPOS(4, 38)
            PROPPOS(50, 38) PROPSIZE(110, 12)} }

PROPERTY "Delete Trigger Name"
        { EDIT Text LENGTH 31
        LABEL "Delete Trigger"
        PLACEMENT {LABELPOS(4, 52)
            PROPPOS(50, 52) PROPSIZE(110, 12)} }
}
```

Some General Sizing Rules

In most cases, you will have to make some modifications to the length (the X coordinate) to make it fit the way you want.

1. Check boxes are PROPSIZE (30, 12)
2. OneOf properties are PROPSIZE (150, 40)
3. ListOf properties are PROPSIZE (320, 98)
4. Short text fields are approximately PROPSIZE (<LENGTH*3>, 12), rounding off the width (x) coordinate for cosmetics
5. Long text fields, such as LENGTH 4074, are about PROPSIZE (150,115).

Some General Placement Rules

Again, you'll probably have to make some modifications once you see the way the dialog is laid out, but these numbers may be helpful to start.

Table 2-4 Positioning and size when the property is in a group and the label is over the property.

PROPERTY Type	PLACEMENT - Left-hand column
ListOf	PLACEMENT { PROPPOS (4, 24) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (4, 24) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (4, 24) PROPSIZE (LENGTH * 3, 12) }
PROPERTY Type	**PLACEMENT - Right-hand column**
ListOf	PLACEMENT { PROPPOS (165, 24) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (165, 24) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (165, 24) PROPSIZE (LENGTH * 3, 12) }

Table 2-4. Positioning and size when the property is in a group and the label is not over the property.

PROPERTY Type	PLACEMENT - Left-hand column
ListOf	PLACEMENT { PROPPOS (4, 12) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (4, 12) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (4, 12) PROPSIZE (LENGTH * 3, 12) }
PROPERTY Type	**PLACEMENT - Right-hand column**
ListOf	PLACEMENT { PROPPOS (165, 12) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (165, 12) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (165, 12) PROPSIZE (LENGTH * 3, 12) }

Table 2-4 Positioning and size when the property is not in a group and the label is not over the property.

PROPERTY Type	PLACEMENT - Left-hand column
ListOf	PLACEMENT { PROPPOS (4, 2) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (4, 2) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (4, 2) PROPSIZE (LENGTH * 3, 12) }
PROPERTY Type	**PLACEMENT - Right-hand column**
ListOf	PLACEMENT { PROPPOS (165, 2) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (165, 2) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (165, 2) PROPSIZE (LENGTH * 3, 12) }

Table 2-4 Positioning and size when the property is not in a group and the label is over the property.

PROPERTY Type	PLACEMENT - Left-hand column
ListOf	PLACEMENT { PROPPOS (4, 14) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (4, 14) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (4, 14) PROPSIZE (LENGTH * 3, 12) }
PROPERTY Type	**PLACEMENT - Right-hand column**
ListOf	PLACEMENT { PROPPOS (165, 14) PROPSIZE (320, 98) }
OneOf	PLACEMENT { PROPPOS (165, 14) PROPSIZE (150, 40) }
EDIT Text (less than 75 characters)	PLACEMENT { PROPPOS (165, 14) PROPSIZE (LENGTH * 3, 12) }

Table 2-4 Positioning and size when properties are below properties in the same dialog.

PROPERTY add to endpoint of previous property	PLACEMENT - Left-hand column
A +14 if label is OVER Property B	PLACEMENT { PROPPOS (4, 14) PROPSIZE (150, 40) }
B + 4 if Property C is OneOf	PROPERTY B PLACEMENT { PROPPOSE (4, 68) PROPSIZE (150, 40) }
C	PLACEMENT { PROPPOS (4, 112) PROPSIZE (150, 12) }
PROPERTY Type	**PLACEMENT - Right-hand column**
D + 4 is label of Property E is on the side	PLACEMENT { PROPPOS (165, 14) PROPSIZE (320, 98) }

E + 4 because a LABELPOS is used for Property F	PLACEMENT { PROPPOS (165, 116) PROPSIZE (150, 40) }
F	PLACEMENT { LABELPOS (165, 160) PROPPOS (165, 170) PROPSIZE (30, 12) }

Note: LABELPOS is optional. It's used to override the LAYOUT command for the group. The y coordinate is 10 units higher than the PROPPOS y coordinate.

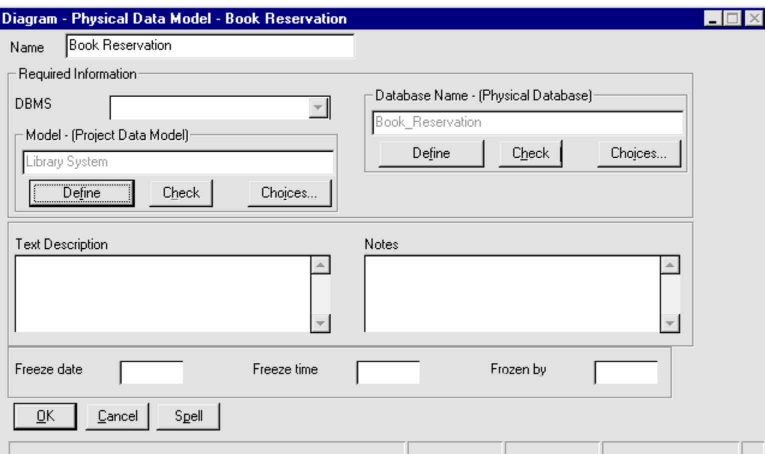


Figure 2-30. The Result of the Chapter Commands

Specifying the Display of Values on Symbols

A symbol represents a definition in the repository. That definition has properties. You may specify that definition properties and their values get displayed on a symbol. By default, the name of a symbol (which is a property of the symbol and its definition) is displayed. To specify that other properties of a symbol's definition are displayable, you use the **DISPLAY** command in each property's declaration. For example:

```
Definition "My Definition"
{
Property "My Property 1" {EDIT TEXT LENGTH 20
DISPLAY { FORMAT String LEGEND "" } }
}
```

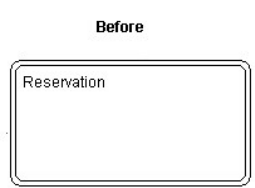
The example code above makes the property "My Property 1" displayable on the symbol – any text that you type into the property's 20-character text box in the definition dialog is displayed on the face of the symbol.

Once you specify certain properties of a symbol's definition as being displayable, those properties are provided in the **Display Mode** dialog for a symbol, where they can be turned on or off at any time. The Display Mode dialog is accessed by selecting a symbol on a diagram, and selecting **View, Display Mode**, or right-mouse-clicking on a symbol and choosing **Display Mode**. The Display Mode dialog enables you to select all displayable properties that you wish to display for the symbol.

The figure below shows an entity symbol – the Before picture shows it with all displayable properties turned off; the After picture shows the Key Data and Non-Key Data turned on.

Figure 2-31. Rectangular Symbol with Properties Displayed and Not Displayed

Before



After

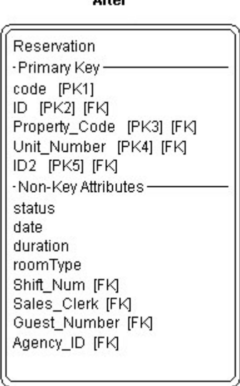
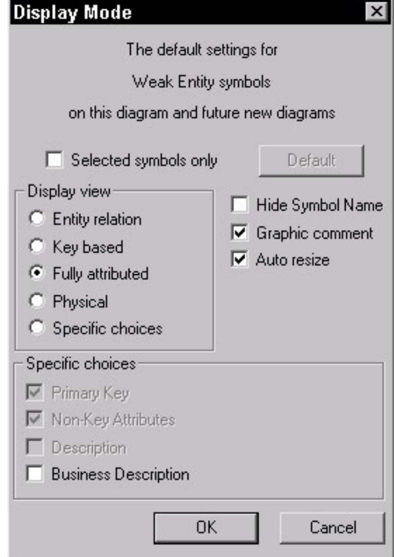


Figure 2-32. Example Display Mode Dialog.



You may also specify displayable properties for line symbols, as shown in the figure below.

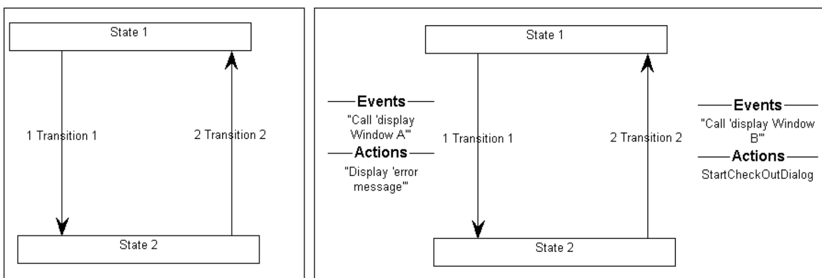


Figure 2-33. Line Symbol with Properties Displayed and Not Displayed

Syntax of the DISPLAY Command

There is a limit of **eight** display statements for one definition. The syntax of the DISPLAY command is as follows:

DISPLAY { FORMAT [STRING | LIST | KEY | NONKEY |
COMPONENT_SCRIPT | COLUMN_SCRIPT | SCRIPT]
LEGEND " (how the block is labeled in the symbol) **" }**

- STRING: This is the default. It causes the values of the property to appear on the symbol exactly as they are typed. This choice is a good one if you want comments to be displayed.
- LIST: Causes items to be displayed on the symbol in a list – each whitespace character causes a new line, unless the whitespace falls within double quotes.
- KEY: Use this keyword for properties designated as keys. They are displayed in a separate section of the symbol. See KEY keyword in Chapter 3 for an example.
- NONKEY: You may use this keyword for non-key properties. They will be displayed in a separate section of the symbol. This keyword was originally used for entities and tables in Rational System Architect's data modeling support. See the NONKEY keyword in Chapter 3 for an example.
- COMPONENT_SCRIPT: calls a script that displays the property value on the symbol in a special format, devised by the script. The script itself is either hard-coded in the product, or written by the user using Rational System Architect Basic language. By convention, the script itself is named with one of the following prefixes:

- fmtxxx – The function itself exists in hard code and cannot be modified. Most functions in SAPROPS.CFG are this way. Hard-coding the function is done to make Rational System Architect's overall response faster.
- _fmtxxx – Exists in the fmtscript.bas file within Rational System Architect's main executable directory, and is coded using SA Basic.

The component scripts are used for ListOf and ExpressionOf properties. The action taken by the script works against each item in the list. For example, a Component Script is used to look at each class attribute in a class definition, and construct how it will be displayed on the class symbol – providing a '-' mark before the name if the attribute's access property is set to private, or a '+' mark if it is public, and displaying the attribute's return type after the attribute name, preceded by a colon. Similarly, a Component Script constructs the way a method is displayed, with a '+' mark preceding the name if its access is public, and a '-' mark if its access is private, and in addition displays its parameters and their type in parenthesis after the method's name.

Diary

- RoomTypeAvailabilty : short

+ confirmReservation (reservationNumber : char) : short
+ decreaseAvailability (date : char, duration : long, roomType : char) : void
+ fillReservation (Reservation : long) : void
+ getAvailability (date : char, duration : long, roomType : char) : long
+ showAvailability (date : char, duration : char, roomType : char) : long
+ showReservation (startDate : char) : long
+ showRoomDirections (room : char) : long
+ showRoomRate (roomType : char) : float

For more information, see COMPONENT_SCRIPT and SCRIPT in Chapter 3.

- COLUMN_SCRIPT: Works like COMPONENT_SCRIPT, calling a script to apply a special formatting to a property

value displayed on a symbol. The script is either a hard-coded script or one written by the user using SA Basic (and placed in the fmtscript.bas file within Rational System Architect's main executable directory). The column scripts are used for displaying columns in table symbols in a physical data model. The action taken by the script works against each column in the list. See COLUMN_SCRIPT in Chapter 3 for more information.

- SCRIPT: Works like COLUMN_SCRIPT and COMPONENT_SCRIPT, calling a script to apply a special formatting to a property value displayed on a symbol. The SCRIPT command calls scripts used for properties that are neither ListOf nor ExpressionOf. The script itself is either a hard-coded script or one written by the user using SA Basic (and placed in the fmtscript.bas file within Rational System Architect's main executable directory). See the SCRIPT keyword in Chapter 3 for more information.

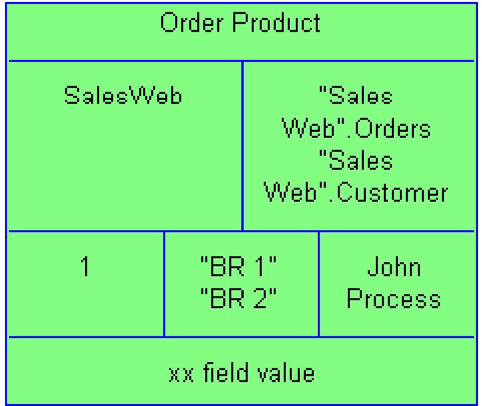
Each property group that is displayed is separated from each other by a dividing line. You can specify a label, or "legend" to appear on the dividing line, using the LEGEND command. If a LEGEND is not supplied, the property name itself is the label. The following LEGEND commands are available:

- **LEGEND "<Your Text>":** Whatever text you place in the quotation marks will be displayed on the symbol above the entry, only if there is a value for the entry.

- **LEGEND "":** Displays a straight line without any words, only if there is a value for the entry.

- **LEGEND "$$FORCE$$":**Displays a horizontal line above the entry on the symbol. This line acts as a divider. The "$$FORCE$$" keyword is different than simply using " ", in that it forces display of a horizontal line even if the property display is suppressed through the display mode dialog.

- **LEGEND "$$NONE$$":** Does not display a horizontal line above the entry on the symbol, whether or not there are values for the entry. This line normally acts as a divider.
- **LEGEND "$$VFORCE$$":** Enables you lay out properties from left to right inside symbols, and draws vertical lines between them. An example is shown below:

  See VFORCE keyword in Chapter 3 for the example USRPROPS.TXT that creates the picture above.
- **LEGEND "$$VNONE$$":** Enables you to lay out properties from left to right, but *does not* provide a dividing line. See VNONE keyword in Chapter 3 for an example.

The typeface and font of the displayable legends are controlled through the **Diagram Format**, **Notation** command under the **Format** menu.

Example

In the following example, we specify a new diagram type, new symbol type, and new definition type. We specify that the new symbol type is defined by the new definition type, and assign the new symbol type to the new diagram type. We create a property for the definition, called "My Important Property". We specify that the legend "My Important Property

Displayed" should be displayed on the symbol on the dividing line above the displayed value.

```
RENAME DIAGRAM "User 1" TO "My Diagram"
RENAME SYMBOL "User 1" TO "My Symbol 1"

SYMBOL "My Symbol 1"
{
DEFINED BY "My Definition 3"
ASSIGN TO "My Diagram"
}
DEFINITION "My Definition 3"
{
PROPERTY "My Important Property" { EDIT Text Length
20 DISPLAY { FORMAT String LEGEND "My Important
Property Displayed" }}
}
```

The figure below shows a subsequent symbol drawn on such a diagram, and the displayed value typed into the property field.

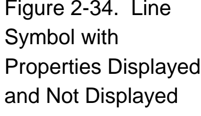
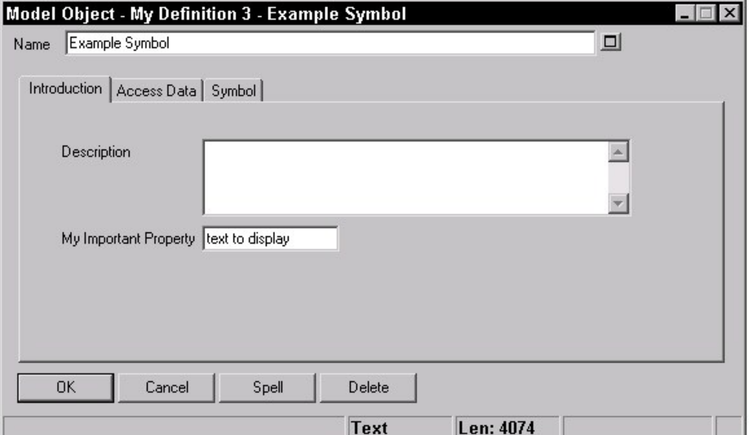


Figure 2-34. Line Symbol with Properties Displayed and Not Displayed



Specifying Key and Keyed By Properties

Establishing KEY Properties

You may specify that a particular definition is ‘keyed’ to one or more other definitions. A key determines the name space of a definition in the encyclopedia. For example, a class attribute definition type is keyed by its containing class definition, and that class’s containing package definition. So you could have two attributes called name, one belonging to the class Customer in the package Reservation_System, and the other belonging to the class Product in the Order_System package. The two attributes, although having the same name, are distinctly different definitions.

By default, every modeling element in an encyclopedia is already secretly keyed to three things -- its *class* (here *class* is used in Rational System Architect terms, distinguishing whether it is a diagram, symbol, or definition), its *type* (whether it is a UML Use Case diagram, a BPMN Process diagram, etc), and its *name* (for example, the Reservation_System Use Case diagram versus the Human_Resource_System Use Case diagram).

You may add key properties to a definition by using the KEY command. You specify the KEY command within the property that you want to be a key of a definition. The KEY command may be placed almost anywhere within the description of a property, but because of its importance, it is customary to place it as the first item within the property’s braces – just before the EDIT keyword.

Note: It is not possible to add a KEY EDIT ONEOF to a diagram.

Example 1:

```
Definition “Use Case”
{
PROPERTY "Package" { KEY EDIT …}
..}
```

Example 2:

```
Definition “Use Case Step”
{
PROPERTY "Use Case Name" { KEY EDIT …  }
PROPERTY "Package" { KEY EDIT …}
             …
```

> **Note:** Key properties of a definition are not shown in a grid formed by an ASGRID command. For example, in a Use Case definition, Use Case Steps are depicted in a grid formed by an ASGRID command, however, the key properties of Use Case Steps (owning package and Use Case) are not shown in the grid of Use Case Steps.

For a property that is a key and that “points at” another object(s) – for example, a LISTOF or ONEOF property, not a simple TEXT or NUMERIC property – the end user must specify the class and the class type of the referenced object(s) when entering a value for the property while working in Rational System Architect.

For example:

```
Definition “Business Process”
{
PROPERTY “System Use Case” {EDIT ONEOF “Use Case” …}
```

The statement above indicates that the property “Use Case Name” refers to a definition of type “Use Case”. Definition is the default when no *class* is specified (*class* in the Rational System Architect sense -- Diagram, Symbol, or Definition).)

The property value itself often contains all the necessary remaining material needed to identify the object(s) actually being referenced. If the referenced class/type of the property has no key properties, the reference value will just be the object’s Name (because the class and type are known), but if the referenced class/type has key properties (such as “Use Case” in the above example, which has key property “package”), Rational System Architect must know the values of these key properties in order to properly identify the reference object.

> **Note:** Heterogeneous reference properties are different in this respect. See HETEROGENEOUS in Chapter 3.

You either code this into USRPROPS.TXT so that Rational System Architect automatically gets the values for the end user or you force the end user to type in the fully qualified name, with periods separating the key parts.

- To have Rational System Architect automatically get the value for users, you use the KEYED BY command.
- If a KEYED BY clause is not given for the property, Rational System Architect expects these additional key values to be given in the reference itself – in other words the user must type in the fully qualified name of the reference object, with periods separating key values (for a Use Case Step called "*Specify email"* in a Use Case called *Order_Product* in a package called "*Order System"* the user would need to type in "*Order System".Order_Product."Specify email"*.

> **Note:** When a component contains a syntactically significant character (such as a space or a period), it must be enclosed in double quotes so that Rational System Architect can parse the reference properly.
>
> Here are two examples of references to a "Use Case Step":
>
> **Order_System.Order_Product."Specify email"**
>
> where *CorrectInvoice* is the name of the Use Case that belongs to the *Accounts_Payable* package.
>
> **"Order System"."Order Product"."Specify email"**
>
> where *Order Product* is the name of the Use Case that belongs to the *Order System* package.

Using KEYED BY to Make Sure All Members of a Group Are Of the Same Type

One other use for the KEYED BY clause is that it enables you to build a list of things that are all related. For example, all the Use Case Steps referred to in the property “Use Case Steps” of a Use Case definition belong to the same Use Case – as it happens, the one containing the “Use Case Steps” property. Where a multiple reference property (like ListOf) refers to objects all belonging to the same parent object, it is advised to use one or more other properties to identify the parent object. In these situations, a KEYED BY clause is used to tell Rational System Architect which other properties to use.

How To Use the KEYED BY Clause

So to summarize, the KEYED BY clause is optionally used to specify how the key components of a referenced object(s) may be found. It provides two key benefits:

1. It eliminates the need for the end user to type in the fully qualified name of a reference value (with periods separating qualifiers). For example, for a property that references a class attribute named ***email*** of the class ***Customer*** of the package ***“Order System”***, instead of typing in ***“Order System”.Customer.email***, the end user simply types in ***email***.
2. It can be used to ensure that all key components of a reference value are the same. For example, the LISTOF “Class Attribute” property in a Class definition contains a list of attributes that all belong to the same class and to the same package.

A KEYED BY clause typically contains a specification of how each of the key components of the referenced object(s) may be found. The KEYED BY clause contains a portion for each key component separated by a comma.

Example:

For example, the KEYED BY clause of the Class’s “Attributes” property could be as follows:

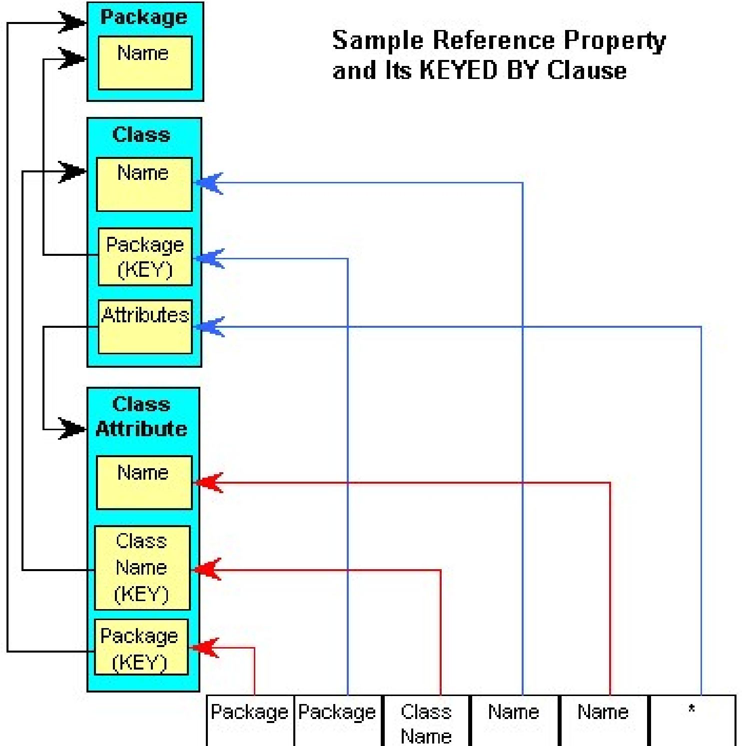
```
DEFINITION "Class"
{
…
```

```
PROPERTY "Attributes" { … LISTOF "Class Attribute"
KEYED BY {Package:Package, "Class Name":Name,
Name:* } … }
```

In the example above, the three *key components* (separated by commas) are Package:Package, “Class Name”:Name, and Name:*. These components refer to the three parts needed to identify the referenced Class Attribute definitions – the Package name, the Class name, and the Class Attribute name. Taking them in reverse order, it states that:

- The name of the Class Attribute will be found in ***this*** property (* means “here”), hence:
  Name:*
- The value of the key property “Class Name” in the Class Attribute definition will be found in this object’s name, hence:
  “Class Name”:Name
- The value of the key property Package in the Class Attribute definition will be found in this object’s Package property, hence:
  Package:Package

The following schematic diagram shows how the KEYED BY clause is used in the example above, and may be useful in understanding the KEYED BY clause generally.



The schematic shows what we have said above – in the definition of a class, a class attribute is entered by specifying its package (stored in the class attribute's Package property and obtained from the Package value of the class you are in), its class name (stored in the class attribute's "Class Name" property and obtained from the class's actual name), and name (stored in the class attribute's "Name" property and obtained from itself).

In summary:

1. For each key component of the **reference object**, the KEYED BY clause has a component.
2. The components of the KEYED BY clause are separated by commas.

3. Each component has two parts:
   - The first part identifies the key component of the reference object,
   - The second part states where the value of that component is to be found, and
   - The two parts are separated by a colon.

However, certain default values may be assumed to simplify the KEYED BY clause. If the two parts of the component are the same, the second may be omitted and if the second part of the last component is omitted, it assumed to be “here” – i.e. the asterisk. Thus, in practice the KEYED BY clause of the Class’s “Attributes” property is coded:

```
KEYED BY {Package, "Class Name":Name, Name }
```

Naturally, all the properties used in the KEYED BY statement must exist. Thus, Rational System Architect checks that there is a “Package” property and a “Class Name” property in the “Class Attribute” definition and that they are both KEY.

Besides saving the end user all the effort of typing in common key components in a LISTOF property like this one (for example, “Order System”.Customer.email), employing a KEYED BY clause using other properties to provide common values **ensures the same values are used for each reference**. Thus, in the example we have been using, all the Class Attributes referred to in the “Attributes” property of the Class are forced to belong to the same class in the same package – a desirable characteristic in this case.

At other times it is convenient to have the key components of the referenced object separated for reasons of clarity and simplicity. Under such circumstances a KEYED BY clause is used to designate the properties supplying the separate components. Indeed, for these reasons, when a property is KEY and refers to an object with KEY properties, Rational System Architect **requires** that the components be in separate properties.

Examples of Key and Keyed By

One Definition Keyed By Another

We wish to categorize automobiles by their “Brand” and “Model”. We create a new definition called “Car Brand” (a “Car Brand” might be Ford, Volkswagen, Toyota, etc), and another called “Car Model” (which would include values such as Mustang, Passat, and Corolla).

A “Car Model” must have its “Car Brand” (otherwise called its ‘Make’) specified. The “Car Brand” is a property that you can use to uniquely identify the “Car Model”; each “Car Model” has one and only one “Car Brand”.

We make “Make” a key property of “Car Model” – but in actuality it is “Car Brand” that is the definition type that is filled in for this property. We also make it REQUIRED, which means that to create the definition, you must fill in this property in the opening dialog to create the definition.

```
RENAME DEFINITION "User 1" To "Car Brand"
RENAME DEFINITION "User 2" To "Car Model"

DEFINITION "Car Brand"
{
PROPERTY "Country of Origin"
{ EDIT  Text  LENGTH 20  }
}

DEFINITION "Car Model"
{
Property "Make"
{KEY EDIT ONEOF "Car Brand" REQUIRED}
}
```

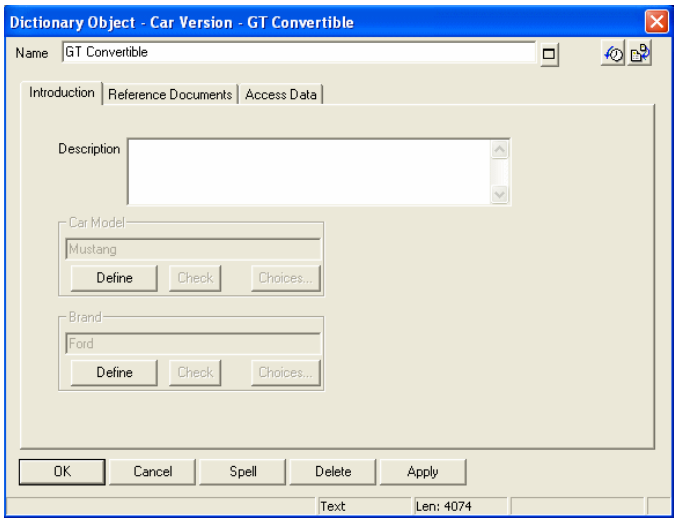
Note: We have introduced a problem in the USRPROPS.TXT above for the purposes of discussion. We will discover it later in this section.

A Third Definition With Two Key Properties

Let’s take it one step further. Each “Car Model” has a version – for instance you can buy a Mustang Coupe, a Convertible, a GT Coupe, a GT Convertible, a Mach 1, or SVT Cobra. This could be an ever changing list so we make it a definition type (versus a static LIST). We call the definition type “Car Version”. When the user creates a “Car Version”, he or she will need to specify the “Car Brand” and “Car Model” of the

version (because there may be many “Car Model”s out there with a GT).

```
RENAME DEFINITION “User 3” TO “Car Version”
Definition "Car Version"
{
Property "Car Model"
{KEY Edit ONEOF "Car Model" RELATE BY "is keyed by"}
Property "Brand"
{KEY EDIT ONEOF "Car Brand" RELATE BY "is keyed by"}
}
```



Creating a ListOf a Keyed Definition

For every “Car Model”, we want to create a list of Car Versions that it provides. We create a LIST OF property that enables the user to enter Car Versions in the “Car Model” definition. Note that Car Versions is a definition type with a compound key – when the user types in the “Car Version”, they must specify the “Car Version”, and the “Car Brand” and the “Car Model” of the “Car Version”.

```
DEFINITION "Car Model"
{
Property "Make"
{KEY EDIT ONEOF "Car Brand" RELATE BY "is keyed by"
REQUIRED}
Property "Versions" {EDIT LISTOF "Car Version"}
}
```

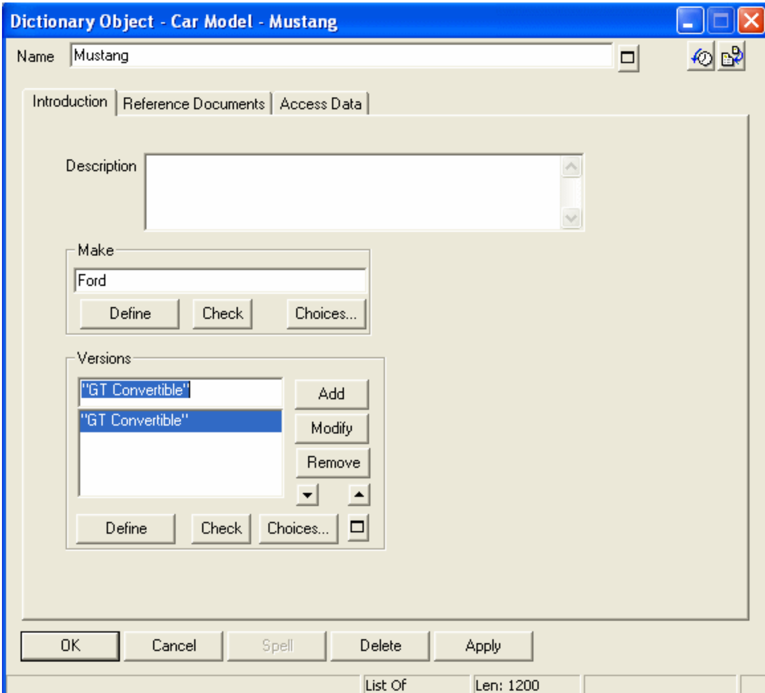
There is a **problem** with the USRPROPS.TXT above. “Car Version” is a compound-key definition – it has its own name as a key, and it has two other key properties, “Car Brand” and “Car Model”. If we specify the USPROPS.TXT above, it will be up to the user to know this. He or she will need to type in the “Car Version”, fully qualified by its Brand and Model, with a period separating each, such as Ford.Mustang.”GT Convertible”.

Adding the KEYED BY Statement

We add the KEYED BY statement to the Property “Version” statement to automatically spell this out.

```
DEFINITION "Car Model"
{
Property "Make"
{KEY EDIT ONEOF "Car Brand" RELATE BY "is keyed by"
REQUIRED}
Property "Versions" {EDIT LISTOF "Car Version" KEYED BY
{"Brand":"Make", "Car Model":Name, Name} }
}
```

In the first part of the KEYED BY statement above, **KEYED BY {“Brand”:”Make”**, we state that the “Car Version” value we are entering has a key property called Brand that must be filled in. This property will be filled with a value that is obtained from the “Make” property of the current definition that we are in – “Car Model”. Note that the actual value is a definition of type “Car Brand”; the KEYED BY statement lists names of properties, not definition types.



If the referenced property ("Brand") and referencing key property ("Make") above were the same, we would not need to specify both (ie, if they were both "Brand", we would only need to type in KEYED BY **{"Brand",** ….

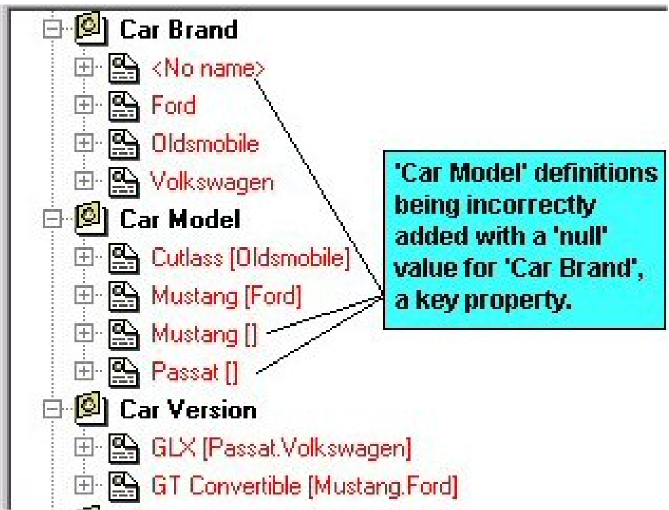
In the second part of the KEYED BY statement above, **"Car Model":Name**, we state that the "Car Version" value we are entering has a second key property called "Car Model" that must be filled in. This property will be filled in with a value that is obtained by the Name of the current definition that we are in – the name of this "Car Model" definition we have open.

In the third and final part of the KEYED BY statement above, Name, we specify that the last key of the "Car Version" value we are entering is keyed by its own name, as any definition is.

So if we have a "Car Model" definition open called Mustang, which has a key property "Make" filled in with Ford, we simply need to type in GT in the LIST OF "Car Version" property,

and the new definition Ford.Mustang.GT will be added to the encyclopedia.

There is still a problem. We notice as we add new “Car Version” definitions into the encyclopedia, “Car Model” definitions are being input that have a null property for “Car Brand”.



This only happens when we add a new “Car Version” definition directly (via the New Definition command in the explorer), not when we add one in the ListOf dialog in the “Car Model” definition.

The reason is that the “Car Version” definition specifies that one of its key properties is “Car Model”, but doesn’t specify that that property has its own key property (“Car Brand”) that needs to be filled in with a value. Every time we add a new “Car Version” definition, we are asked to specify a “Car Model” property. We specify that “Car Model” property, but don’t specify where it gets its key “Car Brand” property value from. So nothing gets filled in for it.

To fix this, we must specify in the “Car Version” definition that its “Car Model” key property is itself keyed by properties, which must be filled in with values. We add the clause **KEYED BY {"Make":"Brand", Name**}, which means that the “Car Model”

definition has a property called “Make” that will be filled in with the value in the current definition’s “Brand” property.

```
Definition "Car Version"
{
Property "Car Model"
{Key Edit oneOf "Car Model" KEYED BY
{"Make":"Brand", Name} RELATE BY "is keyed by"}
Property "Brand"
{KEY EDIT OneOf "Car Brand" RELATE BY "is keyed by"}
}
```

Once we make that change, when we add a new “Car Version” to the encyclopedia, we do not get any inadvertent null “Car Brand” definitions.

Example of “Complete” Keyword

It is arguable whether or not you want users to type in new definitions of Car Versions independently of the “Car Model”. For example, will someone enter in Si as a “Car Version”, and then specify that they are referring to a Honda Accord? Probably not. It may help your users if you force them to enter the “Car Brand” and “Car Model”, and then specify Car Versions in the ListOf property of Car Versions in the “Car Model” definition. What you are saying is that these “Car Version” belong completely to the “Car Model” – to say you have an Si doesn’t mean much by itself. We use the COMPLETE clause to do this.

```
DEFINITION "Car Model"
{
Property "Make"
{KEY EDIT ONEOF "Car Brand" RELATE BY "is keyed by"
REQUIRED}
REM "also contains a list of the versions"
Property "Versions"
{EDIT COMPLETE LISTOF "Car Version" KEYED BY
{"Brand":"Make", "Car Model":Name, Name} }
}
```

Once you set the COMPLETE for a list, you will not see the “Car Version” definition amongst the definition types available when you create a new definition, nor will you be able to open any “Car Version” definitions from the explorer. If you try to, Rational System Architect will give you a message that says

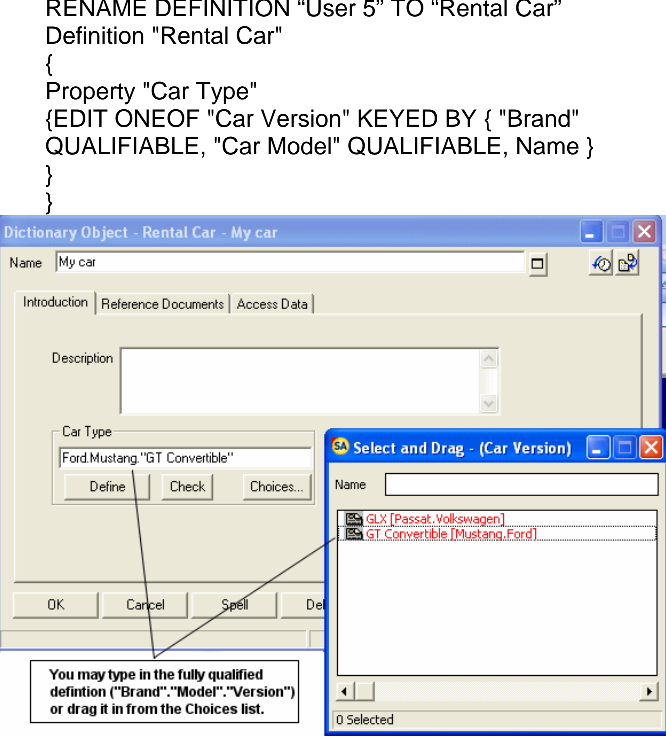
you can only open the definition from its containing definition, which in this case is “Car Model”.

Qualifiable Example

Now we want to add a new definition to the encyclopedia to track Car Rentals. The new definition type Car Rental includes a property to track the Car Type of each rental.

The problem is we do not want to keep track of the “Car Brand” and the “Car Model” as well as the “Car Version” for a Rental Car. We want one property, Car Type, that we can enter a car type into and have it worry about its make and model. So if we specify a ‘Car Type’ for a Rental Car, we do not have a property within the Rental Car definition within which to keep the Car’s Brand or Model. We use the QUALIFIABLE keyword.

```
RENAME DEFINITION “User 5” TO “Rental Car”
Definition "Rental Car"
{
Property "Car Type"
{EDIT ONEOF "Car Version" KEYED BY { "Brand"
QUALIFIABLE, "Car Model" QUALIFIABLE, Name }
}
}
```



The QUALIFIABLE phrase causes the ONEOF “Car Type” property to store the “Brand” and “Model” information. The information is

stored in the value itself, separated by periods. You can either drag in values from the Select and Drag dialog that opens if you press on the Choices button, or type in the values with appropriate periods.

Example of Using the Where Clause

We want to specify that a “Car Version” fits into a category of automobiles – either it will be an SUV, or a sub-compact, or a compact, or a midsize sedan, or a fullsize sedan, or a luxury car, or a truck. Since this list is fairly stable, we don’t need to create a new definition type for it. We create a List of “Vehicle Types”.

```
LIST "Vehicle Types"
{
Value "SUV"
Value "Sub-Compact"
Value "Compact"
Value "Midsize Sedan"
Value "Fullsize Sedan"
Value "Luxury Sedan"
Value “Convertible"
Value "Truck"
}

Definition "Car Version"
{
Property "Car Model"
{Key Edit oneOf "Car Model" KEYED BY {"Make":"Brand", Name}  RELATE BY "is keyed by"}
Property "Brand"
{KEY EDIT OneOf "Car Brand" RELATE BY "is keyed by"}
Property "Vehicle Type"
{EDIT Text List "Vehicle Types" DEFAULT "Midsize Sedan"}
}
```

Where Clause Continued

We create a new definition of type "SUV Ad Campaign". In a property of this definition, we want users to be able to select instances of automobiles of a certain type. In other words, we want this property to be filtered to contain only the instances of definitions in the encyclopedia that satisfy the stated condition of 'vehicle type' = 'SUV'. We use the "Where" clause to provide this filtering.

```
Definition "SUV Ad Campaign"
{
Property "SUV Type"
{ Edit OneOf "Vehicle Types" WHERE "Vehicle
Types" = "SUV" }
}
```

Hiding Standard Entries in the SAPROPS.CFG File

You may hide or make invisible properties in USRPROPS.TXT. The question often arises as to what happens when you hide a property for which information has already been entered into the encyclopedia. For example, assume that you have been adding data elements to the encyclopedia, and have supplied the responsible business areas for each element in the *Business Unit* property. But half way through the project, it has been decided that *Business Unit* is no longer needed. All you have to do is change the property *Business Unit* to an *invisible*, or *hidden,* property, through a modification in USRPROPS.TXT; it no longer appears in the data element definition dialog.

```
DEFINITION "Data Element"
{
PROPERTY "Business Unit"
{ INVISIBLE }
}
```

When you were entering *Business Unit* values, they were added to the encyclopedia, and were saved in the file named ENTITY.DBT. The modification to the encyclopedia's metamodel (above) makes the Business Unit property disappear from the data element definition dialog, but it does not delete previous entries of Business Unit information – they still exist in the .DBT file. If you remove the code above from USRPROPS.TXT, the *Business Unit* property will appear again in the data element definition dialog, and the values previously entered will reappear in their respective data element.

The question arises – how can you remove excess unneeded property information from the ENTITY.DBT file? It can be done by careful use of the **Export Definitions** and **Import Definitions** commands under the **Dictionary** menu.

1. Select Dictionary, Export Definitions. Select to export data element definitions in CSV format to a text file.

2. Open that csv text file in an external editor such as Excel, and delete the column with the unwanted information (in the case above it would be titled Business Unit).
3. Select Dictionary, Import Definitions. Re-import the CSV file using the *delete all fields then add new data* option.

Performing this task is optional. It is only necessary if you wish to regain the memory space used by the excess values.

Error Messages

Error Messages

Whenever Rational System Architect opens an encyclopedia and parses its SAPROPS.CFG File and USRPROPS.TXT, it performs a syntax check on the statements in the file. Any syntax errors are displayed in an **Error** dialog. This **Error** dialog can contain the following error messages. The brackets indicate points at which Rational System Architect inserts variable information, such as a property name or line number.

< > found on Line < > of USRPROPS.TXT

< > has been defined more than once

< > is already defined as a List

Cannot load DLL (STATBAR.DLL).

Cannot load DLL (STATBOX.DLL).

Chapter < > is already defined.

Dictionary class < > is already defined.

Description

Dictionary

Illegal argument < >

Illegal argument < > - must be quoted

Illegal default < > for Boolean edit

Illegal default < > for date edit

Illegal default < > for numeric edit

Illegal default < > for time edit

Illegal default for < > ranged numeric edit

Insufficient resources to load dialog \n%s.

Invalid Dictionary class Name: < >.

Invalid Major Type Name: < >.

Invalid Relation Name: < >.

List < > is already defined.

List-name < > not defined

Name < > already in use

Number of property edits (OneOf, ListOf, ExpressionOf) exceeds limit with < > on

Number of properties exceeds limit with < > on < >

Number of DISPLAYed properties exceeds limit with < > on < >

Number of lists exceeds limit with < > on < >

Number of lists exceeds limit with < > on < > (max=100)
Numeric argument < > out of range
Numeric argument expected but < > was
Out of range or invalid < > length argument
Premature end of file after < >
Previously defined list-name
Property < > is already defined
Referenced List < > is not defined.
Syntax Error in < > Line <line #>.
The < > edit type is only valid for the 'Description' property
Too many Lists.[2]
Too many Properties < >.[3]
Too many Values in List < >.[4]
Unable to open property file
Unbalanced begin-end or { }
Unexpected command < >
Unknown property DISPLAY type < >
Unknown dictionary name < >
Unknown edit-type < >
Unknown initial-type < >
Unknown update-type < >
Warning - RANGE found but no maximum range defined.
Warning - RANGE found but no minimum range defined.

---

[2] Maximum number of lists is 400. This includes SAPROPS and USRPROPS, where the number of lists actually used from the SAPROPS is dependent on the Encyclopedia Configuration.

[3] Maximum number of properties for one Diagram, Symbol or Definition is 128

[4] Maximum number of VALUES in a LIST is 128.

In addition to any error message, Rational System Architect places further information in the error dialog about the syntax error found, as follows.

while checking a **DEFINITION** command.
while checking a **DISPLAY** command.
while checking a **LIST** command.
while checking a **VALUE** command in a **LIST**.
while checking a **PROPERTY** command in a **DEFINITION** .
while checking for a **DEFINITION** or **LIST** command.
Would you like to continue?

For example, the entire error message may look like this:

Unknown property DISPLAY type < > while checking a **DEFINITION** command.

Runtime Edits

"Runtime" is that time when you are drawing diagrams, and, in particular, when you are making encyclopedia entries. The dialogs displayed when you add or modify the dictionary are under control of SAPROPS.CFG and USRPROPS.TXT; the **EDIT** commands act to prevent the user from making erroneous entries. For example, assume SAPROPS.CFG has the following entry:

```
PROPERTY "My Property"
{ EDIT numeric LENGTH 2 MINIMUM 1 MAXIMUM 32 }
```

In this example, an **Invalid Value** error message is displayed if you type in "AB" or "0" in the "My Property" text field, and click OK to close the dialog. This happens because the property has been specified as a numeric (can't be made up of any letters), of minimum value 1.

Rational System Architect performs the following runtime edits:

BOOLEAN	must be T, F, TRUE or FALSE
DATE	numeric, of format MM/DD/YY
Expression, ExpressionOf, ListOf, OneOf	see the entries in the section beginning on page 2-69.
NUMERIC	must be a numeric-string
TEXT	no editing
TIME	numeric, of format HH:MM:SS

3

USRPROPS.TXT Keywords

Introduction

This chapter contains an alphabetical list of all the keywords you can use to make modifications to USRPROPS.TXT.

Certain restrictions apply in the use of the following keywords: CHAPTER, GROUP, LABEL, LIST, and LISTONLY. Please refer to each of those keywords for an explanation of the specific restriction that applies to the use of the keyword.

USRPROPS Keywords

$$FORCE$$ See DISPLAY keyword.

$$NONE$$ See DISPLAY keyword.

$$VFORCE$$ See DISPLAY keyword.

$$VNONE$$ See DISPLAY keyword.

#IFDEF Enables you to switch on commands in USRPROPS.TXT based on whether the clause in quotes after the IFDEF command has been turned on in the Property Configuration dialog. The Property Configuration dialog (Tools, Customize Method Support, Encyclopedia Configuration) modifies the sadeclar.cfg file in an encyclopedia. It is the sadeclar.cfg file that is actually checked when the IFDEF statement is evaluated as SAPROPS.CFG is parsed.

This command must have a matching, ending #endif statement.

Example:

```
#ifdef "Business Enterprise"
DEFINITION "ORGUNIT"
{
LAYOUT { COLS 2 ALIGN OVER }
PROPERTY "RowDefinition"
{ KEY EDIT OneOf "Organizational Unit" RELATE BY "is part of"
ReadOnly LABEL "Organizational Unit"}

PROPERTY "ColumnDefinition"
{ KEY EDIT OneOf "Organizational Unit" RELATE BY "is part of"
ReadOnly LABEL "Organizational Unit"}

PROPERTY "Description"
{ EDIT Text LENGTH 255 HELP "Appears in the cell of a matrix" }

PROPERTY "Intersection?"
{ EDIT Boolean LENGTH 1 }
}
#endif
```

#IFDEF (continued)

In the example above, the definition type “ORGUNIT” only contains the properties specified if Business Enterprise is toggled on in the **Property Configuration** dialog (Tools, Customize Method Support, Encyclopedia Configuration). If you fill in values for these properties and then turn off Business Enterprise, the values remain in the definition of “ORGUNIT” in the repository, but they are not shown in the definition dialog since the property set is turned off.

See also the #IFNDEF keyword.

#IFNDEF

Opposite of the IFDEF command, the IFNDEF command enables you to switch on commands in USRPROPS.TXT if the property listed in quotes after it has **not** been turned on in the Property Configuration dialog. This command must have a matching, ending #endif statement.

Example:

#ifndef "Business Enterprise"

RENAME Symbol "Swim Lane" to "Org. Unit"

#endif

In the example above, if the “Enterprise Architecture” choice is not toggled on in the System Architect Property Configuration dialog (Tools, Customize Method Support, Encyclopedia Configuration), then all ‘Swim Lane’ symbols are renamed to “Org. Unit”. You will notice this change as you select such a symbol on any diagram that it is used on. The “Enterprise Architecture” choice in the configuration dialog used to be named “Business Enterprise”, but was changed on the dialog in V9.0. However, the underlying switch statement that it invokes in SADECLAR.CFG is still called “Business Enterprise”.

#INCLUDE

#include can be used in the USRPROPS.TXT file to break out changes into separate, additional files. Inside each of those files, there can be other includes, which in turn can have other includes, etc. The level of nesting allowed by the parser is 10. Beyond that Rational System Architect will give a warning and ignore the subsequent levels.

Example:

For example, you could create three USRPROPS.TXT files, one for diagrams (arbitrarily named diagrams.txt), one for definitions (arbitrarily named definitions.txt), and one for symbols (arbitrarily named symbols.txt). The USRPROPS.TXT file would look like this:

```
************************************************************************
REM "USRPROPS.TXT"
REM "Copyright IBM.  All rights reserved."
REM "Instructions for modifying this file are in the on-line help."

#include “diagrams.txt”
#include “symbols.txt”
#include “definitions.txt”
************************************************************************
```

Inside each of those files you could place #includes to other files, such as a file for lists (arbitrarily named lists.txt).

This command helps enable coherent reusability of user-defined data.

ADDRESSABLE

In Rational System Architect, symbols may be *addressed by* one or more (addressable) definitions. Thirteen addressable definitions are automatically supplied and they may be addressed to any symbol: *Business Objectives, Business Process, Change Requests, Critical Success Factor, Current Data Collection, Data Class, Deliverable, Functional Organization, Geographic Location, Information Requirement, Organization Goals*, *Requirements, and Test Plans.* These addressable definitions are available by selecting a symbol on a diagram, and choosing Dictionary, Addresses. In addition, any definition may be declared addressable through the syntax in USRPROPS.TXT. Doing so makes the definition available to address a symbol, and places it on the Dictionary, Addresses drop-down list.

Example:

```
DEFINITION "xxxxxx"
 {
  ADDRESSABLE
 }
```

See also keyword NONADDR.

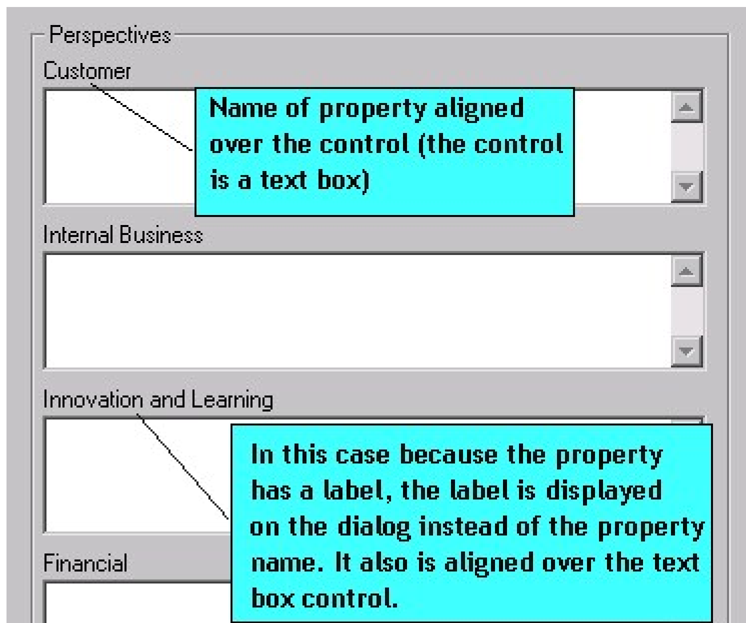
ALIGN

Used to specify the positioning of the name (or label) of a property's control (list box, text box, etc) in a dialog. Valid options are *BODY*, *LABEL*, and *OVER*.

Example:

DEFINITION "Balanced Scorecard"
{..
GROUP "Perspectives"
{
LAYOUT { COLS 2 TAB **ALIGN** OVER }
PROPERTY "**Customer**" { EDIT Text LENGTH 300 }
PROPERTY "Internal Business" { EDIT Text LENGTH 500 }
PROPERTY "Learning" { EDIT Text LENGTH 500 **LABEL "Innovation and Learning**"}
PROPERTY "Financial" { EDIT Text LENGTH 500 }
}

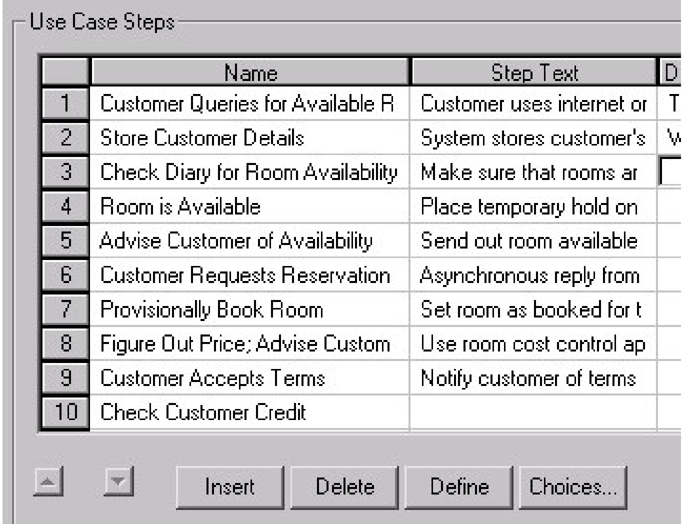
In the example above, the ALIGN OVER command places the names or labels of all properties below the LAYOUT statement over their respective control in a Balanced Scoreboard definition dialog.



See also keywords BODY, LABEL, OVER, JUSTIFY, and TAB

ASGRID

The ASGRID command specifies that a ListOf property is presented in a table, or grid. ASGRID must have an Edit Type that is either ListOf or ParmListOf. Otherwise, Rational System Architect emits a warning when you reopen the encyclopedia and ignores the ASGRID keyword.



Example:

Definition "Use Case"
{CHAPTER "Steps"
PROPERTY "Use Case Steps" { EDIT COMPLETE ListOf "Use Case Step" KEYED BY { "Package", "Use Case Name":Name, Name}
ASGRID LENGTH 1200 } }

Using ASGRID With Keyed Definitions

Key properties of a definition are not shown in a grid formed by an ASGRID command. In the example above, each Use Case Step's package name or Use Case name is not shown in the grid.

Limitation of ASGRID

You cannot use ASGRID in a LISTOF that refers to a definition that is in a COMPLETE ListOf in another definition. So for example, you can add a ListOf "Attribute" to a defintion but it cannot be shown ASGRID. The maximum length of a property that can be seen in the GRID is 400.

ASGRID (continued)

ASGRID COUNT_FIXED

The COUNT_FIXED keyword is used with the ASGRID keyword to specify that the user cannot delete or insert rows to a grid.

See also KEY and KEYED BY and COUNT_FIXED keywords.

ASGUID

This keyword can only be used with text properties. It will automatically populate a property with the value of "GUID" property. This text property can then be used as a key property instead of the actual GUID property. The ASGUID property should be read only. When you re-open the definition the ASGUID property will be filled in.

Example:

```
RENAME DEFINITION "User 1" to "MyDef"
DEFINITION "MyDef"
{
PROPERTY "MyProp"
{KEY EDIT Text LENGTH 100 ASUID READONLY}
Property "HIYA"
{EDIT Text Length 145}
}
```

ASPARMGRID

The ASPARMGRID keyword was specifically created for use with Rational System Architect's data modeling, and works off of specially created code. This keyword is found in SAPROPS.CFG and **should not** be used by users in USRPROPS.TXT.

ASSIGN

You may assign new symbol types or existing symbol types (symbols that already exist in another diagram) to new or existing diagram types. Symbol types may be added to diagram types using the following syntax:

SYMBOL <symbol-type-name> [IN <diagram-type-name1>]
ASSIGN [TO] <diagram-type-name2>

Example:

SYMBOL “Organizatiional Unit” IN “Organization Chart”
{
ASSIGN TO “Enterprise Direction”
}

AUDITID

This keyword represents the characters entered in the **Audit Id** dialog when the user first signs on to Rational System Architect. AUDITID is an allowable keyword type which indicates that a property contains the user's Audit ID. CHECKOUT AUDITID, FREEZE AUDITID, INITIAL AUDITID, and UPDATE AUDITID each have special meanings. Refer to each of these keywords, listed separately in this table, for more information.

Note: Starting in Version 9.1 of Rational System Architect, the INITIAL AUDITID (provided in a field called “Initial Audit”) and UPDATE AUDITID (provided in a field called “Last Change Audit”) are automatically included in the Access Data tab of every definition.

Example:

```
DIAGRAM "Data Flow Gane & Sarson"
  {
    PROPERTY "Frozen by"
  { FREEZE AUDITID }
```

Other uses for the Audit Id might be found in any definition.

Example:

```
DEFINITION "X"
  {
    PROPERTY "Current Owner Name"
  { EDIT Text CHECKOUT AUDITID LENGTH 12 READONLY }
  }
```

AUTOCREATE

The AUTOCREATE command automatically creates a definition behind the value that has been specified in a property, as soon as you click the OK button to close the containing dialog. If you do not use the AUTOCREATE keyword, then the value entered into a property will remain undefined after you click the OK button to close the containing dialog.

Example:

```
DEFINITION "Physical Database"
{..
PROPERTY "Model"
{ KEY EDIT ONEOF "Project Data Model" AUTOCREATE RELATE BY
"is keyed by" READONLY }
..}
```

In the example above, the definition dialog of a Physical Database definition contains a property called “Project Data Model”. If you put a value in this field (for example, “Reservations”), and then click OK to close the Physical Database definition dialog, the “Reservations” Project Data Model definition is automatically created in the encyclopedia.

See also INITIAL USER REQUIRED and OVERRIDABLE.

BEGIN

This keyword indicates the beginning of the definition of a property, or of the series of properties which make up the definition of a diagram, symbol, or definition. You can also use the following syntax: **{**.

See also keyword PROPERTY.

BODY

This is one of the arguments used in the **ALIGN** command. It is used to align all controls one space to the right of the widest label in that column. (Contrast this with the ALIGN OVER keyword pair, which places the name over the property.)

Example:

```
Definition "My Definiition"
{
CHAPTER “My Chapter”
LAYOUT { COLS 1 ALIGN BODY }
PROPERTY "My Property 18"{ EDIT Text Length 10}
PROPERTY "My Property 19"{ EDIT Text Length 10}
PROPERTY "My Property 20"{ EDIT Text Length 10}
PROPERTY "My Property 21"{ EDIT Text Length 10}
PROPERTY "My Property 22"{ EDIT Text Length 10}
PROPERTY "My Really Long Property 23"{ EDIT Text Length 10}
}
```

In the example above, the control for “My Really Long Property 23” is a text box placed one space to the right of the label. All other text-box controls for other properties on the dialog are lined up with this control.

Note – ALIGN BODY used to put all controls one space to the right of the label, but it was subsequently changed to be the same as ALIGN LABEL.

See also keywords OVER, ALIGN, TAB, LABEL, and JUSTIFY.

BOOLEAN

Appears in a **Definition** dialog as a check box. It has one of two values: True (T) or False (F).

Example:
In the following example, the user is allowed to turn on or off the Hierarchical Numbering features on an IDEF0 diagram by selecting true or false.

```
DIAGRAM "IDEF0"
  {
    PROPERTY "Hierarchical Numbering"
    { EDIT Boolean LENGTH 1 DEFAULT "F" }
..
}
```

BROWSER (Explorer)

Specifes whether a property and its value shows up in the Properties box of Rational System Architect's Explorer (browser) when the respective diagram, symbol, or definition is selected in the Explorer.

The following explorer control statements are permitted (the word 'object' is used to mean a Diagram, Symbol, or Definition):

Within the Specification of a Property:

- **BROWSER {SHOW}:** Requires the explorer to display the value of that property when displaying the object containing that property.

Within the Specification of an Object but Not Within the Description of a Property:

- **BROWSER {OMITKEY}:** Requires the explorer to not display key properties of the object under conditions in which it otherwise would.
- **BROWSER {OMITTYPE}:** Requires the explorer to not display the type of the object under conditions in which it otherwise would.

Not Within the Specification of an Object:

- **BROWSER {OMITKEY}:** Requires the explorer to not display key properties of any object under conditions in which it otherwise would.
- **BROWSER {OMITTYPE}:** Requires the explorer to not display the type of any object under conditions in which it otherwise would.

The term "under conditions in which it otherwise would" is used above because the explorer often does not display some (or all) key properties – when the object is being displayed subordinate to one of its key objects - and often does not display the type – when it is being displayed subordinate to a type header.

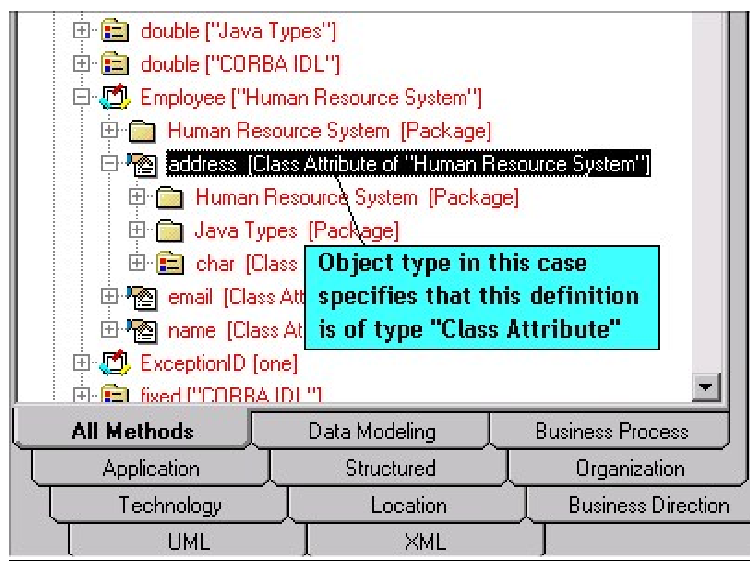
BROWSER (Explorer continued)

Example 1:
DEFINITION "Association End"
{
PROPERTY "Package" { KEY EDIT OneOf "Package" RELATE BY "is keyed by" READONLY **BROWSER { SHOW }}**
..}

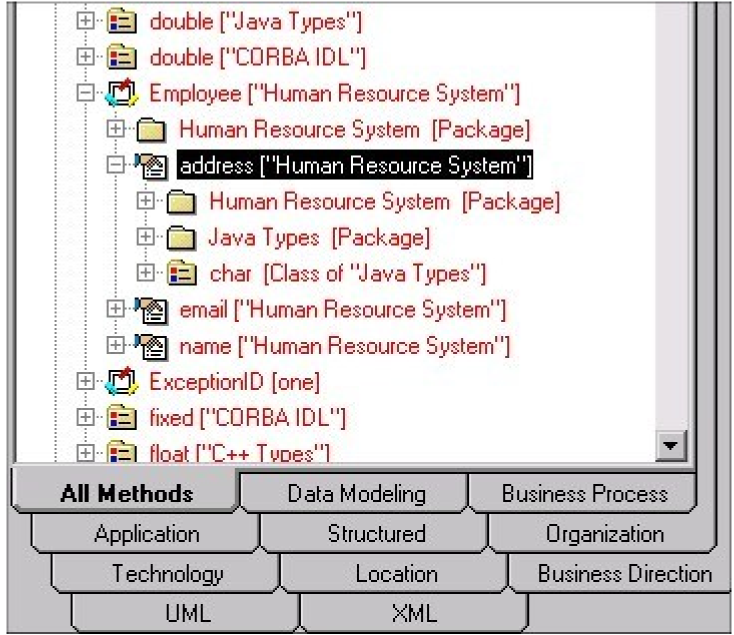
In the example above, the value of the package property is shown in the explorer even though it is normally not shown.

Example 2:
DEFINITION "Class Attribute"
{
BROWSER { OMITTYPE }
..}

In the example above, a class attribute is a definition of type “Class Attribute”. By default, this would be shown in the explorer, which would be a bit redundant and might be considered visually annoying. Without the BROWSER (OMITTYPE) command being used, the explorer would display attributes shown in the diagram below.



Using the BROWSER {OMITTYPE} command makes the explorer display an attribute as shown in the diagram below.



Example 3:

```
DEFINITION "Association"
{
BROWSER { OMITKEY }
LAYOUT { COLS 1 ALIGN OVER TAB }
CHAPTER "Roles"
PROPERTY "Association GUID" { KEY EDIT Text LENGTH 64
INVISIBLE READONLY}
PROPERTY "Class Roles" { EDIT COMPLETE ListOf "Association
End" KEYED BY { "Association GUID":"Association GUID",
"Association":"Name", "Package" QUALIFIABLE, "Class"
QUALIFIABLE, "Role GUID" QUALIFIABLE, "Name" }
RELATE BY "uses" LENGTH 4096 ASGRID COUNT_FIXED
BROWSER { SHOW } }
..
}
```

In the example above, the value of the property “Class Roles” is displayed in the explorer (since the classes that an association attaches to is important information to know), even though it normally is not shown.

BY

An often used keyword used as shown in the following expressions: *DEFINED BY, RELATED BY, RELATE BY,* and *KEYED BY.* For more information, refer to the specific keyword combination.

Example:

```
DEFINITION "Column"
{..
PROPERTY "Database Name"
{ KEY EDIT OneOf "Database" RELATE BY "nothing" }
  ..
}
```

CHAPTER

Creates ***tabs*** in a dialog. Each Chapter statement corresponds to a tab. The syntax is as follows:

CHAPTER <chapter_name>

The Chapter statement does not call for opening or closing brackets to group the items in the tab. All items that fall under a Chapter statement are grouped in that tab. The next grouping is created by the next Chapter statement.

Example:

CHAPTER "Screen Painter properties"

Modifying the Name of a Tab (Chapter):

To change the name of a CHAPTER via USRPROPS.TXT, you use the LABEL command.

Example:

The SAPROPS file provides a Nested Classes tab for a Class definition:

```
DEFINITION "Class"
 { CHAPTER "Nested Classes"
  …
 }
```

You may relabel the CHAPTER “Nested Classes” to “Fred” using the LABEL command in USRPROPS.TXT:

```
DEFINITION "Class"
 { CHAPTER "Nested Classes" LABEL "Fred"
    }
```

CHECKOUT

Displays information concerning the checking out of an object, such as the AUDIT ID of who checked it out, or the DATE or TIME that it was checked out. The displayed fields are always READONLY. Values are automatically kept track of by Rational System Architect, but in order to view the values in a dialog, you must add properties with the following respective characteristics:

CHECKOUT Auditid
CHECKOUT Date
CHECKOUT Time

Example:

```
DIAGRAM "Data Flow Gane & Sarson"
 {
  PROPERTY "Checked out by"
 { CHECKOUT AUDITID }
  PROPERTY "CheckOut Date"
 { CHECKOUT DATE }
  PROPERTY "CheckOut Time"
 { CHECKOUT TIME }
 }
```

Search on *Access Control* in the on-line help for more information on checking objects in and out.

See also keyword FREEZE.

COLS, COLUMNS

Determines the number of columns into which a group of properties are placed in a Diagram, Symbol, or Definition dialog.

Example:

```
DEFINITION "Referent"
{
LAYOUT { COLS 2 ALIGN OVER TAB }
...}
```

COLUMN_SCRIPT

COLUMN_SCRIPT calls a script written in SA Basic. The column scripts are used for the behavior of columns in tables in a physical data model. The action taken by the script works against each column in the list.

By convention, the function itself is named with one of the following prefixes:

- fmtxxx – The function itself exists in hard code and cannot be modified. Most functions in SAPROPS.CFG are this way. Hard-coding the function is done to make Rational System Architect's overall response faster.
- _fmtxxx – Exists in the fmtscript.bas file within Rational System Architect's main executable directory.

Creating Your Own Script

For information on how to create your own script, see the SCRIPT keyword.

Example:

```
DEFINITION "Table"
{
PROPERTY "Description"
{
ZOOMABLE EDIT ListOf Definition "Column"  FROM "Data Element"
KEYED BY {"Database Name","Owner Name","Table
Name":"Name","Name"} LENGTH 2000
DISPLAY { FORMAT Key LEGEND "Key Data" }
DISPLAY { FORMAT NonKey LEGEND "Non-Key Data" }
DISPLAY { FORMAT COLUMN_SCRIPT FmtERAttr LEGEND
"Physical Display" }
}  ..}
```

FmtERAttr returns values for attributes in Entities of Entity Relation diagrams or Columns of Tables in Physical diagrams.

FmtERAttr returns ID, NAME, ADDRESS, STREET, CITY, STATE, FIRST_5_DIGITS, ZIP CODE, and LAST_4_DIGITS.

COLUMN_SCRIPT (continued)

APPLICANT	
Physical Display	
ID	CHARACTER(9) [PK1] [FK]
Reference_Name	CHARACTER(48)
Reference_House	CHARACTER(10)
Reference_Street	CHAR(48)
Reference_City	CHAR(31)
Reference_State	CHAR(2)
Reference_ZIP	CHAR(9)
Reference_Description	CHAR(999)

See also SCRIPT, COMPONENT_SCRIPT, VALUESCRIPT, and FORMAT keywords.

COMPLETE

Causes the referenced definition to belong to the referencing definition, so the referenced definition cannot be referenced by another definition, and can only be edited from within the containing referencing definition.

An example is an attribute in an entity, which completely belongs to an entity (and does not belong to another definition), and can only be opened from within the entity definition (you cannot open an attribute definition directly in Rational System Architect's explorer, for instance).

Example:

```
DEFINITION "Entity"
{
PROPERTY "Attributes"
   {ZOOMABLE EDIT COMPLETE ListOf "Class Attribute" KEYED BY
{"Class Name":"Name", Name} ASGRID LENGTH 4096 DISPLAY {
FORMAT List } }
..
}
```

COMPONENT_ SCRIPT

Calls a function written in Basic, using function calls to Rational System Architect that are included in what is referred to as SA Basic. The component scripts are used for ListOf and ExpressionOf lists. The action taken by the script works against each item in the list. For instance, the syntax

COMPONENT_SCRIPT ***fmtomtattr*** returns all attributes and their corresponding C- storage types, separated by a colon (:).

By convention, the function itself is named with one of the following prefixes:

- fmtxxx – The function itself exists in hard code and cannot be modified. Most functions in SAPROPS.CFG are this way. Hard-coding the function is done to make Rational System Architect's overall response faster.
- _fmtxxx – Exists in the fmtscript.bas file within Rational System Architect's main executable directory.

<u>Creating Your Own Script</u>

For information on how to create your own script, see the SCRIPT keyword.

<u>Explanation of Existing Scripts:</u>

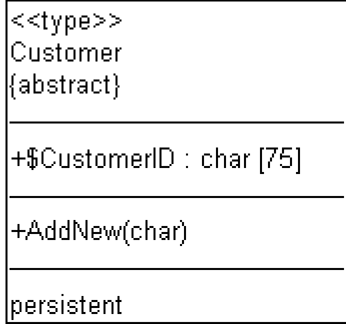
fmtUMLAttr returns all attributes and their corresponding types, separated by a colon.

fmtOMTOperation returns all operations and their corresponding C-storage types, enclosed within parenthesis (type).

FmtOMTObjInstAttr returns all attributes for the class that an object instantiates.

FmtOMTActivity returns the script **do**: and the name of the activity for all activities listed in a state definition.

FmtOMTStateActions returns the name of the internal action for all internal actions listed in a state definition.



Example (using *fmtomattr*):

Definition “Class” {
  PROPERTY "Attributes"
  { PROPERTY "Attributes" {ZOOMABLE EDIT COMPLETE ListOf "Class Attribute" KEYED BY {"Package", "Class Name":"Name", Name } LENGTH 4096 ASGRID DISPLAY { **FORMAT COMPONENT_SCRIPT _FmtNewUMLAttr** LEGEND "$$FORCE$$"} LABEL "Attributes" }
 }

CONTROL

The Control keyword is equivalent to the Property keyword, when used with TESTPROC’s to set up a switch within a definition.

There are two ways to specify that a property appears in a definition dialog depending on the value of a switch. You may use #ifdef’s, which act upon values that you set for the encyclopedia in the Encyclopedia Configuration dialog (for example, setting the language type of the encyclopedia to Java or C++). The Encyclopedia Configuration dialog actually sets values in the sadeclar.cfg file.

You may also specify that a PROPERTY appears in a dialog (and what its initial value is) based on a switch that is itself a property (TESTPROPERTY) within the definition dialog. For example, you may specify within an entity that it’s DBMS type is Oracle or SQL Server. Subsequent properties will appear or not appear in the definition, and have certain default values, based on the value that you set for DBMS type. You use the TESTPROC keyword to specify the TESTPROPERTY switch. You use the PROPERTY keyword the first time you specify a particular property in the definition, and the Control keyword for every other occurance of that property in the definition. The REFPROP keyword is used to specify what PROPERTY each CONTROL is referencing. For this reason, the CONTROL and REFPROP keywords are often used in conjunction with TESTPROC's.

To summarize, for a CONTROL to be used, there must be an initial reference to the PROPERTY that the CONTROL references, at the top of the definition. The REFPROP keyword is used in conjunction with the CONTROL keyword.

Example:

Definition "Index"
{
CHAPTER "Modeling Properties"
{ **TESTPROC** TestProperty**Not**Value **TESTPROPERTY** "DBMS"
TESTSTRING { "ORACLE 8" } }
PROPERTY "Primary Key" {EDIT Boolean LENGTH 1 DEFAULT "F"
READONLY }
PROPERTY “Unique” {EDIT Boolean LENGTH 1 VALUESCRIPT
ProcessIndexUnique DEFAULT "F" }
PROPERTY “Clustered” {EDIT Boolean LENGTH 1 DEFAULT "F" }
..
CHAPTER "Modeling Properties "
{ **TESTPROC** TestPropertyValue **TESTPROPERTY** "DBMS"
TESTSTRING { "ORACLE 8" } }
CONTROL "Primary Key" { **REFPROP** "Primary Key" }
CONTROL “Unique” { **REFPROP** "Unique" }
CONTROL “Clustered” {**REFPROP** "Clustered"}
..
}

In the example above, the REFPROP keyword is used in conjunction with the CONTROL keyword to specify that the “Primary Key”, “Unique”, and “Clustered” properties are provided to the Index definition when Oracle 8 is selected as the DBMS – these properties will be exactly the same as their referenced property.

COPY PROPERTIES FROM

This command enables you to copy properties into the current definition type from other definition types. It enables you to consolidate similar concepts into a single definition type. This applies to definitions only. The syntax is as follows:

DEFINITION <object-1>
{
…
COPY PROPERTIES FROM <object 2> {[, <object n>]}…

Example:

DEFINITION "Elephant"
{
…
CHAPTER "Properties copied from Change Request"
COPY PROPERTIES FROM "Change Request"
CHAPTER "Properties copied from Dependency and Node"
COPY PROPERTIES FROM "Dependency", "Node"
…
}

The copy is performed at that point in the input where the copy statement is encountered. If, in the above example, properties are added to Change Requests, Dependencies or Nodes later in the property file(s), or existing properties are changed later in the property file(s), the additions and changes are **not** copied.

COPYSCRIPT

This keyword is used to specify an SABasic script to be invoked for a specific property when a copy of the definition is made.

Creating Your Own Script

For information on how to create your own script, see the SCRIPT keyword.

Example:

```
DEFINITION "Entity"
{
CHAPTER "Attributes"
PROPERTY "Description"
{ EDIT COMPLETELISTOF "Attribute" FROM "Data" KEYED BY
{Model, "Entity Name":"Name", "Name"} RELATE BY "uses" ASGRID
COPYSCRIPT OnCopyEntityDesc EDITCLASS
SACPropertyAttributeGrid
..}
```

COUNT_FIXED

The COUNT_FIXED keyword is used with the ASGRID keyword to specify that the user cannot delete or insert rows to a grid. The number of rows is fixed.

Example:

```
DEFINITION "Association"
{
BROWSER { OMITKEY }
LAYOUT { COLS 1 ALIGN OVER TAB }
CHAPTER "Roles"
PROPERTY "Association GUID" { KEY EDIT Text LENGTH 64
INVISIBLE READONLY}
PROPERTY "Class Roles" { EDIT COMPLETE ListOf "Association
End" KEYED BY { "Association GUID":"Association GUID",
"Association":"Name", "Package" QUALIFIABLE, "Class"
QUALIFIABLE, "Role GUID" QUALIFIABLE, "Name" }
RELATE BY "uses" LENGTH 4096 ASGRID COUNT_FIXED
BROWSER { SHOW } }
```

In the example above, an association between classes has a row in the grid for each class that the association attaches to (normally two, but can be three or more if additional classes are attached to the association line – this behavior is hard-coded in the software). Because of the COUNT_FIXED keyword, users cannot add to or delete rows in the grid.

Contrast this with other grids, for example the Use Case Step grid, wherein users may add new steps or delete steps from the grid.

DATA

This is ***not*** a keyword. It is a special word used as an argument of the ONEOF, LISTOF, and EXPRESSIONOF commands, providing a reference to data elements and data structures, which make up Rational System Architect’s data dictionary.

DATE

This keyword is an edit type whose length must be 10. The graphic display is based on the date format set in Windows. DATE is also an allowable field type which indicates that a property contains a date stamp in the notation appropriate to the time format defined to Windows.

CHECKOUT DATE, FREEZE DATE, INITIAL DATE, and UPDATE DATE each have special meanings.

Example 1:

```
DIAGRAM "Data Flow Gane & Sarson"
  {
    PROPERTY "Freeze date"
  { FREEZE DATE }
```

Other uses for the DATE might be found in any definition.

Example 2:

```
DEFINITION "X"
  { PROPERTY "Creator Date"
    { EDIT Text INITIAL DATE LENGTH 12 READONLY }
  }
```

DEASSIGN

The keyword DEASSIGN is used for removing symbols from a diagram type.

Example:

```
SYMBOL "Message Flow" in "Business Process"
{
DEASSIGN from "Business Process"
}
```

DEFAULT

The value assigned by Rational System Architect to a property which may be overridden by the user. On the graphic screen, the default value is initially displayed in a text box, or determines whether a check box is initially toggled on or off.

Example:

```
PROPERTY "Not a table"
  { EDIT Boolean LENGTH 1 DEFAULT F }
```

DEFINED BY

This keyword associates a definition to a symbol. It also enables you to reassociate a symbol to a different definition.

Meaning 1: If you add new symbols to an encyclopedia in USRPROPS.TXT, you must specify what definition type they are associated with using this keyword. If a new symbol specified in USRPROPS.TXT is missing this clause, Rational System Architect will give a parsing warning when opening the encyclopedia, and default to the null definition for the symbol.

Example 1:

```
RENAME DIAGRAM "User 1" to "My Diagram"
RENAME SYMBOL "User 1" to "Direction"
RENAME DEFINITION "User 1" to " Direction"

SYMBOL "Direction"
{
DEFINED BY " Direction"
ASSIGN TO "My Diagram"
}
```

In the example above, a new diagram type, symbol type, and definition type have been specified in USRPROPS.TXT. The DEFINED BY keyword is used to specify that the symbol “Direction” is defined by the “Direction” definition. In addition, the symbol is assigned to the “My Diagram” diagram. (Note: You could also specify a definition statement for the new definition, “Direction”, but this is not mandatory. If not specified, the new definition will simply have a default properties “Name” and “Description”.)

Meaning 2: The DEFINED BY keyword also enables you to define a symbol by a different definition than that specified in SAPROPS.CFG. When using this keyword to reassociate a symbol to a different definition, be sure to specify what diagram the symbol you are referring to is represented in (for example, Symbol “Class” in “Class” versus Symbol “Class” in “Component” – in the first case we specify we are redefining

the class symbol definition in a Class diagram; in the later case, the class symbol in a Component diagram.)

Example 2:

```
SYMBOL Process IN "Data Flow Gane & Sarson"
{
DEFINED BY "Control Transform"
}
```

In the example above, the Process symbol in a “Data Flow Gane & Sarson” diagram is now defined by “Control Transform”. Normally, it is defined by “Process”.

DEFINITION

This keyword is the first word in a block in which the properties of a DEFINITION, as opposed to a DIAGRAM or a SYMBOL, are listed.

Example:

```
DEFINITION "Data Element"
  {
    PROPERTY "Length"
  { EDIT number LENGTH 2 }
  .
  .
  .
  }
```

See also keywords DIAGRAM and SYMBOL.

DEFINITION REFERENCED IN

See ‘OF DEFINITION REFERENCED IN’.

DEPICT LIKE

The DEPICT LIKE keyword combination is used to specify how a symbol is depicted on a diagram. You may use this keyword combination when creating a new symbol, and specifying what it should look like on a diagram. You may specify that it looks like a symbol on another diagram.

You may use the DEPICT LIKE keyword combination with 'node' symbols and with 'line' symbols.

Example (Node Symbol):

```
SYMBOL "Communications Connection"
{
ASSIGN TO "OV-01 Highlevel Op. Concept"
..
DEPICT LIKE "Event Flow" IN "Data Flow Ward & Mellor"
```

Example (Line Symbol):

```
SYMBOL "Need Line"
{
PROPERTY "From Operational Node"
{EDIT ONEOF "Operational Node" READONLY INVISIBLE}
PROPERTY "To Operational Node"
{EDIT ONEOF "Operational Node" READONLY INVISIBLE}
DEPICT LIKE "Transition" IN "OMT State"
DEFINED BY "Need Line"
ASSIGN TO "OV-02 Op. Node Connectivity"
}
```

DEPICTIONS

Identifies how a symbol can be represented by an image file that you supply. You may depict a symbol with a bitmap or metafile. You may specify how this symbol is depicted on the diagram workspace using the DEPICTIONS keyword combined with the DIAGRAM keyword. You may also specify how the symbol is depicted in the toolbox and Draw menu using the DEPICTIONS keyword combined with the MENU keyword. The syntax is as follows:

```
SYMBOL <symbol-type-name>
        { ...
        DEPICTIONS { DIAGRAM <depiction-file> }
        DEPICTIONS { MENU <depiction-file> }
        ...}
```

where <depiction-file> is the name and full path of a bitmap or a metafile.

Example:

Rename Symbol "User 3" To "Radar
SYMBOL "Radar"
{ASSIGN To "Wireless Network"
DEPICTIONS { DIAGRAM "C:\Program Files\IBM\pictures\radar.bmp" }
DEPICTIONS { MENU "C:\Program Files\IBM\pictures\radartoolbar.bmp" }}

You may also use the DEPICTIONS keyword within a list, so that the symbol is depicted in different ways based on the value of the list that is selected.

Example:

List "Class Stereotypes"

{

Value "actor"**DEPICTIONS** {DIAGRAM images\slctact.wmf MENU images\slctact.bmp}

Value "boundary"**DEPICTIONS** { DIAGRAM images\slctbndy.wmf MENU images\slctbndy.bmp}

..}

DEFINITION “Class” {

PROPERTY "Stereotype" { EDIT Text LIST "Class Stereotypes" INIT_FROM_SYMBOL Default "" LENGTH 20 } ..}

DIAGRAM

The DIAGRAM command is used in two different ways.

Specifying Diagram Properties:

The DIAGRAM command is used as the first word in a block in which the properties of a diagram, as opposed to a DEFINITION or a SYMBOL, are listed.

Example:

```
DIAGRAM "Booch Class"
  { PROPERTY "DGX File Name"
    { EDIT Text LENGTH 255 }
  PROPERTY "Notes"
    { EDIT Text LENGTH 4000 }
     }
```

See also keywords DEFINITION and SYMBOL.

Used with DEPICTIONS Command:

References the graphic used to represent a symbol on the diagram workspace, as compared to on the Draw toolbar or menu.

Example:

```
SYMBOL "Satellite"
{ASSIGN To "Wireless Network"
DEPICTIONS { DIAGRAM "C:\Program
Files\IBM\pictures\satellite.bmp" }
DEPICTIONS { MENU "C:\Program
Files\IBM\pictures\satellitetoolbar.bmp" }}
```

DISPLAY

Causes a property and its value to be displayable on a diagram symbol. There is a limit of 37 display statements for one definition.

The syntax is as follows:

DISPLAY { FORMAT [STRING | LIST | KEY | NONKEY | COMPONENT_SCRIPT | COLUMN_SCRIPT | SCRIPT] **LEGEND "** (how the block is labeled in the symbol) **" }**

You have the option of specifying one of the following FORMAT keywords:

STRING: Causes the values of the property to appear on the symbol exactly the way they are typed. See the STRING keyword for an example.

LIST: Causes items to be displayed on the symbol in a list – each whitespace character causes a new line, unless the whitespace falls within double quotes. See the LIST keyword for more information.

KEY: Use this keyword for properties designated as keys. They are displayed in a separate section of the symbol. See the KEY keyword for an example and further information.

NONKEY: You may use this keyword for non-key properties. They will be displayed in a separate section of the symbol. This keyword was originally used for entities and tables in Rational System Architect's data modeling support. See the NONKEY keyword for an example.

COLUMN_SCRIPT: See COLUMN_SCRIPT keyword.

COMPONENT_SCRIPT: See COMPONENT_SCRIPT keyword.

SCRIPT:. See SCRIPT keyword.

DISPLAY (continued)

Within the quotes after the LEGEND keyword, you specify how the block is labeled in the symbol. Your choices are as follows:

LEGEND "<Your Text>": Whatever text you place in the quotation marks will be displayed on the symbol above the entry, only if there is a value for the entry.

LEGEND "": Displays a straight line without any words, only if there is a value for the entry.

LEGEND "$$FORCE$$": Displays a horizontal line above the entry on the symbol. This line acts as a divider. The "$$FORCE$$" keyword is different than simply using " ", in that it forces display of a horizontal line even if the property display is suppressed through the display mode dialog.

LEGEND "$$NONE$$": Does not display a horizontal line above the entry on the symbol, whether or not there are values for the entry. This line normally acts as a divider.

LEGEND "$$VFORCE$$": Enables you lay out properties from left to right inside symbols, and draws vertical lines between them. See VFORCE keyword.

LEGEND "$$VNONE$$": Enables you to lay out properties from left to right, but *does not* provide a dividing line. See VNONE keyword.

Example:

```
DEFINITION "Organizational Entity"
  { PROPERTY "Incumbent Name"
    { EDIT Text LENGTH 100  HELP "Name of person
          currently in position"
  DISPLAY { FORMAT String LEGEND "" } }
```

EDIT

In conjunction with the keyword BEGIN (or {), indicates the beginning of the definition of a property. The keyword EDIT carries the meaning, "This is the beginning argument."

Example:

```
SYMBOL "Process" IN "Data Flow Gane & Sarson"
  { PROPERTY "Short Description"
    { EDIT Text LENGTH 1500 }
  PROPERTY "Number" { EDIT Numeric LENGTH 4 }
```

EDIT COMPLETE

See the COMPLETE keyword.

EDITCLASS

Do not use this keyword. This keyword is a special keyword developed specifically for a certain situation in Rational System Architect, inheritance of Data Element properties by an Attribute in an Entity. You will see this command in SAPROPS.CFG used for this situation. This is the only situation that this keyword can be applied to. Use in other situations may cause errors.

EDIT URLS

You may specify that a listof property is designated as one which can reference external documents. The command causes buttons to be presented at the bottom of the listof property -- an Open button, a Browse Externally button, and a Browse Internally button. You may use these buttons to browse and select external documents, or type in external hyperlinks, or browse the internal Files table of the encyclopedia's database, or open an external or internally referenced document.

Syntax:

PROPERTY property name { **EDIT URLS** }

Restrictions:

SA enforces the following restrictions on URLS property -- It may not be Key.

Example:

Definition "Use Case"
{
PROPERTY "Reference Documents" { EDIT URLS }
}

END

Indicates the end of the specification of a property, or of the group of properties making up the definition of a diagram, symbol, or definition. It is combined with the BEGIN statement to enclose the specification. Instead of the BEGIN and END statements, you can also use opening and closing braces, **{ }**

Example:

PROPERTY "<property_name>"
  BEGIN EDIT <edit_type> <property_parameter>
  END

EXPRESSION

Indicates that the value of the definition must be entered as a series of strings separated by a + sign, or white space.

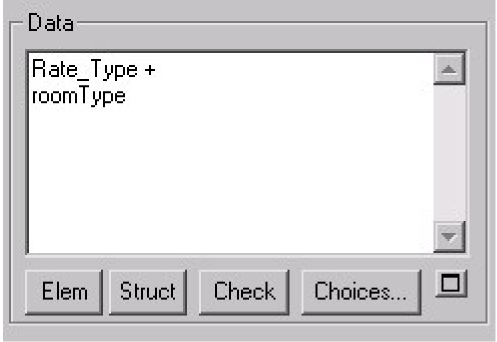
Example:

VendorName +
VendorCity +
VendorState

See also keyword EXPRESSIONOF which has replaced this keyword.

EXPRESSIONOF

EXPRESSIONOF allows you to express references to objects using complex operators and delimiters. EXPRESSIONOF is normally used with the special argument DATA, which refers to data elements and data structures – in other words, EXPRESSIONOF DATA. This keyword combination provides a text box within which definition values are entered as a series of strings. The division between one definition value and the next is determined by white space. By convention, a + sign is used to divide the individual definition values, but it is not required.



Example:

```
DEFINITION "Control Flag"
{
PROPERTY "Description"
{ EDIT EXPRESSIONOF DATA LENGTH 4074 LABEL "Data" }
}
```

See also keywords ONEOF and LISTOF, and refer to Chapter 2, section on ExpressionOf, for more information and a list of operators and delimiters that can be used.

fmtxxx or_fmtxxx

These two name prefixes are used, by convention, at the beginning of the name of any function called by the SCRIPT, COLUMN_SCRIPT, or COMPONENT_SCRIPT keywords. The function itself (for example, _fmtUMLAttr) usually provides a special formatting display of a property value (such as an attribute and all of its properties) on the symbol. The naming convention is as follows:

- _fmt (for example, _fmtUMLAttr): The function itself exists in hard code and cannot be modified. Most functions in SAPROPS.CFG are this way. Hard-coding the function is done to make Rational System Architect's overall response faster.
- fmt (for example, fmtUMLAttr): Exists in the fmtscript.bas file within Rational System Architect's main executable directory.

Example:

DEFINITION “Class”
PROPERTY "Attributes" {ZOOMABLE EDIT COMPLETE ListOf "Class Attribute" KEYED BY {"Package", "Class Name":"Name", Name } LENGTH 4096 ASGRID DISPLAY { FORMAT COMPONENT_SCRIPT **_FmtNewUMLAttr** LEGEND "$$FORCE$$"} LABEL "Attributes" }

In the example above, a script is called that is used to display attributes on a class diagram in a particular fashion (for example, if an attribute's access property is ‘public’, a ‘+’ mark is placed before the attribute on the class symbol, etc.

Creating Your Own Functions

To create your own functions, see the SCRIPT keyword.

See also the DISPLAY, FORMAT, SCRIPT, COLUMN_SCRIPT, COMPONENT_SCRIPT, and VALUESCRIPT keywords.

FORCE

Actually the $$FORCE$$ keyword, used with the DISPLAY keyword.

For information, see the DISPLAY keyword.

FORMAT

Indicates the way data is to be presented for a specific displayable property.

Example:

```
PROPERTY "Description"
  { EDIT ExpressionOf "Data"
  Display { FORMAT List LEGEND "Data" } }
```

Refer to DISPLAY keyword for more information.

FREEZE

Displays information concerning the freezing of an object, such as the AUDIT ID of who froze it, or the DATE or TIME of freezing. The displayed fields are always READONLY. Values are automatically kept track of by Rational System Architect, but in order to view the values in a dialog, you must add properties with the following respective characteristics:

FREEZE Auditid
FREEZE Date
FREEZE Time

Example:

```
DIAGRAM "Data Flow Gane & Sarson"
  {
    PROPERTY "Frozen by"
  { FREEZE Auditid }
    PROPERTY "Freeze Date"
  { FREEZE Date }
    PROPERTY "Freeze Time"
  { FREEZE Time }
  }
```

Search on *Access Control* in the on-line help for more information on freezing objects.

See also keyword CHECKOUT.

FROM_CHOICES_ONLY

Restricts a user to only select a definition from a choices list, without being able to type in a new definition. A message box appears prompting a user to select only from the "Choices" list. This is used with ListOf and OneOf.

For example:

```
DEFINITION "Product"
{
CHAPTER "Technical Reference Model"
PROPERTY "Status" {Zoomable EDIT Oneof "Product Status"}
Group "Involvements"{
LAYOUT { COLS 2 ALIGN OVER }
PROPERTY "Lead Proponent" {Zoomable EDIT Oneof "Organizational Unit" FROM_CHOICES_ONLY}
PROPERTY "Others Involved" {Zoomable EDIT Listof "Organizational Unit" FROM_CHOICES_ONLY}
}
}
```

GROUP

Used to produce a group box with specific layout parameters, such as a series of radio buttons, within which two or more properties are located.

Example 1:

```
GROUP "Referential Integrity"
  {
    LAYOUT { ALIGN OVER TAB COLS 3 }
      PROPERTY "Parent Delete"
  { EDIT Text LISTONLY LIST RDC
      LENGTH 15 }
  …
  }  REM "End of Group Referential Integrity"
```

You cannot modify any GROUP name that is already predefined by Rational System Architect in the SAPROPS file.

Example 2:

```
DEFINITION "Class Attribute"
  { CHAPTER "Class, Source Data, Desc."
     GROUP "Source Data" {
  LAYOUT { COLS 2 ALIGN OVER TAB }PROPERTY
        "Description"
  { EDIT Text LENGTH 1500 }
```

In the third line of the example above, if you try to change 'GROUP "Source Data"' to 'GROUP "Original Data" in the USRPROPS.TXT file, your change will have no effect. The text contained in the SAPROPS GROUP entry, "Source Data" will not be overridden. It will continue to be the Class Attribute Group display text.

HELP

This is the string that is displayed on the status line in the ***lower left-hand corner*** of a **Diagram** or **Definition** dialog when a given property is selected.

Syntax:

HELP "<text_string>"

Example:

```
PROPERTY Length
  { EDIT Numeric LENGTH 2 MIN 1 MAX 99
    HELP "Length of this field"
  }
```

HETEROGENEOUS (ONEOF, LISTOF)

Enables a single property to refer to definitions of more than one type. (A normal list references definitions of a single type.) The HETEROGENEOUS keyword is used to modify either the ONEOF or the LISTOF keyword.

For example, when you click on the Choices button of a class list, only "class" definitions are provided to choose from. If you click on the Choices button of a heterogeneous list, you are provided with various types of definitions that you have specified in the Heterogeneous list clause, such as "class", "process", "entity", etc.

Syntax for ONEOF:

PROPERTY *property name* **{ EDIT HeterogeneousOneOf [** *class* **]** *type-1* **{ [,** *type-n* **] } … .etc. }**

Syntax for LISTOF:

PROPERTY *property name* **{ EDIT HeterogeneousListOf [** *class* **]** *type-1* **{ [,** *type-n* **] } … .etc. }**

Restricitons

There are certain restrictions for a Heterogeneous list property. It cannot also be one of the following (in other words, you cannot use any of the following keywords along with the HETEROGENEOUSLISTOF or HETEROGENEOUSONEOF keyword in the same property):

- The property may not be KEY.
- It may not contain a KEYED BY clause.
- It may not be COMPLETE.
- It may not have a FROM clause.
- It may not be ASGRID.
- It may not have a DEFAULT.
- It may not be INITIAL USER REQUIRED.
- It may not have a restriction (REFERENCED IN or WHERE) clause.
- It may not have the INIT_FROM_SYMBOL attribution.
- No type name may be listed more than once

Adding New Values to the List

Although most of the time users are expected to drag in values into a Heterogeneous list from the Select and Drag browser provided by clicking on the Choices button, users may add new values to the heterogeneous list. However, to add new values into a Heterogeneous list, users must enter the new values with their fully qualified name, in the following format:

ClassName:TypeName:FullyQualifiedName

Where:

- ClassName is the System Archtiect encyclopedia class types – Diagram, Symbol, or Definition.
- TypeName is the specific name of the Diagram, Symbol, or Definition type, such as Class (definition) or Use Case Step (definition).
- Each part of the FullyQualifiedName is separated by periods, so, for example, a Use Case Step, which is keyed by its Use Case, which is keyed by its package, would be entered as follows:

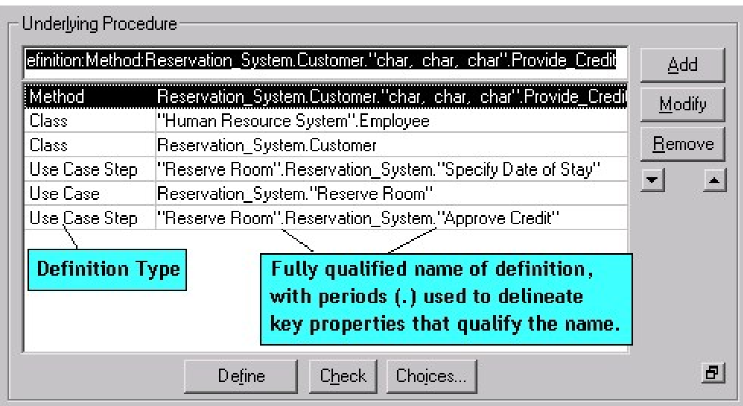
  Definition:"Use Case Step":"Package Name"."Use Case Name"."Use Case Step Name"

Example:

```
Definition " Procedure"
{
PROPERTY "Underlying Procedure" { EDIT
HETEROGENEOUSLISTOF " Use Case","Class", "Method", "Use
Case Step" READONLY}
```

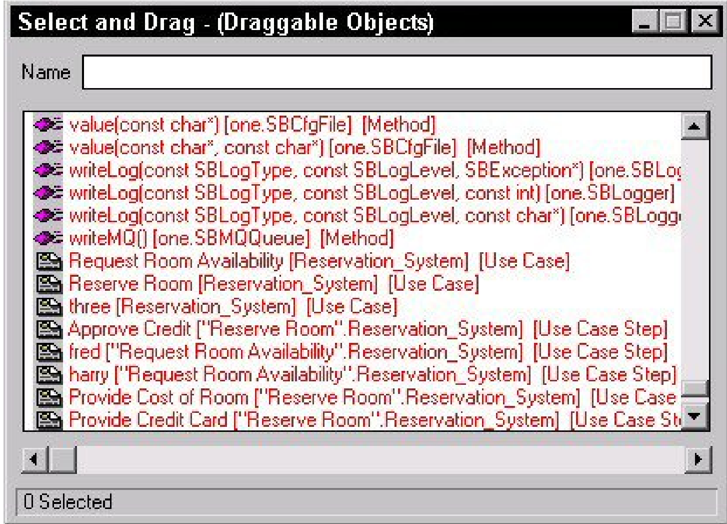
In the example above, the "Underlying Procedure" property of the "Procedure" definition can be populated with definitions of the type Use Case, Class, Method, and Use Case Step.

The user interface provided by the HETEROGENEOUSONEOF or HETEROGENEOUSLISTOF keyword displays a column that contains the name of each definition type, and the fully qualified name of the particular definition dragged into the list.



The key properties that qualify the name of a definition are provided in the user interface, separated from each other by periods (.). For example a Use Case Step is keyed to its containing Use Case, which is keyed by its containing package. In the HETEROGENEOUSONEOF or HETEROGENEOSLISTOF field, a Use Case Step is represented by “Package Name”.”Use Case Name”.”Use Case Step Name”. If there are embedded spaces in the name of any item, that item is enclosed in quotation marks. For example, in the picture above, the Use Case Step “Approve Credit” is in the Use Case Reservation_System, which belongs to the package “Reserve Room”.

When you click on the Choices button for a Heterogeneous list, all called for diagram, symbol, or definition types are presented, with their type listed in brackets after their name.



The Properties window also presents values of a heterogeneous list. You may drag on the borders of the Properties rows or columns to get a full look at the values. Each value is preceded by its class type (diagram, symbol, or definition), type name (ie, Use Case Step definition), and value itself.

Properties

Property	Value
Initial Date	12/21/2003
Initial Time	10:26:31
Initial Audit	LouV
GUID	80269da7-0626-459a-ad2c-99cd1dd8b393
Underlying Procedure	Definition:Method:Reservation_System.Customer."char, char, char".Provide_Credit Definition:Class:"Human Resource System".Employee Definition:Class:Reservation_System.Customer Definition:"Use Case Step":"Reserve Room".Reservation_System."Specify Date of Stay" Definition:"Use Case":Reservation_System."Reserve Room" Definition:"Use Case Step":"Reserve Room".Reservation_System."Approve Credit"
Last Change Date	12/21/2003
Last Change Time	10:32:22

HIDE DEFINITION

Removes the referenced definition type from the **New Definition** and **Open Definition** dialogs.

Syntax:

HIDE DEFINITION <definition name>

Example:

HIDE DEFINITION "SQL Server Table"

WARNING: You should exercise care when hiding definitions, especially if they are used by symbols that you have made active by choices in the Property Configuration dialog (Tools, Customize Method Support). You may find yourself in a situation where you are drawing symbols with no underlying definitions.

HIDE DIAGRAM

Removes the referenced diagram type from the **Diagram New** and **Diagram Open** dialogs.

Syntax:

HIDE DIAGRAM <diagram name>

Example:

HIDE DIAGRAM "Booch Process"

Note: Instead of using this keyword, a less drastic change is to simply deselect the diagram type from the **Property Configuration** dialog (select Tools, Customize Method Support, Encyclopedia Configuration, and either toggle off the method employing the diagram type or click the Advanced button on the Property Configuration dialog and move the diagram type from the “Selected Diagrams” to the “Available Diagrams” list).

HIERARCHICAL

By default, user diagrams are networks (of symbols), but if this keyword is included in a diagram type's description, the diagram type is treated as a hierarchical diagram. This means that all node symbols assigned to it will have the capability of being arranged in a hierarchy and other related hierarchical functionality (such as hierarchical numbering) is supported.

The HIERARCHICAL keyword can only be used with user-defined diagram types – it cannot be applied to existing diagram types. In any other context it is ignored after a warning to the user. (For information on how to create a new user-defined diagram type, see RENAME DIAGRAM keyword.)

Example:

```
RENAME DIAGRAM “User 1” to “Zoo”
RENAME SYMBOL "User 1" to "Mammals"
RENAME SYMBOL "User 2" to "Reptiles"
RENAME DEFINITION “User 1” to “Mammal”
RENAME DEFINITION “User 2” to “Reptile”

SYMBOL "Mammals"
{DEFINED by "Mammal"
ASSIGN TO "Zoo"}

SYMBOL "Reptiles"
{DEFINED by "Reptile"
ASSIGN TO "Zoo"}

DIAGRAM "Zoo"
{HIERARCHICAL
PROPERTY "Hierarchical Numbering"
{ EDIT Boolean LENGTH 1 DEFAULT "T" }
PROPERTY "First Node Number"
{ EDIT Text  Length 20 DEFAULT "1" }
}
```

The diagram created by the USRPROPS.TXT above will be hierarchical in nature – it will be similar to an Organizational Chart, etc.

IFDEF

See #IFDEF command.

IFNDEF

See #IFNDEF command.

IN

Establishes the context for the **RENAME** command when applied to a symbol. This may also be used for **DEPICT LIKE**.

Examples:

For example, in order to rename an Application symbol:

```
#ifdef "Business Enterprise"
RENAME SYMBOL "Application" IN "System Architecture" TO
"My Symbol"
#endif
```

To make a symbol look like one in another diagram:

```
#ifdef "Business Enterprise"
SYMBOL "System" IN "System Context"
{
DEPICT LIKE "Process" IN "Data Flow Gane & Sarson"
}
#endif
```

INCLUDE

See #INCLUDE.

INITIAL

Used to stamp any diagram, symbol, or definition with the AUDIT ID, DATE, and TIME of its creation. The value of this field is never changed by Rational System Architect.

Variants:

INITIAL DATE

INITIAL TIME

INTIAL AUDITID

Starting in Rational System Architect V9, INITIAL DATE, INITIAL TIME, and INITIAL AUDITID are provided by default in the Access Data tab of each diagram or definition dialog. This is hard-coded in the product – in other words, you will not find the INITIAL keyword in each definition in SAPROPS.CFG, nor do you need to add it to USRPROPS.TXT for new diagram or definition types that you create.

Example:

```
DEFINITION "X"
{ PROPERTY "Creation Auditid"
{ EDIT Text INITIAL AUDITID LENGTH 12 READONLY }
}
```

See also UPDATE keyword.

INIT_FROM_SYMBOL

The INIT_FROM_SYMBOL keyword is used within a definition that defines a symbol. It specifies that a property in the definition initially inherits its value from a similarly named property in the symbol. This is used in cases where a property must exist in both symbol and definition, and should have the same value. A case where this is necessary is in specifying user-provided metafiles for a symbol based on a property such as Stereotype. The stereotype must be specified for the symbol (because this is what drives how the symbol is represented on the diagram) and in the corresponding definition.

Example 1:

```
LIST "Class Stereotypes"
{
VALUE "actor" DEPICTIONS {diagram images\slctact.wmfmenu
images\slctact.bmp}
VALUE "boundary" DEPICTIONS {diagram images\slctbndy.wmfmenu
images\slctbndy.bmp}
VALUE "case worker" DEPICTIONS {diagram images\slctcwkr.wmf
menu images\slctcwkr.bmp}
}

SYMBOL "Class" in "Class"
{
PROPERTY "Stereotype" { INVISIBLE EDIT Text ListOnly List "Class
Stereotypes" DEFAULT "" LENGTH 20}..}

DEFINITION “Class”
{
PROPERTY "Stereotype" { EDIT Text LIST "Class Stereotypes"
INIT_FROM_SYMBOL Default "" LENGTH 20 } ..}
```

In the example above, the Stereotype property is declared in both the specification of the class symbol and the class definition. It must have the same value. The stereotype property in the SYMBOL causes the drop-down display of possible stereotype values to select from in Rational System Architect’s Draw menu (which themselves are bitmaps specified by the DEPICTIONS clause in the LIST statement). Once you select a stereotyped class from the list in the Draw toolbar and place it on the diagram, the class’s definition is created and its stereotype property is automatically filled in by the stereotype you have chosen for the symbol. Note that if you change this value in the definition, it will change in the symbol.

INIT_FROM_SYMBOL (continued)

Note also that you do not see this stereotype value in the symbol tab of the class because it has been made INVISIBLE.

Example 2:

DIAGRAM “Class”
{
PROPERTY "Programming Language"{ EDIT Text Listonly LIST "Programming Languages" Default "CORBA" LENGTH 30 **INITIAL USER REQUIRED** }
}

SYMBOL "Class" in "Class"
{
PROPERTY "Package" { EDIT OneOf "Package" READONLY }
PROPERTY "Stereotype" { INVISIBLE EDIT Text ListOnly List "Class Stereotypes" DEFAULT "" LENGTH 20}
PROPERTY "Programming Language" { INVISIBLE EDIT Text ListOnly List "Programming Languages" DEFAULT "" LENGTH 30}
}

DEFINITION “Class”
{
PROPERTY "Package" { KEY EDIT OneOf "Package" RELATE BY "is keyed by" READONLY}
PROPERTY "Stereotype" { EDIT Text LIST "Class Stereotypes" INIT_FROM_SYMBOL Default "" LENGTH 20 }
PROPERTY "Programming Language"{ EDIT Text ListOnly LIST "Programming Languages" **INIT_FROM_SYMBOL** Default "CORBA" LENGTH 30 **INITIAL USER REQUIRED** READONLY }
}

In the example above, the Programming Language property exists in the diagram, and the Class symbol inherits the value of this property from the diagram. The Class symbol’s definition also inherits the value of this property through the symbol, because of the INIT_FROM_SYMBOL keyword.

If a Class definition is created via the explorer, the required property MUST be supplied at the time of its creation because of the INITIAL USER REQUIRED keyword in the Class definition.

INITIAL USER REQUIRED

This keyword specifies that at the creation time of the modeling element (either diagram, symbol, or definition), a value for the property **must** be supplied. If you do not supply it, and try to close the dialog by pressing OK, Rational System Architect will give you a message that says “The xxx property must be supplied.” You will not be able to click OK to close the dialog and create the diagram, symbol, or definition. You will either need to supply a value for the property, or cancel the dialog.

Example:

DIAGRAM "Activity"
{
PROPERTY "Package" { EDIT OneOf "Package" RELATE BY "is part of" **INITIAL USER REQUIRED** OVERRIDABLE }
PROPERTY "Activity Model" { EDIT OneOf "Activity Model" ReadOnly **INITIAL USER REQUIRED** } ..}

In the example above, both properties “Package” and “Activity Model” must be filled in before you can click the OK button in the diagram dialog when creating an Activity diagram.

Note that in the example above, the property “Package” is also OVERRIDABLE, while the property “Activity Model” is not. The OVERRIDABLE keyword only has meaning to symbols drawn on this diagram that inherit values of the property from the diagram.

Example 2:

Diagram “XML”
{..
PROPERTY "XML Schema" { Edit OneOf "XML Schema" **AUTOCREATE** Relate By "is part of" **INITIAL USER REQUIRED OVERRIDABLE READONLY** } ..}

In the example above, the READONLY keyword used in conjunction with INITIAL USER REQUIRED keyword specifies that the dialog cannot be closed unless a value is entered for this property by the user, and that after the initial value is supplied, the property becomes readonly and cannot be changed by the user. OVERRIDABLE only has meaning to the definitions inheriting this property value from the diagram.

INITIAL USER REQUIRED (continued)

So the INITIAL USER REQUIRED keyword mandates that a value for the XML Schema property is be supplied upon creation of the diagram. The AUTOCREATE keyword automatically creates a definition for any value entered into this property. Therefore, when the user clicks OK to close the Diagram dialog, a defined XML Schema definition is created.

See also OVERRIDABLE, READONLY, and AUTOCREATE keywords.

INVISIBLE

Renders a property nonvisible in the graphic dialog without deleting it. Invisible properties are used in situations where a property is needed for a definition, but is meaningless to the user.

Example:

SYMBOL "Class" in "Class"
{
PROPERTY "Package" { EDIT OneOf "Package" READONLY }
PROPERTY "**Stereotype**" { **INVISIBLE** EDIT Text ListOnly List "Class Stereotypes" DEFAULT "" LENGTH 20}
PROPERTY "Programming Language" { INVISIBLE EDIT Text ListOnly List "Programming Languages" DEFAULT "" LENGTH 30}
}

DEFINITION “Class”
{
PROPERTY "Package" { KEY EDIT OneOf "Package" RELATE BY "is keyed by" READONLY}
PROPERTY "**Stereotype**" { EDIT Text LIST "Class Stereotypes" INIT_FROM_SYMBOL Default "" LENGTH 20 }
PROPERTY "Programming Language"{ EDIT Text ListOnly LIST "Programming Languages" INIT_FROM_SYMBOL Default "CORBA" LENGTH 30 INITIAL USER REQUIRED READONLY }
}

In the example above, the Stereotype property is used with both the Symbol and the Definition of a class. It must match. Users may choose the stereotype property in the class, and that value is automatically given to the symbol, where it is used to determine how the symbol is displayed. However, the user does not need to see the Stereotype property within the Symbol tab of the Class definition, since it is already in the Class definition dialog. To have it in both places would only confuse the user. It is made invisible.

See also keyword VISIBLE.

JUSTIFY

This command is **no longer used** in SAPROPS.CFG or USRPROPS.TXT. It will not cause an error if specified in USRPROPS.TXT, it will simply be **ignored** by the USRPROPS.TXT parser. It used to be one of the arguments of the LAYOUT command. When used, it lined up all controls to the edge of the right and left margin of the dialog page.

See also keywords LAYOUT and ALIGN.

KEY

The KEY keyword is used to establish a property as a key. Keys are used to determine the name space of modeling elements in the encyclopedia. The KEY keyword also has a second usage – it is one of the allowed arguments following the FORMAT keyword in the DISPLAY command. For that latter usage, see KEY (Used for Display).

By default, every modeling element in an encyclopedia is distinguished by its class (whether it is a diagram, symbol, or definition), its type (whether it is a UML Use Case diagram, a BPMN Process diagram, etc), and its name (for example, the Reservation_System Use Case diagram versus the Human_Resource_System Use Case diagram). In addition to these built-in defaults, you may also specify additional keys for a definition modeling element – for example, a class attribute definition is keyed by its containing class definition, and that class's containing package definition.

To use the KEY command, you specify it within the property that you want to be a key of a definition. The KEY command may be placed almost anywhere within the description of a property, but because of its importance, it is customary to place it as the first item within the property's braces – just before the EDIT keyword.

Example:

```
Definition "Use Case Step"
{
PROPERTY "Use Case Name" { KEY EDIT …  }
PROPERTY "Package" { KEY EDIT …}
…
```

KEY (continued)

For a property that is a key and that "points at" another object(s) – for example, a LISTOF or ONEOF property, not a simple TEXT or NUMERIC property – the end user must specify the class and the class type of the referenced object(s) when entering a value for the property while working in Rational System Architect.

For example:

```
Definition "Business Process"
{
PROPERTY "System Use Case" {EDIT ONEOF "Use Case" …}
```

The statement above indicates that the property "Use Case Name" refers to a definition of type "Use Case". Definition is the default when no *class* is specified (*class* in the Rational System Architect sense -- Diagram, Symbol, or Definition).)

The property value itself often will contain all the necessary remaining material needed to identify the object(s) actually being referenced. If the referenced class/type of the property has no key properties, the reference value will just be the object's Name (because the class and type are known), but if the referenced class/type has key properties (such as "Use Case" in the above example, which has key property "package"), Rational System Architect must know the values of these key properties in order to properly identify the reference object.

You either code this into USRPROPS.TXT so that Rational System Architect automatically gets the values for the end user or you force the end user to type in the fully qualified name, with periods separating the key parts.

- To have Rational System Architect automatically get the value for users, you use the KEYED BY command.
- If a KEYED BY clause is not given for the property, Rational System Architect expects these additional key values to be given in the reference itself – in other words the user must type in the fully qualified name of the reference object, with periods separating key values (for a Use Case Step called "*Specify email"* in a Use Case called *Order_Product* in a package called "*Order System"* the user would need to type in "*Order System".Order_Product."Specify email"*.

KEY (continued)

Note: Heterogeneous reference properties are different in this respect. See HETEROGENEOUS.

One other use for the KEYED BY clause is that it enables you to build a list of things that are all related. For example, all the Use Case Steps referred to in the property “Use Case Steps” of a Use Case definition belong to the same Use Case – as it happens, the one containing the “Use Case Steps” property. Where a multiple reference property (like ListOf) refers to objects all belonging to the same parent object, it is advised to use one or more other properties to identify the parent object. In these situations, a KEYED BY clause is used to tell Rational System Architect which other properties to use.

Note: Key properties of a definition are not shown in a grid formed by an ASGRID command. For example, in a Use Case definition, Use Case Steps are depicted in a grid formed by an ASGRID command, however, the key properties of Use Case Steps (owning package and Use Case) are not shown in the grid of Use Case Steps.

Note: It is not possible to add a KEY EDIT ONEOF to a diagram.

See also KEYED BY keyword.

KEYED BY

A KEYED BY clause is optionally used to specify how the key components of a referenced object(s) may be found. The KEYED BY clause contains a portion for each key component separated by a comma.

The KEYED BY clause provides two benefits:

1. It eliminates the need for the end user to type in the fully qualified name of a reference value (with periods separating qualifiers). For example, for a property that references a class attribute named ***email*** of the class ***Customer*** of the package ***"Order System"***, instead of typing in ***"Order System".Customer.email***, the end user simply types in ***email***.
2. It can be used to ensure that all key components of a reference value are the same. For example, the LISTOF "Class Attribute" property in a Class definition contains a list of attributes that all belong to the same class and to the same package.

Example:

For example, the KEYED BY clause of the Class's "Class Attribute" property could be as follows:

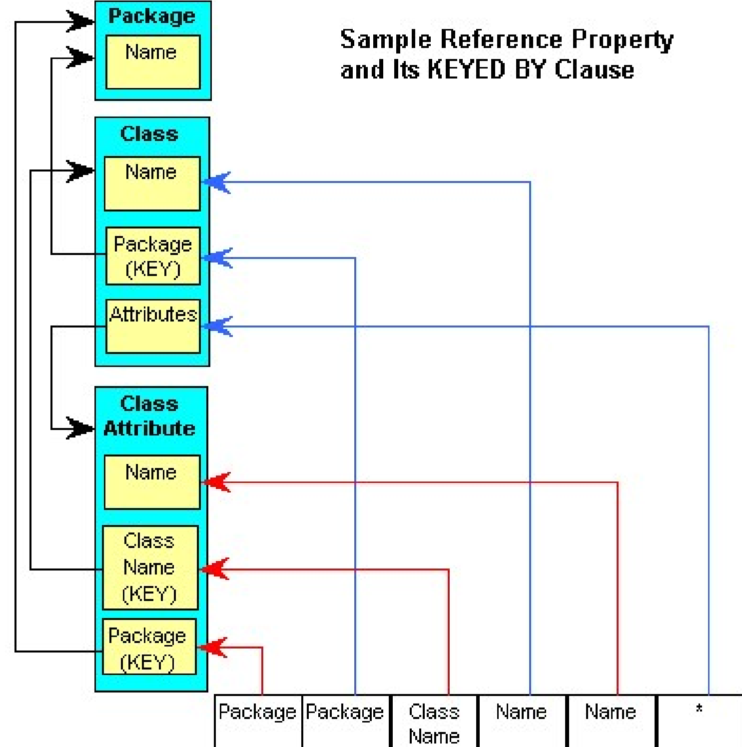
```
DEFINITION "Class"
{
…
PROPERTY "Attributes" { … LISTOF "Class Attribute"
KEYED BY {Package:Package, "Class Name":Name, Name:* } … }
```

In the example above, the three *key components* (separated by commas) are Package:Package, "Class Name":Name, and Name:*. These components refer to the three parts needed to identify the referenced Class Attribute definitions – the Package name, the Class name, and the Class Attribute name. Taking them in reverse order, it states that:

- The name of the Class Attribute will be found in ***this*** property (* means "here"), hence: **Name:***
- The value of the key property "Class Name" in the Class Attribute definition will be found in this object's name, hence: **"Class Name":Name**
- The value of the key property Package in the Class Attribute definition will be found in this object's Package property, hence: **Package:Package**

KEYED BY (continued)

The following schematic diagram shows how the KEYED BY clause is used in the example above, and may be useful in understanding the KEYED BY clause generally.



The schematic shows what we have said above – in the definition of a class, a class attribute is entered by specifying its package (stored in the class attribute's Package property and obtained from the Package value of the class you are in), its class name (stored in the class attribute's "Class Name" property and obtained from the class's actual name), and name (stored in the class attribute's "Name" property and obtained from itself).

KEYED BY (continued)

In summary:

1. For each key component of the **reference object**, the KEYED BY clause has a component.
2. The components of the KEYED BY clause are separated by commas.
3. Each component has two parts:
   - The first part identifies the key component of the reference object,
   - The second part states where the value of that component is to be found, and
   - The two parts are separated by a colon.

However, certain default values may be assumed to simplify the KEYED BY clause. If the two parts of the component are the same, the second may be omitted and if the second part of the last component is omitted, it assumed to be "here" – i.e. the asterisk. Thus, in practice the KEYED BY clause of the Class's "Attributes" property is coded:

```
KEYED BY {Package, "Class Name":Name, Name }
```

Naturally, all the properties used in the KEYED BY statement must exist. Thus, Rational System Architect checks that there is a "Package" property and a "Class Name" property in the "Class Attribute" definition and that they are both KEY.

Besides saving all the effort of coding common key components in a LISTOF property like this one, employing a KEYED BY clause using other properties to provide common values **ensures the same values are used for each reference**. Thus, in the example we have been using, all the Class Attributes referred to in the "Attributes" property of the Class are forced to belong to the same class in the same package – a desirable characteristic in this case.

At other times it is convenient to have the key components of the referenced object separated for reasons of clarity and simplicity. Under such circumstances a KEYED BY clause is used to designate the properties supplying the separate components. Indeed, for these reasons, when a property is KEY and refers to an object with KEY properties, Rational System Architect **requires** that the components be in separate properties.

KEYED BY (continued)

Often it is desirable that some key component values besides the names be provided in the reference itself rather than taken from another property. This may happen when there is no suitable property to provide a value or when it is not desirable that the key component be the same value for all references in the property. In this case, the keyword QUALIFIABLE is used. For example, in the class definition there is this property:

PROPERTY "Operations" {Edit … ParmListOf "Method"
KEYED BY {"Package","Class Name":Name,"Formal Parameters"
QUALIFIABLE, Name } … }

This indicates that although the values of the “Package” and “Class Name” key properties of the Methods referenced should be taken from the Class’s “Package” property and Name respectively, the values of the “Formal Parameters” property of the Methods and their names should be taken from the Class’s“ Operations” property itself. Thus each reference will contain two components, the value of the “Formal Parameters” property and the value of the name separated by a period.

Note that Rational System Architect requires that KEY properties that have a KEYED BY clause **not** use the QUALIFIABLE keyword. This is for the reasons of clarity and simplicity mentioned above.

KEY (Used for Display

KEY is also one of the allowed arguments following FORMAT in the **DISPLAY** command. Properties designated as keys are displayed in a separate section of the symbol.

Example:

```
DEFINITION "Entity"
{
PROPERTY "Description"
{ EDIT COMPLETE LISTOF "Attribute" FROM "Data Element"
KEYED BY {Model, "Entity Name":"Name", "Name"} RELATE BY
"uses" ASGRID COPYSCRIPT OnCopyEntityDesc EDITCLASS
SACPropertyAttributeGrid Label "Attribute List" LENGTH 4096
ZOOMABLE DISPLAY { FORMAT KEY LEGEND "Primary Key" }
DISPLAY { FORMAT NONKEY LEGEND "Non-Key Attributes" }
...}
```

In the example above, the FORMAT KEY command places all attributes designated by the user as primary keys under the “Primary Key” legend on an entity symbol. The FORMAT NONKEY command places all non-primary-key attributes under the “Non-Key Attributes” legend on the entity symbol.

LABEL

The **LABEL** command is used for two purposes.

Purpose 1: LABEL is one of the arguments of the **ALIGN** command. It is used to align all controls one space to the right of the widest label in that column. (Contrast this with the ALIGN OVER keyword pair, which places the name over the property.)

Example:

```
Definition "My Definiition"
{
CHAPTER “My Chapter”
LAYOUT { COLS 1 ALIGN LABEL }
PROPERTY "My Property 18"{ EDIT Text Length 10}
PROPERTY "My Property 19"{ EDIT Text Length 10}
PROPERTY "My Property 20"{ EDIT Text Length 10}
PROPERTY "My Property 21"{ EDIT Text Length 10}
PROPERTY "My Property 22"{ EDIT Text Length 10}
PROPERTY "My Really Long Property 23"{ EDIT Text Length 10}
}
```

In the example above, the control for “My Really Long Property 23” is a text box placed one space to the right of the label. All other text-box controls for other properties on the dialog are lined up with this control.

See also keywords ALIGN, BODY, and OVER.

LABEL

(continued)

Purpose 2: LABEL is used to **relabel** the name of tabs (chapters), groups, or properties in a dialog. You cannot remove a property name which has been defined in SAPROPS. However you can modify the text that is displayed for the property by using the LABEL command in USRPROPS.TXT.

Example 2:

```
DIAGRAM "Data Flow Diagram" {
  PROPERTY "Event Label Prefix"
    { EDIT Text LENGTH 10 }
  PROPERTY "Key Letters"
    { EDIT text LENGTH 10 LABEL "Process Prefix" } ..}
```

Adding the code above to USRPROPS.TXT (and reopening your encyclopedia) causes the words "Processing Prefix" to be displayed as the label of the “Key Letters” control.

Renaming a Group in a Definition

You can use the LABEL command to rename a Group. If you specify an empty text string (“ “), no words will appear for the Group box.

Example 2:

```
DEFINITION “Attribute” {
GROUP "other stuff" LABEL "" }
```

To rename a CHAPTER, see the CHAPTER command. See also keywords ALIGN and BODY.

LABELPOS

A parameter of the PLACEMENT command that you use to specify exact placement of a property's name (or label) on a DIAGRAM, SYMBOL, or DEFINITION dialog. The LABELPOS command has two arguments – the horizontal position (from the top of the dialog) in Windows units, and the vertical position (from the left of the dialog) in Windows units.

Syntax:

PLACEMENT { **LABELPOS(4, 52)** PROPPOS (horizontal-positioning, vertical-positioning) PROPSIZE (width, height) }

Example:

```
DEFINITION “My Definition”
{
PROPERTY "Table Name" { EDIT Text LENGTH 31 PLACEMENT {
LABELPOS (4, 24) PROPPOS (20, 24) PROPSIZE(150, 12)} }
}
```

Notice in the above example that LABELPOS and PROPPOS have the same y coordinate (24) – this means that the tops of their letters will both be 24 units from the top of the dialog. This means that the label will be to the left of the control (not over it). Notice also that the difference between the PROPPOS x coordinate and the LABELPOS x coordinate (20 - 4 = 16) leaves plenty of room (16 -10 = 6 units) for the 10-character label name, which is “Table Name”, since it has to go to the left of the starting point of the property control's starting position.

Important: See Chapter 2 of this manual for general placement and sizing tips.

See also PLACEMENT, PROPPOS, PROPSIZE, and FORMAT keywords.

LAYOUT

This keyword specifies the layout of properties in a Diagram, Symbol, or Definition dialog. (Note that the Symbol dialog is included as the last tab of a Definition dialog.)

Within the LAYOUT command's opening and closing brackets, you use arguments to specify the layout of all properties called out under that LAYOUT command. You may specify how many columns the properties of the dialog should be laid out into, and how the properties should be aligned.

You may have more than one LAYOUT command specified for a Diagram, Symbol, or Definition dialog. You may specify a LAYOUT command for an entire dialog, and/or override it within each GROUP in a dialog, or within each tab (CHAPTER) in a dialog.

Syntax:

LAYOUT { [alignment_criteria] [PACK_TAB_criteria] [Number of Columns] [JUSTIFY] }

Or more specifically:

LAYOUT {[ALIGN BODY | ALIGN LABEL | ALIGN OVER] [PACK | TAB] [COLS <number>] [JUSTIFY] }

Example:

```
SYMBOL "Object" IN "Sequence"
{
LAYOUT { COLS 2 ALIGN OVER }
PROPERTY "Package" { EDIT OneOf "Package" READONLY }
PROPERTY "Class" { EDIT OneOf "Class" KEYED BY { "Package",
Name } REQUIRED READONLY }
..}
```

In the example above, all properties in the object's symbol dialog are laid out in two columns.

See also ALIGN, BODY, LABEL, OVER, PACK, TAB, COLS, and JUSTIFY keywords.

LEGEND

The string of the displayable property in a rectangular symbol which overrides the property name.

Example:

```
PROPERTY "Description"
  { EDIT ListOf Data
    DISPLAY { FORMAT Key LEGEND "Key data" }
  }
```

The syntax:

LEGEND "<Your Text>": Whatever text you place in the quotation marks will be displayed on the symbol above the entry, only if there is a value for the entry.

LEGEND "": Displays a straight line without any words, only if there is a value for the entry.

LEGEND "$$FORCE$$": Displays a horizontal line above the entry on the symbol. This line acts as a divider. The "$$FORCE$$" keyword is different than simply using " ", in that it forces display of a horizontal line even if the property display is suppressed through the display mode dialog.

LEGEND "$$NONE$$": Does not display a horizontal line above the entry on the symbol, whether or not there are values for the entry. This line normally acts as a divider.

LEGEND "$$VFORCE$$": Enables you lay out properties from left to right inside symbols, and draws vertical lines between them. See VFORCE keyword.

LEGEND "$$VNONE$$": Enables you to lay out properties from left to right, but *does not* provide a dividing line. See VNONE keyword.

See also keyword DISPLAY.

LENGTH

Indicates the number of characters the user may enter in the property field.

Example:

```
PROPERTY "From Entity"
   { EDIT TEXT LENGTH 80 }
```

In the example above, *From Entity* may be 80 characters long.

LINES

Sets the number of lines, in depth, for a property field.

Example:

```
DEFINITION "Constructor"
 { CHAPTER "Desc., Formal Parm"
 GROUP "" {
 LAYOUT { COLS 2 TAB ALIGN OVER }
 PROPERTY "Formal Parameters"   { KEY EDIT Text LENGTH 1020 }
 PROPERTY "Initializer List"    { EDIT Text LENGTH 1000 LINES 4 }
```

This keyword is only useful if you want the space automatically provided to be much bigger or smaller than the default.

See also keyword ZOOMABLE.

LIST

The list keyword has two purposes in USRPPROPS.TXT. The default length is 1200.

Purpose 1: The LIST keyword establishes a list of possible text values. It must be defined in two places – at the top of the USRPROPS.TXT file, wherein you specify the list of possible values, and within the property that is using the list. All List specification statements must be at the top of the USRPROPS.TXT file, before any DIAGRAM, DEFINITION, or SYMBOL specification statements.

Example:

```
List "Method Stereotypes"
{
VALUE "Get"
VALUE "Let"
VALUE "Set"
}
DEFINITION “Method”  {..
PROPERTY "Stereotype"{ EDIT Text LIST "Method Stereotypes"
Default "" LENGTH 30  }   …}
```

Radio Buttons Versus Drop-Down List

Rational System Architect automatically displays a list as a list of radio button choices if the number of values in the LIST statement is four or less. If the number of values is five or more, the list is automatically displayed as a drop-down list box. Users may type in their own value in a drop-down list box. If you wish to have a drop-down list box but only have four or less LIST values, use the LISTONLYCOMBO keyword.

Purpose 2: The LIST keyword is also one of the allowed arguments following FORMAT in the **DISPLAY** command. The LIST keyword causes items to be displayed on the symbol in a list – each whitespace character causes a new line, unless the whitespace falls within double quotes bounds.

Example:

```
DEFINITION "Operational Node"
{PROPERTY "Operational Activities" {EDIT LISTOF "Operational
Activity" LENGTH 2000 DISPLAY {FORMAT LIST Legend "Activities"}
..}
```

See also keywords LISTONLY and LISTONLYCOMBO.

LISTOF

One of the allowed types for a property. Is used with the EDIT keyword to specify that the property references a **list of other definitions**. For example, a Class contains a property called Attributes, which is a list of class attributes. Class attribute is a definition type in of itself, which has its own set of properties. Contrast this to the property of a class called Access Type, which is a list of simple textual choices, such as Public, Private, Protected, etc. (The LIST command is used to define this simple text list; see LIST.) Also contrast with ONEOF, which specifies that a property references exactly one other definition – an example is that a Class contains a property called Package, which specifies the one package the class resides in; Package is a definition in of itself.

LISTOF is used with the EDIT keyword. The syntax is as follows:

```
PROPERTY "Your Property" { EDIT LISTOF "Referenced Definition Type" } LENGTH 1200}
```

The ASGRID keyword is often used with LISTOF – ASGRID presents the list of definitions in a grid; if it is not used, the definitions are listed in a default list structure. LISTOF is sometimes used with the keyword ZOOMABLE and also COMPLETE (described elsewhere in this chapter). For a LISTOF property, the LENGTH keyword by default is set to 1200. LENGTH specifies how many characters the user may enter in the property field – in this case the total number of characters of the names of definitions that can fit into the list.

Example:

```
DEFINITION "Use Case"
{
PROPERTY "Preconditions" { ZOOMABLE EDIT ListOf "Pre/Post Condition" LENGTH 1200  }...}
```

See also keywords ONE OF, EXPRESSIONOF, COMPLETE, and ZOOMABLE.

LISTONLY

Indicates that the values for a property must be taken from the displayed list (created via the LIST keyword at the top of the USRPROPS.TXT file) – the user is ***not allowed*** to type in their own value into the list.

Example:

```
List "Importance"
{
Value "Mandatory"
Value "Strongly Desired"
Value "Should Have"
Value "Icing on the Cake"
Value “Not Important”
}

Definition "Use Case Step"
{
PROPERTY "Importance" {Edit Text ListOnly List "Importance"
Length 20 Default "Should Have" }
..}
```

In the example above, the list is provided in the Use Case Step definition dialog as a drop-down list that you can type in your own entry into. Note that there are five Values in the List statement. If there were four or less, the list in the Use Case Step definition dialog would be provided as a selection of toggle boxes. If you wished to have a drop-down list even though you only had four or less List values, you would use the LISTONLYCOMBO keyword.

See also keyword LIST and LISTONLYCOMBO.

LISTONLYCOMBO

Provides a drop-down list no matter how many LIST values there are. In addition, the user cannot type in their own values to the list.

The LISTONLYCOMBO keyword provides functionality that the LIST command doesn't – when using the LIST command, Rational System Architect automatically displays a list as a list of checkbox choices if the number of values in the LIST statement is four or less. If the number of values is five or more, the list is automatically displayed as a drop-down list box. Users may type in their own value in a drop-down list box. If you wish to have a drop-down list box but only have four or less LIST values, use the LISTONLYCOMBO keyword.

Example:

```
List "Importance"
{
Value "Mandatory"
Value "Strongly Desired"
Value "Should Have"
}

Definition "Use Case Step"
{
PROPERTY "Importance" {EDIT TEXT LISTONLYCOMBO LIST
"Importance" LENGTH 20 DEFAULT "Should Have" }
..}
```

In the example above, the list is provided in the Use Case Step definition dialog as a drop-down list even though there are only three values in the List statement. If you had used the simple LIST statement, then the values would have been shown as toggle boxes since there are less than five values.

See also LIST and LISTONLY keywords.

MAX; MAXIMUM

Indicates the maximum allowed number for a property defined as numeric. A numeric field is one in which you can only place numbers.

Example:

```
PROPERTY Length
  { EDIT numeric LENGTH 2 MINIMUM 1 MAXIMUM 99  }
```

MENU

References the graphic used to represent a symbol on the Draw menu and the Draw toolbar, as compared to the diagram workspace.

Example:

SYMBOL "Satellite"
{ASSIGN To "Wireless Network"
DEPICTIONS { DIAGRAM "C:\Program Files\IBM\pictures\satellite.bmp" }
DEPICTIONS { **MENU** "C:\Program Files\IBM \pictures\satellitetoolbar.bmp" }}

In the example above, the satellitetoolbar.bmp picture is placed on the Draw menu of the “Wireless Network” diagram.

See also DEPICTIONS keyword.

MIN; MINIMUM

Indicates the minimum allowed number for a property defined as numeric. A numeric field is one in which you can only place numbers.

Example:

PROPERTY Length
  { EDIT numeric LENGTH 2 **MINIMUM 1** MAXIMUM 99 }

MINISPEC

In Rational System Architect, a minispec is the statement that expresses the processing logic of a process symbol. Minispecs are written using a formal syntax often referred to as Structured English. The MINISPEC keyword is used with the EDIT keyword.

Example:

```
DEFINITION "Process"
{
PROPERTY "Description"
{ ZOOMABLE EDIT MINISPEC LENGTH 1500 LABEL "Minispec" }
...}
```

Minispec is a statement that expresses the processing logic of a process symbol – how the process transforms input data into output data.

The following is an example of a Minispec statement:

```
If ISBN number brand new,
Create "ISBN MASTER LIST"
Else
Update "Borrower Request"
```

Rational System Architect can balance the input and output flows of a process using the minispec words against the data elements and data structures on data flows. The balancing function requires that the system analyze the text word-by-word, looking for significant words. Significant words are flagged by delimiting them with either single or double quotes. You can choose to have the system consider every word, or only the significant words flagged for consideration.

By default, the system considers only the significant words specifically flagged with quotes, for example:

```
Compute "extended_cost" = "unit_cost" times "quantity"
```

If you want the system to consider every word contained in minispecs, and not only those delimited by double quotes, you must set MinispecUsesQuotes to "N" in the SA2001.INI file. The sample Minispec above could then be written:

```
Compute extended_cost = unit_cost times quantity
```

NAME

Used to indicate that part of the key of a definition is the name of the object itself, and also may be the name of the parent object.

Example:

```
DEFINITION "SQL Server Trigger"
{
PROPERTY "Table Name"
{ EDIT OneOf Definition "Table" RELATE BY "is keyed by" KEYED BY
{"Database Name", "Owner Name","Table Name":Name, Name} }
...}
```

In the example above, the key of the property "Table Name" in the definition of a trigger is the name of the table in which that trigger is defined. The trigger's own name is also part of the key.

NODESC; NODESCRIPTION

This keyword specifies that the definition does not have a property DESCRIPTION.

Syntax:

```
DEFINITION <def_name>
  { NODESC
  }
```

Example:

```
DEFINITION "XML Attribute Type"
{
NODESC
PROPERTY "Data Type" { Edit Text LIST "XML Data Type" Length 100
}
PROPERTY "Required" { Edit Text List "XML yesno" Length 100 }
PROPERTY "Default" { Edit Text Length 1000 }
..}
```

There are a number of definition types in Rational System Architect for which the Description field has been removed through use of the NODESC keyword. They are definition types where a Description is not necessary and would only get in the user’s way. Examples are Trigger Template, Table Synonym, Table, Stored Procedures, and Views.

NONADDR, NONADDRESSABLE

Used to remove definitions from the address list. There are 13 definition types that have been predefined as 'Addressable', meaning that you can 'address' a symbol on a diagram with them (select any symbol and choose Dictionary, Addresses, and then the definition type). Definition types specified as 'Addressable' are generally things like requirements, rules, test plans, etc – things that the symbol on the diagram is 'addressing' or satisfying. To remove any one of those definitions from the Dictionary, Addresses drop-down menu, modify the statement in USRPROPS.TXT.

Example:

```
DEFINITION "Change Request"
  {
   NONADDR
  }
```

See also keyword ADDRESSABLE.

NONE

Actually the $$NONE$$ keyword, used with the DISPLAY keyword. For more information, see the DISPLAY keyword.

NONKEY

One of the allowed arguments following FORMAT in the **Display** command. Elements that are not designated as keys can be displayed in a separate section of the symbol.

Example:

```
DEFINITION "Entity"
{
PROPERTY "Description"
{ EDIT COMPLETE LISTOF "Attribute" FROM "Data Element"
KEYED BY {Model, "Entity Name":"Name", "Name"} RELATE BY
"uses" ASGRID COPYSCRIPT OnCopyEntityDesc EDITCLASS
SACPropertyAttributeGrid Label "Attribute List" LENGTH 4096
ZOOMABLE DISPLAY { FORMAT KEY LEGEND "Primary Key" }
DISPLAY { FORMAT NONKEY LEGEND "Non-Key Attributes" }
...}
```

In the example above, the FORMAT NONKEY command places all attributes not designated as primary keys under the "Non-Key Attributes" legend on an entity symbol.

NOTHING

Used in the RELATE BY NOTHING command.

See RELATE BY.

NUMERIC

This is one of the allowed types for a property. It specifies that the property is a number – only numbers are allowed to be entered into the field (and plus or minus marks). The LENGTH statement determines the amount of numbers that may be entered into the field. The user will not be able to enter decimal points or any characters into the field; only numbers and plus or minus marks.

Example:

```
SYMBOL "Process" IN "Data Flow Gane & Sarson"
  {PROPERTY "Short Description"
    { EDIT Text LENGTH 1500 }
  PROPERTY "Number" { EDIT Numeric LENGTH 4 }
```

OF DEFINITION REFERENCED IN

Enables you to specify a restricted list of definitions that you can choose from when you click on the Choices button for a property. It is used to add further refinement to an EDIT LISTOF or EDIT ONEOF statement. You may specify that only the definitions belonging to a particular referencing definition are listed.

Example 1:

```
DEFINITION "Object"
{
PROPERTY "Package" { KEY EDIT OneOf "Package" RELATE BY "is
   keyed by" READONLY}
PROPERTY "Class" { KEY EDIT OneOf "Class" KEYED BY
   { "Package", Name } RELATE BY "is keyed by" READONLY }
PROPERTY "Attributes" { ZOOMABLE EDIT LISTOF  "Class
   Attribute" OF DEFINITION REFERENCED IN "Class"
  KEYED BY {"Package", "Class Name":"Class", Name} LENGTH
   4096 DISPLAY {FORMAT COMPONENT_SCRIPT
   _FmtNewUMLObjInstAttr LEGEND "$$FORCE$$"}  }
..}
```

In the example above, when you click on the Choices button in an object’s Attribute grid, only the attributes of the object’s containing class are provided. The “Class” property that OF DEFINITION REFERENCED IN is referencing must also be specified in the object’s definition, as is shown above. (As an aside, also in the example above, the attribute itself is specified to be keyed by its package, class name, and its own attribute name via the statement KEYED BY {"Package", "Class Name":"Class", Name.)

Example 2:

```
DEFINITION "Message"
{
PROPERTY "To Class"  { KEY EDIT OneOf "Class" }
...
PROPERTY "Operation"  { EDIT ParmOneOf "Method" OF
DEFINITION REFERENCED IN "To Class"  KEYED BY { "Class
Name" : "To Class", Name, "Formal Parameters"} LENGTH 1000}
..}
```

In the example above, the definition of a message line is refined so that only the methods of the class of the object that

the message line is drawn to are listed. These are methods of the “To Class”. This is possible since the object symbol (object lifeline) that the message line is drawn to contains properties for its referencing class, and the message line contains that property, “To Class”. Note that this is a definition for an OMT Sequence diagram; the UML Sequence diagram has additional keying (by package) than this example. See keyword OF DEFINITION AND SUPERS REFERENCED IN for an example of a UML message line definition.

See also keyword OF DEFINITION AND SUPERS REFERENCED IN.

OF DEFINITION AND SUPERS REFERENCED IN

Enables you to specify a restricted list of definitions that you can choose from when you click on the Choices button for a property – this restricted list references elements of a particular definition **and** elements of any other definition that it inherits from (is attached to via an inheritance line). It is used to add further refinement to an EDIT LISTOF or EDIT ONEOF statement, and is used in UML modeling.

Example:

```
DEFINITION "Message/Stimulus"
{
PROPERTY "To Package" { KEY EDIT OneOf "Package" RELATE
  BY "is keyed by" READONLY}
PROPERTY "To Class" { KEY EDIT OneOf "Class" KEYED BY
  { "Package":"To Package", Name } }
PROPERTY "To Object"{ KEY EDIT OneOf "Object" KEYED BY
  { "Package":"To Package", "Class":"To Class",Name} }
PROPERTY "Operation" { EDIT ParmOneOf "Method"
  OF DEFINITION AND SUPERS REFERENCED IN "To Class"
  KEYED BY { "Package" QUALIFIABLE,"Class Name"
  QUALIFIABLE,"Formal Parameters" QUALIFIABLE ,Name}
  LENGTH 1000 DISPLAY {FORMAT COMPONENT_SCRIPT
  _FmtNewUMLEventOperation LEGEND "$$NONE$$" } LABEL
  "Method" HELP "Choose a method from the proper class" }
```

In the example above, when you click on Choices in the message definition, you get methods of the class that the message line is attached to (the “To Class”, not the “From Class”), and any methods of any class that is a superclass of that class (connected to that class via an inheritance line).

See also keyword OF DEFINITION REFERENCED IN.

ONEOF

One of the allowed types for a property. Is used with the EDIT keyword to specify that the property references **one of** the definitions of another definition type.

Example:

```
SYMBOL "Relation" IN "Entity Relation"
{
PROPERTY "From Entity" { EDIT ONEOF Entity READONLY }
PROPERTY "To Entity"  { EDIT ONEOF Entity READONLY }
}
```

In the example above, the Relation line between two entities contains, on its Symbol tab, the entities that it connects – both the entity that the line is drawn to and the entity that the line is drawn from. In each case, one and only one entity is listed. This information is supplied automatically (Rational System Architect keeps track of from and to information), and therefore the property is made READONLY.

Example 2:

```
Definition "Extends"
{
PROPERTY "Use Case Steps" { ZOOMABLE EDIT ONEOF "Use Case Step" KEYED BY {"Model Name":"Model Name", "Use Case Name":"From Use Case", Name}
}
```

In the example above, the definition behind the Extends line contains a reference to the Use Case Step (in the referencing Use Case) at which the extension (to the other Use Case that the line connects to) takes place. When you click on Choices for this property in the Extends definition, you get a list of all Use Case Steps – however there is only room in the property field to drag in one Use Case Step (contrast to LISTOF which would allow multiple definitions to be dragged in).

See also keywords LISTOF and EXPRESSIONOF.

OVER

An argument of the **ALIGN** command; it places the name of the property (or its label) over the property's control (such as a text field, drop-down list box, etc).

Example:

```
Definition "Use Case Step"
{
Chapter "My Properties"
LAYOUT { COLS 2 ALIGN OVER TAB }
PROPERTY "Importance" {Edit Text ListOnlycombo List "Importance"
Length 20 Default "Should Have" }
PROPERTY "Number" { EDIT Numeric LENGTH 4  LABEL “Ranking”}
}
```

Importance | Ranking
Should Have | +78

In the example above, all properties of the tab (Chapter keyword) are laid out so that the name or label of each property is placed over its control. The Number property is relabeled to be “Ranking”. Its label is placed over its control, which is a simple Numeric field. The Importance property has its name placed over its control, which is a drop-down list.

Contrast this keyword with the BODY keyword, which places the name or label of the property to the left of the control.

See also keywords BODY, TAB, ALIGN, LABEL, and JUSTIFY.

OVERRIDABLE

This keyword enables a read-only property of a definition, inherited from the diagram, to be changed (or written to) when the definition is initially created, despite the fact that it is read-only. OVERRIDABLE is **only** used at the diagram level to specify that a property belonging to a definition representing a symbol drawn on that diagram, can be changed when the symbol is initially created, even though it is read-only.

Example:

```
Diagram "XML"
{
CHAPTER "Diagram"
PROPERTY "XML Schema" { EDIT ONEOF "XML Schema"
AUTOCREATE RELATE BY "is part of" INITIAL USER REQUIRED
OVERRIDABLE READONLY }
..}

DEFINITION "XML Element"
{
..
PROPERTY "XML Schema" { Key Edit ONEOF "XML Schema" Relate
By "is keyed by" Initial User Required Readonly }
..}
```

In the example above, the XML Element definition inherits its value for the “XML Schema” property from the XML Element symbol it is defining, which in turn inherits the value of its “XML Schema” property from the diagram it is placed on. When you initially place the XML Element symbol down on the diagram workspace, you are enabled to change the value of the “XML Schema” property of the XML Element definition. Once you click OK to close the definition, then reopen it, you will notice that the “XML Schema” property is read-only and can no longer be changed.

Note that the keyword is used in the Diagram specification, not the XML Element definition specification.

PACK

Controls vertical positioning within the **LAYOUT** command. This command separates sets of controls and labels in multiple columns from the set of controls/labels located directly to the right by the minimum amount of space.

Example:

GROUP "Power Builder Headings/Labels" {
  LAYOUT { COLS 2 ALIGN OVER PACK }

See also keywords LAYOUT and TAB.

PARENT

In object-oriented and methodological terminology, designates the object from which the current object inherits all key properties. Must include RELATE BY "is keyed by" in the statement.

Example:

```
DEFINITION "Use Case Step"
{
PROPERTY "Use Case Name" { KEY EDIT OneOf "Use Case"
  PARENT RELATE BY "is keyed by" READONLY  HELP "Name of
  Owning Use Case" }
...}
```

PARMONEOF

This keyword is only used in particular cases in SAPROPS.CFG and **should not** be used in USRPROPS.TXT. This keyword specifies that a reference property in Rational System Architect syntax is displayed like a typical UML operation. In other words, a qualified reference property is displayed using parenthesis around the qualified part of the key, rather than double quotes which Rational System Architect normally displays, and the order of the keys is changed so that the name of the referenced object appears first. For example, methods are shown as **meth(int, char)** instead of **“int, char”.meth**.

Example:

DEFINITION "Activity Model"
{ ..
PROPERTY "Operation" { EDIT **PARMONEOF**"Method" OF
DEFINITION REFERENCED IN "Active Class" KEYED BY
{ "Package" QUALIFIABLE, "Class Name":"Active Class", "Formal
Parameters" QUALIFIABLE, Name} LENGTH 1000 DISPLAY {
FORMAT STRING LEGEND "$$NONE$$" } LABEL "Method" HELP
"Specify class and then click choices button" }
}

PARMLISTOF

This keyword is only used in particular cases in SAPROPS.CFG and **should not** be used in USRPROPS.TXT. It is the same as PARMONEOF but is applied to reference list of properties, such as a grid of methods. It specifies that a reference property in Rational System Architect syntax is displayed like a typical UML operation. In other words, a qualified reference property is displayed using parenthesis around the qualified part of the key, rather than double quotes which Rational System Architect normally displays, and the order of the keys is changed so that the name of the referenced object appears first. For example, methods are shown as **meth(int, char)** instead of **"int, char".meth**.

Example:

Definition “Class” {
CHAPTER "Methods"
PROPERTY "Operations" { ZOOMABLE EDIT
COMPLETE_ALLOW_NEW **PARMLISTOF** "Method" KEYED BY {
"Package", "Class Name":Name, "Formal Parameters" QUALIFIABLE,
Name }
LENGTH 1200 ASGRID DISPLAY { FORMAT COMPONENT_SCRIPT
_FmtNewUMLOperation LEGEND "$$FORCE$$"} LABEL "Methods" }

PLACEMENT

This command is used to specify exact placement of properties on a DIAGRAM, SYMBOL, or DEFINITION dialog. The PLACEMENT command has the following parameters:

LABELPOS (x, y) – specifies the starting point of the upper left-hand corner of a property's name (or label). The x specifies the horizontal position (from the left edge of the dialog) and the y specifies the vertical position (from the top edge of the dialog). Both x and y are in Windows units.

PROPPOS (x, y) – specifies the starting point of the upper left-hand corner of a property's control on a dialog. The x specifies the horizontal position (from the left edge of the dialog) and the y specifies the vertical position (from the top edge of the dialog). Both x and y are in Windows units.

PROPSIZE (x, y) – specifies the rectangular size of the control. The x specifies the width and the y specifies the height of the control, in Windows units.

Example:

```
DEFINITION “Class”
{
CHAPTER "Entity Information”
LAYOUT { COLS 2 TAB ALIGN OVER }
PROPERTY "Table Name" { EDIT Text LENGTH 31
   PLACEMENT {PROPPOS (4, 24) PROPSIZE(150, 12)} }
PROPERTY "Naming Prefix" { EDIT Text LENGTH 8 LABEL
   "Column Prefix" HELP "Prefix of column name"
   PLACEMENT {PROPPOS (175, 24) PROPSIZE(40, 12)} }
..
}
```

In the example above, the PLACEMENT command override the LAYOUT command for the tab (CHAPTER). The Table Name property's text box control is positioned on the Class definition dialog (Entity Information tab) at a position starting 4 Windows units from the left edge of the dialog, and 24 Windows units down from the top edge of the dialog. The text box is 150 units wide by 12 units deep.

Important: See Chapter 2 of this manual for general placement and sizing tips.

See also keywords PROPPOS, PROPSIZE, and LABELPOS.

PROPERTY

Begins the argument that establishes a characteristic of a diagram, symbol, or definition. You must follow the PROPERTY keyword with the name of the property, enclosed in quotation marks. You must then specify the characteristics of the property, within either a pair of opening and closing braces, or within the BEGIN and END statements.

Syntax:

PROPERTY “<property_name>”
  { EDIT <edit_type> <property_parameter> **}**

Or

PROPERTY “<property_name>”
  BEGIN EDIT <edit_type> <property_parameter>
  END

PROPPOS, PROPSIZE

A pair of parameters of the PLACEMENT command that you use to specify exact placement of properties on a DIAGRAM, SYMBOL, or DEFINITION dialog. The PROPPOS command has two arguments – the horizontal position (from the top of the dialog) in Windows units, and the vertical position (from the left of the dialog) in Windows units. The PROPSIZE command also has two arguments, x and y, which specify the width and height of the property's control, respectively, in Windows units.

Syntax:

PLACEMENT { **PROPPOS (horizontal-positioning, vertical-positioning) PROPSIZE (width, height)** }

Example:

DEFINITION "My Definition"
{
PROPERTY "Table Name" { EDIT Text LENGTH 31
PLACEMENT {**PROPPOS (4, 24) PROPSIZE(150, 12**)} }
}

The example above places the beginning (upper left edge of its text box) of the Table Name property 4 Windows units from the left edge of the definition dialog, and 24 Windows units down from the top edge of the definition dialog. The text box is also 150 Windows units wide and 12 Windows units long. This statement does not specify anything about the name (or label, which is "Table Name") that goes along with this textbox. Since nothing is mentioned, the label is placed to the left of the text box, by default. You may change the positioning of the label using the FORMAT command or the PLACEMENT {LABELPOS} command.

Important: See Chapter 2 of this manual for general placement and sizing tips.

See also PLACEMENT, LABELPOS, and FORMAT keywords.

PUBLISHER

This keyword enables you to specify whether or not the values of a property are published in the output of **SA Information Publisher**. It has two arguments – **PUBLISHER SHOW** and **PUBLISHER ORDER**.

Syntax:

PROPERTY "Some user property” {
PUBLISHER
{
SHOW (YES|NO)   ‘ default is YES
ORDER nnnn   ‘ default is zero (do not sort)}
}

PUBLISHER ORDER

This argument of the **PUBLISHER** command enables you to specify the order in which the values of the property are shown in the published output of **SA/Publisher**. It is used with the **PUBLISHER SHOW** argument.

Syntax:

PROPERTY "Some user property” { ... **PUBLISHER** {SHOW YES|NO **ORDER** nnnn } ...}

The default is zero (do not sort).

Example:

DEFINITION "Business Requirement"
{
PROPERTY "Benefit" { EDIT Text LENGTH 50 **PUBLISHER {ORDER 2 }** }
PROPERTY "Status" { EDIT Text LENGTH 50 **PUBLISHER {ORDER 1 }** }
PROPERTY "Difficulty" { EDIT Text LENGTH 50 **PUBLISHER {ORDER 3 }** }
PROPERTY "Assigned to" { EDIT Text LENGTH 50 **PUBLISHER {ORDER 4}** }
}

PUBLISHER SHOW

This argument of the **PUBLISHER** command enables you to specify whether or not the values of a property are published in the output of **SA/Publisher**. You may also use the **PUBLISHER ORDER** command with this keyword.

Syntax:

PROPERTY "Some user property" {.....PUBLISHER {SHOW YES|NO} ... }

The default is YES.

Example:

DEFINITION "Business Requirement"
{
PROPERTY "Benefit" { EDIT Text LENGTH 50 **PUBLISHER** {**SHOW NO**} }
}

In the above example, the property Benefit will not show up in a website generated by SA/Publisher, even if this property has been specified to be output by a report.

QUALIFIABLE

QUALIFIABLE is used in a reference property where one or more key components of the referenced object(s) need not be taken from other properties in the referring object, but may be supplied in the property itself. It is used when all key data cannot be stored within properties of a referencing definition, but the name of the referenced definition must be qualified by the key property.

For example, this KEYED BY clause:

KEYED BY {key_component-1: property_name_1, name}

states that the value of key_component_1 should be taken from property_name_1 and so the reference property would contain just the name(s) of the reference object(s). Whereas this KEYED BY clause:

KEYED BY {key_component-1 QUALIFIABLE, name}

states that the value of key_component_1 should be taken from this property – i.e. the one with this KEYED BY clause, and so the reference property could contain the values of both the name(s) and the key_component_1(s) of the reference object(s). Under these conditions, the values of the names are separated from the values of the key-component_1(s) by periods.

Example:

PROPERTY "Operations”

{ ZOOMABLE EDIT ParmListOf "Method”

KEYED BY {"Class Name":Name, "Formal Parameters" **QUALIFIABLE**, Name}

LENGTH 1200

ASGRID

DISPLAY { FORMAT COMPONENT_SCRIPT FmtUMLOperation LEGEND "$$FORCE$$"}

}

READONLY

Designates that a property is readable but not modifiable. READONLY is used in SAPROPS for properties whose value is inserted by the software, but should be visible to the user. It is always used for Initial AuditId, Date and Time, and Update AuditId, Date and Time. Relation lines, constraints, and other lines linking symbols, where the From and To symbols of the line are significant, are always READONLY.

Example:

```
SYMBOL "Link" IN "Booch (94) Object"
REM "defined by Object Link"
{
PROPERTY "From Class"
{ EDIT OneOf "Class" READONLY }
PROPERTY "From Object"
{ EDIT OneOf "Object" KEYED BY {"Class":"From
Class", Name}  READONLY }
PROPERTY "To Class"
{ EDIT OneOf "Class"  READONLY }
PROPERTY "To Object"
{ EDIT OneOf "Object" KEYED BY {"Class":"To
Class", Name} READONLY  }
...}
```

REFPROP

A Property can only be used once in a definition (unless it is surrounded by #ifdef's). If the user wants to use the same property more than once in a definition, they must use the Control and RefProp keywords. For this reason, the Control and RefProp keywords are often used in conjunction with TESTPROC's.

For a Control to be used, there must be an initial reference to the Property that the Control references, at the top of the definition. The REFPROP keyword is used in conjunction with the Control keyword.

Example:

```
Definition "Index"
{
CHAPTER "Modeling Properties"
  { TESTPROC TestPropertyNotValue TESTPROPERTY
"DBMS" TESTSTRING { "ORACLE 8" } }
  PROPERTY "Primary Key"
  {EDIT Boolean LENGTH 1 DEFAULT "F" READONLY }
  PROPERTY Unique
  {EDIT Boolean LENGTH 1 VALUESCRIPT
ProcessIndexUnique DEFAULT "F" }
  PROPERTY Clustered
  {EDIT Boolean LENGTH 1 DEFAULT "F" }

  ...


CHAPTER "Modeling Properties "
  { TESTPROC TestPropertyValue TESTPROPERTY "DBMS"
TESTSTRING { "ORACLE 8" } }
  Control "Primary Key"
  { REFPROP "Primary Key" }
  Control Unique
  { REFPROP "Unique" }
  Control Clustered
  {REFPROP "Clustered"}
  ...
  }
```

See Also CONTROL keyword.

RELATE (BY), RELATED (BY)

The default relationship type for a reference property is “uses”. The RELATE BY keyword is used to override this default with a different relationship (such as “keyed by”) or no relationship (when “RELATE BY nothing” should be coded).

The following relationships may be used with the RELATE BY keyword:

Nothing – no relationship.

Uses – the default. Means that the definition contains definition.

Explained By – Means a symbol is explained by a definition.

Defined By – Means a symbol is defined by a definition.

Is A – Means a definition "is an instance of" a definition (for example, a column is a data element)

Identifies – Means an object identifies another object.

Comprises – Means an object comprises objects (for example, model comprises entities, relationships, etc. Categorization comprises Category Relations).

Originated From – Means an object originated from a definition.

Is Based On – Means an object is based on a definition (usually a data element).

Is Part Of – Means that a definition is part of a definition. This is used with OneOf or ListOf.

Is Keyed by – Means that a definition is identified by a definition.

User-Defined Relationships – There are also 20 user-defined relationship types (USER 1 through USER 20) available if the user creates them via a RENAME command, for example RENAME RELATION “USER 1” to “XXXX”.

Example:

Definition “Use Case Step”
{
PROPERTY "Use Case Name" { KEY EDIT ONEOF "Use Case" KEYED BY {"Package", Name} **RELATE BY** "is keyed by" READONLY HELP "Name of Owning Use Case" }

PROPERTY "Package" { KEY EDIT ONEOF "Package" **RELATE BY** "is keyed by" READONLY}}

In the example above, the first KEY EDIT indicates that the “Use Case Name” property is a key property of the Use Case Step definition. That “Use Case Name” property refers to a Use Case definition – the ONEOF “Use Case” specifies this. You must specify the full key of that Use Case (that the step is being keyed to); in this case you use the KEYED BY command to specify the keys, which are the package that the Use Case is contained in, and the Name of the Use Case itself. Finally, you must specify that the Use Case Step is keyed by this Use Case, which is what the RELATE BY “is keyed by” command does. The RELATE BY clause is added because the default relationship type for a reference property is “uses”. If a different relationship type is wanted (such as “keyed by”), then the default must be overridden.

The second KEY EDIT in the example above specifies that the “Package” property is also a key of the Use Case Step definition – specifically, the Use Case Step is keyed by a package definition. Note, however, that a package definition is not itself keyed by any additional properties besides its own name, therefore the KEYED BY command is not used. The package is related to a Use Case Step by the “is keyed by” relationship.

RELATION

The relation in effect between a property and its referents.

See also keywords RELATE [BY], RELATED [BY].

REM, REMARK

Causes text following this command, and placed within single or double quote marks to be ignored.

Example:

```
GROUP "Connections"
  { LAYOUT { COLS 2 TAB ALIGN OVER }
    PROPERTY "From Entity"
  { KEY EDIT OneOf "Entity" RELATE BY "is keyed by"
       READONLY}
    PROPERTY "To Entity"
  { KEY EDIT OneOf "Entity"  RELATE BY "is keyed by"
       READONLY}
  } REM "End of group Connections"
```

RENAME

Enables references to an object by a name other than that normally used by Rational System Architect.

Example:

```
RENAME SYMBOL "Control Transform"
  IN "Data Flow Ward & Mellor" TO "Process"
RENAME DIAGRAM "Data Flow Ward & Mellor"
  TO "Ward Mellor"
```

RENAME DEFINITION

Enables the use of 150 User-provided definitions. These 150 user-provided definitions are named User 1 through User 150. You use the RENAME DEFINITION command to rename any number of these user-provided definitions to a new name, thus creating in essence new definition types. RENAME DEFINITION statements should be listed near the top of the USRPROPS.TXT file.

Example:

```
Rename Definition "User 10" to "System Requirement "
```

RENAME DIAGRAM

Enables the use of 20 User-provided diagrams. These 20 user-provided diagrams are named User 1 through User 20. You use the RENAME DIAGRAM command to rename any number of these user-provided diagrams to a new name, thus creating in essence new diagram types. RENAME DIAGRAM statements should be listed near the top of the USRPROPS.TXT file.

Example:

Rename Diagram "User 10" to "Requirements Hierarchy"

RENAME SYMBOL

Enables the use of the 150 User symbols. . These 150 user-provided symbols are named User 1 through User 150. You use the RENAME SYMBOL command to rename any number of these user-provided symbols to a new name, thus creating in essence new symbol types. RENAME SYMBOL statements should be listed near the top of the USRPROPS.TXT file.

Example:

Rename Symbol "User 10" to "System Requirement"

REQUIRED

Specifies that a property must be filled in by the user to enable the diagram or definition to be created. The property will automatically appear in the initial **Name** dialog for the diagram or definition.

Example:

This example requires that the user fills in a value for the XML Schema property to create the XML Element Entity definition.

```
DEFINITION "XML Element Entity"
{
PROPERTY "XML Schema" { Key Edit ONEOF "XML Schema"
Relate By "is keyed by" Required
Readonly }
}
```

RETAIN STYLE

This keyword specifies that user-provided metafiles retain their original graphical style and coloring when used in the tool. This keyword is used with the DEPICTIONS clause.

When you use external, user-provided images to represent symbols on a diagram, the default behavior is that you may specify features of these symbols, such as the fill color and line color, as you would with any other symbol in Rational System Architect. If you specify the RETAIN STYLE keyword in the DEPICTIONS clause, the colors of the user-defined symbol remain as they are – unchangeable.

Example:

```
LiST "Node Stereotypes"
{
Value "Client" DEPICTIONS {diagram images\client.wmf menu
images\client.bmp}
Value "Database" DEPICTIONS {diagram images\database.wmf menu
images\database.bmp}
Value "Firewall" DEPICTIONS {diagram RETAIN STYLE
images\firewall.wmf menu images\firewall.bmp}

SYMBOL "Node" in "Deployment"
{
PROPERTY "Stereotype" { INVISIBLE EDIT Text ListOnly List "Node
Stereotypes" DEFAULT "" LENGTH 32}
}

DEFINITION "Node"
{
PROPERTY "Stereotype"
{ EDIT Text LIST "Node Stereotypes" Default "" LENGTH 32 }
```

In the example above, the firewall.wmf can be used to depict a node symbol on a Deployment diagram if the node's stereotype is set to “Firewall”. When drawn on the diagram, the user-provided metafile, firewall.wmf (added by the user to the FILES table of the encyclopedia's database), is drawn with exactly the same colors as it is outside of Rational System Architect, and cannot be changed by Rational System Architect's color tools.

SACPropertyOnOf Base	Used in the command EDITCLASS SACPropertyOneOfBase. **Do not use this keyword combination.** This keyword combination was specially designed for a certain situation in Rational System Architect, inheritance of Data Element properties by an Attribute in an Entity. You will see this keyword combination in SAPROPS.CFG used for this situation. This is the only situation that this keyword combination can be applied to. Use in other situations may cause errors.

SCRIPT

Calls a script written in SA Basic. The script is used for properties that are neither ListOf nor ExpressionOf. A Script takes the value of a property and performs an action, usually to display a particular type of annotation on a symbol on a diagram. The naming convention for the script itself is as follows:

- _fmt (for example, _fmtUMLAttr): The function itself exists in hard code and cannot be modified. Most functions in SAPROPS.CFG are this way. Hard-coding the function is done to make Rational System Architect's overall response faster.
- fmt (for example, fmtUMLAttr): Exists in the fmtscript.bas file within Rational System Architect's main executable directory.

Creating Your Own Functions

You may create your own functions to display items in a particular way on a symbol, or to compute a particular value. Functions that you create ***should not*** be placed in fmtscript.bas, since this file is overwritten for every new installation or update of Rational System Architect. If you create your own functions, you should place them in a **usr_fn.bas** file, which you must create (it is not provided by default) and place in the main Rational System Architect directory (<C>:\Program Files\IBM\Rational\11.3.1\System Architect Suite\System Architect). (The sarules.bas file has a #include to the usr_fn.bas.)

Most functions called in SAPROPS.CFG (which are hard-coded and by convention have an underscore at the start of their name, such as _fmtUMLAttr) have an equivalent function call in the fmtscript.bas file (without the underscore). If you wish to create your own function, you may use the scripts in fmtscript.bas as a guide.

Explanation of Existing Functions:

FmtOMTAbstractClass returns the script **{abstract}** if this property has been set; otherwise returns nothing

FmtBOOClassConstraint returns a set of braces enclosing the name of the constraint, i.e., **{*constraint name*}**, if one has been set; otherwise, returns nothing

SCRIPT (continued)

FmtOMTObjInstClass returns a set of parenthesis enclosing the name of the class, i.e. **(*class name*)**, if one has been set; otherwise, returns nothing

FmtEntryAction returns the script **entry /** and the name of the entry action if one has been set; otherwise, returns nothing

FmtExitAction returns the script **exit /** and the name of the exit action if one has been set; otherwise, returns nothing

FmtOMTTransition returns the following values, within the following punctuation marks, if they are set in a state transition definition: **(*attribute name*) [*condition*] /*action***

Examples:

```
CHAPTER "OMT Object-oriented"
GROUP "OMT Object-oriented"  {
..
PROPERTY "Abstract"
{ EDIT Boolean LENGTH 1 DEFAULT "F" DISPLAY { FORMAT
SCRIPT FmtOMTAbstractClass  LEGEND "$$FORCE$$"} }
...
PROPERTY "Constraints" { EDIT Text LENGTH 500 DISPLAY {
FORMAT SCRIPT FmtBOOClassConstraint  LEGEND "$$NONE$$"}
}
} REM "end of group OMT Object-oriented"
```

See also keywords FORMAT, COLUMN_SCRIPT, COMPONENT_SCRIPT, and fmtxxx.

STRING

The default argument used after the keyword FORMAT in a **Display** command. Use of *string* causes the contents of the dictionary entry to appear on the screen exactly the way it was typed.

Example:

```
SYMBOL "Relation" IN "Shlaer Information Model"
{
PROPERTY "Description"  { EDIT Text LENGTH 100 }
PROPERTY "Reverse Phrase"  { EDIT Text LENGTH 65 DISPLAY {
FORMAT STRING LEGEND "$$NONE$$" } }
..}
```

See also keywords FORMAT and DISPLAY.

SUPERS

See keyword OF DEFINITION AND SUPERS REFERENCED IN.

SYMBOL

This is the first word in the block in which the properties of a symbol, as opposed to a DEFINITION or a DIAGRAM, are listed.

Example:

```
SYMBOL "Entity" IN "Entity Relation"
{
PROPERTY "Description"  { EDIT Text LENGTH 500 }
…
}
```

See also keywords DIAGRAM and DEFINITION.

TAB

This keyword controls vertical positioning in the **LAYOUT** command by separating sets of controls and labels in multiple columns by tabs so that the entries in each row line up directly below the entries in the row above.

Example:

```
GROUP "SQL Server Schema Check Constraint"
{
LAYOUT { TAB ALIGN Over COLS 2 }
PROPERTY "SQL Server Check Constraint Name"
{ EDIT Text LENGTH 30 Label "Constraint Name"}
PROPERTY "SQL Server Check Constraint"
{ EDIT TEXT LENGTH 256 LINES 10 LABEL "Constraint Check"}
} REM "End of Schema Check Constraint group "
```

See also keywords LAYOUT and PACK.

TESTPROC, TESTPROPERTY, TESTSTRING command group

The TESTPROC, TESTPROPERTY, and TESTSTRING command group provides a conditional capability for properties on a per-diagram basis. This command group provides similar functionality to #ifdef's, except that #ifdef's provide a conditional capability based on an encyclopedia-wide level. The TESTPROC command group works off of a **diagram property --** a definition will contain a certain property set if a value is selected for a "test property" within the diagram's properties.

The TESTPROC command group is especially used in logical data models, to specify data modeling property sets depending on the RDBMS chosen.

TESTPROC stands for Test Procedure. There are two values that can follow a TESTPROC keyword: **TestPropertyValue** and **TestPropertyNotValue**. If TESTPROC is followed by TestPropertyValue, it means "test the property and if it is the same as one of the values in the TESTSTRING specified, then apply the properties in this TESTPROC section to the definition in question". If TESTPROC is followed by TestPropertyNotValue, it means "test the property and if it is **Not** the same as one of the values in the TESTSTRING specified, then apply the properties in this TESTPROC section to the definition in question". There is case sensitivity when using **TestPropertyValue** and **TestPropertyNotValue**, the case must be exactly as specified. It will not work if you use all lower case or all upper case.

TESTPROPERTY is the diagram property that will be queried.

TESTSTRING are the values that are queried. You can list one or more values in the string.

Controls and RefProps: A Property can only be used once in a definition (unless it is surrounded by #ifdef's). If the user wants to use the same property more than once in a definition, he or she must use the Control and RefProp keywords. For this reason, the Control and RefProp keywords are often used in conjunction with TESTPROC's. For a Control to be used, there must be an initial reference to the Property that the Control references, at the top of the definition.

TESTPROC, TESTPROPERTY, TESTSTRING command group (continued)

Example:

Definition "Index"
{
CHAPTER "Modeling Properties"
 { **TESTPROC** TestProperty**Not**Value **TESTPROPERTY** "DBMS" **TESTSTRING** { "ORACLE 8" } }
 PROPERTY "Primary Key"
 {EDIT Boolean LENGTH 1 DEFAULT "F" READONLY }
 PROPERTY Unique
 {EDIT Boolean LENGTH 1 VALUESCRIPT ProcessIndexUnique DEFAULT "F" }
 PROPERTY Clustered
 {EDIT Boolean LENGTH 1 DEFAULT "F" }
 ...

CHAPTER "Modeling Properties "
 { **TESTPROC** TestPropertyValue **TESTPROPERTY** "DBMS" **TESTSTRING** { "ORACLE 8" } }
 Control "Primary Key"
 { **REFPROP** "Primary Key" }
 Control Unique
 { **REFPROP** "Unique" }
 Control Clustered
 {**REFPROP** "Clustered"}
 ...
 }

TestPropertyValue

See TESTPROC, TESTPROPERTY, TESTSTRING command group.

TestPropertyNotValue

See TESTPROC, TESTPROPERTY, TESTSTRING command group.

TEXT

This is an allowable field type. The definition defined as text may come from a list, or may be any alphanumeric characters typed by the user.

Example:

```
DEFINITION "Relationship"
{
CHAPTER "Relations and Connections"
GROUP "Relation"
{ LAYOUT { COLS 2 TAB ALIGN OVER }
PROPERTY "Role" { EDIT Text LENGTH 31 }
PROPERTY "Role Prefix"  { EDIT Text LENGTH 31 }
}
```

TIME

This is an allowable field type, indicating the property contains a time stamp in the notation appropriate to the time format defined to Windows. CHECKOUT TIME, FREEZE TIME, INITIAL TIME, and UPDATE TIME each have special meanings.

Example:

```
DIAGRAM "Data Flow Gane & Sarson"
{
PROPERTY "Freeze time"
{ FREEZE  TIME }
..
}
```

Other uses for the TIME might be found in any definition.

Example:

```
DEFINITION "X" {
PROPERTY "Creation Time"
{ EDIT Text INITIAL TIME LENGTH 12 READONLY }
}
```

TO

Used in the **Rename** command to separate the original name of the object from the new name.

Example:

```
RENAME SYMBOL Class IN "Booch Class"
  TO "Booch Class"
```

UPDATE

An allowable field type which indicates that the system automatically updates the field when the property is changed. It is used by default for *Audit ID, Update Date* and *Update Time*.

The UPDATE keyword provides the same information that the LAST CHANGED keyword provides. Both specify the last time that a definition was changed – meaning that someone opened up a definition dialog, made a modification ('dirtied' the definition in some way, such as adding a space or deleting a letter in one of the properties, or removing a letter and then adding the letter back), and then clicked the SAVE button to save the change. If a user opens a definition dialog and does not touch anything, and clicks SAVE, then the definition was not changed (not 'dirtied'), and it is not considered a change. (Note: When a definition is opened by a user it is 'locked' temporarily by that user. If he or she does not make a change and either Saves or Cancels out of that definition, they have not changed the definition and the LAST CHANGED or UPDATE properties will not know about them. However, Rational System Architect internally tracks who last 'locked' a definition. This Last Locked information is not available to you via USRPROPS keywords.)

Starting in Rational System Architect V9, LAST CHANGED AUDITID, LAST CHANGED DATE, and LAST CHANGED TIME are provided by default in the Access Data tab of each diagram or definition dialog. This is hard-coded in the product – in other words, you will not find the LAST CHANGED keyword in each definition in SAPROPS.CFG, nor do you need to add it to USRPROPS.TXT for new diagram or definition types that you create.

Example:

```
DEFINITION "X"
  { PROPERTY "Modified Time"
    { EDIT Text UPDATE TIME LENGTH 12 READONLY }
  }
```

See also INITIAL keyword.

VALUE

This keyword prefaces a value string in a LIST.

Example:

List "Importance"
{
VALUE "Mandatory"
VALUE "Strongly Desired"
VALUE "Should Have"
VALUE "Icing on the Cake"
VALUE "Not Important"
}

Definition "Use Case Step"
{
PROPERTY "Importance" {EDIT TEXT LIST "Importance" LENGTH 20 DEFAULT "Should Have" }
}

In the example above, a new list is created in USRPROPS.TXT (at the top of the file). There are five values assigned to the list. Later in the USRPROPS.TXT, within the definition of a Use Case Step, this list is employed within the property “Importance”. Note that in this type of list, the user can type in their own value in the Importance field.

See keywords LIST, LISTONLY, and LISTONLYCOMBO.

VALUESCRIPT

The VALUESCRIPT calls a function written in SA Basic. VALUESCRIPTS involve enforcing consistency checks of property values in an open dialog. The functions generally compute values set for properties in a dialog, and make necessary changes, in real time, to values of other properties in the dialog, so that certain consistency rules are enforced.

Creating Your Own Function

You may create your own functions to enforce consistency checks of property values in an open dialog. For information on how to create your own function, see the SCRIPT keyword.

Example:

```
DEFINITION "Index"
{
PROPERTY Unique
{ PLACEMENT {PROPPOS(84, 0) PROPSIZE(100, 12)} EDIT Boolean
LENGTH 1 VALUESCRIPT ProcessIndexUnique DEFAULT "F" }
.. }
```

In the example above, the function ProcessIndexUnique is called. It is located in the fmtscript.bas file. This function is only called when Oracle is the DBMS chosen. The function checks to see if the Bitmap property for the Index is toggled on – if it is, the ProcessIndexUnique function toggles the Index property off if it has been set to on. The reason is that in Oracle, an Index cannot be Unique if it has been specified as a Bitmap index.

Example:

```
Definition “Data Element”
{
PROPERTY "SQL Data Type"
{ EDIT text LIST "Standard Data Types" LENGTH 30 VALUESCRIPT
ProcessSQLDataType LABEL "Data Type" "Type" "DT" PLACEMENT
{PROPPOS(4, 26) PROPSIZE(80, 12)} }
..}
```

In the example above, the ProcessSQLDataType checks to see if the Data Element inherits its type from an underlying Data Domain, and if so, automatically fills it in. If it does not inherit its type, and the user leaves the type field empty, then the function automatically fills in Character 10 as the default when the user hits the Enter key or changes fields in the attribute grid.

VFORCE

Enables you draw vertical lines inside symbols, as opposed to horizontal lines, which are the default. VFORCE lays out properties from left to right, and separates them by a vertical line. (Note: The VNONE command does the same thing but does not show the vertical line.)

Syntax:

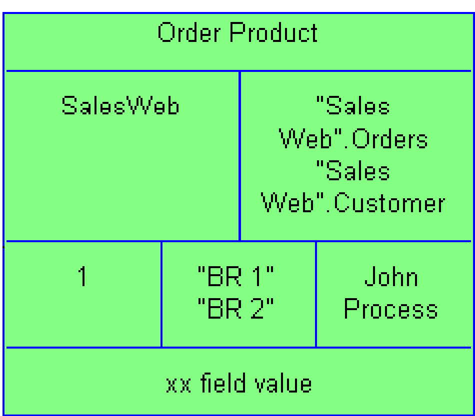
{FORMAT String LEGEND **"$$VFORCE$$"**}

Example:

```
DEFINITION "Elementary Business Process"
{
PROPERTY "Supporting Applications"
{  Edit Listof "Application" Label "Applications" LENGTH 2000 HELP
"Must be entered through Matrix"  READONLY DISPLAY { FORMAT
String LEGEND "$$FORCE$$"} }
PROPERTY "Referenced Data"
{ EDIT ListOf "Entity" KEYED BY {Model QUALIFIABLE,
Name} LENGTH 5000 READONLY DISPLAY { FORMAT String
LEGEND "$$VFORCE$$"}}
..}
```

Notice that the first property listed does not have VFORCE, just FORCE. Subsequent properties that you want to line up to the right of the first property are given the VFORCE specification. In the picture below, the “Sales Web”.Orders and “Sales Web”.Customer values are listed in a box for “Referenced Data”. The VFORCE command was used to make this box appear to the right of the “Supporting Applications” property box, which in the picture has the value SalesWeb listed.

VFORCE (continued)



Note also that the VFORCE command was also used to make the boxes containing “BR 1” and “BR 2” and John Process appear to the right of the box containing the value 1, but this is not shown in the USRPROPS.TXT sample provided.

VISIBLE

If a property is denoted as INVISIBLE in SAPROPS.CFG, using the keyword VISIBLE will make it appear in the definition dialog.

Example (SAPROPS):

```
DEFINITION "Watcom Stored Procedure"
       { CHAPTER "Keys and Parameters"
    PROPERTY "Owner Name"
       { EDIT Text KEY  LENGTH 31 }
    PROPERTY "Procedure Number"
       { INVISIBLE EDIT Numeric LENGTH 9 }
    PROPERTY "Description"
       { EDIT Text  LENGTH 400 }
```

Example (USRPROPS):

```
DEFINITION "Watcom Stored Procedure"
    PROPERTY "Procedure Number"
       { VISIBLE }
```

See also keyword INVISIBLE.

VNONE

Actually the $$VNONE$$ keyword, used with the DISPLAY keyword. For more information, see the DISPLAY keyword.

Enables you draw vertical lines inside symbols, as opposed to horizontal lines, which are the default. VNONE lays out properties from left to right, separating them but not showing any vertical line between them. (Note: The VFORCE command does the same thing but shows the vertical line.)

Syntax:

{FORMAT String LEGEND **"$$VNONE$$"**}

Example:

Notice that in the example USRPROPS.TXT snippet below, the first property listed specifies FORCE. Subsequent properties that you want to line up to the right of the first property (without a dividing line) are given the VNONE specification.

Example:

```
DEFINITION "Elementary Business Process"
{

PROPERTY "Supporting Applications"
{  Edit Listof "Application" Label "Applications" LENGTH 2000 HELP
"Must be entered through Matrix"  READONLY DISPLAY { FORMAT
String LEGEND "$$FORCE$$"} }
PROPERTY "Referenced Data"
{ EDIT ListOf "Entity" KEYED BY {Model QUALIFIABLE,
Name} LENGTH 5000 READONLY DISPLAY { FORMAT String
LEGEND "$$VNONE$$"}}

..}
```

WHERE

Displays only those definitions in the Choices dialog that contain a fixed value in a named property of the definition.

Example:

```
Rename Definition "User 1" To "Aircraft Type"
Rename Definition "User 2" To "Filtered Aircraft"


List "Engine"
{
  Value "Propeller"
  Value "Jet"
  Value "Glider"
}

Definition "Aircraft Type"
{
  Property "Engine Type"
   { EDIT Text List "Engine" Length 48 }
}
Definition "Filtered Aircraft"
{
   Property "Selected Aircraft Type"
        { edit listof "Aircraft Type" WHERE "Engine Type" = "Jet"}
}
```

If the above USRPROPS.TXT were applied to an encyclopedia, and the following Aircraft Type definitions were created in the encyclopedia:

Mustang (engine = Propeller)
Spitfire (engine = Propeller)
F-16 Fighting Falcon (engine = Jet)
F-86 Sabre (engine = Jet)

then upon creating a new definition of type Filtered Aircraft (named Current Jet Fighters, for example), clicking on the **Choices** button for this definition would only reveal two choices -- F-16 Fighting Falcon and F-86 Sabre; all definitions with Engine Type set to Propeller will not appear in the **Choices** list.

ZOOMABLE

Places a button into a list box which enables it to expand to fill the entire dialog page.

Example:

PROPERTY "User Roles"
  {ZOOMABLE EDIT ListOf "User Role with Access Rights"
  LENGTH 1500 LABEL "User Role(s)"}

See also keyword LINES.

4

IBM support

Introduction

There are a number of self-help information resources and tools to help you troubleshoot problems. If there is a problem with your product, you can:

Refer to the release information for your product for known issues, workarounds, and troubleshooting information.

Check if a download or fix is available to resolve your problem.

Search the available knowledge bases to see if the resolution to your problem is already documented.
If you still need help, contact IBM® Software Support and report your problem.

Topics in this chapter	Page
Contacting IBM Rational Software Support	4-2

Contacting IBM Rational Software Support

If you cannot resolve a problem with the self-help resources, contact IBM® Rational® Software Support.

Note: If you are a heritage Telelogic customer, you can find a single reference site for all support resources at http://www.ibm.com/software/rational/support/telelogic/

Prerequisites

To submit a problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage at http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html.
- For further assistance, contact your IBM representative.

To submit a problem online (from the IBM Web site) to IBM Rational Software Support:

- Register as a user on the IBM Rational Software Support Web site. For details about registering, go to http://www.ibm.com/software/support/.
- Be listed as an authorized caller in the service request tool.

Other information

For Rational software product news, events, and other information, visit the IBM Rational Software Web site: http://www.ibm.com/software/rational/.

Submitting problems

To submit a problem to IBM Rational Software Support:

1. Determine the business impact of the problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem.

To determine the severity level, use the following table.

Severity	Description
1	The problem has a critical business impact: you are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.
2	The problem has a significant business impact: the program is usable, but it is severely limited.
3	The problem has some business impact: the program is usable, but less significant features (not critical to operations) are unavailable.
4	The problem has minimal business impact: the problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2. Describe the problem and gather background information. When you describe the problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

- To determine the exact product name and version, use the option applicable to you:
    - Start the IBM Installation Manager and click **File** > **View Installed Packages**. Expand a package group and select a package to see the package name and version number.
    - Start your product, and click **Help** > **About** to see the offering name and version number.

- What is your operating system and version number (including any service packs or patches)?

- Do you have logs, traces, and messages that are related to the problem symptoms?

- Can you recreate the problem? If so, what steps do you perform to recreate the problem?

- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?

3. Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

4. Submit the problem to IBM Rational Software Support in one of the following ways:

    - Online: Go to the IBM Rational Software Support Web site at https://www.ibm.com/software/rational/support/. In the Rational support task navigator, click **Open Service Request**. Select the electronic problem

reporting tool, and open a Problem Management Record (PMR) to describe the problem.

- For more information about opening a service request, go to http://www.ibm.com/software/support/help.html.

- You can also open an online service request by using the IBM Support Assistant. For more information, go to http://www.ibm.com/software/support/isa/faq.html.
- By phone: For the phone number to call in your country or region, visit the IBM directory of worldwide contacts at http://www.ibm.com/planetwide/ and click the name of your country or geographic region.

- Through your IBM Representative: If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. For complete contact information for each country, visit http://www.ibm.com/planetwide/.

5

Appendix:

Introduction

This chapter contains information about the legal uses and trademarks of IBM® Rational® System Architect®.

Topics in this chapter	Page
Notices	5-2
Trademarks	5-5

Notices

© Copyright IBM Corporation 1986, 2010.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or

implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software
IBM Corporation
1 Rogers Street
Cambridge, MA 02142
U.S.A

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been

estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Copyright license

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 2000 2010.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.html

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other company, product or service names mentioned may be trademarks or service marks of others.

Index

B

Begin
  Keyword
    Syntax in USRPROPS.TXT, 2-52

C

Chapter
  Command, 2-87
  Keyword
    Syntax in USRPROPS.TXT, 2-52
Cols
  Keyword
    Syntax in USRPROPS.TXT, 2-91
    Syntax in USRPROPS.TXT, 2-53
  Layout Command, 2-86
Columns
  Keyword, 3-18
CONFIG.PRP, 1-9

D

Default
  Keyword
    Syntax in USRPROPS.TXT, 2-64
Definition
  Defining in USRPROPS.TXT, 2-60
Definition Dialog, 2-76
Diagram
  Defining properties in USRPROPS.TXT, 2-54
Display
  Keyword
    Syntax in USRPROPS.TXT, 2-64

E

Edit
    Keyword
        Syntax in USRPROPS.TXT, 2-63
Error message
    SAPROPS.CFG file, 2-123

G

Group
    Keyword
        Syntax in USRPROPS.TXT, 2-52

H

Hide Definition
    Keyword, 3-46
Hide Diagram
    Keyword, 3-46

I

Indent
    Text
        USRPROPS.TXT, 2-16
Invisible
    Command
        Syntax in USRPROPS.TXT, 2-121

J

Justify
    Keyword
        Syntax in USRPROPS.TXT, 2-91
        Syntax in USRPROPS.TXT, 2-53
    Layout Command, 2-86

K

Key
Option of Display command
USRPROPS.TXT, 2-101

L

Label
Keyword
Syntax in USRPROPS.TXT, 2-63
Layout
Keyword
Syntax in USRPROPS.TXT, 2-53
Legend
Keyword
Syntax in USRPROPS.TXT, 2-64
On display mode divider lines, 2-64
Length
Keyword
Syntax in USRPROPS.TXT, 2-64
List
Keyword
Syntax in USRPROPS.TXT, 2-64
Of Values
USRPROPS.TXT, 2-27
Option of Display command
USRPROPS.TXT, 2-101
Syntax in USRPROPS.TXT, 2-27
ListOnly List
Keyword
Syntax in USRPROPS.TXT, 2-64

M

Maximum
Keyword
Syntax in USRPROPS.TXT, 2-64
Meta-model
definition, 1-3
modifying, 1-3
Minimum
Keyword
Syntax in USRPROPS.TXT, 2-64

N

NonAddr
Keyword, 3-74
NonAddressable
Keyword, 3-74
NonKey
Option of Display command
USRPROPS.TXT, 2-101

P

Pack
Keyword
Syntax in USRPROPS.TXT, 2-53, 2-91
Layout Command, 2-86
Parent
Keyword, 3-81
Property
Arguments
Case sensitivity, 2-7
Dialog, 2-76
Keyword
Case sensitivity, 2-7
Syntax in USRPROPS.TXT, 2-62
Syntax in USRPROPS.TXT, 2-53, 2-63
Referenced in reports, 2-7

R

REM
    Keyword, 3-91
Remark
    Keyword, 3-91
RENAME
    Definition, 2-32
    Keyword, 2-32
Rename Definition
    Command
        Syntax, 2-29, 2-31, 2-32
Rename Diagram
    Command
        Syntax, 2-29
Rename Symbol
    Command
        Syntax, 2-29, 2-30
Renaming Existing Diagram, Symbol, or Definition Types, 2-29
Runtime Edits
    SAPROPS.CFG file, 2-126
    USRPROPS.TXT, 2-126

S

SAPROPS.CFG file
    Error messages, 2-123
    Hiding Standard Entries, 2-121

T

Tab
    Keyword
        Syntax in USRPROPS.TXT, 2-53, 2-91
    Layout Command, 2-86

U

User Definition Type
- User n, 2-32

User n
- Definition type, 2-32

USRPROPS.TXT
- Error message, 2-123
- Hiding Standard Entries, 2-121
- List
  - Syntax, 2-27
- List of Values, 2-27
- Syntax
  - Indenting text, 2-16

Z

Zoomable
- Command, 2-74
  - Syntax in USRPROPS.TXT, 2-74
- Keyword, 3-108