

Software through Pictures® Structured Environment

Release 7

Creating SE Models

UD/UG/ST1000-10116/001



Aonix

Software through Pictures Structured Environment

Creating SE Models

Release 7

December 1999

Aonix reserves the right to make changes in the specifications and other information contained in this publication without prior notice. In case of doubt, the reader should consult Aonix to determine whether any such changes have been made. The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Copyright © 1999 by Aonix.™ All rights reserved.

This publication is protected by Federal Copyright Law, with all rights reserved. Unless you are a licensed user, no part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, by any means, without prior written permission from Aonix. **Licensed users may make copies of this document as needed solely for their internal use—as long as this copyright notice is also reproduced.**

Trademarks

Aonix and the Aonix logo are trademarks of Aonix. ObjectAda is a trademark of Aonix. Software through Pictures is a registered trademark of Aonix. All rights reserved.

HP, HP-UX, and SoftBench are trademarks of Hewlett-Packard Inc. Sun and Solaris are registered trademarks of Sun Microsystems, Inc. SPARC is a registered trademark of SPARC International, Inc. Products bearing SPARC trademarks are based on an architecture developed by Sun Microsystems, Inc. Open Software Foundation, OSF, OSF/Motif, and Motif are trademarks of the Open Software Foundation, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. X Window System is a trademark of X Consortium, Inc. Windows NT is a trademark and Microsoft and Windows are registered trademarks of Microsoft Corporation in the United States and other countries. Adobe, Acrobat, the Acrobat logo, and PostScript are trademarks of Adobe Systems, Inc. Sybase and the Sybase logo are registered trademarks of Sybase, Inc. Adaptive Server, Backup Server, Client-Library, DB-Library, Open Client, PC Net Library, SQL Server, SQL Server Manager, SQL Server Monitor, Sybase Central, SyBooks, System 10, and System 11 are trademarks of Sybase, Inc.



© 1999 Aonix. All rights reserved.

World Headquarters
5040 Shoreham Place
San Diego, CA 92122
Phone: (800) 97-AONIX
Fax: (858) 824-0212
E-mail: info@aonix.com
<http://www.aonix.com>

Table of Contents

Preface

Intended Audience	xvii
Typographical Conventions	xviii
Contacting Aonix.....	xviii
Technical Support	xviii
Website	xix
Reader Comments	xix
Related Reading	xx

Chapter 1 Introduction to StP/SE

What are SE Models?	1-2
Supported Methodologies	1-2
StP/SE Diagram and Table Types	1-4
Data and Control Flow Diagrams	1-4
State Transition Diagrams	1-5
Control Specification Tables	1-5
Data Structure Diagrams	1-5
Structure Charts	1-6
Flow Charts	1-6
Features of StP/SE	1-6
Using the Editors	1-8

Adding Object Annotations to an SE Model.....	1-9
Allocating Requirements to a Model	1-10
Requirements Table Editor Overview	1-10
Entering Requirements Information with the OAE.....	1-11
How StP/SE Stores Information.....	1-12
Diagram, Table, and Annotation Files	1-13
The Repository	1-13
StP/SE Summary	1-16

Chapter 2 Using the StP Desktop

Overview of the StP/SE Desktop	2-1
Starting the StP Desktop	2-3
Starting the StP Desktop from Windows NT.....	2-3
Opening an StP System.....	2-3
Understanding the Model Pane.....	2-3
Using StP/SE Desktop Commands.....	2-5
Starting an Editor.....	2-6
Invoking Other Desktop Commands	2-6
Desktop Commands Specific to StP/SE.....	2-7

Chapter 3 Creating Data Flow and Control Flow Diagrams

Using the Data Flow Editor	3-2
Notation Methods.....	3-3
Starting the Data Flow Editor	3-4
Using the Data Flow Editor Symbols.....	3-5
Navigating to Object References	3-6
Using the DFE Menu.....	3-9
Using Display Marks.....	3-10

Creating a Data Flow and Control Flow Model	3-12
Diagram Names and Process Indexes	3-12
Representing Objects in Data Flow Diagrams.....	3-15
Creating a Context Diagram	3-22
Decomposing Processes.....	3-24
Creating a Process Specification (Pspec)	3-27
Representing Control Information.....	3-32
Defining Data and Control Information.....	3-35
Using Filters to Hide Data or Control Flows	3-37
Reorganizing the Model.....	3-38
Renaming a Hierarchy	3-38
Changing a Process Index	3-40
Collapsing Multiple Processes into One	3-43
Exploding a Process into its Components.....	3-46
Removing Qualifications	3-50
Splitting a Flow	3-52
Merging Flows	3-53
Reversing a Flow	3-55
Validating Data Flow Diagrams	3-56
Syntax Checks	3-56
Semantic Checks	3-57

Chapter 4 Creating Data Structure Diagrams

Using the Data Structure Editor	4-2
Starting the Data Structure Editor.....	4-3
Using the Data Structure Editor Symbols	4-3
Navigating to Object References	4-4
Using the DSE Menu	4-6
Using Display Marks.....	4-6

Creating a Data Structure Diagram.....	4-8
Elements of a Well-Defined Data Structure	4-8
Representing Data Objects	4-10
Creating Root, Intermediate, and Leaf Nodes.....	4-14
Defining an Object in Another Diagram	4-15
Defining Control Information.....	4-15
Setting Properties of Data Objects	4-17
Using the Properties Dialog Box	4-17
Selecting Values from a Fill List.....	4-19
Summary of Object Properties and Dialog Options	4-20
Specifying Scope Information.....	4-21
Assigning Data Types	4-27
Specifying Array Size.....	4-28
Adding C Structure Tags	4-29
Defining a Data Element as a System Type.....	4-30
Defining an Abstract Data Type	4-30
Decomposing an Object	4-33
Navigating to a Parent	4-36
Showing All Children.....	4-36
Generating a BNF File	4-38
Validating a Data Structure Diagram.....	4-40
Syntax Checks	4-40
Semantic Checks	4-41

Chapter 5 Creating State Transition Diagrams

Using the State Transition Editor.....	5-2
Starting the State Transition Editor	5-3
Using the State Transition Editor Symbols	5-4
Navigating to Object References	5-4

Creating a State Transition Diagram	5-7
Representing the Initial State	5-7
Representing States and Transitions	5-8
Representing Events and Actions	5-10
Moving Event/Action Bars	5-12
Annotating the Diagram	5-13
Using Filters to Hide or Show Event/Action Bars	5-13
Creating Associated Cspec Tables	5-14
Validating State Transition Diagrams	5-15
Syntax Checks	5-15
Semantic Checks	5-15

Chapter 6 Creating Control Specifications

Anatomy of a Cspec Table	6-2
Flow within the Cspec	6-2
Overview of Table Elements	6-4
Using the Control Specification Editor	6-8
Starting the Control Specification Editor	6-9
Navigating to Object References	6-9
Using Editor-Specific Menu Commands	6-14
Creating and Editing Cspec Tables	6-15
Creating a Cspec Table by Navigation	6-16
Editing an Existing Table	6-18
Switching Between Cspec Tables	6-21
Table Descriptions	6-22
Action Logic Table (ALT)	6-23
Decision Table (DET)	6-24
Event Logic Table (ELT)	6-25
Process Activation Matrix (PAM)	6-26

Process Activation Table (PAT)	6-27
State Event Matrix (SEM)	6-28
State Transition Table (STT).....	6-29
Validating a Cspec	6-30
Syntax Checks	6-30
Semantic Checks	6-30

Chapter 7 Creating Structure Charts

Using the Structure Chart Editor.....	7-2
Starting the Structure Chart Editor	7-4
Using the Structure Chart Editor Symbols.....	7-4
Navigating to Object References	7-6
Using the SCE Menu	7-8
Using Display Marks.....	7-9
Creating a Structure Chart.....	7-11
Representing Program Modules.....	7-11
Representing Global Data.....	7-12
Representing Library Modules.....	7-13
Drawing Modules and Arcs.....	7-13
Showing the Transfer of Data and Control Information.....	7-15
Using Procedural Symbols	7-18
Using Offpage Connectors	7-20
Setting Properties of Structure Chart Objects	7-22
Using the Properties Dialog Box	7-23
Selecting Values from a Fill List.....	7-24
Summary of Object Properties and Dialog Options	7-25
Changing Scope Values for an Object Reference.....	7-27
Choosing a C Storage Class.....	7-29
Specifying Data and Return Types.....	7-31
Adding a Lexical Include Annotation to a Module.....	7-32

Specifying an Access Mode to Global Data	7-34
Specifying Array Size for Global Data	7-36
Creating a Formal Definition for a Module	7-37
Defining an Object's Data or Return Type	7-40
Generating Module PDL Files	7-41
Creating a Module PDL Annotation.....	7-42
Generating the PDL File	7-44
Using Filters.....	7-47
Validating a Structure Chart.....	7-48
Syntax Checks	7-48
Semantic Checks	7-48

Chapter 8 Creating Flow Charts

Using the Flow Chart Editor	8-2
Starting the Flow Chart Editor	8-3
Using the Flow Chart Editor Symbols.....	8-3
Navigating to Object References	8-5
Creating a Flow Chart.....	8-7
Associating a Flow Chart with a Function.....	8-7
Creating an Entry Point	8-7
Representing Function and Library Calls	8-8
Representing Code Blocks and Macros	8-9
Representing Conditional Statements	8-10
Creating Exit Points.....	8-12
Adding Source Comments	8-15
Validating Flow Charts	8-15

Chapter 9 Checking SE Models

Syntax versus Semantic Checks.....	9-1
------------------------------------	-----

Types of Semantic Checks.....	9-2
Data Flow Editor Checks	9-3
Data Structure Editor Checks.....	9-6
State Transition Editor Checks.....	9-6
Control Specification Editor Checks	9-7
Structure Chart Editor Checks	9-9
Flow Chart Editor Checks	9-9
Using Semantic Checking Commands	9-9
Checking Semantics from the StP Desktop.....	9-11
Checking Semantics from Within an Editor	9-11
Specifying a File and Format for Error Messages	9-12
Checking Semantics Selectively.....	9-12
Navigating to the Source of an Error	9-14
Checking the Model for C Code Generation	9-14
Checking the Semantic Model	9-14

Chapter 10 Generating C Code

What Code is Generated?	10-2
How is C Code Generated?	10-2
Assigning Annotations for Code Generation	10-3
Structure Chart Annotations Used in Code Generation	10-3
Data Structure Annotations Used in Code Generation.....	10-5
SCE Support for Code Generation	10-6
ANSI versus K&R C	10-6
Descriptive Function Headers	10-8
Function Definitions with Formal Parameters	10-9
Variable Argument Functions.....	10-9
Function Pointers.....	10-10
Function Return Types.....	10-11
Function Calls with Actual Parameters.....	10-12

Storage Class for Functions, Globals, and Parameters.....	10-12
Function Code Bodies	10-13
Arrays.....	10-13
Global Data.....	10-13
DSE Support for Code Generation.....	10-14
Data Structure Comments	10-14
Creation of Data Definitions	10-15
System Types.....	10-19
Arrays.....	10-19
Ifndef Statements.....	10-19
Include File Generation.....	10-20
Illegal Variable Names	10-21
Generating Code	10-22
Output Files and Directories.....	10-22
Using the Generate C Code Dialog Box	10-23
Summary of Code Generation Options.....	10-25
Checking the Model for Code Generation	10-26
Updating and Editing Code	10-27
Incremental Updates versus All New Code	10-27
Editing Generated Code	10-28

Chapter 11 Reverse Engineering

What is Generated?.....	11-1
Overview of Reverse Engineering.....	11-2
Using Reverse Engineering.....	11-4
Reverse Engineering Commands.....	11-4
Parsing the Source Files	11-6
Using the Parse Source Code Dialog Box.....	11-6
Parsing the Files.....	11-8
Using the Makefile Reader	11-12

Using the Parser Preprocessor Options Dialog Box	11-15
Parsing C Code Containing Embedded Code	11-19
Common Parsing Problems.....	11-22
Listing the Files in the Semantic Model	11-24
Removing File Contents from the Semantic Model.....	11-25
Semantic Model Locks	11-26
Checking the Semantic Model	11-27
Extracting Comments.....	11-27
Using the Extract Source Code Comments Dialog	11-28
Evaluating and Improving Comment Extraction	11-33
Detection of Inappropriate Language.....	11-33
Generating a Model from Parsed Files	11-34
Using the Generate New Model Dialog Box	11-34
Using the Set Diagram Generation Options Dialog	11-36
Specifying Library Functions	11-39
Setting Control Levels	11-43
Including Unused and System Types	11-43
Maintaining and Updating Your Model.....	11-44
Modifying the Source Code	11-44
Regenerating an Entirely New Model	11-45
Synchronizing the Model with Modified Code.....	11-45
Generated Structure Charts	11-56
Defining Points.....	11-57
Control Levels for Procedural Symbols.....	11-58
Parameters	11-63
Library Modules	11-65
Global Data	11-66
Generated Data Structure Charts	11-66
Nested Structures.....	11-68
Enumeration	11-69

Typedefs	11-70
Generated Flow Charts	11-71
Entry Point.....	11-71
Code Blocks	11-71
Calls	11-72
Decision Constructs.....	11-73
Loops	11-74
Exits and Aborts.....	11-76
Generated Annotations	11-81
Reverse Engineering Directory Structure.....	11-83
Db_dir Directory	11-84
Tree_dir Directory	11-84
Code_dir Directory	11-85
Files File	11-85
SystemIncludes File.....	11-85
UserIncludes File	11-85
Defines File	11-85
Specifiers File.....	11-86
Comment_template File	11-86

Chapter 12 Browsing Models and Code

Browsing SE Models.....	12-2
SE Browser Overview	12-2
Starting the SE Browser	12-5
SE Browser Menus.....	12-5
Executing OMS Queries.....	12-6
Navigating from Query Results	12-10
Browsing from Query Results to Related Objects.....	12-10
Using the SE Menu	12-12

Browsing C Code	12-13
Starting the C Code Browser.....	12-14
Using the C Code Browser	12-14
Examples.....	12-25
Read-Only Locking on Semantic Model	12-37
Interpreting Search Results	12-37
Troubleshooting	12-38
Navigating from Model to Source Code.....	12-38
Setting a Default Editor	12-39
Using the Navigation	12-40

Chapter 13 Renaming Symbols

What Are the Options for Renaming Symbols?	13-1
Choosing a Renaming Method	13-2
How Renaming Affects Objects and Object References.....	13-2
Retyping a Label	13-4
Remapping to an Object without Annotations	13-4
Remapping to an Object with Annotations	13-6
Editing the Symbol or Cell Label.....	13-7
Deleting Unreferenced Objects	13-7
Cloning Objects by Relabeling Symbols.....	13-7
Renaming Objects Systemwide.....	13-8
Objects That Can Be Renamed.....	13-10
Renaming an Object from a Diagram or Table	13-11
Renaming SEFile and SEDirectory Objects.....	13-12
Rename Options.....	13-13
Renaming Errors	13-15
Effects on Dependent Objects	13-15
Automatic Updates.....	13-16
Optional Updates.....	13-17

Chapter 14 StP/SE Reports

Analysis Review Report.....	14-1
Analysis Review Report Example.....	14-2
Design Review Report.....	14-12
Design Review Report Example.....	14-13
C Metrics Reports	14-22
Top Function Report.....	14-23
Complex Function Report	14-25
Include Hierarchy Report.....	14-26
Function Definition Report	14-28
Function Localization Report.....	14-29
Uncommented Objects Report.....	14-30
Generating Reports.....	14-31
Using the Report Generation Dialog Boxes.....	14-31
Summary of Analysis Review Report Options	14-32
Summary of Design Review Report Options	14-34
Summary of C Code Metrics Report Options	14-36

Appendix A Application Types and PDM Types

Persistent Data Model Types.....	A-1
Application Types.....	A-2
Control Specification Editor (CSE) Types.....	A-3
Data Flow Editor (DFE) Types.....	A-5
Data Structure Editor (DSE) Types.....	A-6
Flow Chart Editor (FCE) Types.....	A-7
State Transition Editor (STE) Types.....	A-8
Structure Chart Editor (SCE) Types	A-9

Appendix B StP/SE Symbol Reference

Data Flow Editor Symbols.....	B-2
Data Structure Editor Symbols	B-7
Flow Chart Editor Symbols	B-10
Structure Chart Editor Symbols.....	B-16
State Transition Editor Symbols.....	B-22
Display Marks	B-25
Data Flow Editor.....	B-25
Data Structure Editor	B-27
Structure Chart Editor.....	B-28

Index

Preface

This manual describes Software through Pictures/Structured Environment (StP/SE), a set of interactive graphical and tabular tools for performing structured analysis, structured design, and real-time system specification.

This manual is part of a complete StP documentation set that also includes “Core” manuals: *Fundamentals of StP*, *Customizing StP*, *Query and Reporting System*, *Object Management System*, *StP Administration*, and *StP Guide to Sybase Repositories*. Basic information about the StP user interface is documented in *Fundamentals of StP*.

Intended Audience

The audience for this manual includes analysts and developers who:

- Perform structured analysis
- Create structured designs
- Create specifications for real-time systems

The information in this manual assumes that you are familiar with:

- Fundamentals of StP, including using diagram editors, table editors, and the Object Annotation Editor (OAE)
 - The methodologies supported by the StP/SE editors (see “Supported Methodologies” on page 1-2)
 - Your operating and windowing systems
-

Refer to the appropriate documentation if you have any questions on these topics.

Typographical Conventions

This manual uses the following typographical conventions:

Table 1: Typographical Conventions

Convention	Meaning
Palatino bold	Identifies menus, commands, buttons, dialog boxes and their elements
<code>Courier</code>	Indicates system output and programming code.
<code>Courier bold</code>	Indicates information that must be typed exactly as shown, such as command syntax.
<i>italics</i>	Indicate pathnames, filenames, and ToolInfo variable names.
<angle brackets>	Surround variable information whose exact value you must supply.
[square brackets]	Surround optional information.

Contacting Aonix

You can contact Aonix using the following methods.

Technical Support

If you need to contact Aonix Technical Support, you can do so by using the following email aliases:

Table 2: Technical Support Email Aliases

Country	Email Alias
Canada	info@aonix.com
France	info@aonix.fr
Germany	info@aonix.de
Sweden	info@aonix.se
United Kingdom	info@aonix.co.uk
United States	info@aonix.com

Users in other countries should contact their StP distributor.

Website

You can visit the Aonix website at <http://www.aonix.com>.

Reader Comments

Aonix welcomes your comments about its documentation. If you have any suggestions for improving *Creating SE Models*, you can send email to support@aonix.com.

Related Reading

Creating SE Models is part of a set of Software through Pictures documentation. For more information about StP/SE and related subjects, refer to the sources listed in Table 3.

Table 3: Further Reading

For Information About	Refer To
Using the StP Desktop	<i>Fundamentals of StP</i>
Using StP editors	<i>Fundamentals of StP</i>
StP/SE tutorial	<i>Getting Started with StP/SE</i>
Printing diagrams, tables, and reports	<i>Fundamentals of StP, Query and Reporting System</i>
Internal components of StP and how to customize an StP installation	<i>Customizing StP</i>
Using the Script Manager, writing Query and Reporting Language (QRL) scripts	<i>Query and Reporting System</i>
StP Object Management System (OMS)	<i>Object Management System</i>
Installing StP	<i>Installing StP for Windows NT</i>
Managing an StP installation	<i>StP Administration</i>
Interacting directly with the relational database management system you use as the StP storage manager	Documentation provided with your relational database management system, and <i>StP Guide to Sybase Repositories</i> (for Sybase databases only)
Annotation items	Object Annotation Editor Help menu

1 Introduction to StP/SE

Creating SE Models presents a comprehensive, task-oriented approach to using Software through Pictures/Structured Environment (StP/SE), a client-server, multi-user development environment for:

- Building graphical system models based on structured engineering methods during the analysis and design phases of a software development project
- Automatically generating a graphical model of a system from existing C code, to create design documentation for existing software
- Generating C code from a graphical model
- Synchronizing C code and design models during the development and maintenance phases of a software development project

Additionally, StP/SE allows you to:

- Navigate between the component parts of the model
- Validate SE models for consistency and completeness
- Browse the graphical model and the related C code to locate specific objects in diagrams or sections of code you want to edit
- Generate various design and analysis reports

This chapter provides an introduction to StP/SE and is intended for all users. Topics covered are:

- “What are SE Models?” on page 1-2
 - “StP/SE Diagram and Table Types” on page 1-4
 - “Features of StP/SE” on page 1-6
 - “Using the Editors” on page 1-8
 - “Adding Object Annotations to an SE Model” on page 1-9
-

- “Allocating Requirements to a Model” on page 1-10
- “How StP/SE Stores Information” on page 1-12
- “StP/SE Summary” on page 1-16

What are SE Models?

SE models are graphical and tabular representations of structured engineering system analysis and design elements. These include:

- Data flows—Showing the transformation of data
- Control flows—Adding real-time considerations to the transformation of data
- Processes—Functional components that use or generate data
- Data definitions—Data structure definitions
- Implementation design elements— Including functional relationships between program modules, state transitions, and program flow for each function

Using StP/SE diagram and table editors, you can create several interrelated models that describe the requirements and design aspects of a system. The models help ensure the system's ability to meet customer needs and the integrity of the system design, before any code is written. This reduces the risk of design flaws and the number of potential bugs.

Once your analysis and design models are complete, you can produce the system according to the model, either manually or with the help of the StP/SE-provided C code generation tool.

Supported Methodologies

StP/SE uses functional concepts, notation, and methodologies for SE modeling that are described in Table 1. The Reference column shows the name of the book that describes the corresponding methodology.

Table 1: StP/SE Supported Methodologies

Functional Concept	Diagram/ Table	StP/SE Editor	Method	Reference
Structured analysis	Data flow diagrams	Data Flow Editor	DeMarco/ Yourdon	<i>Modern Structured Analysis</i> by Edward Yourdon (Yourdon Press, Englewood Cliffs, NJ, 1991)
			Gane/ Sarson	<i>Computer-Aided Software Engineering</i> by Chris Gane (Prentice-Hall, Englewood Cliffs, NJ, 1990)
Real-time aspects of structured analysis	Control flow diagrams, state transition diagrams, control specification tables	Data Flow Editor (Cspec) State Transition Editor Control Specification Editor	Hatley/ Pirbhai	<i>Strategies for Real-Time System Specification</i> by Derek J. Hatley and Imtiaz A. Pirbhai (Dorset House, New York, NY, 1988)
Structured analysis, structured design	Data structure diagrams	Data Structure Editor	Based on Jackson	<i>Principles of Program Design</i> by M. A. Jackson (Academic Press, 1975)
Structured design	Structure charts	Structure Chart Editor	Yourdon/ Constantine	<i>Structured Design: Fundamentals of a Discipline of Computer Program and System Design</i> by Edward Yourdon and Larry L. Constantine (Prentice Hall, 1986)
Structured design	Flow charts	Flow Chart Editor		<i>Software Engineering—A Practitioner's Approach</i> by Roger S. Pressman (Fourth Edition, McGraw-Hill, 1996)

Additionally, the StP/SE design editors support scoping capabilities similar to modern programming languages.

StP/SE Diagram and Table Types

You can use the StP/SE editors to create the following graphical and tabular representations of your system:

- Data flow and control flow diagrams
- State transition diagrams
- Control specification tables
- Data structure diagrams
- Structure charts
- Flow charts

This section briefly describes each type of diagram.

You can also use the Object Annotation Editor (OAE) to define StP/SE objects in more detail. For a brief description of the editors, see “Features of StP/SE” on page 1-6; each SE editor is discussed in more detail in its own chapter.

StP/SE diagrams, tables, and annotations are stored in your system’s repository. Through StP/SE, you can access and check this information against the entire model of your software system.

Data and Control Flow Diagrams

Data and control flow diagrams allow you to view the system from a purely functional perspective. They model the purpose and functions of your system without describing the procedural details. Data and control flow diagrams can be leveled, such that each lower-level diagram in a hierarchical collection provides successively more detail about a specific portion of the diagram above it.

You can create data flows, control flows, or both in one diagram. Control flow diagrams model the event-driven aspects of real-time systems, and show the interaction between discrete-valued control signals and processes. Interacting control signals can be modeled with the Control Specification Editor for combinational and sequential machines and with a State Transition Editor for sequential machines.

StP/SE supports both the DeMarco/Yourdon and the Gane/Sarson notations for data flow diagrams and Hatley/Pirbhai for modeling the real-time aspects of structured analysis related to control flows.

State Transition Diagrams

State transition diagrams show the relationships between states, events, and actions. They model the time-dependent behavior of a system by depicting events that cause a transition from one state to another, and the actions performed in connection with the transition. StP/SE supports Hatley/Pirbhai methodology and notation for modeling the real-time aspects of structured analysis.

Control Specification Tables

A control specification (Cspec) is a collection of tables and/or matrices that define the real-time aspects of an analysis model. The real-time aspects of a model are represented by control flows and processes in control flow diagrams and by states, events, and actions in state transition diagrams. A Cspec provides information in tabular form about the real-time objects. StP/SE supports Hatley/Pirbhai methodology for modeling the real-time aspects of structured analysis.

Data Structure Diagrams

Data structure diagrams consist of data symbols linked hierarchically into a tree-like structure. They depict data relationships and define both simple and complex hierarchical data objects in your system. The diagrams specify the types, ranges, and values associated with data and control parameters or data and control flows in your system.

In StP/SE, you use data structure diagrams to define:

- Actual data, represented by data and control flows on data flow diagrams created in the Data Flow Editor
- Data types for data parameters on structure charts created in the Structure Chart Editor
- Return types for modules and data types for global data modules on structure charts created in the Structure Chart Editor

StP/SE supports a notation that is based on the Jackson notation for unambiguous and concise hierarchical data structure modeling.

Structure Charts

Structure charts show the functional interconnections between separate, identifiable program modules. They graphically represent the system hierarchy, as well as the data and control signals passed between modules. Structure charts also allow you to provide a textual description of each program module in the system. StP/SE supports the Yourdon/Constantine notation and methodology for structured design for creating structure charts.

Flow Charts

Flow charts are used in StP/SE to model the programmatic constructs and detailed functionality of your system's program modules. A flow chart diagram generally contains only one flow chart, which is associated with a single function or program module. A flow chart graphically represents an initial state, calls to other functions, code blocks and macros, conditional statements, exit points, and final states.

Features of StP/SE

StP/SE provides a variety of editors and other tools that enable you to develop SE models and generate C code from them. Table 2 summarizes some of the product's major features.

Table 2: StP/SE Features

Feature	Use for
StP Desktop	Opening, copying, and destroying systems, starting all StP/SE editors and utilities, and performing operations on an entire model or selected diagrams and tables (see “Overview of the StP/SE Desktop” on page 2-1).
Data Flow Editor (DFE)	Drawing data flow and control flow diagrams in support of structured analysis methods.
Structure Chart Editor (SCE)	Drawing structure charts depicting identifiable program modules and their functional interrelationships, in support of structured design methodology.
Flow Chart Editor (FCE)	Drawing flow charts that model the programmatic constructs needed for each functional module in your system.
Data Structure Editor (DSE)	Creating data structure diagrams to model data represented in your system model.
State Transition Editor (STE)	Drawing state transition diagrams that show relationships between states, events, and actions, typically in conjunction with Cspec tables and control flows in data flow/control flow diagrams, in support of the real-time extensions to structured analysis.
Requirements Table Editor	Capturing system requirements (described in <i>Fundamentals of StP</i>).
Control Specification Editor (CSE)	Creating and modifying a Cspec table for control flows in a data flow/control flow diagram.
SE Browser	Querying and browsing the repository for SE constructs.

Table 2: StP/SE Features (Continued)

Feature	Use for
C Code Browser	Locating user-specified C constructs in source code that has been reverse engineered to create an SE model.
C Code Generator	Automatically generating and updating C code from a graphical SE model of your system.
Report Generator	Generating C code metrics reports from a semantic model of your system. Generating analysis and design review reports from an SE model of your system.
Reverse Engineering	Automatically generating a graphical SE model from multiple C source files.
Model/Code Synchronizer	Synchronizing design models and C code during software development and maintenance.

Using the Editors

This manual describes how to use the StP/SE diagram and table editors to create structured analysis and structured design models, specifically. It does not provide instructions on how to use the generic features of StP editors. For complete information about standard StP editor features and usage, see *Fundamentals of StP*.

Table 3 provides an overview of some frequently used functions and commands available from the standard StP editor menus.

Table 3: Frequently Used StP Editor Commands

To	Use
Load an existing diagram or table	File > Open
Save changes to an existing diagram or table	File > Save
Create a new diagram or table by saving the current drawing or table entries with a new name	File > Save As
Use the Object Annotation Editor to annotate a diagram or table	Edit > Diagram/Table Annotation Edit > Object Annotation
Print a diagram or table	File > Print
Exit an editor	File > Exit

Menus and commands that are unique to each of the individual StP/SE editors are described in the chapters about each editor.

Adding Object Annotations to an SE Model

You add annotations to an SE object to provide more detailed information about it. An annotation is a group of descriptive notes about an object in the system being modeled. The object can be represented by a symbol in a diagram, the entire diagram, a cell in a table, or an entire table. Each note is of a specific type, and each allows you to add descriptive text and a set of items and values to the object being annotated. The annotations are stored in the repository along with the objects that they describe.

To annotate objects in an SE model, you use the:

- **Properties** dialog box, available for most SE editors
- Object Annotation Editor (OAE)

The types of annotations you can assign are unique to the particular object you are annotating. StP/SE provides predefined annotations for most editors. You can generally add the most common annotations to an object as properties on the editor's **Properties** dialog box. The editor-specific chapters of this manual describe the annotations that can be entered as object properties and explain how to use the **Properties** dialog box.

Several annotations support C code generation. For more information about annotating a diagram for code generation, see Chapter 10, "Generating C Code."

Annotations that do not appear as properties on the **Properties** dialog box are assigned to objects using the Object Annotation Editor. You can also use the Object Annotation Editor to create your own user-defined annotations. The Object Annotation Editor's **Help** menu provides online descriptions of available annotation notes and items. To display the descriptions, select a note or item and choose **On Selection**. For general information about using the Object Annotation Editor, see *Fundamentals of StP*.

Allocating Requirements to a Model

To create, allocate, and track requirements for your project in StP/SE, you can use either:

- Requirements Table Editor (RTE)
- Object Annotation Editor (OAE)

Requirements Table Editor Overview

A requirements table associates project requirements with specific objects in your model. You can create requirements in the Requirements Table Editor and then link them to objects in a diagram in either of two ways:

- By dragging the requirement from the requirements table to an object on a diagram, or alternatively, by dragging the object to the requirement.

- By selecting the object on the diagram and navigating to the requirements table, using the **Allocate Requirements** command.

Table 4 lists the objects to which you can allocate requirements. For a detailed description of the Requirements Table Editor and instructions on creating and allocating requirements, see *Fundamentals of StP*.

Table 4: Requirement Allocations

Editor	Can Allocate Requirements To
Data Flow Editor	Process, Store, Offpage Process, External Store, Offpage External, Data Flow, Control Flow
Data Structure Editor	Sequence, Selection, Enumeration, Typedef
State Transition Editor	State, Event/Action bar
Structure Chart Editor	Program Module, Library Module, Global Data, Parameter,

Entering Requirements Information with the OAE

Alternatively, you can use the Object Annotation Editor to enter requirements information for your model. This information is unrelated to the StP Requirements Table Editor. Table 5 lists the standard annotation notes and items available for StP/SE.

Table 5: Requirement Annotation Notes and Items for StP/SE

Note	Purpose	Item
Requirement	Adds references to requirements documentation to an StP/SE object	Requirement Name
		Requirement Document
		Requirement Paragraph
		Requirement Paragraph Number
		Qualification Method
		Qualification Level
		Test Paragraph
Requirement Allocation	Associates a requirement name and its unique identifier with an StP/SE object	Analysis Requirement Id
		Design Requirement Id
		Implementation Requirement Id
		Test Requirement Id

For more information on using the Object Annotation Editor to enter requirements information, see *Fundamentals of StP*.

How StP/SE Stores Information

As analysis and design proceeds, you describe the functions of the system in a set of diagrams, tables, and their associated annotations. In diagrams, the information is represented as symbols and labels; in tables, it is represented as text or data in cells. Annotations are represented by annotation notes, items, and values in the Object Annotation Editor, and in some cases, as display marks on objects in diagrams.

StP manages elements of an SE model by storing all information associated with the model both as objects in your system's repository and as information in ASCII files.

Diagram, Table, and Annotation Files

A diagram, table, or annotation file is an ASCII representation of information about a model. Each file contains information about a single diagram, table, or set of annotations. A diagram file stores information about the relationships among objects represented by symbols in the diagram. A table file stores information about the table cells in a table. An annotation file stores information about the annotations for a single element in a diagram or table. Every occurrence of a symbol, table cell, or annotation within a system model is recorded in its related diagram, table, or annotation file. Thus, information about a single element can be stored repeatedly in several different ASCII files, or more than once in the same file. For example, if a single data structure appears as a symbol twice in one diagram and once in another diagram, the information about this data structure symbol is recorded three times in the ASCII files—twice in one diagram file and once in the other diagram file.

The Repository

When you save your SE model, StP writes information (application types) from the SE editors to the repository, a central pool of data that StP can access in a consistent manner. Each object in an StP/SE model is stored in the repository only once, regardless of the number of times that object is used in your model's diagrams. Each use of an object in a diagram is a reference to the single object that is stored in the repository. The repository ensures that the information about an object is consistent across the entire model.

The repository stores application types as persistent objects. Each object in the repository is unique. An object contains all the information about a construct in the design domain.

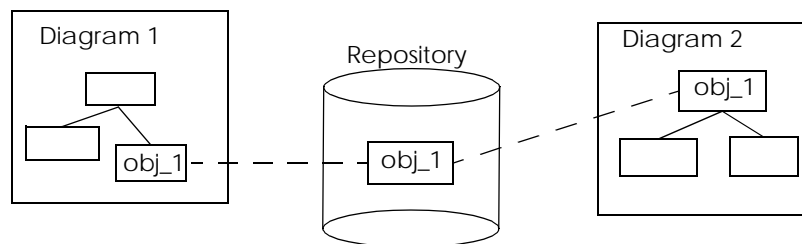
Each application type maps to a persistent object type. The scheme that determines the meaning of objects and their interrelationships is the "Persistent Data Model" (PDM).

For more information about persistent objects, the Persistent Data Model, and the repository schema, see *Object Management System*.

Object Mapping

SE symbols in diagrams and cells in tables are associated with, or “mapped to,” objects in the repository for that particular system. The single object provides the identity for all occurrences (“references”) of the symbol or cell. For example, a data structure that appears twice in the SE model diagrams and files appears only once as an object in the repository, but with multiple references (see Figure 1).

Figure 1: Multiple References to One Repository Object



Consequently, you enter information about an object only once. StP/SE automatically associates the information with each symbol or table cell representing that object. Also, because all related information is stored with the object, you cannot enter conflicting information about anything in the model.

For specific StP/SE-to-repository mappings, see Appendix A, “Application Types and PDM Types.”

Renamed Repository Objects

StP/SE allows you to rename an object, using the **Rename Object System Wide** command on the **Tools** menu. The command changes the name of the object stored in the repository and updates all occurrences of the object, accordingly. This ensures that all references to the object throughout the entire system reflect the name change.

Retyping the label on an object in a diagram does *not* rename the object in the repository. Rather, it creates an entirely new object. For more information about renaming objects, see Chapter 13, “Renaming Symbols.”

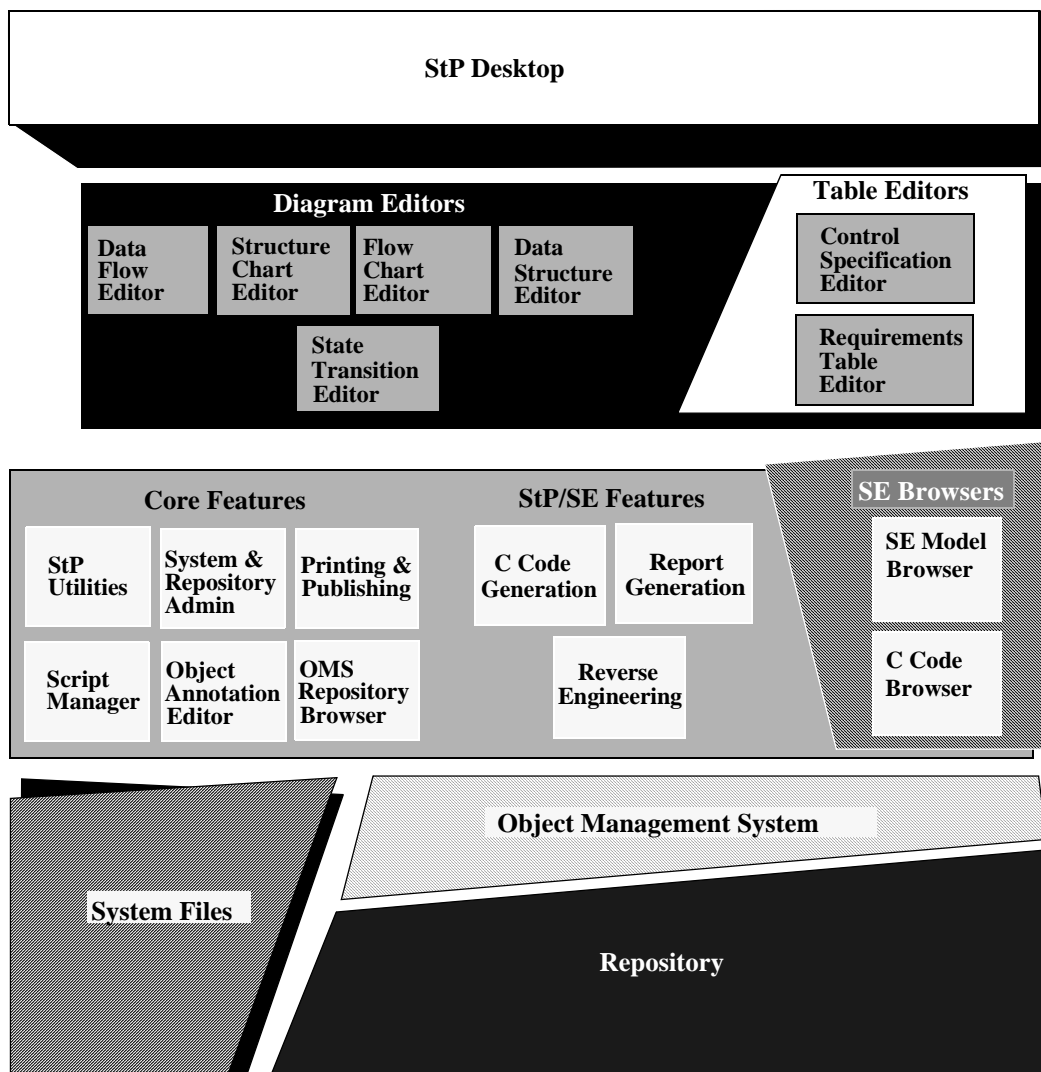
Browsing the Repository

You can search the repository for objects that match user-specified criteria, using either the OMS repository browser or the SE model-specific browser. For instructions on using the browsers, see Chapter 12, “Browsing Models and Code.”

StP/SE Summary

Figure 2 provides a graphical representation of StP/SE.

Figure 2: Summary of StP/SE



2 Using the StP Desktop

This chapter describes:

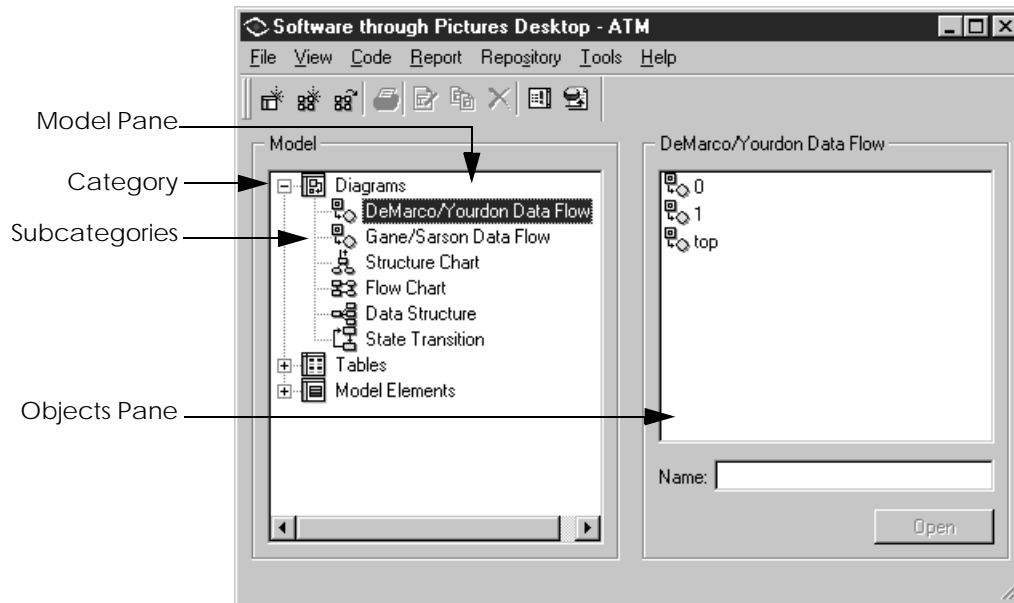
- “Overview of the StP/SE Desktop” on page 2-1
- “Starting the StP Desktop” on page 2-3
- “Understanding the Model Pane” on page 2-3
- “Using StP/SE Desktop Commands” on page 2-5

This chapter focuses on using the StP Desktop with StP/SE. For general information about the StP Desktop, see *Fundamentals of StP*.

Overview of the StP/SE Desktop

The Software through Pictures Desktop (Figure 1) provides access to all StP/SE editors and tools, as well as to all Core commands and utilities.

Figure 1: StP/SE Desktop



Using the StP Desktop, you can perform a variety of tasks, including:

- Opening, copying, or destroying an StP system
- Starting any StP/SE editor
- Generating C code from a model, or a model from C source code
- Browsing generated code for C constructs or a structured model for SE constructs
- Printing a diagram, table, or report
- Checking consistency and completeness of diagrams and tables
- Managing locks for diagrams and tables

Starting the StP Desktop

This section describes how to start the StP Desktop and open an StP system.

Starting the StP Desktop from Windows NT

To start the StP Desktop from Windows NT:

1. Click the **Start** button.
2. Choose **Programs > Aonix Software through Pictures > StP SE**.

Opening an StP System

When you first start the StP Desktop, all items in the Model pane are inactive and appear dimmed until you open an StP system. To open a system, use either of these commands on the **File** menu, as described in *Fundamentals of StP*:

- **Open System**—Opens an existing system of your choice
- **New > System**—Creates a new system with a specified name and database type

You can set the *projdir* and *system* ToolInfo variables in your StP ToolInfo file to automatically open a particular system when you start StP (see *StP Administration* for more information).

Understanding the Model Pane

The Model pane on the StP Desktop (Figure 1 on page 2-2) contains an icon for each category: **Diagrams**, **Tables**, and **Model Elements**. Each category contains subcategories, representing collections of objects (such as diagrams or tables) for the installed product. To open a category and display the subcategories, click the plus sign (+); to close it, click the minus sign (-).

Each subcategory has associated commands on the Desktop menus. When you select a subcategory, the associated commands appear active in the Desktop menus. Also, any objects of that type residing in the current system appear in the objects pane.

The subcategories that appear on the StP Desktop are determined by your installation configuration. Your Desktop might include additional subcategories from other installed StP products. Table 1 provides an overview of StP/SE categories and their subcategories.

Table 1: StP/SE Categories and Subcategories

Category	Subcategory	Description
Diagrams	DeMarco/Yourdon Data Flow	Provides access to, and commands that operate on, data flow diagrams that use DeMarco/Yourdon notation
	Gane/Sarson Data Flow	Provides access to, and commands that operate on, data flow diagrams that use Gane/Sarson notation
	Structure Chart	Provides access to, and commands that operate on, structure chart diagrams
	Flow Chart	Provides access to, and commands that operate on, flow chart diagrams
	Data Structure	Provides access to, and commands that operate on, data structure diagrams
	State Transition	Provides access to, and commands that operate on, state transition diagrams

Table 1: StP/SE Categories and Subcategories (Continued)

Category	Subcategory	Description
Tables	Requirements	Provides access to, and commands that operate on, requirements tables (this is an StP Core editor, described in <i>Fundamentals of StP</i>)
	All Control Spec	Provides access to, and commands that operate on, all control specification tables
	Decision Tables	Provides access to, and commands that operate on, these specific types of control specification tables
	Process Activation	
	Process Activation Matrix	
	State Transition	
	State Event Matrix	
	Event Logic	
	Action Logic	
Model Elements	Structured Model	Provides commands that operate on the whole StP/SE model
	File Objects	Provides commands that operate on file objects in the repository
	Directory Objects	Provides commands that operate on directory objects in the repository

Using StP/SE Desktop Commands

This section describes how to start an StP/SE editor and invoke a Desktop command. It also briefly describes Desktop commands that are specific to StP/SE. For more detailed information about using the StP Desktop, see *Fundamentals of StP*.

Starting an Editor

To start an empty StP/SE editor for creating a new diagram or table, do one of the following:

- Open a Desktop category, select a diagram or table type, and click the **Start New Editor** toolbar button.
- From the Desktop Model pane, open the **Diagrams** or **Tables** category, select a diagram or table type, and choose **New** from the Model pane's shortcut (right-click) menu
- From the **File** menu, choose **New**; from the **New** submenu, choose a diagram or table type.

To open an existing diagram or table in an StP/SE editor:

1. From the Desktop Model pane, open the **Diagrams** or **Tables** category and select a diagram or table type.
2. Double-click a diagram or table in the objects pane.

Alternatively, you can select a diagram or table in the objects pane and do one of the following:

- Click the **Open** button
- Right-click the diagram or table name and choose **Open** from the shortcut menu
- Click the **Edit Diagram/Table** toolbar button
- Choose **Open Diagram** or **Open Table** from the **File** menu

Invoking Other Desktop Commands

To invoke a Desktop command for StP/SE:

1. From the Desktop Model pane, open the **Diagrams**, **Tables**, or **Model Elements** category and select a subcategory.
2. If the command you want to execute requires a specific object, select one or more objects from the objects pane.
3. Select the command from the appropriate menu.

If the command requires no additional user input, it is executed (skip the remaining steps).

4. If a dialog box appears, enter the required information and select options, as desired.
5. Click **OK** or **Apply**.
OK executes the command and dismisses the option dialog box;
Apply executes the command but retains the option dialog box for future use.

Desktop Commands Specific to StP/SE

Of the seven StP Desktop menus (**File**, **View**, **Code**, **Report**, **Repository**, **Tools**, and **Help**), only the **Code**, **Report**, and **Tools** menus contain functions specific to StP/SE. The tables in this section describe these StP/SE-specific commands. See *Fundamentals of StP* for information about Desktop commands common to all StP products.

Code Menu Summary

Table 2 describes **Code** menu commands that are specific to StP/SE.

Table 2: StP/SE Code Menu Commands

Submenu	Command	Description	For Details, See
C	Generate C	Generates C code for each selected structure chart and associated data structure diagram(s), or for the objects scoped to the selected SEFile or SEDirectory object(s)	"Generating Code" on page 10-22
	Generate C for Entire Model	Generates C code for all structure charts and associated data structure diagrams in the SE model	Chapter 10, "Generating C Code"
BNF	Generate BNF	Generates a Backus-Naur format (BNF) file for each selected data structure diagram	"Generating a BNF File" on page 4-38
	Generate BNF for All Diagrams	Generates Backus-Naur format (BNF) files for all data structure diagrams in the entire model	

Table 2: StP/SE Code Menu Commands (Continued)

Submenu	Command	Description	For Details, See
Pspec	Generate Pspec	Creates a process specification for each pspec-annotated process on the selected data flow diagram(s)	“Creating a Process Specification (Pspec)” on page 3-27
	Generate Pspec for All Diagrams	Creates a process specification for all pspec-annotated processes in all data flow diagrams for the entire model	
PDL	Generate PDL	Generates a Program Design Language (PDL) file for each selected structure chart	“Generating Module PDL Files” on page 7-41
	Generate PDL for All Diagrams	Generates Program Design Language (PDL) files for all structure charts in the entire model	
Reverse Engineering	(various)	Provides commands that generate and update (synchronize) an StP/SE models from C source code	Chapter 11, “Reverse Engineering”

Report Menu Summary

Table 3 describes **Report** menu commands that are specific to StP/SE.

Table 3: StP/SE Report Menu Commands

Submenu	Command	Description	For Details, See
Generate	Generate Analysis Review Report for Entire Model	Generates a report with information about data and control flow diagrams	“Analysis Review Report” on page 14-1
	Generate Design Review Report for Entire Model	Generates design and code review reports	“Design Review Report” on page 14-12

Tools Menu Summary

Table 4 describes **Tools** menu commands that are specific to StP/SE.

Table 4: StP/SE Tools Menu Commands

Submenu	Command	Description	For Details, See
Check	Check Semantics for Selected Objects	Validates that the contents of the selected diagram(s) or table(s) are properly defined	“Checking Semantics from the StP Desktop” on page 9-11
	Check Semantics for Whole Model	Validates that all diagrams for the entire model are properly defined	
	Check Semantics Selectively	Allows you to choose which semantic checks to apply to the selected diagrams; then executes the selected checks to validate the diagrams	“Checking Semantics Selectively” on page 9-12
(none)	Edit Annotation	Starts the Object Annotation Editor (OAE), which allows you to annotate the selected SEFile or SEDirectory scoping object	<i>Fundamentals of StP</i>
(none)	Browse Structured Models	Starts the SE Browser, which allows you to search the repository for specified SE constructs	“Browsing SE Models” on page 12-2
(none)	Rename Object System Wide	Enables you to rename an SEFile or SEDirectory object in the repository, which effectively specifies a new directory and/or filename to which C code for its scoped objects will be generated	“Renaming SEFile and SEDirectory Objects” on page 13-12

Table 4: StP/SE Tools Menu Commands (Continued)

Submenu	Command	Description	For Details, See
(none)	Rename Hierarchy	Renames and moves a decomposition hierarchy from under one process to under another; renames selected data flow diagram and all decomposition diagrams; renumbers process indexes in diagrams	“Renaming a Hierarchy” on page 3-38

3

Creating Data Flow and Control Flow Diagrams

In support of structured analysis, StP/SE provides the Data Flow Editor (DFE) for drawing data flow and control flow diagrams to model a system. Data flow and control flow diagrams allow you to view the system you are designing from a “functional” perspective. The top-level diagram, called the context diagram, shows the system’s overall purpose and how it interacts with constructs that are external to the system. Lower-level diagrams show the system partitioned into functional components or processes, using decomposition techniques.

This chapter describes:

- “Using the Data Flow Editor” on page 3-2
- “Creating a Data Flow and Control Flow Model” on page 3-12
- “Using Filters to Hide Data or Control Flows” on page 3-37
- “Reorganizing the Model” on page 3-38
- “Validating Data Flow Diagrams” on page 3-56

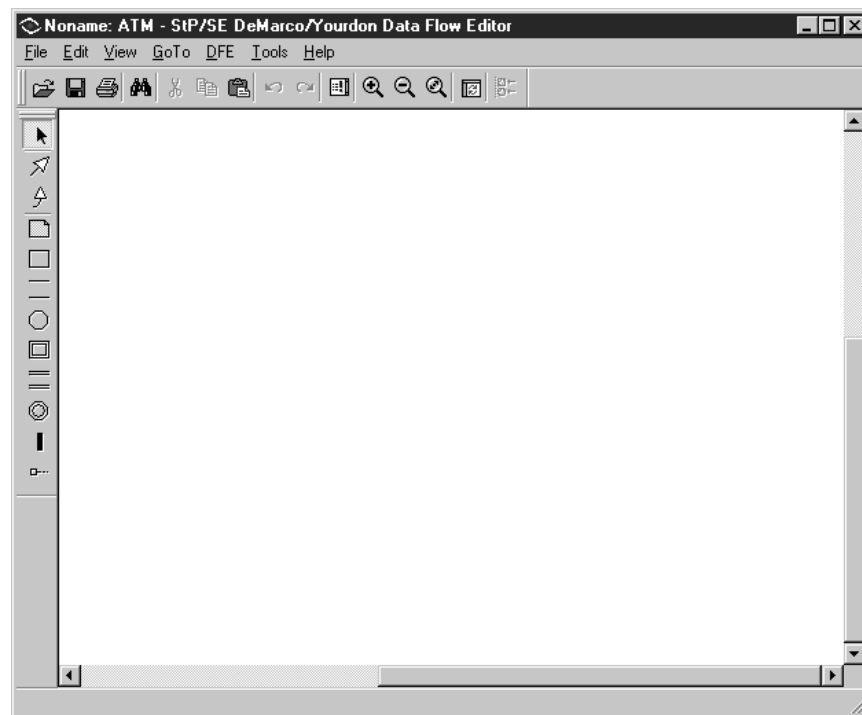
Although this chapter provides a brief explanation to data flow and control flow diagramming, it should not be treated as an introduction to or explanation of structured analysis. For complete details on the notation and methodology of this diagramming technique, refer to:

- *Modern Structured Analysis* by Edward Yourdon (Yourdon Press, Englewood Cliffs, NJ, 1991)
 - *Computer-Aided Software Engineering* by Chris Gane (Prentice-Hall, Englewood Cliffs, NJ, 1990)
 - *Strategies for Real-Time System Specification* by Derek J. Hatley and Imtiaz A. Pirbhai (Dorset House, New York, NY, 1988)
-

Using the Data Flow Editor

The Data Flow Editor (Figure 1), is an interactive graphical tool for drawing data flow and control flow diagrams in support of structured analysis methodology. The example shown here is the DeMarco/Yourdon Data Flow Editor. Alternatively, you can use the Gane/Sarson Data Flow Editor. For more information on these alternatives, see “Notation Methods” on page 3-3.

Figure 1: DeMarco/Yourdon Data Flow Editor



The Data Flow Editor provides the standard StP diagram editor functions and menu options. For general information about using diagram editors, see *Fundamentals of StP*.

In addition to the standard StP diagram editor features, the Data Flow Editor provides:

- Symbols for drawing data flow and control flow diagrams
- Navigations to related diagrams and tables
- The **DFE** menu, providing commands specific to the Data Flow Editor
- Display marks representing information specific to Data Flow Editor objects

These features are described briefly in this section. For more information about using each of these features to create or edit a data flow diagram, see “Creating a Data Flow and Control Flow Model” on page 3-12.

Notation Methods

You can use either the DeMarco/Yourdon or Gane/Sarson notation to create and edit data flow diagrams. There are two corresponding Data Flow Editors, each of which uses its own notation for an entire editor session. Both notations are completely interchangeable in the StP environment. The examples in this manual were created with the DeMarco/Yourdon Data Flow Editor.

By default, navigating from another tool to the Data Flow Editor starts the DeMarco/Yourdon editor. To change the default, set the *dfe_editor* ToolInfo variable in the ToolInfo file to either:

- *gsdfe*, for the Gane/Sarson editor
- *dydfe*, for the DeMarco/Yourdon editor

For more information on ToolInfo variables and files, see *StP Administration*.

Starting the Data Flow Editor

StP/SE provides access to the Data Flow Editor from the:

- StP Desktop
- Control Specification Editor
- State Transition Editor

Starting from the Desktop

To start the Data Flow Editor from the Desktop, choose one of the following tools from the **Diagrams** category in the Model pane, depending on the notation method you want to use in this session:

- DeMarco/Yourdon Data Flow Diagrams
- Gane/Sarson Data Flow Diagrams

For information about starting an editor from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

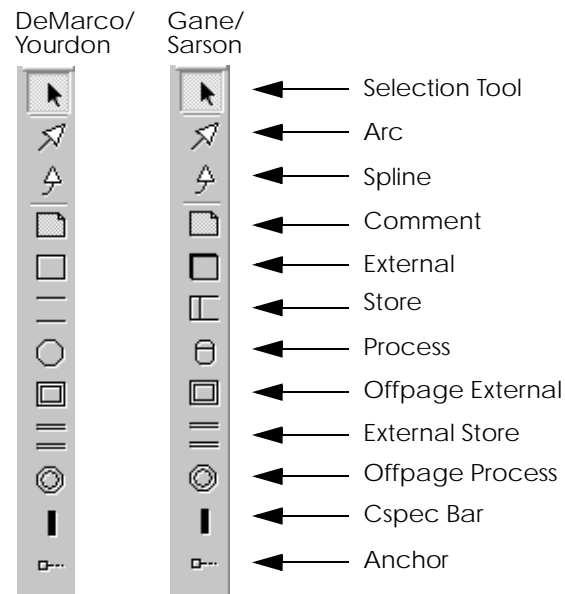
Starting the Data Flow Editor from Another Editor

You can start the Data Flow Editor from the Control Specification Editor or State Transition Editor by using an appropriate navigation command from that editor’s **Go To** menu. If the Data Flow Editor has already been started, the navigation command uses the current session, rather than starting another copy of the editor. For more information about starting the Data Flow Editor from the Control Specification Editor or State Transition Editor, see the chapter describing that editor.

Using the Data Flow Editor Symbols

You select symbols for drawing data flow and control flow diagrams from the Symbols toolbar (Figure 2).

Figure 2: Data Flow Editor Symbols Toolbar



Procedures for using the symbols are described in this chapter. For a summary description of each symbol, see Appendix B, “StP/SE Symbol Reference.” For general instructions on using the Symbols toolbar, see *Fundamentals of StP*.

The Data Flow Editor supports the use of both data flows and control signals in the same diagram. Both are represented by the Arc or Spline symbol. For instructions on setting the default arc type, see “Data and Control Flows” on page 3-19.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object whose symbol is selected on the current diagram. You navigate using the **GoTo** menu. In the Data Flow Editor, the commands on the **GoTo** menu allow you to:

- Display diagrams and tables that are related to the selected object
- Display other levels in a hierarchy of data flow diagrams
- Create new lower-level diagrams

In some cases, navigation starts another editor from a current editor session. When you navigate to another editor, the current editor session continues. You can navigate to these StP editors from the Data Flow Editor:

- Data Structure Editor
- State Transition Editor
- Requirements Table Editor
- Control Specification Editor

The **GoTo** menu provides context-sensitive choices for the selected symbol.

To navigate to a target:

1. Select a symbol on the current diagram.
2. Choose a command from the **GoTo** menu.

The navigation target appears.

If the selected symbol has multiple labels, or if there is more than one possible target, a selection list appears from which you can choose the appropriate target. In some cases, if a navigation target does not exist, you are given the option to create it.

Table 1 describes the navigation targets and commands available for each Data Flow Editor symbol.

Table 1: GoTo Menu Commands

Navigate From	Navigate To	Command
Entire diagram	Higher-level diagram	Parent
Process	Higher-level diagram	Parent
	Lower-level diagram	Decomposition
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements
Store	Higher-level diagram	Parent
	Data definition in a data structure diagram that defines the selected symbol	DSE Data Definition
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements
External Store	Data definition in a data structure diagram that defines the selected symbol	DSE Data Definition
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements

Table 1: GoTo Menu Commands (Continued)

Navigate From	Navigate To	Command
Cspec bar	Higher-level diagram	Parent
	State transition diagram for the selected Cspec bar	State Transition Diagram
	State transition table for the selected Cspec bar	State Transition Table
	Decision table for the selected Cspec bar	Decision Table
	Process activation table for the selected Cspec bar	Process Activation Table
	Process activation matrix for the selected Cspec bar	Process Activation Matrix
Cspec bar	State event matrix for the selected Cspec bar	State Event Matrix
	Event logic table for the selected Cspec bar	Event Logic Table
	Action logic table for the selected Cspec bar	Action Logic Table
Data Flow	Higher-level diagram	Parent
	Data definition in a data structure diagram that defines the selected flow	DSE Data Definition
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements
Control Flow	Higher-level diagram	Parent
	Data definition in a data structure diagram that defines the selected symbol	DSE Data Definition
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements

Table 1: GoTo Menu Commands (Continued)

Navigate From	Navigate To	Command
Offpage Process	Higher-level diagram	Parent
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements
Offpage External	Higher-level diagram	Parent
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements

Using the DFE Menu

In addition to the standard diagram menus described in *Fundamentals of StP*, the Data Flow Editor provides the **DFE** menu. This menu allows you to perform a variety of functions that modify data flow diagrams. Table 2 describes the commands available from the **DFE** menu.

Table 2: DFE Menu Commands

Command	Description	For Details, See
Edit PSpec Note	Allows you to create or edit a Pspec annotation note for a process in order to generate a process specification for it.	“Creating a Pspec Annotation” on page 3-28
Generate Pspec	Creates a process specification (a type of mini-specification) that describes the function performed by a process as well as the input and output data flows and control flows associated with the process. If a process is not selected, Pspecs are generated for all processes on the diagram.	“Generating the Pspec” on page 3-29

Table 2: DFE Menu Commands (Continued)

Command	Description	For Details, See
View Generated PSpec	Displays the generated Pspec for a selected process in a View Text window.	“Viewing a Pspec” on page 3-31
Change Process Index	Changes the index number assigned to a process.	“Changing a Process Index” on page 3-40
Collapse	Combines multiple processes into a single process and adjusts their hierarchy appropriately.	“Collapsing Multiple Processes into One” on page 3-43
Explode	Splits a single process that has a decomposition (or was created by Collapse) into its constituent processes and re-levels the decomposition.	“Exploding a Process into its Components” on page 3-46
Remove All Qualifications in Diagram	Removes obsolete qualifications from all objects in the diagram that have them.	“Removing Qualifications” on page 3-50
Split Flow	Alters a data or control flow so that a single source flows to multiple targets.	“Splitting a Flow” on page 3-52
Merge Flow	Alters a data or control flow so that a multiple sources flow to a single target.	“Merging Flows” on page 3-53
Reverse Flow	Changes the direction of a data or control flow (changes source to target and target to source).	“Reversing a Flow” on page 3-55

Using Display Marks

A display mark is a symbol or string that appears on or near an object and conveys additional information about that object. Several data flow diagram annotations cause display marks to appear in the diagram.

You can control the behavior of these display marks using the **Display Marks** tab of the **Options** dialog box. To display this dialog box, choose **Options** on the **Tools** menu. For details about using the dialog box, see *Fundamentals of StP*.

Table 3 describes the display marks that can appear on a data flow diagram. For more information and an example of each mark, see the section referenced in the table.

Table 3: Display Marks

Display Mark For	Name	Description	For Details, See
Process index	Pindex	Indicates the decomposition level of the process in terms of the parent process. The process index appears in decimal format (x.y), where x is the parent process index and also the current diagram level (diagram's name) and y is an integer that is incremented for each process added to the current diagram.	"Diagram Names and Process Indexes" on page 3-12
Existence of a decomposition for a process		Indicates the existence of a lower-level diagram that defines this process in more detail.	"Decomposing Processes" on page 3-24
Existence of a Pspec annotation for a process		Indicates the existence of a process specification annotation that describes this process.	"Creating a Process Specification (Pspec)" on page 3-27

Table 3: Display Marks (Continued)

Display Mark For	Name	Description	For Details, See
Flow type	FlowType	Indicates a discrete flow (default; uses single arrowhead) or continuous flow (uses double arrowhead).	“Data and Control Flows” on page 3-19
Duplicate externals	DuplicateExternal	(Gane/Sarson only) Indicates that you have used this external more than once.	
Duplicate stores	DuplicateStore	(Gane/Sarson only) Indicates that you have used this store more than once.	

Creating a Data Flow and Control Flow Model

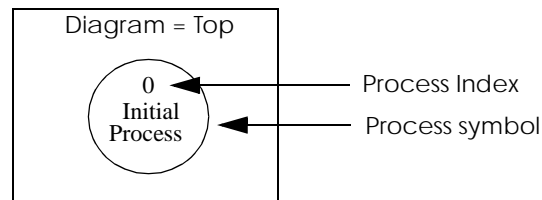
This section describes how to create a data flow and control flow model, using examples based on an automated teller machine (ATM) system.

To create the data flow/control flow model, you create a hierarchy of data flow diagrams, with a context diagram as the top-level diagram. The context diagram, shows the system’s overall purpose and how it interacts with external constructs. Lower-level diagrams show how the system partitions functionality into functional components or processes. The Data Flow Editor allows you to specify the processes to be analyzed either by decomposition into a hierarchy of diagrams and tables or by writing a process specification (Pspec).

Diagram Names and Process Indexes

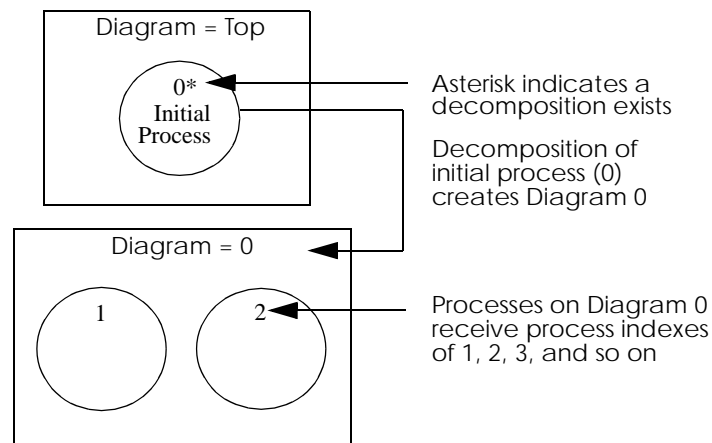
The Data Flow Editor uses a decimal system for naming diagrams and indexing processes to keep track of the diagram and process hierarchy. The context (top-level) diagram must be named *top* to enable the Data Flow Editor to assign the correct index to the initial process. The initial process is given a process index of 0 on the diagram named *top*. The process index appears as a display mark on the process symbol, as shown in Figure 3.

Figure 3: Initial Process on the Top Diagram



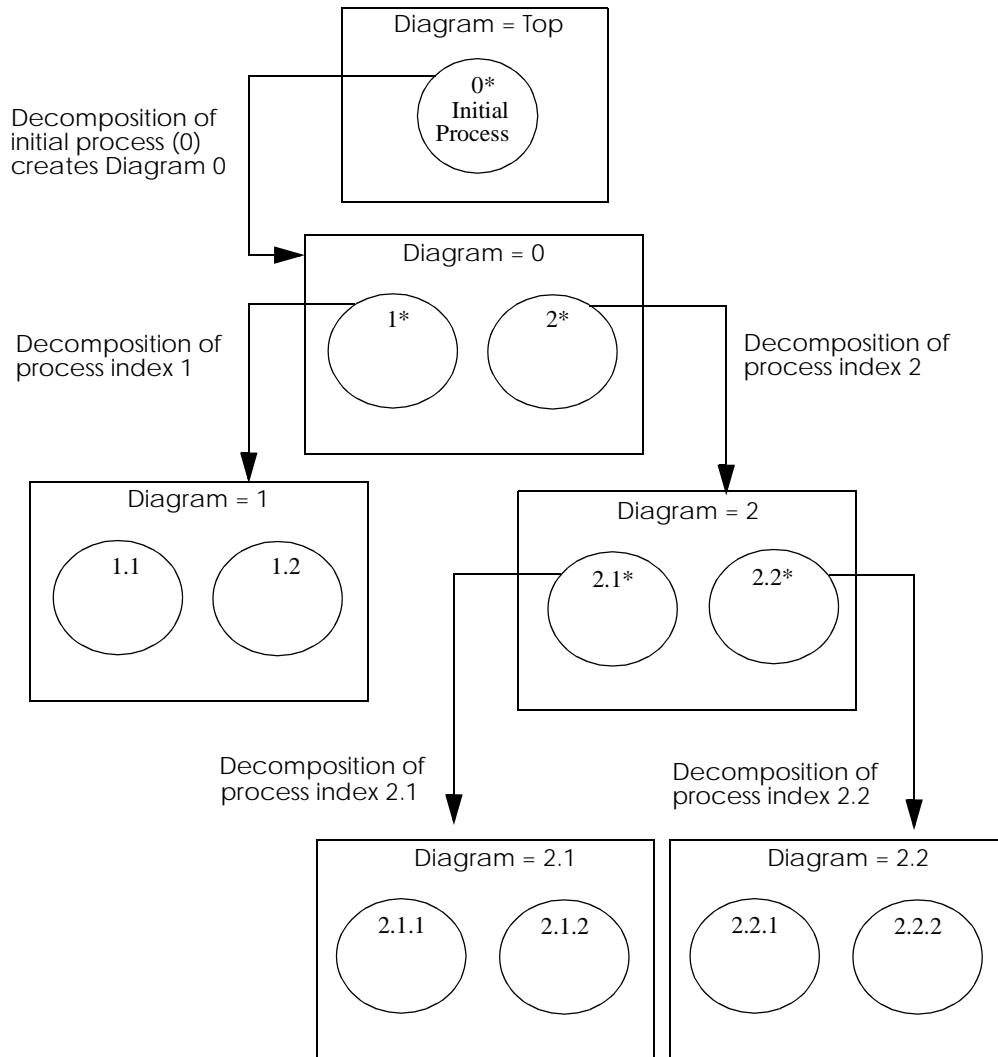
Decomposing a process on a diagram creates a decomposition diagram whose name is identical to the process index of the parent process, as shown in Figure 4. The first process on Diagram 0 is assigned an index of 1. The Data Flow Editor automatically increments the process number for each new process you add to diagram 0, resulting in indexes of 2, 3, and so on.

Figure 4: Decomposing the Initial Process



Decomposing processes 1, 2, 3, and so forth results in diagrams named 1, 2, 3, and so forth. Processes on these decomposition diagrams automatically receive process indexes of $x.1$ through $x.y$, where x is the diagram name and y is a digit that is incremented for each new process added to the diagram, as shown in Figure 5.

Figure 5: Diagram and Process Hierarchy



By default, StP/SE displays full process indexes, which include the name of the diagram as the first element of the process index display mark. If you prefer to use relative process indexes, which do not include the

diagram name, you can add the *relative_pindex* ToolInfo variable to the ToolInfo file and set it to True. For more information on ToolInfo variables and files, see *StP Administration*.

Representing Objects in Data Flow Diagrams

To create data flow diagrams, you select symbols from the Symbols toolbar and insert them in the drawing area. This section describes each of the symbols you use to create data flow diagrams.

For information about drawing the control aspects of your system, see “Representing Control Information” on page 3-32.

Externals

An external is any construct outside of the system being modeled and can represent a physical device, a process, or a data source or sink. It is symbolized by a plain box in DeMarco/Yourdon notation, as shown in Figure 6, and a shadowed box in Gane/Sarson notation. The system accepts input from externals and sends output to externals but is not concerned with their internal structure or operation.

Figure 6: External Symbol



You cannot connect an external directly to another external with a data flow. How externals communicate with each other and what they pass between them is outside the scope of the system you are modeling. The system is not concerned with externals except as the source of data used by the system or the destination of data generated by the system.

Offpage Externals

Offpage externals appear only on decomposition diagrams and never on the context diagram. An offpage external symbol in a decomposition diagram represents an external to which the parent process is connected in the parent diagram. It appears automatically in the decomposition diagram with a label identical to the external it represents.

This symbol is initially smaller than an external symbol and is bordered with a double line. Figure 7 shows the offpage external symbol used for both the DeMarco/Yourdon and Gane/Sarson.

Figure 7: Offpage External Symbol

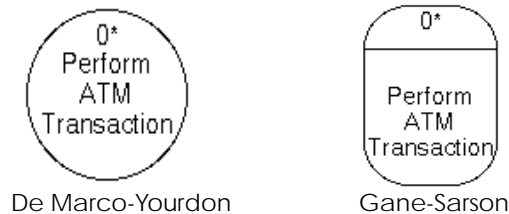


Alternatively, you can instruct the Data Flow Editor to use anchors instead of offpage externals, by changing the default setting of the *use_offpages* ToolInfo variable (see “Anchors” on page 3-20 for details).

Processes

A process symbol represents an operation that uses or generates data. A process is symbolized by a circle in DeMarco/Yourdon notation, as shown in Figure 8, and by a rectangle with rounded corners in Gane/Sarson notation. Throughout the diagram hierarchy, process symbols are automatically indexed (numbered), based on the indexes of their parent processes. The process index appears as a display mark above the label on a process symbol.

Figure 8: Process Symbol



The status of a process is represented by the presence or absence of a status marker that appears as a display mark (* or p) beside the process index number. An asterisk (*) indicates that this process is further defined in a decomposition diagram; a small p indicates that the process has no decompositions and is defined in a process specification (Pspec). An undefined process is one that has neither a decomposition diagram nor a Pspec. If there is no status marker, the process is undefined.

Offpage Processes

Offpage processes appear only on decomposition diagrams and never on the context diagram. An offpage process symbol in a decomposition diagram represents a process to which the parent process is connected in the parent diagram. It appears automatically in the decomposition diagram with a label identical to the parent process it represents.

This symbol, shown in Figure 9, is initially smaller than a process symbol and is bordered with a double line.

Figure 9: Offpage Process Symbol



Alternatively, you can instruct the Data Flow Editor to use anchors instead of offpage processes, by changing the default setting of the *use_offpages* ToolInfo variable (see “Anchors” on page 3-20 for details).

Data Stores

A store symbol, shown in Figure 10, represents a place in which data is kept until it is used by a process. It is symbolized by two parallel lines in DeMarco/Yourdon notation and by a long narrow box open at one end in Gane/Sarson notation. The data in a data store must be defined by using the Data Structure Editor. For details, see “Defining Data and Control Information” on page 3-35.

Figure 10: Store Symbol



External Stores

StP/SE allows you to show a data store that is external to the system on a top diagram (Figure 11). There can be flows into and out of an external store; labels are not mandatory on such flows. These flows can connect to other external symbols.

Figure 11: External Store



An external store does not participate in balance checking unless data passes into it through an arc.

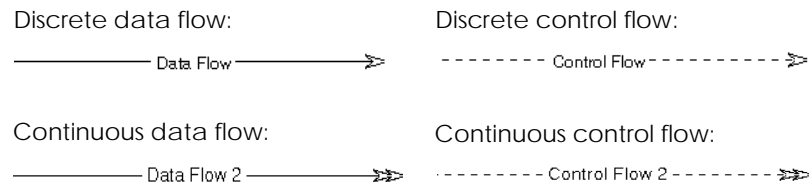
An external store decomposes to an anchor. If an arc flowing into or out of the store is unlabeled on the top diagram, the decomposed anchor is labeled with the name of the store.

Data and Control Flows

Data flows represent the movement of data between processes or between processes and externals or data stores in your system. Control flows represent the movement of control information between processes, externals, and Cspec bars.

A flow appears as a line with a single or double arrowhead at one end that points to the destination symbol, as shown in Figure 12. Data flows have solid lines; control flows have broken lines.

Figure 12: Data and Control Flows



You use the Arc or Spline symbol to draw both data flows and control flows. In some cases, the arc type is context sensitive; for example, it automatically defaults to the ControlFlow arc type for arcs drawn to or from a Cspec bar. For an existing arc, you can toggle the arc type between DataFlow and ControlFlow by selecting the arc and choosing **Replace** on the **Edit** menu.

To change the default arc type:

1. From the **Tools** menu, choose **Options**.
2. On the **Options** dialog box, select the **Default Arc** tab.
3. In the **Default Arc Type** field, select **Data Flow** or **Control Flow**.

You can also designate the flow type's dynamic characteristics as continuous or discrete (default) using the Object Annotation Editor. A continuous flow is one whose value is available at all points in time. A discrete flow is one whose value is available only at certain points in time and is otherwise undefined. The Flow Type is indicated by a display mark that appears as a single or double arrowhead on the arc. The data flow arc with a single arrowhead on the Symbols toolbar represents a discrete

Creating Data Flow and Control Flow Diagrams

flow. To designate a continuous flow, annotate the flow with a Flow Type note and a Flow Type item with a value of Continuous. The flow is redisplayed with a double arrowhead.

Each flow must be labeled with the name(s) of the data or control information flowing along it, with the exception of data flowing to or from a store as discussed in the next paragraph. You must define the data or control information that flows on the arc, using the Data Structure Editor (see “Defining Data and Control Information” on page 3-35).

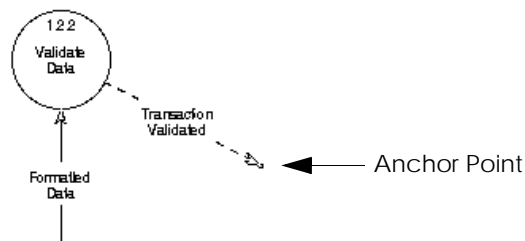
When data flows into or out of a store, the Data Flow Editor assumes that the flow carries whatever data structures are kept in the store, and implicitly gives the flow the same name as the store. Labeling the store overrides this assumption. (This applies only to data, as control information does not flow into or out of data stores.) If a data flow originates in a data store, you can label the flow with a data item only if the data item has been defined in the Data Structure Editor as part of the data store. The checking programs that apply to data flow diagrams do not allow data to flow out of a data store unless it has been explicitly represented in the diagram as having flowed into that data store.

For more information on control flows, see “Representing Control Information” on page 3-32.

Anchors

The anchor is an alternative representation of an offpage external or offpage process. An anchor point is symbolized by a small dot, as shown in Figure 13, and cannot be labeled.

Figure 13: Anchor Point



Although the anchor is the traditional methodological symbol for offpage references, it can be difficult to see; therefore, in StP/SE the default offpage references are offpage externals and offpage processes.

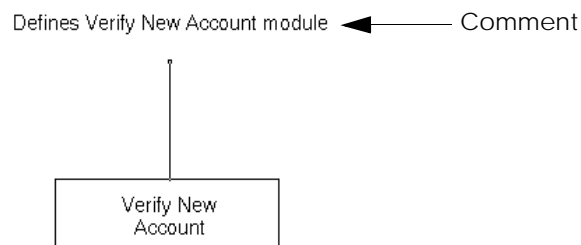
The *use_offpages* ToolInfo variable specifies whether or not the Data Flow Editor uses offpage externals and offpage processes. The default setting is **True**. To instruct the Data Flow Editor to use anchor points instead, change the default setting to False.

The *use_offpages* ToolInfo variable is not included in the default ToolInfo file, but you can add it to the file if you want to change its value. For more information on ToolInfo variables and files, see *StP Administration*.

Comment

The comment symbol, shown in Figure 14, allows you to insert comments of any length anywhere in the diagram.

Figure 14: Comment Example



You use the Arc or Spline symbol to draw links connecting a comment to the object to which the comment applies.

Creating a Context Diagram

The context diagram is the highest-level data flow diagram for your system. It contains the initial process symbol and shows all the external interfaces between the system and the outside world, including the data flowing into and out of the system. The only symbols that can appear on the context diagram are:

- Process symbol for initial process only
- Externals
- Data flows
- Control flows
- External stores

The initial process (process 0) describes the overall system. It is the only process allowed on the context diagram and appears on no other diagrams. Externals also appear only on the context level. Once you create a new level by decomposing process 0, the external symbols from the context level appear only optionally as offpage externals on the lower-level diagrams.

To identify the context diagram as the top-level data flow diagram, you must name it *top*, as explained in “Diagram Names and Process Indexes” on page 3-12.

Note: If the top-level diagram is named anything other than *top*, the initial process is assigned an index of <diagram_name>.0. To reassign it a correct process index of 0, save the diagram as *top*.

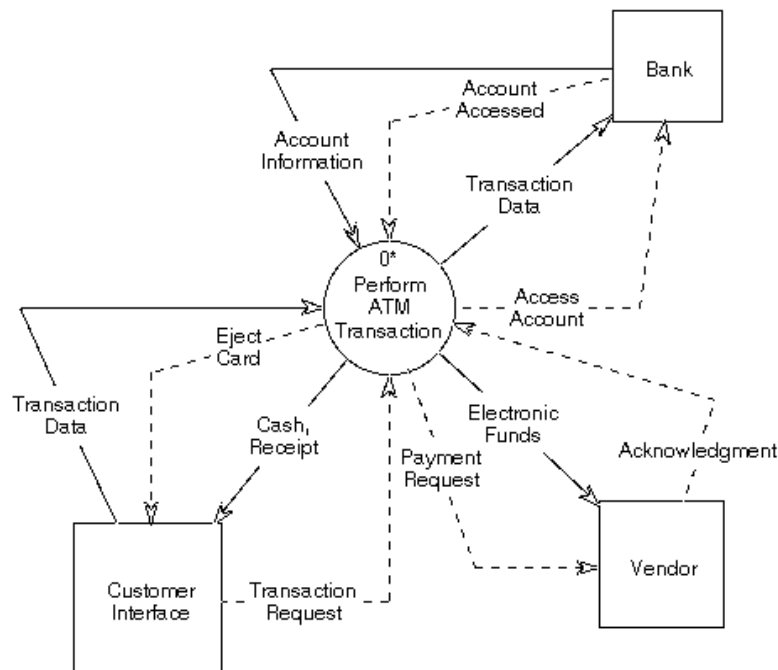
To create the context diagram:

1. From the Desktop **File** menu, choose **New**; from the **New** submenu, choose **DeMarco/Yourdon Data Flow Diagram** or **Gane/Sarson Data Flow Diagram**.
2. Insert a process symbol into the drawing area and type a label.
The process symbol is assigned a process index of 0, which appears as a display mark above the label.
3. Insert the necessary external symbols into the drawing area and label them.

4. Using an arc connection symbol, draw data flow and control flow arcs to connect each external to the process; then label each arc.
The connecting arcs are directional, and must be drawn from the source to the target. For information on drawing control flows, see “Representing Control Information” on page 3-32.
As needed, change the flow type with the **Edit > Replace** command or by selecting a different default arc type on the **Default Arc** tab of the **Options** dialog box (see “Data and Control Flows” on page 3-19).
5. When done creating the diagram, choose **Save As** from the **File** menu.
6. In the **Selection** field of the **Save As** dialog box, type **top** and click **Save**.

Figure 15 shows the context diagram for a sample ATM system.

Figure 15: ATM System Context Diagram



Decomposing Processes

Each process in a data flow diagram must be defined in either a:

- Decomposition diagram
- Process specification (Pspec)

Decomposition diagrams show progressively more detailed breakdowns of the functions performed by the parent processes in higher-level diagrams. After drawing the context diagram, you decompose process 0 to create the level 0 diagram, which shows the next level of detail in the system. All other processes in the system are drawn either on the level 0 diagram or on lower-level decomposition diagrams.

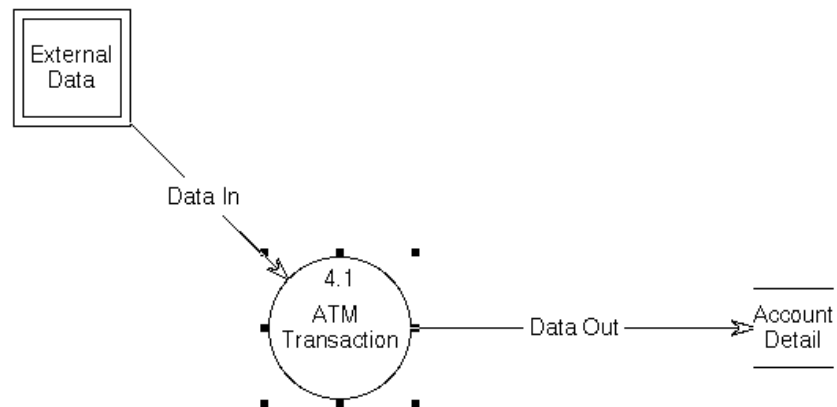
A primitive process, which needs no further decomposition, requires a process specification (Pspec) to describe it. For more information, see “Creating a Process Specification (Pspec)” on page 3-27.

Decomposing a process creates a decomposition diagram whose name is identical to the process index of the parent process, as described in “Diagram Names and Process Indexes” on page 3-12. For example, if process 4.1 is decomposed, a diagram named 4.1 is created. The diagram is named automatically.

To create a decomposition diagram for a process:

1. Double-click the process, or select the process to be decomposed.
2. From the **GoTo** menu, choose **Decomposition**.

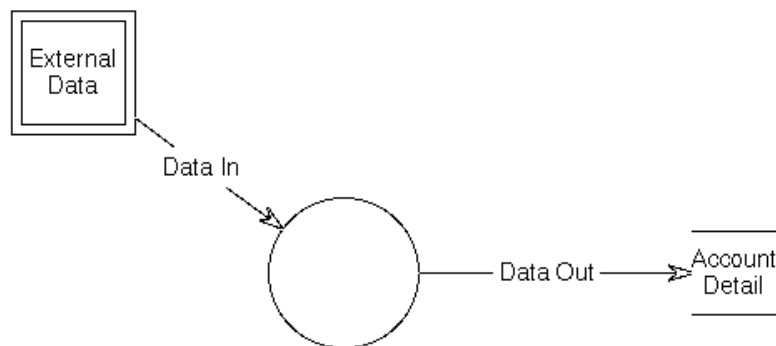
Figure 16: Selecting a Process for Decomposition



3. In the confirmation box, click **Yes** to confirm that you want to create the diagram.

The new decomposition diagram appears with an unlabeled process, as shown in Figure 17.

Figure 17: Unlabeled Process in Decomposition Diagram



The process symbol is connected by arcs representing data and control flows to offpage externals or offpage processes representing the externals or processes to which the parent process is connected. The Data Flow Editor uses offpage externals and offpage processes by

default. To change the default to anchor points, set the *use_offpages* ToolInfo variable to False. For more information on ToolInfo variables and files, see *StP Administration*.

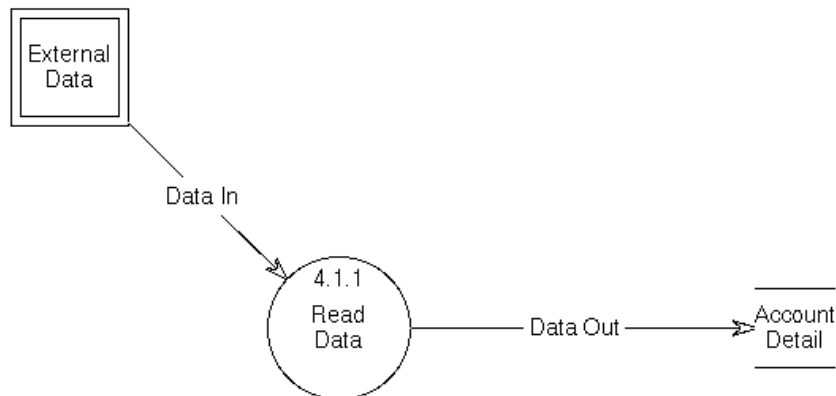
Loopback flows that begin and end in the parent process without connecting to any other symbol, do not appear in the decomposition and must be drawn manually to ensure proper balancing between diagrams.

4. Label the process on the decomposition diagram.

The process index for the decomposed process appears as a display mark above the label. The index is automatically incremented from the parent process to the next decimal point level for the decomposition. For example, if the parent process has an index of 4.1 the process appearing on the decomposition diagram has an index of 4.1.1, as shown in Figure 18.

If the process index does not appear, from the **View** menu choose **Refresh Display Marks** (for information on setting the display mark to Continuous Update, see “Using Display Marks” on page 3-10).

Figure 18: Decomposition Result



5. Add and label additional processes, as needed.

As each new process is added, the process number in the process index is incremented by one. For example, if the decomposition process in the diagram has an index of 4.1.1, additional processes on this diagram will be numbered 4.1.2, 4.1.3, and so on.

6. Add and label other symbols, such as data stores, and draw and label all connecting arcs needed to complete the diagram.
7. From the **File** menu, choose **Save**.
The diagram is saved with the name automatically assigned to it by the decomposition procedure.

Creating a Process Specification (Pspec)

A primitive process needs no further decomposition. The Data Flow Editor allows you to create process specification scripts (also called mini-specs or Pspecs) for primitive processes. A Pspec is a detailed description of a process derived from information represented symbolically in the diagram.

According to the rules of structured analysis, it is permissible to have either a Pspec or a decomposition for a process, but not both. If you decompose a process, and then decide you would rather define it with a process specification, you must delete or rename the hierarchy before you can generate the Pspec. If you want to expand the detail of a process by decomposing it instead of generating a process specification, you must delete the Pspec and Pspec note before you can create a decomposition.

The Pspec generated for process 1.1 in the *atm* example system is:

```
Process 1.1: Dispense Cash
Pspec generated from data flow diagram "1" of the "atm"
system of the "C:\StP\Examples\" projdir at 06/15/99
20:18:43 by cpanzer@CENTAURI
```

```
This process has 2 data flows:
Amount,
Cash
```

```
input data flows
Amount : DataFlow
```

```
output data flows
Cash : DataFlow
```

```
This process has 2 control flows:
```

Creating Data Flow and Control Flow Diagrams

```
Cash Dispensed,  
Cash Approved  
  
input control flows  
Cash Approved : ControlFlow  
  
output control flows  
Cash Dispensed : ControlFlow  
  
description  
  
The Dispense Cash process dispenses cash to the customer.  
end pspec
```

The Pspec associated with a process in a diagram is stored in an ASCII file in the *dfe_files* directory for your project and system. The filename is the process index plus the extension *.pspec*.

Creating a Pspec involves two separate procedures:

- Adding a Pspec annotation to each process for which you want to generate a process specification
- Generating process specifications for all Pspec-annotated processes

If you modify the diagram and make changes that affect the specified process, you must also regenerate the Pspec. The Data Flow Editor automatically makes all of the appropriate substitutions based on the updated diagram.

Creating a Pspec Annotation

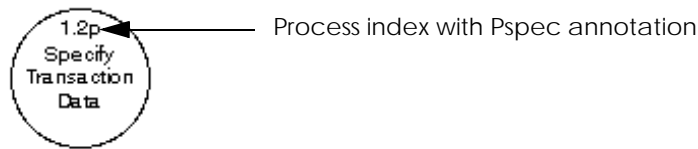
To create or edit a Pspec annotation:

1. Select a process for which you want to create or edit a process specification.
2. From the **DFE** menu, choose **Edit PSpec Note**.
This creates a Pspec annotation note with a Has Pspec? item for the selected process and displays a **Description** dialog box for the note description entry.
3. In the **Description** dialog, type a textual description that explains what the process does.

4. Click **OK** to save the note description.

A small **p** appears next to the process index as part of the Pindex display mark (Figure 19), indicating that a Pspec annotation is now associated with the process.

Figure 19: Display Mark for Pspec Annotation



Generating the Pspec

You can generate Pspecs for

- One or more processes on the current diagram in the Data Flow Editor
- All processes on selected diagrams in the Desktop objects pane
- All processes in your entire system model

Each process for which a Pspec is to be generated must have a Pspec annotation, as described in “Creating a Pspec Annotation” on page 3-28.

To generate a Pspec for any or all Pspec-annotated processes in the current diagram:

1. In the Data Flow Editor, select the processes for which Pspecs are to be generated.
If you do not select a process, a Pspec will be generated for every Pspec-annotated process in the current diagram.
2. From the **DFE** menu, choose **Generate Pspec**.
StP/SE generates the Pspec(s).

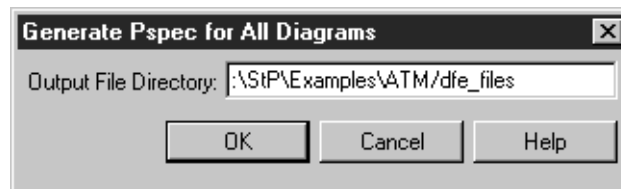
Creating Pspecs for Selected Diagrams or Entire Model

To generate Pspecs for all Pspec-annotated processes in selected diagrams or for the entire model:

1. In the Model pane on the StP Desktop, open the **Diagrams** category and select **DeMarco/Yourdon Data Flow** or **Gane/Sarson Data Flow**.
2. To generate Pspecs for processes in particular diagrams only, select the diagrams in the objects pane.
3. From the **Code** menu, choose **Pspec**; then choose either:
 - **Generate Pspec**—For one or more selected diagrams in the objects pane
 - **Generate Pspec for All Diagrams**—For all data flow diagrams in your system model

A dialog box similar to the one in Figure 20 appears.

Figure 20: Generate Pspec Dialog Box



4. Optionally, edit the contents of the **Output File Directory** field (the default output location for the generated files is the *dfe_files* directory in the current project and system).
5. Click **OK** to generate the Pspecs.

Generating a Pspec Report Using the Script Manager

If you prefer, you can use the StP Script Manager to generate a Pspec report in a selected publishing format, such as RTF or FrameMaker. You can execute one of the StP-provided scripts, *all_pspec* or *one_pspec*, or define a new script using StP's Query and Reporting Language (QRL).

See *Query and Reporting System* for more information on running scripts in the Script Manager, or on modifying scripts using QRL.

Viewing a Pspec

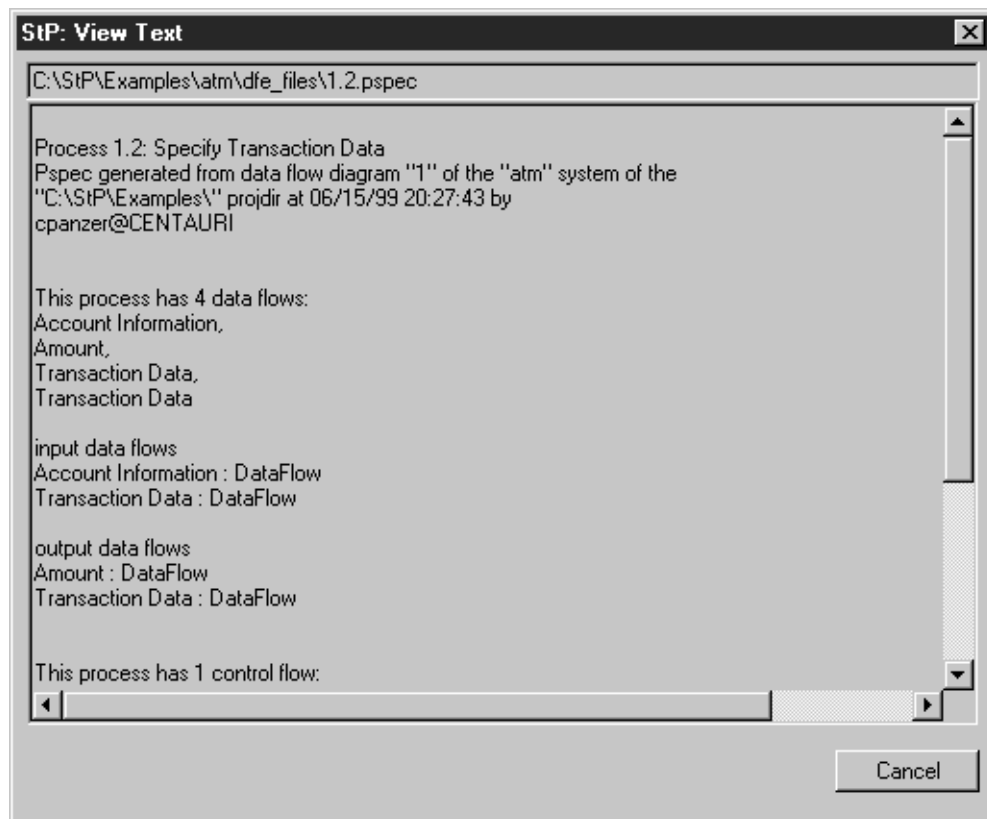
After generating the process specification, you can view the results directly from the Data Flow Editor using the **View Generated PSpec** command. This command opens a read-only window containing the Pspec. You can also use a standard text file editor to view the Pspec.

To view a Pspec from the Data Flow Editor:

1. Select the process.
2. From the **DFE** menu, choose **View Generated Pspec**.

The **View Text** window appears, as shown in Figure 21.

Figure 21: Pspec in View Text Window



Representing Control Information

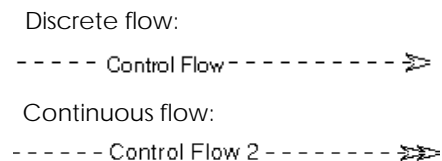
In the Data Flow Editor, you can represent control information along with data on the data flow diagram, using control flows (signals) and control bars. To view the diagram as either a control flow diagram or a data flow diagram, you apply a filter to temporarily hide data flows or control flows (see “Using Filters to Hide Data or Control Flows” on page 3-37 for more information).

Showing Control Flows

Control flows represent the movement of control information between processes, externals, and Cspec bars. To draw control flows, use the Arc or Spline symbol with the default flow type set to ControlFlow, as described in “Data and Control Flows” on page 3-19.

As shown in Figure 22, a control flow appears as a broken line with a single or double arrowhead at one end, which points to the destination symbol. A discrete flow is the default flow type. To designate the control flow as continuous, use the Object Annotation Editor to annotate the flow with a Flow Type note and a Flow Type item with a value of Continuous.

Figure 22: Control Flow



A control flow must be labeled with the name(s) of the control information flowing along it and defined in the Data Structure Editor (see “Defining Data and Control Information” on page 3-35).

Using Cspec Bars

You use the control specification (Cspec) bar, shown in Figure 23, to represent the real-time events that occur in the subset of the system represented by the diagram containing the Cspec bar. Only decomposition diagrams can contain Cspec bars. Using a Cspec bar in the context diagram results in a syntax error. Cspec bars do not have labels.

Figure 23: Cspec Bar



Cspec bars decompose into control specification tables and/or a state transition diagram, which detail the events that can occur in response to various inputs. From Cspec bars you can navigate to the Control Specification Editor or State Transition Editor.

The Control Specification Editor allows you to create a set of tables defining the real-time events for the part of the system represented by the Cspec bar. The various tables for a particular Cspec bar together compose an entire control specification. For instructions on creating Cspec tables, see Chapter 6, “Creating Control Specifications.”

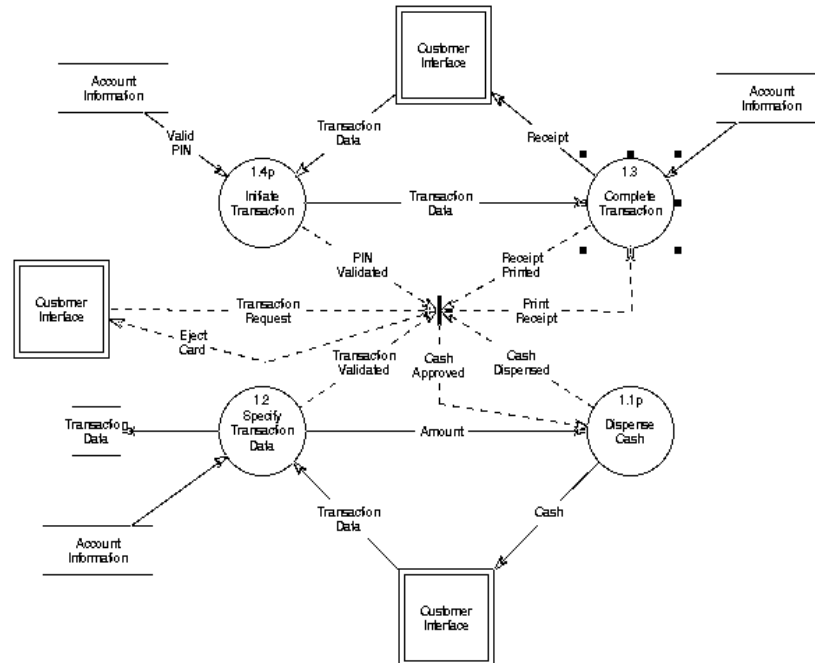
The State Transition Editor allows you to create a state transition diagram to model the state changes for this part of the system. For more information, see Chapter 5, “Creating State Transition Diagrams.”

To avoid clutter, you can use multiple Cspec bars in a single data flow diagram. However, all the Cspec bars in a diagram refer to the same control specification.

Drawing Cspec Bars and Control Flows

To draw control specification bars and control flows in a data flow diagram:

1. Insert a Cspec bar into the drawing area.
Figure 24 shows a Cspec bar in an ATM system diagram.



- Draw the arcs in the direction of the movement of the control information.

Defining Data and Control Information

In data flow diagrams, all data is represented by either data stores or data flows, and control information is represented by control flows. The data stores, data flows, and control flows must be labeled and defined. A data store implicitly holds all the data shown flowing into it, but as you proceed with the model, you should define the data more explicitly and completely. Such definitions are essential for building a complete model, checking the decomposition of a model, and checking the model for completeness.

You use the Data Structure Editor to define data and control information, including its types, constraints, structure and interrelationships. You can navigate to the Data Structure Editor by choosing the appropriate commands from the **GoTo** menu. If the defining diagram already exists, it is displayed in the appropriate editor. If the data structure diagram does not exist, you are given the opportunity to create it.

For the Data Flow Editor to find an existing data structure diagram that defines the selected data object, the following criteria must be met:

- The name of the data store, data flow, or control flow in the data flow diagram must be the same as in the data structure diagram.
- If StP/SE cannot determine which data definition corresponds to a data store, data flow, or control flow, you may be required to select the appropriate data structure diagram from a list displayed in a selector dialog.

In a defining data structure diagram, the object being defined appears as either a:

- Data element (leaf node)
- Parent object with additional structure

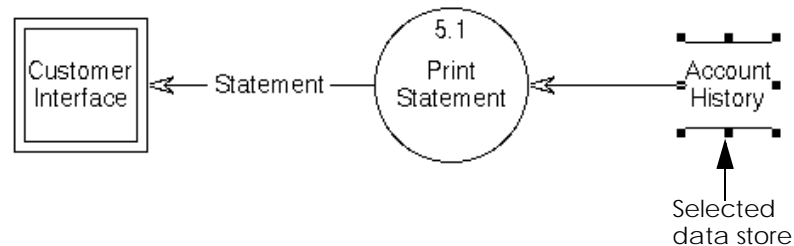
If it is a data element, it must have been assigned a data type value, indicated by a display mark in the lower-left corner. If it is not a data element, the object must have structure—that is, it must be the parent object for one or more other data objects that are defined in data structure diagrams.

Creating Data Flow and Control Flow Diagrams

To locate or define data structures for data stores, data flows, and control flows in data flow diagrams:

1. Select a data store, data flow, or control flow, as shown in Figure 25.

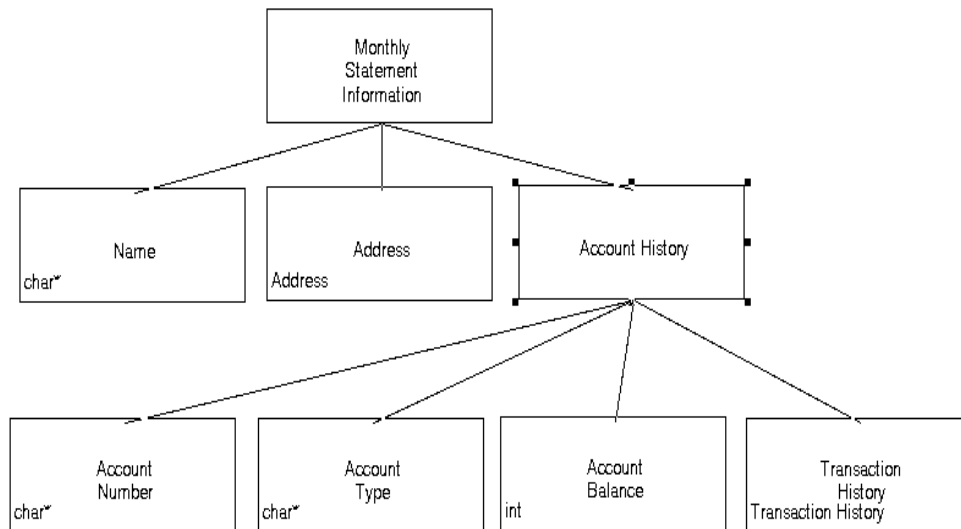
Figure 25: Defining Data



2. From the **GoTo** menu, choose **DSE Data Definition** to create or view an existing definition in the Data Structure Editor.

If the data structure exists, it appears in the Data Structure Editor with the associated data element selected, as shown in Figure 26.

Figure 26: Displaying a Data Element in the Data Structure Editor



3. If the data structure does not exist, click **Yes** to create it when the confirmation dialog box appears.

A sequence symbol appears by default in the drawing area. Both the sequence symbol and the diagram have the same name as the data store or data flow in the data flow diagram. (You can change the name of the diagram, if you wish.) For information about data structures, see Chapter 4, “Creating Data Structure Diagrams.”

Using Filters to Hide Data or Control Flows

Data flow diagrams may contain both data and control flows. To view data flows and control flows separately, you apply a filter that hides either data flows or control signals. These filters are:

- HideAllControlFlowsShowAllDataFlows
- HideAllDataFlowsShowAllControlFlows

To apply an editor-specific filter to a data flow diagram:

1. From the **View** menu, choose **Apply Filter**.
2. On the **Apply Filter** dialog:
 - Make sure the **Location** group is set to either **StP Installation** or **Both**.
 - Select **Editor Specific Filters** to display a list of filters specific to data flow diagrams.
3. Select one or more filter(s) in the list.
4. Click **OK**.

The diagram reappears with the appropriate flows hidden or visible, depending on your selection(s).

To redisplay the diagram with both data and control flows visible:

1. From the **View** menu, choose **Apply Filter**.
2. On the **Apply Filter** dialog, make sure the:
 - **Location** group is set to either **StP Installation** or **Both**
 - **Editor Specific Filters** option is not selected

3. In the **Filters** list, select **ShowAll**.
4. Click **OK**.

The diagram reappears with all data and control flows visible.

For more information about using filters, see *Fundamentals of StP*.

Reorganizing the Model

The Data Flow Editor provides a set of commands that allow you to reorganize data flow and control flow diagrams:

- **Rename Hierarchy** (see “Renaming a Hierarchy” on page 3-38)
- **Change Process Index** (see “Changing a Process Index” on page 3-40)
- **Collapse** (see “Collapsing Multiple Processes into One” on page 3-43)
- **Explode** (see “Exploding a Process into its Components” on page 3-46)
- **Remove All Qualifications in Diagram** (see “Removing Qualifications” on page 3-50)
- **Split Flow** (see “Splitting a Flow” on page 3-52)
- **Merge Flow** (see “Merging Flows” on page 3-53)
- **Reverse Flow** (see “Reversing a Flow” on page 3-55)

Renaming a Hierarchy

Renaming a hierarchy renames a specified diagram and all of its lower-level decomposition diagrams, and also renumbers the process indexes in those diagrams. This command is generally used when changes that are made to the system being analyzed require changes in logical organization and therefore in the order in which process decomposition diagrams are numbered.

For example, if diagram 4, shown in Figure 16 on page 3-25, is to be renamed to 5, all diagrams named *4[x.y.z...]* are renamed to *5[x.y.z...]*. All *4[x.y.z...]* process indexes are changed to *5[x.y.z...]*. No change is made to the index of process 4 on the parent of diagram 4.

Note: There is no **Undo** function for this command; therefore, it should be used with care. Because it renames diagrams, this command can profoundly affect the organization of the system being analyzed.

This command can be executed only from the StP Desktop. For information about the Desktop, see Chapter 2, “Using the StP Desktop.”

To rename a hierarchy:

1. In the Model pane, select the **Diagrams** category.
2. Select a data flow diagram subcategory.
3. From the objects pane, select a data flow diagram.
4. From the **Tools** menu, choose **Rename Hierarchy**.
5. In the **New Name** field of the **Rename Hierarchy** dialog box, type the new diagram name (number).

Figure 27: Rename Hierarchy Dialog Box



6. Click **OK**.

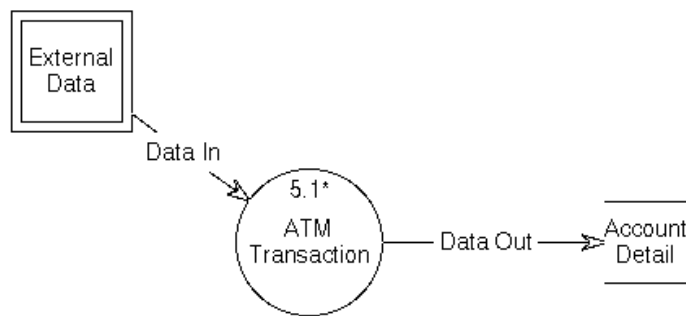
If the diagram name already exists, and you have selected the **Ask for Confirmation** option, you are given a choice of overwriting the existing diagram or cancelling the rename.

If the rename proceeds, the diagram and any subdiagrams and associated Cspecs are renamed in the repository. All process indexes on the affected diagrams are renumbered, as well.

Note: These changes are not immediately apparent on any currently open diagrams. To see the changes, reopen the diagram in the Data Flow Editor.

Figure 28 shows the diagram previously named 4 now named 5, with the index number of process 4.1 changed to 5.1. The asterisk following the process index indicates that this process has a decomposition. The rename operation also renames the decomposition diagram 4.1 to 5.1, and the process 4.1.1 to 5.1.1.

Figure 28: Rename Hierarchy Result



Changing a Process Index

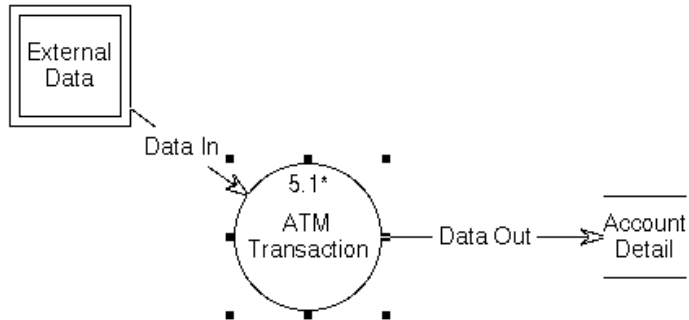
The **Change Process Index** command allows you to renumber a process index. The Data Flow Editor automatically increments index numbers, as you add processes to a diagram. You may want to change the sequence because an insertion followed by a subsequent deletion has left gaps in the numbering.

To change a process index:

1. Select the process that has the index number you want to change, as shown in Figure 29.

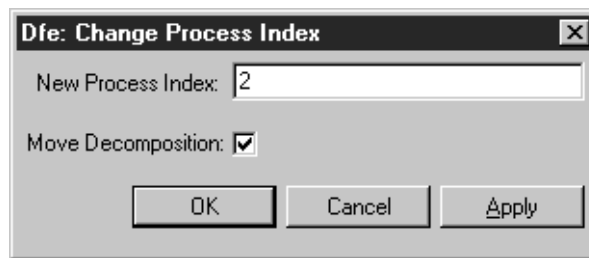
In this example, process 5.1 on diagram 5 will be changed to process 5.2.

Figure 29: Change Process Index



2. From the **DFE** menu, choose **Change Process Index**.
3. In the **New Process Index** field of the **Change Process Index** dialog box, type the new relative process index (the last decimal digit(s) only), as shown in Figure 30.

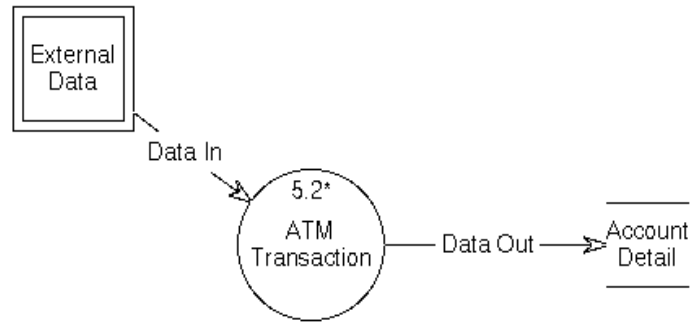
Figure 30: Change Process Index Dialog Box



4. Optionally select **Move Decomposition** to change the process indexes throughout the decomposition hierarchy.
5. Click **OK**.

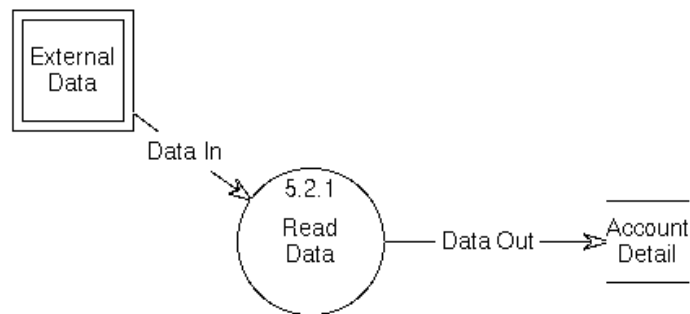
The index number is changed, as shown in Figure 31; the name of the diagram remains the same.

Figure 31: Change Process Index Result



If the process has a decomposition and you selected **Move Decomposition** in the dialog box, all process indexes throughout the decomposition hierarchy are renamed as well. In this example, the asterisk following the process index indicates the existence of a decomposition process 5.1.1. As a result of the process index change in Figure 31, the decomposition diagram named 5.1 is automatically moved to diagram 5.2, and the decomposition process index 5.1.1 is changed to 5.2.1 (see Figure 32).

Figure 32: Change Decomposition Process Index



The Data Flow Editor prevents duplicate process indexes.

For example, if process numbers 1 to 4 are used on a diagram, and you want to swap numbers 2 and 3:

1. Change the index number of process 2 to an unused number, such as 5.
2. Change the index number of process 3 to 2.
3. Change the index number of process 5 (originally 2) to 3.

Collapsing Multiple Processes into One

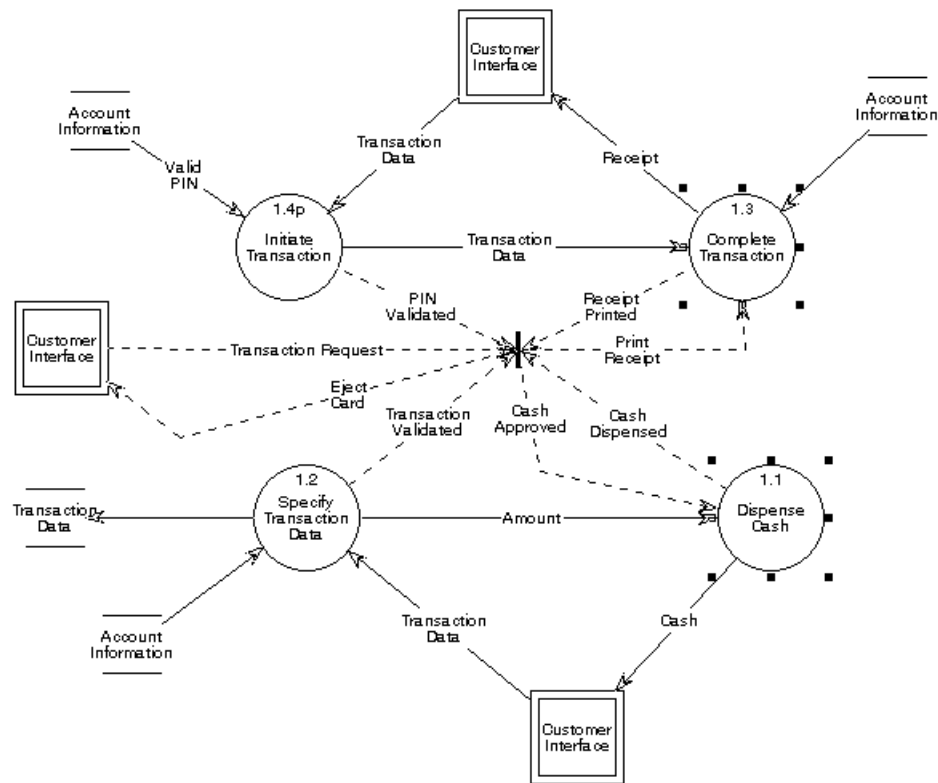
The **Collapse** command allows you to combine multiple processes into a single process. All links into and out of the symbol set are preserved. You can use **Collapse** on any level 1 or lower level diagram.

Note: This command renames diagrams, and should be used with caution. It cannot be undone.

1. Select the multiple processes to be collapsed into a single process, as shown in Figure 33.

In this example, the name of the diagram is *1*.

Figure 33: Collapse - Before

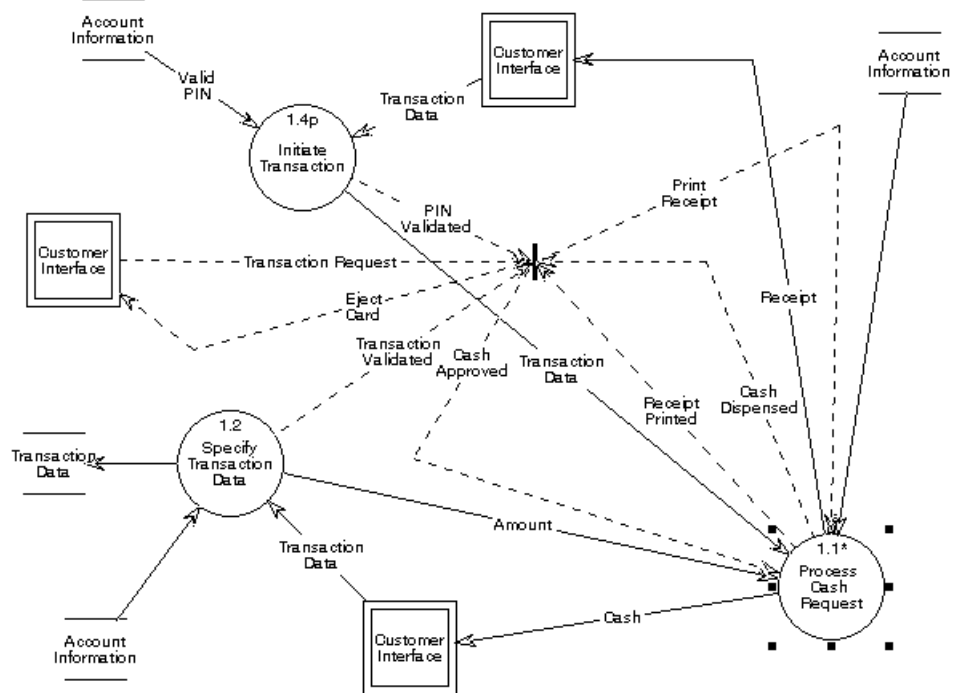


2. From the **DFE** menu, choose **Collapse**.
3. In the **Collapse Process** dialog box, type the name to be assigned to the collapsed process, and click **OK**.

In this example, the new name is *Process Cash Request*.

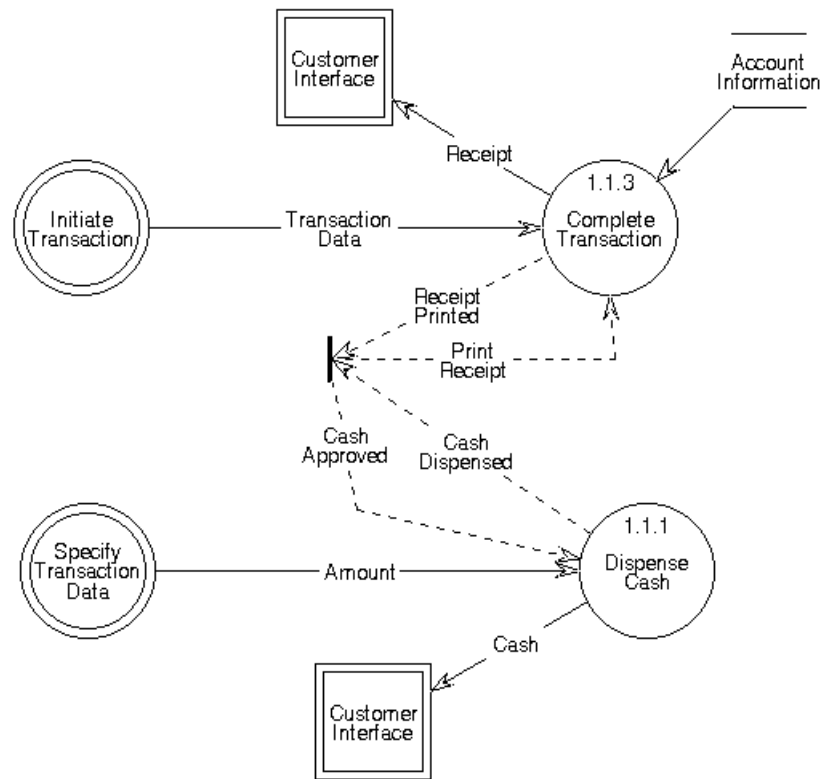
As shown in Figure 34, the original processes are now combined in the new collapsed process 1.1 on diagram 1. You may need to reposition some elements of the diagram for easier readability.

Figure 34: Collapse - After



The newly collapsed process 1.1 has a decomposition, as indicated by the asterisk display mark. The decomposition, named diagram 1.1, contains the original set of processes from diagram 1 that illustrate the detail for the collapsed process (see Figure 35). StP ensures that the process numbers in the decomposition diagram are unique and reflect the level of the diagram. In this case, the new process indexes are 1.1.1 and 1.1.3.

Figure 35: Decomposition of Collapsed Process



Any pre-existing decompositions of the processes in diagram 1.1 are moved to a lower-level decomposition.

Exploding a Process into its Components

The **Explode** command allows you to merge a single process with its set of decomposition processes. You can use the **Explode** command on any process in a level 1 or lower level diagram that has a decomposition.

Note: This command renames diagrams, and should be used with caution. It cannot be undone.

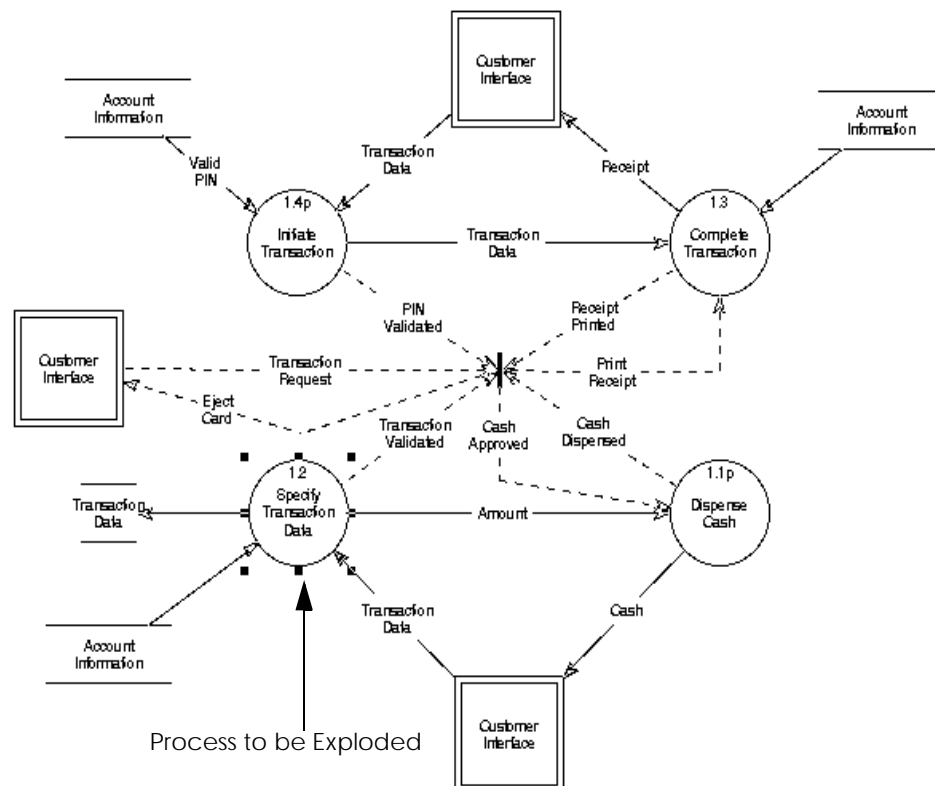
When you execute **Explode**:

- The lower-level processes are moved to the diagram with the higher-level process.
- The decomposition diagram is deleted, and any child diagrams are renumbered to reflect the new levels.
- The exploded process is initially replaced with an anchor that connects with all links into and out of the original symbol. To merge the decomposition with the diagram, you must move the links to the new processes.

To explode a process:

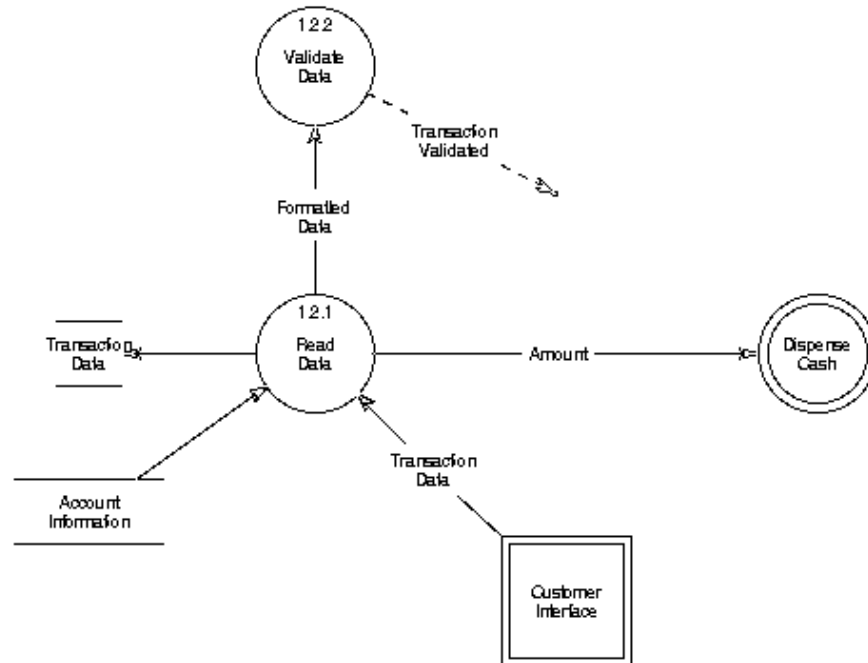
1. Select the process to be exploded, shown in Figure 36.

Figure 36: Explode - Before



This process has an existing decomposition, shown in Figure 37.

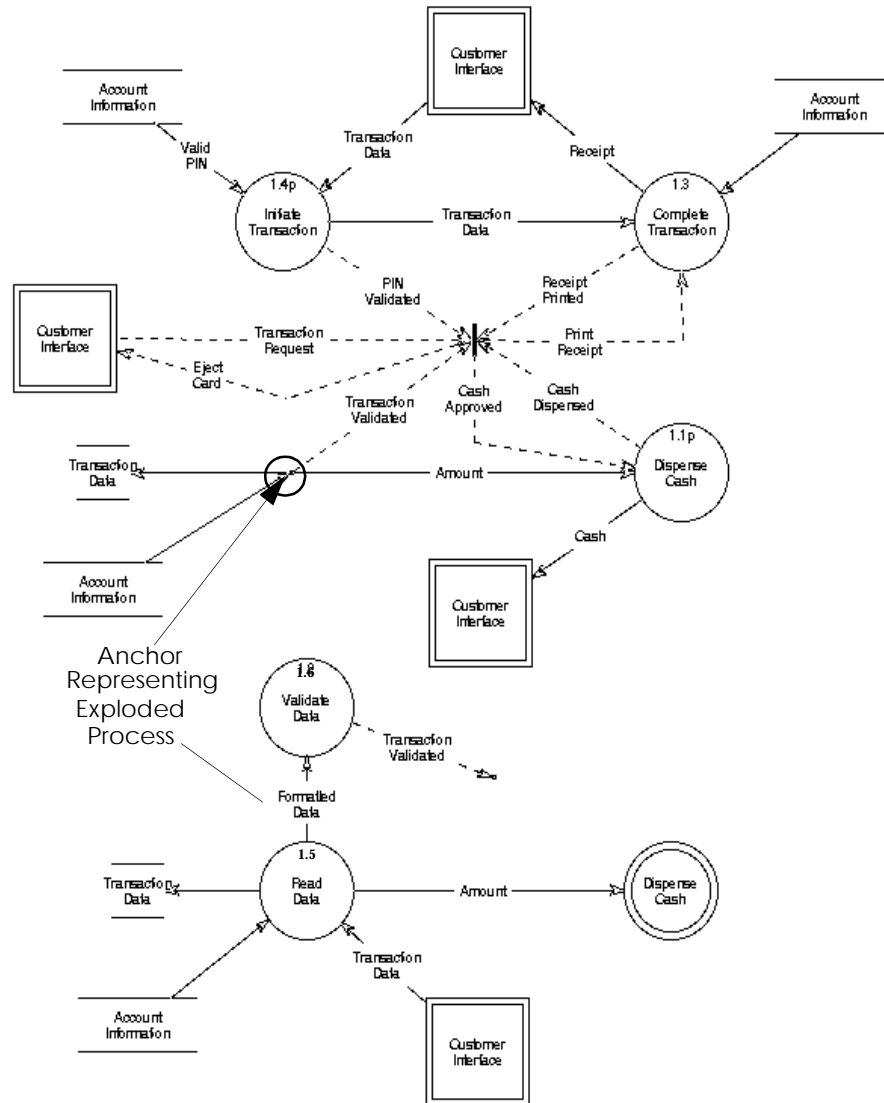
Figure 37: Decomposition of Specify Transaction Data



2. From the **DFE** menu, choose **Explode**.

The decomposed set of processes in Figure 37 appears on the higher-level diagram (Figure 38). The original exploded process, *Specify Transaction Data*, is replaced by an anchor point.

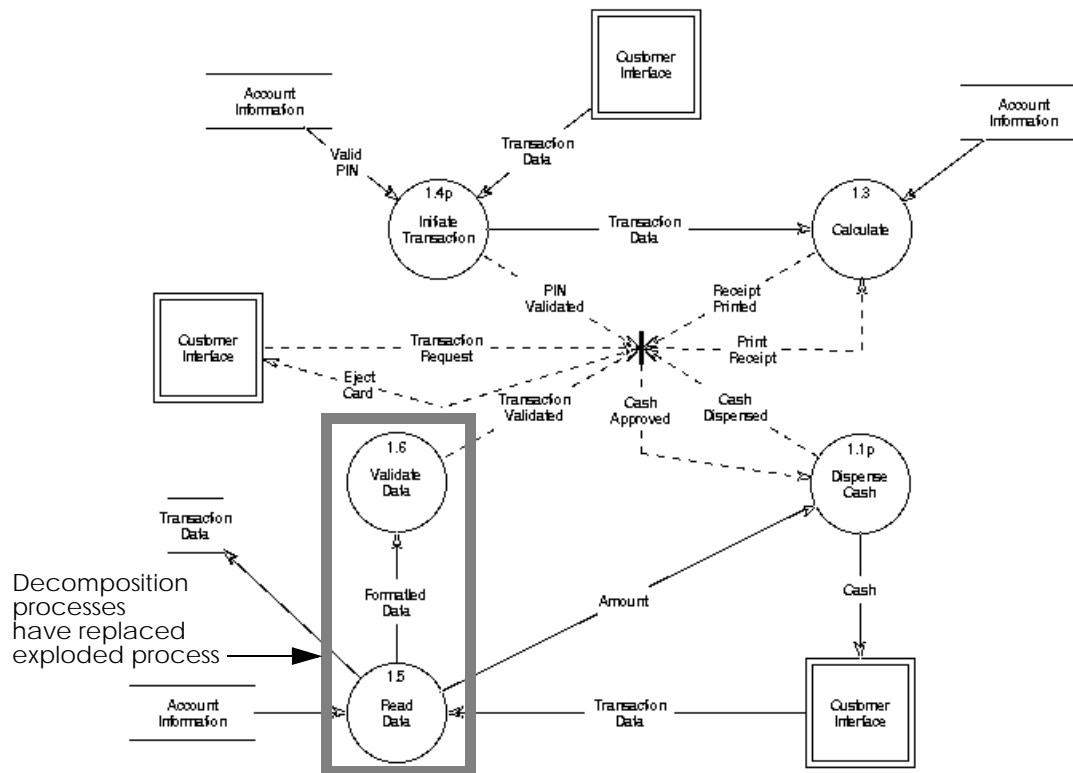
Figure 38: Decomposition Drawn on Higher-level Diagram



3. Reconnect the flows from the exploded process anchor to the decomposition processes, repositioning the processes as needed.
4. Delete any unneeded duplicate stores, processes, and flows that were created when the decomposition was moved to the parent diagram.

5. Delete the anchor that temporarily represented the exploded process.
The finished exploded process diagram is shown in Figure 39.

Figure 39: Exploded Process Diagram - After



Removing Qualifications

In prior releases of StP/SE, it was sometimes necessary to qualify a data flow, control flow, or data store with a unique signature to differentiate it from like-named objects in the repository and to identify its corresponding data structure diagram. This is no longer required. Except in the case of modeling errors, it is now possible for StP/SE to deduce the proper data structures without qualifications. Qualifications in existing

systems need not be removed for use with the current StP/SE release. However, you may want to remove them for compatibility with future releases.

You can remove qualifications from the:

- Current data flow diagram, using the **Remove All Qualifications in Diagram** command on the Data Flow Editor's **DFE** menu
- Entire model, including all data flow diagrams and Cspec tables, using an StP Query and Reporting Language (QRL) script

Removing Qualifications from the Current Diagram

The **Remove All Qualifications in Diagram** command operates only on the current data flow diagram. The command appears dimmed on the **DFE** menu if no qualifications exist in the current diagram. There is no equivalent command in the Control Specification Editor to remove qualifications in individual Cspec tables.

Removing Qualifications from an Entire Model

To remove all qualifications from your entire model, execute the following QRL script at a command prompt:

```
grp -s <sys> -p <proj>
<stp_path>\templates\se\qrl\check\RemoveQualifications.qrl
```

Table 4: grp Command Arguments

Flag	Argument	Description
-s	<sys>	Sets the current system name to a system directory you specify. The new system name overrides the default value set by the <i>system</i> ToolInfo variable.
-p	<proj>	Sets the current project name to a project directory you specify. The new project name overrides the default value set by the <i>projdir</i> ToolInfo variable.
	<stp_path>	Specifies the StP installation directory, including the drive designation, if required.

In order for the RemoveQualifications script to run to completion, diagrams must not contain syntax errors. This script takes longer to execute on larger systems, as it must load and save every data flow diagram and control specification table.

Splitting a Flow

The **Split Flow** command allows you to split a data flow or control flow to show data or control flowing from a single source to two or more targets.

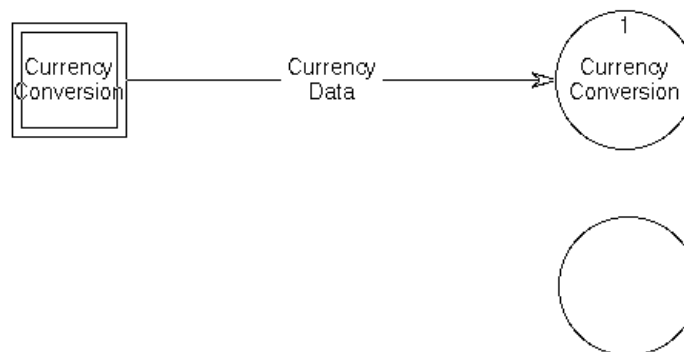
Two Targets

To split a flow between two targets:

1. Select the flow to be split.

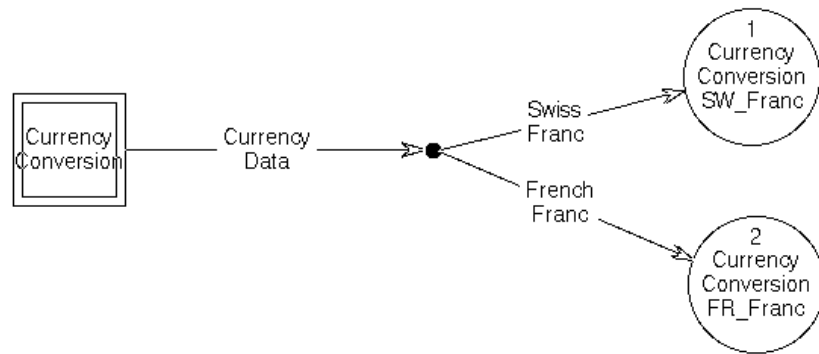
In this example, *Currency Data*, shown in Figure 40, is to be split so that it flows not only to process 1 but also to new process 2.

Figure 40: Split Flow - Before



2. From the **DFE** menu, choose **Split Flow**.
A vertex appears in the selected data flow as the source of a new data flow. The target of the new data flow is attached to the mouse pointer.
3. Drag the new data flow to the new target, and click the left mouse button.
The split flow appears, as shown in Figure 41, and can be labeled.

Figure 41: Split Flow - After



Adding a Flow to an Existing Split Flow

To add a flow to an existing split flow, begin a new arc at the point of the split.

Merging Flows

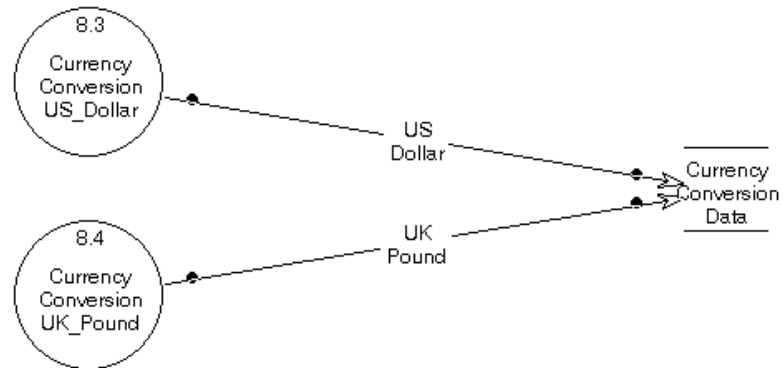
The **Merge Flow** command also allows you to merge data flows or control flows to show data or control flowing from multiple sources to a single target.

To merge a flow:

1. Select the flows to be merged.

In this example, *US Dollar* and *UK Pound*, shown in Figure 42, are to be merged so that only a single data flow enters the data store.

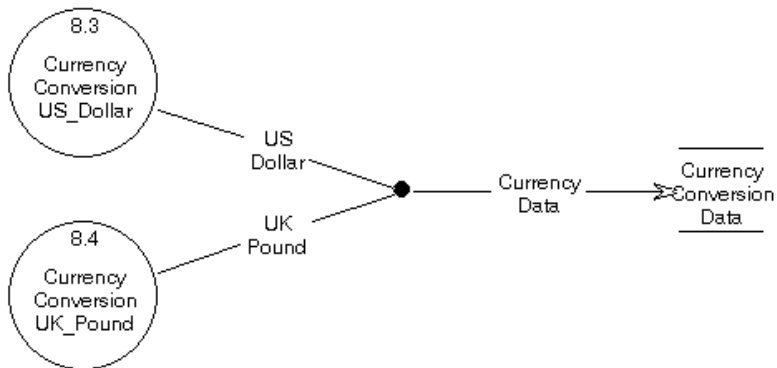
Figure 42: Merge Flow - Before



2. From the **DFE** menu, choose **Merge Flow**.

The data flows are merged at a vertex, as shown in Figure 43. The merged section of the data flows can be labeled.

Figure 43: Merge Flow - After



Reversing a Flow

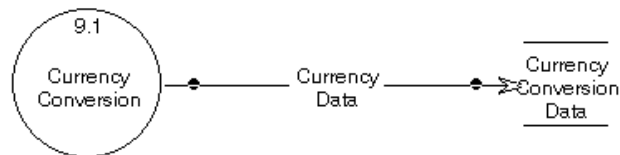
The **Reverse Flow** command allows you to reverse the direction of selected data flows or control flows, including split flows.

To reverse a flow:

1. Select one or more flows to be reversed.

In this example, the direction of the *Currency Data* flow, shown in Figure 44, is to be reversed.

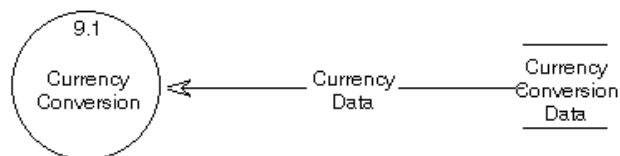
Figure 44: Reverse Flow - Before



2. From the **DFE** menu, choose **Reverse Flow**.

The direction of the data flow is reversed, as shown in Figure 45.

Figure 45: Reverse Flow - After



Validating Data Flow Diagrams

StP/SE editors provide two kinds of validation checks for the diagrams that comprise your system model:

- Syntax checks
- Semantic checks

This section describes how to check syntax and semantics for the current diagram in the Data Flow Editor. You can also check semantics for the entire model or for one or more selected diagrams from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, “Checking SE Models.”

Syntax Checks

A data flow diagram is syntactically complete and correctly drawn if:

- The context diagram contains at least one external, one process, no offpage connectors, and no cspec bars.
- Non-context diagrams have no externals or external stores.
- Each process has an outgoing flow.
- All processes, externals, stores, external stores, offpage processes, and offpage externals are labeled.
- All anchor points, externals, stores, Cspec bars, offpage processes, and offpage externals have at least one connection.
- All flows are named (data flows to or from a store need not be named; they are implicitly given the same name as the store).
- No split flow in a given diagram has both multiple in and multiple out links (multiple in *or* multiple out links are OK).

Checking diagram syntax does not check the contents of the repository.

Syntax checks are applied automatically when you save the current diagram, or you can choose the **Check Syntax** command on the **Tools** menu.

Semantic Checks

A data flow diagram is semantically correct if all data and control flows are balanced between diagram levels and are properly defined in data structure diagrams. In a decomposition diagram, flows to and from offpage processes, offpage externals, and anchors must balance with the flows to and from the parent process on the parent diagram. For a complete list of Data Flow Editor semantic checks, see “Data Flow Editor Checks” on page 9-3.

For information on checking Cspec semantics, see “Validating a Cspec” on page 6-30.

To invoke semantic checking for the current diagram, choose either of the following commands from the **Tools** menu:

- **Check Semantics**—In addition to displaying results in the Message Log, optionally allows you to save results to a file in a selected format
- **Check Semantics Selectively**—Allows you to apply only selected semantic checks

4

Creating Data Structure Diagrams

In support of structured design and structured analysis, StP/SE provides the Data Structure Editor (DSE) for drawing hierarchical data structures that define data objects in design models and data types in analysis models. The DSE diagrams define actual data and control signals used in data flow diagrams and data types of objects used in structure chart diagrams. You can define both simple data objects and complex, hierarchically structured ones. Through annotations, the diagrams also specify the types, ranges, and values associated with data, control signals, and data types.

This chapter describes:

- “Using the Data Structure Editor” on page 4-2
- “Creating a Data Structure Diagram” on page 4-8
- “Setting Properties of Data Objects” on page 4-17
- “Defining an Abstract Data Type” on page 4-30
- “Decomposing an Object” on page 4-33
- “Navigating to a Parent” on page 4-36
- “Showing All Children” on page 4-36
- “Generating a BNF File” on page 4-38
- “Validating a Data Structure Diagram” on page 4-40

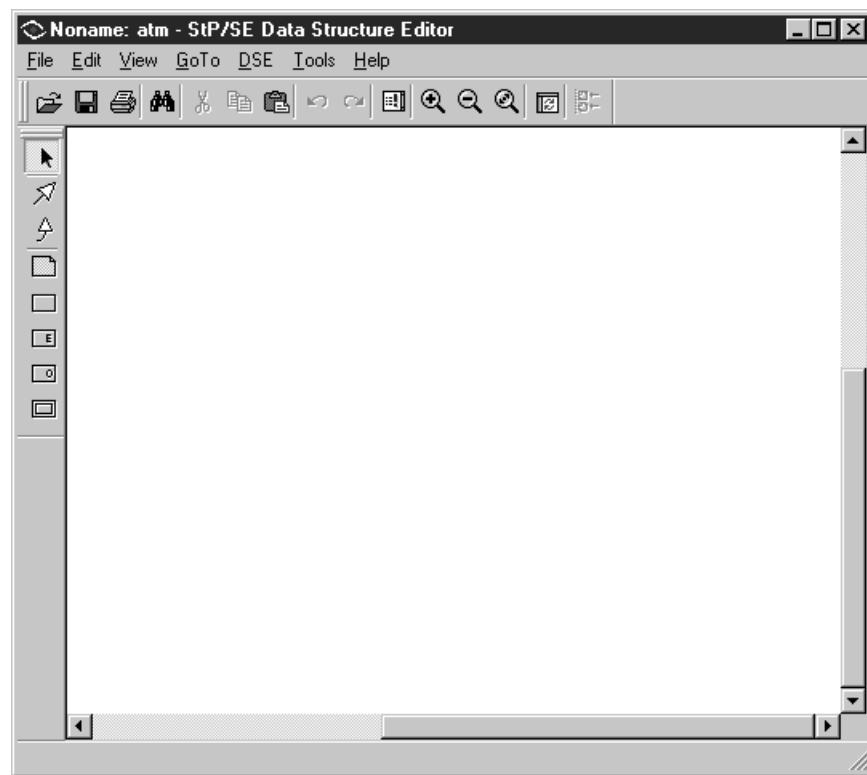
This chapter provides a brief explanation of data structure notation and methodology. The Data Structure Editor implements a notation based on the Michael Jackson data structure notation. For complete details on the Jackson notation, refer to *Principles of Program Design* by Jackson (Academic Press, 1975).

For instructions on automatically generating data structure diagrams from existing C code, see Chapter 11, “Reverse Engineering.”

Using the Data Structure Editor

The Data Structure Editor (Figure 1) is an interactive graphical tool for drawing hierarchical data structures.

Figure 1: Data Structure Editor Window



The Data Structure Editor provides the standard StP diagram editor functions and menu options. For general information about diagram editors, see *Fundamentals of StP*.

In addition to the standard StP Diagram editor features, the Data Structure Editor provides:

- Symbols for drawing data structure diagrams
- Navigations to related diagrams and tables
- The **DSE** menu, providing commands specific to the Data Structure Editor
- Display marks representing additional information about data structure objects

These features are described briefly in this section. For more information about using these features, see “Creating a Data Structure Diagram” on page 4-8.

Starting the Data Structure Editor

StP/SE provides access to the Data Structure Editor from:

- The StP Desktop
- Data Flow Editor
- Structure Chart Editor

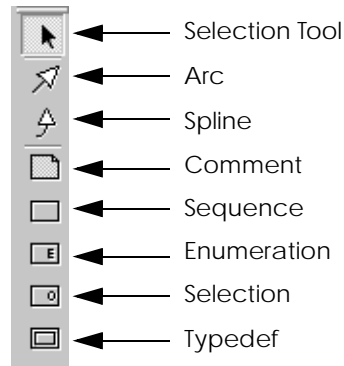
For information about starting DSE from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

You can start the Data Structure Editor from the Data Flow Editor or Structure Chart Editor by using an appropriate navigation command from that editor’s **GoTo** menu. If the Data Structure Editor has already been started, the navigation command uses the current session rather than starting another copy of the editor. For information about starting the Data Structure Editor from the Data Flow Editor or Structure Chart Editor, see the chapter describing that editor.

Using the Data Structure Editor Symbols

You can select symbols for drawing data structure diagrams from the Symbols toolbar (Figure 2).

Figure 2: Data Structure Editor Symbols Toolbar



Procedures for using the symbols are described in this chapter. For a summary description of each symbol, see Appendix B, “StP/SE Symbol Reference.” For general instructions on using the Symbols toolbar, see *Fundamentals of StP*.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object whose symbol is selected on the current diagram. Use the **GoTo** menu to navigate. In the Data Structure Editor, the commands on the **GoTo** menu allow you to:

- Display different levels in a hierarchy of data structure diagrams
- Create new lower-level data structure diagrams
- Access a selected object’s C source code in a code viewer or editor
- Navigate to the Requirements Table Editor from the current Data Structure Editor session (the current session continues, as well)

The **GoTo** menu provides context-sensitive choices for the selected symbol.

To navigate to a target:

1. Select a symbol on the current diagram.

2. Choose a command from the **GoTo** menu.

The navigation target appears.

Table 1 describes the navigation targets and commands available for each Data Structure Editor symbol.

Table 1: GoTo menu Commands

Navigate From	Navigate To	Command
All	Other references to the same object	All References
	Object's source code definition in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38)	Source Code Definition
	Requirements table (see "Allocating Requirements to a Model" on page 1-10)	Allocate Requirements to Object
Sequence, Selection, Enumeration	Depending on whether you're navigating from a component structure or a type definition diagram: - A higher-level object to which the selected object is connected by a component link, - Your choice (from a selection list) of an object whose type is defined by the structure whose root is selected	Parent
Sequence, Selection	Diagram defining the data structure of the data type indicated by the object's data type annotation, if any	Type's Definition
	Lower-level diagram in which a node without children in a higher-level diagram is defined by its decomposition into a substructure containing children	Scoped Decomposition
Typedef	Diagram defining the data structure of the data type indicated by the object's data type annotation	Type

Using the DSE Menu

In addition to the standard diagram menus described in *Fundamentals of StP*, the Data Structure Editor provides the **DSE** menu. This menu allows you to perform a variety of tasks that modify data structure diagrams.

Table 2 describes the commands available from the **DSE** menu.

Table 2: DSE Menu Commands

Command	Target Description	For Details, See
Generate BNF for Diagram	Generates a Backus-Naur format (BNF) file for the current diagram. The file is written to the <i>dse_files</i> directory in the path defined by the project and system names. The name of the generated file is <i><diagram>.bnf</i> , where <i><diagram></i> is the name of the data structure diagram.	“Generating a BNF File” on page 4-38
Show All Children	Draws on the current diagram all children of the selected symbol, whether the children are from a decomposition or another diagram.	“Showing All Children” on page 4-36

Using Display Marks

A display mark is a symbol or string that appears on or near an object and conveys additional information about that object. Several data structure diagram annotations cause display marks to appear in the diagram.

You can control the behavior of these display marks using the **Display Marks** tab of the **Options** dialog box. To display this dialog box, choose **Options** on the **Tools** menu. For details about using the dialog box, see *Fundamentals of StP*.

Table 3 describes the display marks that can appear on a data structure diagram. For more information and an example of each mark, see the section referenced in the table.

Table 3: Display Marks

Display Mark For	Name	Description	For Details, See
SEDirectory scope object	SEDirectory	Period (.) for default or user-specified text in outside upper right corner of root nodes that identifies the directory object to which this data object is scoped.	“Specifying Scope Information” on page 4-21
SEFile scope object	SEFile	Text in outside lower right corner of root nodes that identifies the file object to which this data object is scoped.	“Specifying Scope Information” on page 4-21
C structure tags	Tag	Text preceded by the word Tag: in the upper left corner of intermediate nodes refers to the C structure tag for the struct, union, or enum. The tag is stored as the Tag item on a Data Definition annotation note.	“Adding C Structure Tags” on page 4-29
Data type	LeafType, TypedefLeafType	Text in lower-left corner indicates the data type of the object. The data type is stored as the Data Type item on a Data Definition annotation note for sequences and selections, and on a Typedef Definition annotation note for typedefs.	“Assigning Data Types” on page 4-27
Array size	ArraySize	Text in lower right corner identifies array size. Array bounds are stored as the Array Size item on a Data Definition annotation note.	“Specifying Array Size” on page 4-28

Creating a Data Structure Diagram

You need to create data structures to define data, control signals, or data types on any StP/SE diagram. A data structure can contain the:

- Definition of a data store in a data flow diagram
- Definition of a data flow or control flow in a data flow diagram
- Definition of a return type for a module or a data type for global data or parameters in a structure chart

The data structures modeled in this section are taken from or based on the ATM system distributed with StP/SE.

Elements of a Well-Defined Data Structure

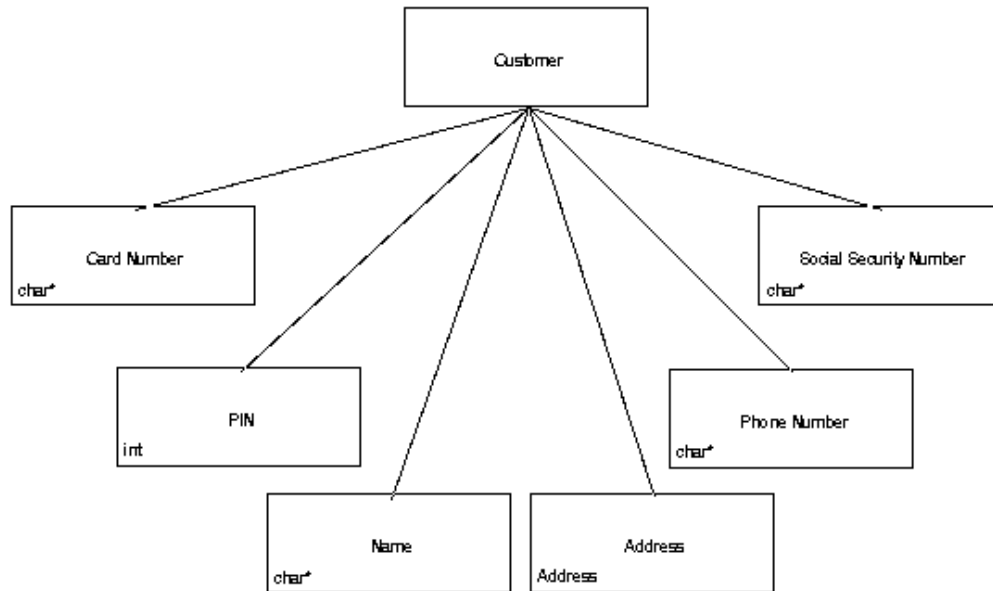
A data structure consists of sequence, selection, and enumeration symbols linked hierarchically into a tree-like structure. The root of the tree is drawn at the top of the diagram, and can be a sequence, selection, or enumeration data symbol. The intermediate nodes in the tree structure and the leaves of the tree (which represent data elements) can be represented by any of the preceding symbol types, although some restrictions apply, as described in “Representing Data Objects” on page 4-10. In most cases, intermediate nodes and leaf nodes on the same level can be of mixed symbol types. Links must always be drawn from the root to the leaves (from parent to child).

Note: The remaining symbol type, typedef, is used only to create a pointer to a defined data object and has no in or out links. It may appear on a data structure diagram as a discrete symbol, but cannot be a root or leaf node in the hierarchical data structure itself.

Figure 3 shows an example of a data structure for the *Customer* data object.

The symbols in a data structure diagram are usually arranged from left to right, in the order in which the fields occur in a record structure (as in Figure 3), and from top to bottom in hierarchical order (as in Figure 4 on page 4-11).

Figure 3: Example of a Data Structure



A data structure is considered fully defined when every symbol in the structure is defined. Objects must be defined consistently. The same drawing rules and procedures that produce well-defined data can also be used to define control signals. Symbols that are data elements and symbols that are intermediate nodes are defined differently.

Symbols that Are Data Elements

A data element in a data structure diagram is a data object that has no dependent children in any diagram. Data elements are the lowest-level elements in a data structure diagram. A data element is also called a leaf, or leaf node, in the tree structure. Data elements can be represented only by sequence and selection symbols (described in “Representing Data Objects” on page 4-10).

Generally, a data element is considered fully defined when it has been assigned a data type value, as indicated by the display mark in the lower-left corner of the symbol (see Figure 3). One exception to this is that children of an enumeration represent finite values that cannot have data

types (see “Using Enumeration Symbols” on page 4-13). Another exception is if the data element is designated as a system type (see “Defining a Data Element as a System Type” on page 4-30).

Typically, the data type value assigned to an object is a simple data type such as char, float, or int. Alternatively, the data type value can be the name of an abstract data type that is represented by an object defined somewhere in the system—for example, as an associated data structure diagram created with the **Type’s Definition** command (see “Defining an Abstract Data Type” on page 4-30).

An undefined leaf node can be decomposed in order to further define its structure in another diagram, rather than assigning it a data type. However, when decomposed, it ceases to be a leaf node and becomes an intermediate node with children. A leaf node cannot have both a data type and a decomposition.

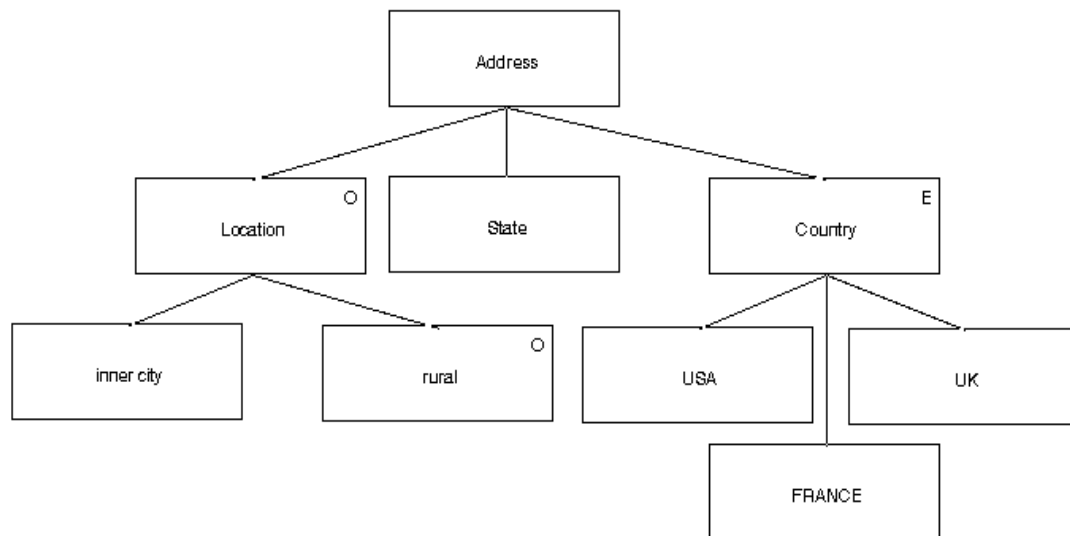
Symbols that Are Root or Intermediate Nodes

If a symbol is not a data element in a particular diagram, it is a root node or an intermediate node in that diagram. A root node is the highest-level symbol on a data structure diagram. Intermediate nodes are children of higher-level nodes that also have children of their own somewhere in the system—for example, on the same diagram or in a decomposition diagram. Root nodes and intermediate nodes are considered fully defined when they have children, and are sometimes called parent nodes. A parent can have any number of children, which can be any combination of data elements and intermediate nodes.

Representing Data Objects

Figure 4 shows an example of the beginning of a data structure diagram in which sequence, selection, and enumeration symbols are used to represent parts of a customer’s address.

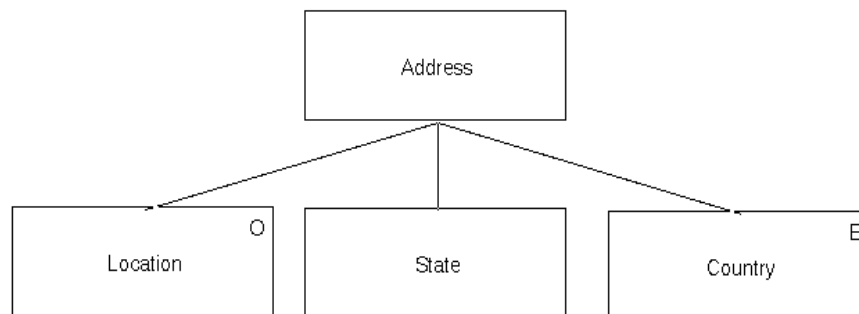
Figure 4: Beginning of a Data Structure for Address



Using Sequence Symbols

A sequence symbol implies an “and” relationship among its children, which together compose the complete structure of the parent. For example, in Figure 5 the data structure of the *Address* sequence symbol is composed of its children, *Location*, *State*, and *Country*.

Figure 5: Sequence Symbol Example



The children of a sequence symbol can be any combination of sequence, selection, or enumeration symbols. In this case, *Location* is a selection, *State* is a sequence, and *Country* is an enumeration.

The sequence symbol can represent the root of a data structure, a single data object, a field in a record structure, or a set of data objects with the same data structure (such as an array).

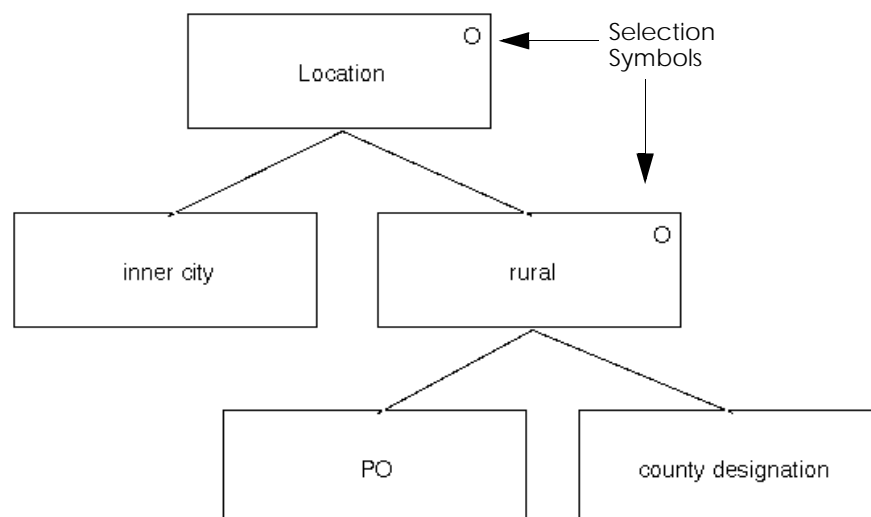
Using Selection Symbols

A selection symbol represents an “or” relationship among its children that implies a choice between two or more structures. A selection symbol is distinguished by an \circ (for “or”) in the upper-right corner.

In Figure 6, the data object named *Location* is drawn with a selection symbol because there are two, mutually exclusive ways to indicate a location (*inner city* or *rural*). The object *rural* is also a selection symbol, because it is further defined by either of two children that represent different ways to specify a rural address (*PO* or *county designation*).

In order to be semantically correct, a selection symbol must have at least one child on either the current diagram or in a decomposition diagram.

Figure 6: Selection Symbol Example



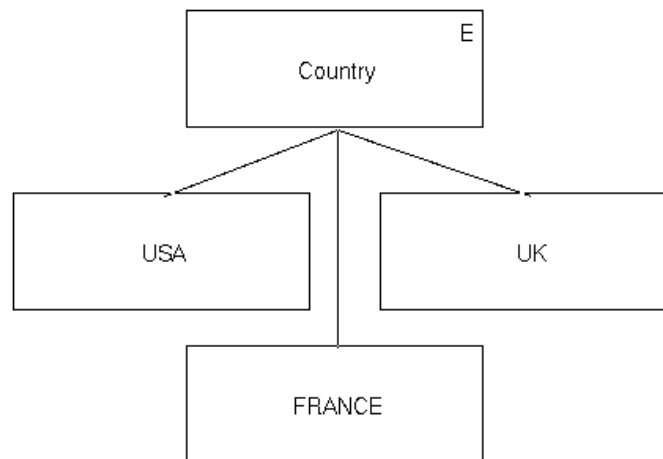
Using Enumeration Symbols

An enumeration symbol represents a data element that is limited to a finite number of specified values, which are represented by the enumeration's children. An enumeration symbol is distinguished by an \mathbb{E} in the upper-right corner.

The *country* data object in Figure 7 is drawn with an enumeration symbol because, in this example, there are a finite number of countries (USA, UK, and FRANCE) that are legal data values for this object.

An enumeration cannot be decomposed. It must have at least one child in the current diagram in order to be semantically correct. The children can be sequence symbols only. Since they represent finite values, children of an enumeration cannot have data types or children of their own.

Figure 7: Enumeration Example

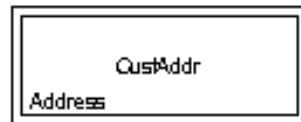


Using Typedefs

A typedef symbol represents a pointer to another defined data object. Typically, it is used to associate a data or return type in a structure chart with a data definition that has a different name in a data structure diagram. Typedefs have no in or out links and are not included in a hierarchical data structure, although they may appear on the same diagram. To indicate what a typedef represents, you specify its data type

as the name of another data object in a data structure diagram. For example, Figure 8 shows a typedef symbol named *CustAddr* with a data type of *Address*. The data type indicates that this typedef is a pointer to the *Address* data object.

Figure 8: Typedef



Creating Root, Intermediate, and Leaf Nodes

Use the following guidelines to create a data structure with root, intermediate, and leaf nodes:

1. Do one of the following to begin the diagram:
 - Navigate from the Data Flow Editor (see “Defining Data and Control Information” on page 3-35)
 - Navigate from the Structure Chart Editor (see “Defining an Object’s Data or Return Type” on page 7-40)
 - Insert a sequence, selection, or enumeration symbol into the drawing area for the root of the data structure.
2. Insert additional sequence, selection, enumeration, or typedef symbols as needed (see “Representing Data Objects” on page 4-10).
3. Label the symbols.
4. Draw the links, from parent symbol to child symbol(s).
5. From the **Edit** menu, choose **Properties** and specify the additional properties for each of the objects, as needed:
 - Assign a data type to each leaf node (see “Assigning Data Types” on page 4-27).
 - Add array bounds to sequence, selection, and typedef symbols, as needed (see “Specifying Array Size” on page 4-28).
 - Add structure tags to sequence and selection symbols, as needed (see “Adding C Structure Tags” on page 4-29).
 - Change the default directory and file scope for the root node, if needed (see “Specifying Scope Information” on page 4-21).

6. Define undefined data types and decompose nodes, as needed (see “Defining an Object in Another Diagram” on page 4-15).
7. From the **File** menu, choose **Save As** to name and save the data structure diagram.

For more specific instructions and examples, refer to the sections noted in the preceding guidelines.

Defining an Object in Another Diagram

To define a data object, you add additional structure to it. You can include the additional level of detail in the same diagram or in an associated diagram. Adding substructure to an object in the same diagram may make the diagram too crowded. Or an object’s substructure may be more appropriately defined on another diagram.

You can use either of these methods to define an object’s structure on another diagram:

- Assign the object an abstract data type and use the **Type’s Definition** command to create an associated data structure diagram defining the abstract data type (preferred method; see “Defining an Abstract Data Type” on page 4-30 for details).
- Use the **Scoped Decomposition** command to create an associated lower-level diagram that is a continuation of the original data structure, in which you add substructure to a particular object (see “Decomposing an Object” on page 4-33 for details).

These methods are mutually exclusive. If an object has a data type, it should not have a decomposition. If it has a decomposition, it should not have a data type.

Defining Control Information

Generally, you define a data flow diagram’s control flows as single objects of type `int` or `boolean` in a data structure diagram. Although a control flow typically has no hierarchical structure, defining it in the Data Structure Editor allows you to provide additional information about the flow in the form of annotations. For example, you can annotate the data definition of a control flow to specify allowed control values for the flow.

You can then use the Control Specification Editor to populate control value cells in a Cspec table with the annotated values (see “Setting Control Values” on page 6-20).

To specify allowed values for a control flow in a data structure diagram:

1. Do one of the following:
 - From the Data Flow Editor, select the control flow and from the **GoTo** menu, choose **DSE Data Definition**.
 - From the Data Structure Editor, select the object representing the control flow.
2. From the Data Structure Editor's **Edit** menu, choose **Object Annotation**.

This starts the Object Annotation Editor (OAE) and creates a Data Definition annotation note for the data structure object representing the control flow.
3. Select the Data Definition note in the OAE **Annotations** list and add an Allowed Value item to the note, using one of the following methods:
 - In the **Items** field, select **Allowed Value** and click the **Add** button.
 - From the OAE **Edit** menu, choose **Add Item** and select **Allowed Value** from the list of items.
4. In the **Description** field, enter a user-defined string or number representing one possible control value this flow can have; then click the **Set** button.
5. Repeat the previous two steps for each possible control value the flow could have.
6. Save the annotation and exit the OAE.

Setting Properties of Data Objects

A data object can have various properties, including:

- Directory and file scope
- Data type
- Array size
- Structure tag

These properties represent some of the most common relationships and annotations for data objects in a data structure diagram. You can assign these properties to data objects by editing the object property sheet, as explained in “Using the Properties Dialog Box” on page 4-17. Most assigned properties are stored as annotations of the object in the repository. The directory and file properties are stored as SEDirectory and SEFile objects.

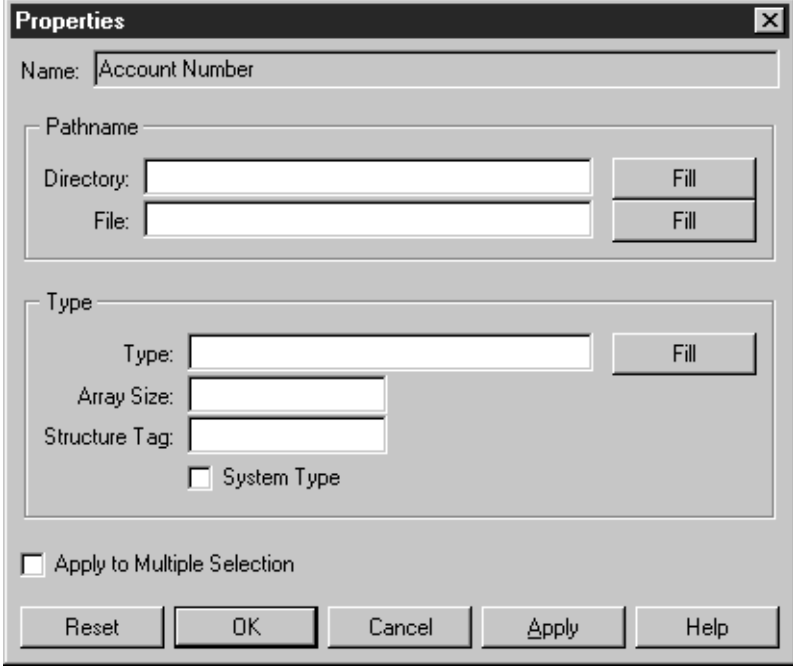
Alternatively, you can use the Object Annotation Editor (OAE) to add these or other annotations to data objects. Use the **On Selection** option on the OAE **Help** menu to provide a description of available annotations. For information about the OAE, see *Fundamentals of StP*.

Some properties are specifically used for C code generation from data structure diagrams. For information about these properties, see “Data Structure Annotations Used in Code Generation” on page 10-5.

Using the Properties Dialog Box

The **Properties** dialog box (Figure 9) enables you to specify or change any or all of the characteristics of a selected object at one time. The dialog box always displays the properties for the currently selected object. If you select a different object, the properties shown in the dialog box change to those of the newly selected object.

Figure 9: Properties Dialog Box



The Properties dialog box is a standard Windows-style window with a title bar that says "Properties" and a close button (X). It contains several sections for configuring object properties:

- Name:** A text field containing "Account Number".
- Pathname:** A section containing two text fields: "Directory:" and "File:". To the right of each field is a "Fill" button.
- Type:** A section containing three text fields: "Type:", "Array Size:", and "Structure Tag:". To the right of the "Type:" field is a "Fill" button. Below these fields is a checkbox labeled "System Type".
- Apply to Multiple Selection:** A checkbox at the bottom of the main content area.
- Buttons:** At the bottom of the dialog are five buttons: "Reset", "OK", "Cancel", "Apply", and "Help".

To use the **Properties** dialog box:

1. Select a single object or multiple objects.
2. From the **Edit** menu, choose **Properties**.

The **Properties** dialog box appears with current settings for the selected object. If multiple objects are selected, properties for the first object appear, but the dialog fields are initially dimmed.
3. To apply properties to multiple selected objects, select **Apply to Multiple Selection** to make the previously dimmed fields editable.
4. Enter or change the values for the selected object(s), as needed.

For details about each setting, see "Summary of Object Properties and Dialog Options" on page 4-20 and other subsequent sections.
5. Click **OK** or **Apply**.
6. From the **File** menu, choose **Save** to save the object's new properties in the repository.

Selecting Values from a Fill List

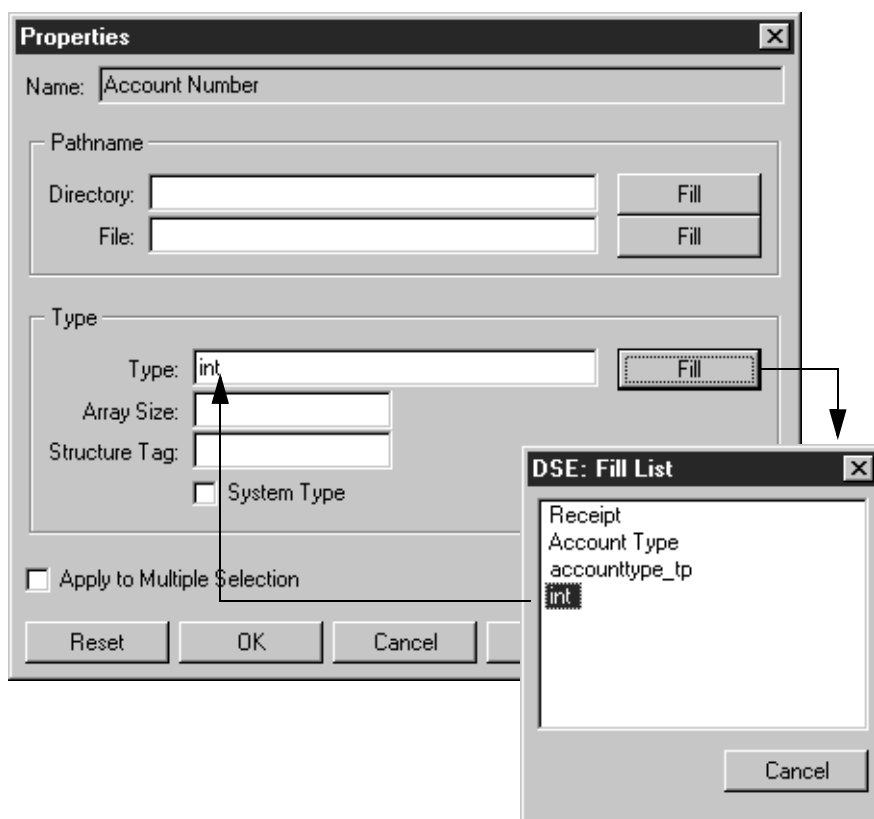
The **Fill** buttons on the **Properties** dialog allows you to choose an existing value for the adjacent field from a list.

To select a property value from the **Fill** list:

1. On the **Properties** dialog, click the **Fill** button to the right of the **Directory**, **File**, or **Type** field.
2. Select one of the values in the list as shown in Figure 10; then close the **Fill** window.

The selected value appears in the **Properties** dialog.

Figure 10: Choosing a Value from the Fill List



Summary of Object Properties and Dialog Options

Table 4 provides a summary of the **Properties** dialog box options and their possible settings.

Table 4: Object Properties Summary

Property or Option	Description	Settings	For Details, See
Name	Name of the object to which the properties will be applied.	(Read-only field)	
Fill button	Displays a list of choices for the adjacent property specification.		“Selecting Values from a Fill List” on page 4-19
Directory, File	SEDirectory and SEFile scope objects, respectively, to which the root node of a data structure is scoped, and which determine the directory and filename for the data structure’s C code.	Directory and file pathnames.	“Specifying Scope Information” on page 4-21
Type	Defines the data type of the object.	Pre-defined C types are char, double, float, int, long, short, or void. You can also specify the named of another data object as the data type.	“Assigning Data Types” on page 4-27
Array Size	Defines a sequence or selection data object as an array with a limited number of occurrences in your system.	Text that specifies the array bounds; for example, 1000 or 1-10.	“Specifying Array Size” on page 4-28

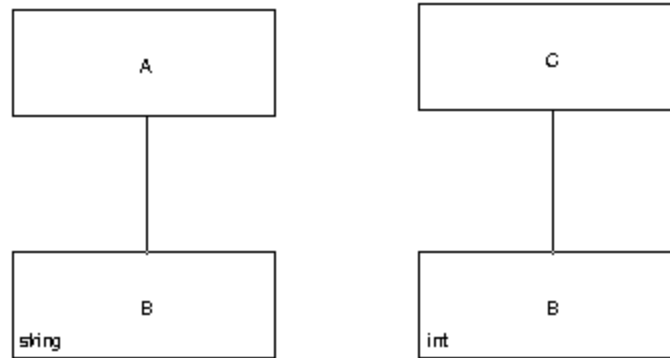
Table 4: Object Properties Summary (Continued)

Property or Option	Description	Settings	For Details, See
Structure Tag	Contains text representing the name for the abstract data type represented by an intermediate node and its children, for use in generating C code or appearing in diagrams reverse engineered from C code.	Can be applied to any intermediate node with children. Structure tags do not apply to leaf nodes.	“Adding C Structure Tags” on page 4-29
System Type	Defines a data element as a system type that is not defined within the scope of this system, and for which no definition is generated by the C code generator.	Selected or unselected.	“Defining a Data Element as a System Type” on page 4-30
Apply to Multiple Selection	Applies the property values to multiple selected objects.	Selected or unselected.	“Using the Properties Dialog Box” on page 4-17

Specifying Scope Information

Data object names are not unique within a system model. Objects in a data structure are distinguished from identically-named objects through their connections to parent objects. In StP, object identification through the parent-child hierarchy is called scoping (not to be confused with scoping in the C programming sense). For example, in Figure 11 the object *B* that is scoped to parent *A* is different from object *B* scoped to parent *C*, as the former is of type string while the latter is of type int.

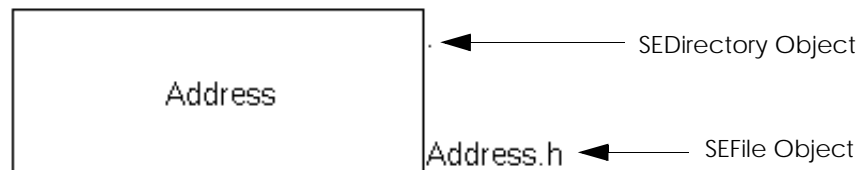
Figure 11: Scoping Data Objects through the Parent-Child Hierarchy



Each root object is scoped to an SEDirectory and SEFile object in the repository. The SEDirectory and SEFile objects indicate the directory and file location for any C code for this object. By default, root objects are scoped to a default directory represented by a period (.) in the object's scope specification string. At code generation time, the period is interpreted as the current directory from which you start the C code generator. The default file is *<object_name>.h*. Identically named root symbols with default file and directory scope always refer to the same repository object.

The directory and file scope information appears as display marks in the outside upper-right and lower-right corners of the root symbol when the SEDirectory and SEFile display marks are turned on. The default directory appears as a small dot, as shown in Figure 12.

Figure 12: Default SEDirectory and SEFile Display Marks



You may want to scope the root of a data structure to a different SEDirectory or SEFile object in order to:

- Specify a different directory or filename for the data structure's C code
- Differentiate between this root object and one with the same name in the repository

Differentiating Between Identically Named Roots

You can use different SEFile objects to differentiate between two identically named root nodes that represent different data objects with different substructures. For example, suppose your model contains an *Address* object defined in one diagram as a postal address and an *Address* object defined in another diagram as an Internet email address. As shown in Figure 13, you could assign a file specification of *Address_E.h* to the email *Address* object, to distinguish it from the postal *Address* object. In this example, both of the objects are scoped to the default SEDirectory object.

Figure 13: Root Objects with Same Name, Scoped Differently

Diagram 1:

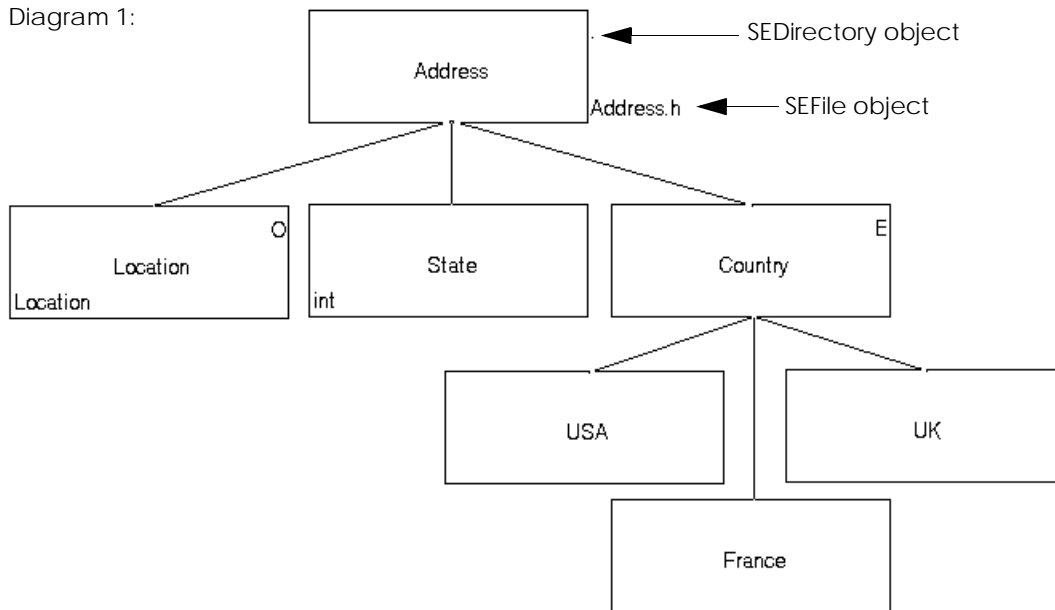
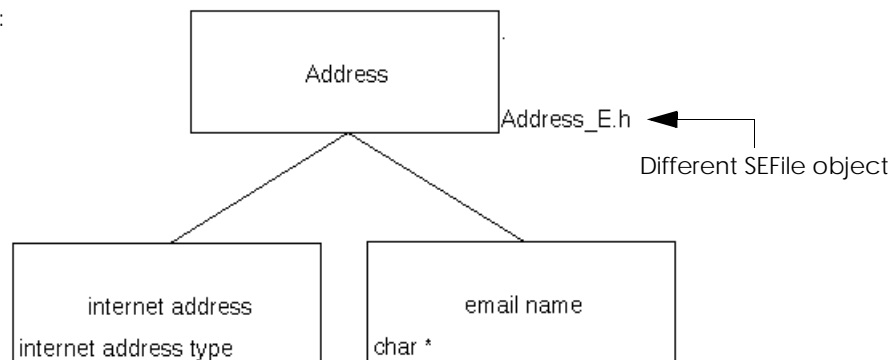


Diagram 2:



Scoping Partial Structures to the Same Root

Generally, you should keep all same-level child nodes together on one diagram, to provide a complete view of that structure level. However, in some cases, you may want to create separate diagrams, each of which contains only some of the children. In such cases, even if the root nodes in these diagrams have the same name, you may need to scope them to the same SEDirectory and/or SEFile object to ensure that they are differentiated from any other identically named root object.

Figure 14 on page 4-25 illustrates this strategy. The root objects are scoped to the same SEDirectory and SEFile objects. Each substructure diagram contains only some of the root node's children, and therefore represents only a portion of the entire data definition for the root.

Figure 14: Two Diagrams Defining Parts of the Same Root Object

Diagram 1:

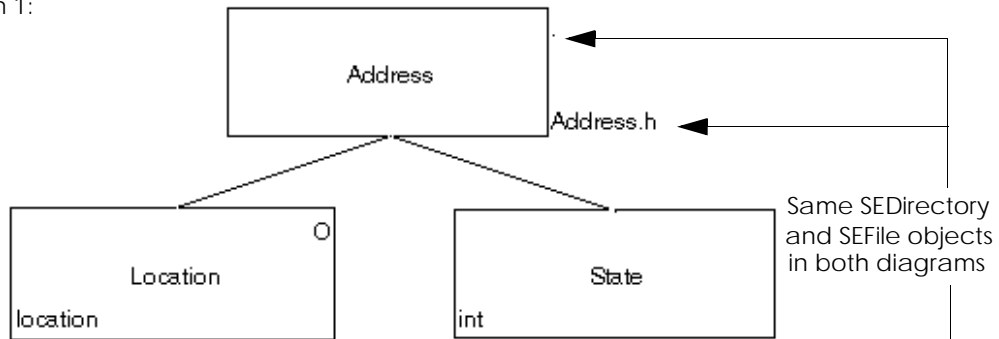
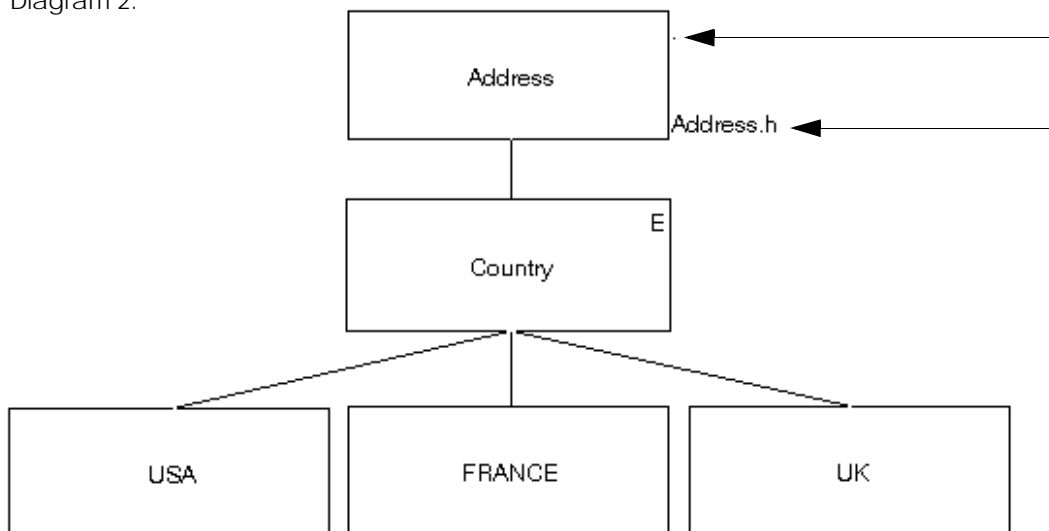


Diagram 2:



Scoping a Root Node to a Different Directory or File Object

To scope the root node of a data structure to a different SEDirectory or SEFile object, you edit its Directory and/or File properties on the **Properties** dialog box.

Changing the scope of the root node effectively creates a new data structure with its own scope chain in the repository. StP/SE copies all annotations for the original data structure to the newly scoped one. The original object remains in the repository, along with the newly scoped one. Other references to the original object remain unchanged. If you want some of the other references to point to the newly scoped object, use the same procedure to change their scope properties on the **Properties** dialog.

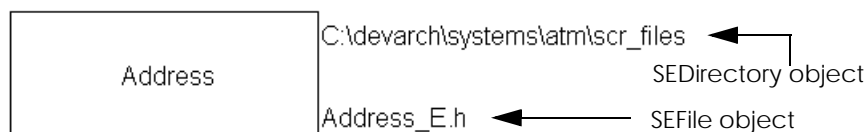
Note: To change the value of an SEFile or SEDirectory object for all occurrences in the current system, see “Renaming SEFile and SEDirectory Objects” on page 13-12.

To scope the root node of a data structure to a different SEDirectory or SEFile object:

1. Select the root node.
2. From the **Edit** menu, choose **Properties**.
3. On the **Properties** dialog, edit the specifications in the **Directory** and/or **File** fields, either by typing them manually or by using the **Fill** button to select from a list (see “Selecting Values from a Fill List” on page 4-19).
4. Click **OK** or **Apply**.

If the display marks are turned on, the changed directory and/or file scoping information appears, as shown in Figure 15.

Figure 15: Root with User-Specified Directory and File Scope



Assigning Data Types

In order to be fully defined, each typedef and each sequence and selection leaf node must either be designated as a system type (see “Defining a Data Element as a System Type” on page 4-30) or you must assign it one of these data types:

- Simple, predefined basic C type, such as char*, float, or int
- Abstract data type (ADT), represented by a data object that has a defining structure in a data structure diagram

If the data type is not defined, int is assumed. Enumerations cannot have data types.

Once an object’s type is defined, all other occurrences of the same object in a data structure diagram are automatically defined as having the same type, because they are all references to the same object.

Assigning a data type to a sequence, selection, or typedef creates a Type item for a:

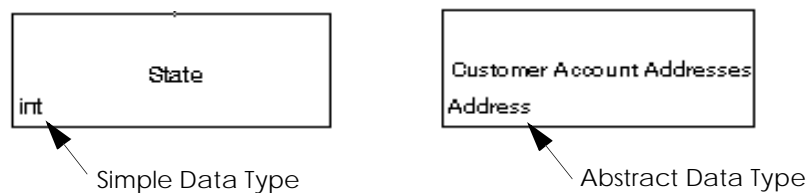
- Data Definition annotation note (sequences and selections)
- Typedef Definition annotation note (typedefs)

To assign a data type to a data object, do one of the following in the **Type** field on the **Properties** dialog:

- Enter one of these simple, predefined C data types: char, double, float, int, long, short, or void.
- Enter the name of an object that is the root of a data structure that defines an abstract data type (for details, see “Defining an Object in Another Diagram” on page 4-15).
- From a **Fill** list, select a data object that defines an abstract data type, as explained in “Selecting Values from a Fill List” on page 4-19.

The data type assigned to the selected object appears in the lower-left corner of the object’s symbol, as shown in the following figures.

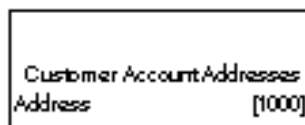
Figure 16: Data Type Display Marks



Specifying Array Size

You can assign an array size to a sequence, selection, or typedef symbol to indicate that it represents an array of a limited set of data objects that all have the same data type. *Customer Account Addresses*, shown in Figure 17, is an example of a sequence symbol that represents an array—in this case, of 1000 addresses, all of which have the data structure defined by the *Address* data type.

Figure 17: Sequence Symbol with Array Bounds



To specify array size on the **Properties** dialog, enter the array limits as text in the **Array Size** field. For example, you could enter 1000 for an upper limit only, or 1-10 to specify both lower and upper limits. The text you enter appears as a display mark enclosed in square brackets in the lower-right corner of the symbol. If you include the brackets in your text entry, the extra pair of enclosing brackets are omitted.

Specifying the array size creates an Array Size item for a

- Data Definition annotation note (sequences and selections)
- Typedef Definition annotation note (typedefs)

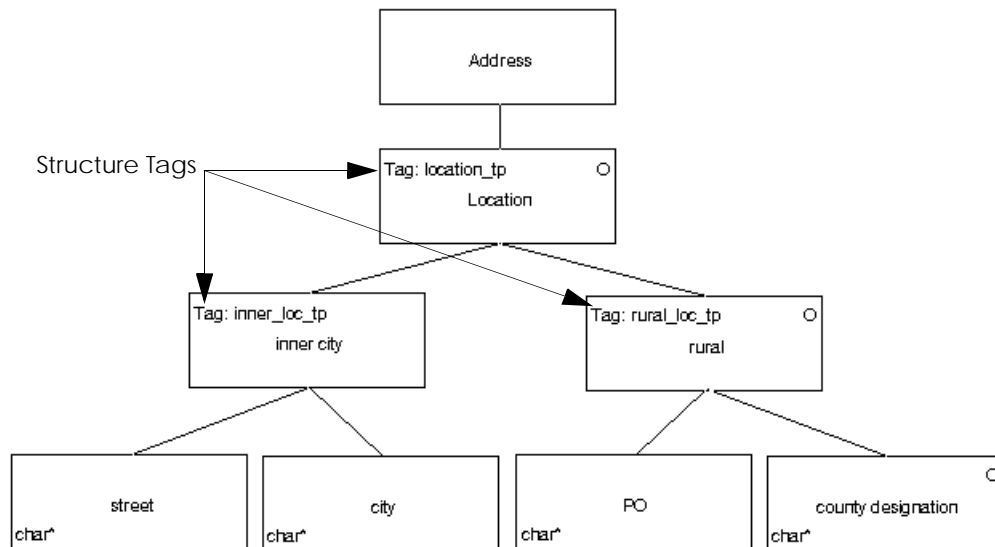
Adding C Structure Tags

If you plan to generate C code from your model, you may need to assign C structure tags to the intermediate nodes in your data structure diagram. The structure tag contains text representing the name for the abstract data type represented by an intermediate node and its children. For more detailed information about generating C code for abstract data types and how structure tags affect the generated code, see “Creation of Data Definitions” on page 10-15.

To add structure tags using the **Properties** dialog, enter the name as text in the **Structure Tag** field, as explained in “Using the Properties Dialog Box” on page 4-17. The word **Tag:**, followed by the text you entered, appears as a display mark in the upper-left corner of the symbol, as shown in Figure 18.

Specifying a structure tag creates a Data Definition annotation note with an item called Tag.

Figure 18: Data Structure Symbols With Structure Tags



Defining a Data Element as a System Type

You can define a data element (a sequence or selection object with no children) as a system type, rather than assigning it a data type. System types are not defined within the scope of the current system. Objects defined as system types do not require a data type annotation.

To define an object as a system type on the **Properties** dialog, leave the **Type** field blank and select the **System Type** option.

For information on how the C code generator handles system types, see “System Types” on page 10-19.

Defining an Abstract Data Type

To define the structure of an object, you can assign it an abstract data type on the **Properties** dialog and define the structure of the abstract data type on an associated data structure diagram. The easiest way to create a type definition diagram is to use the **Type's Definition** command on the **GoTo** menu. The newly created diagram and its root node are automatically given the name of the abstract data type you assigned to the original object. Alternatively, you can create the data type's defining structure in a diagram you create directly, without navigating. The type definition diagram's name and root node label associate it with any data object that has that name as a data type. Figure 19 illustrates the association.

Figure 19: Type Definition Diagram for an Object's Data Type

Diagram: Customer

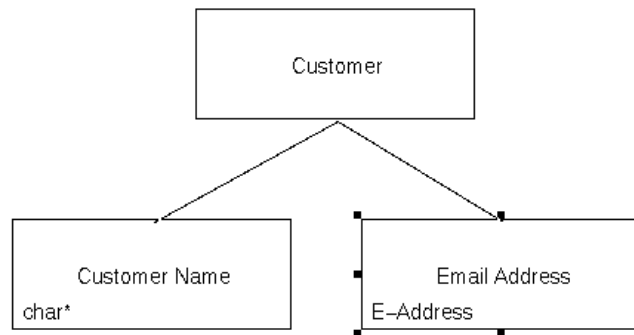
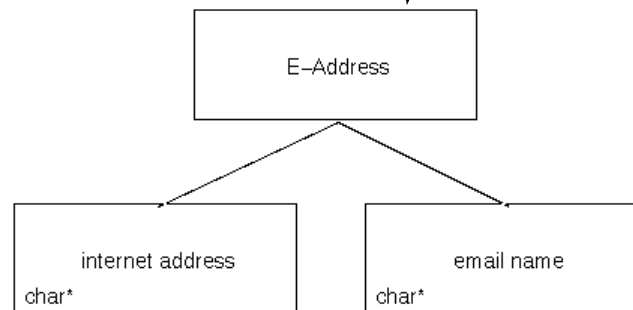


Diagram: E-Address



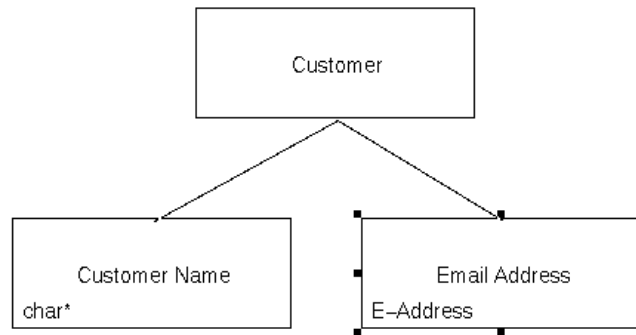
The symbol for the root node in the type's defining data structure is the same symbol (sequence, selection, or enumeration) as the leaf node from which you navigated.

To define an abstract data type using the **Type's Definition** command:

1. Assign a new abstract data type name to a typedef, or to a sequence or selection leaf node (see "Assigning Data Types" on page 4-27).

For example, in Figure 20, the *Email Address* sequence leaf node has been assigned an abstract data type name of *E-Address*.

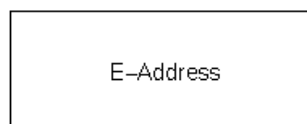
Figure 20: Object Whose Data Type Will Be Defined



2. With the leaf node still selected, from the **GoTo** menu choose **Type's Definition**.
3. In the confirmation dialog, click **Yes** to confirm that you want to create the diagram.

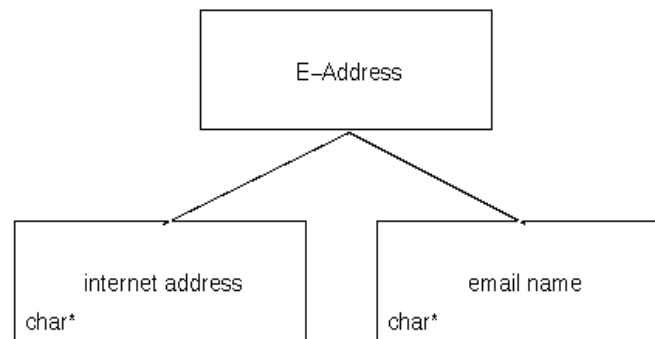
StP/SE creates a new diagram, as shown in Figure 21, with the name of the selected object's data type. The new diagram's root node is represented by the same kind of symbol as the selected node in the original diagram (in this example, a sequence symbol). The root node is labeled with the name of the selected object's data type (in this example, *E-Address*).

Figure 21: Newly Created Type Definition Diagram for *E-Address*



4. Add detail to the new data type structure diagram, as shown in Figure 22.

Figure 22: Detail Added to *E-Address* Data Structure



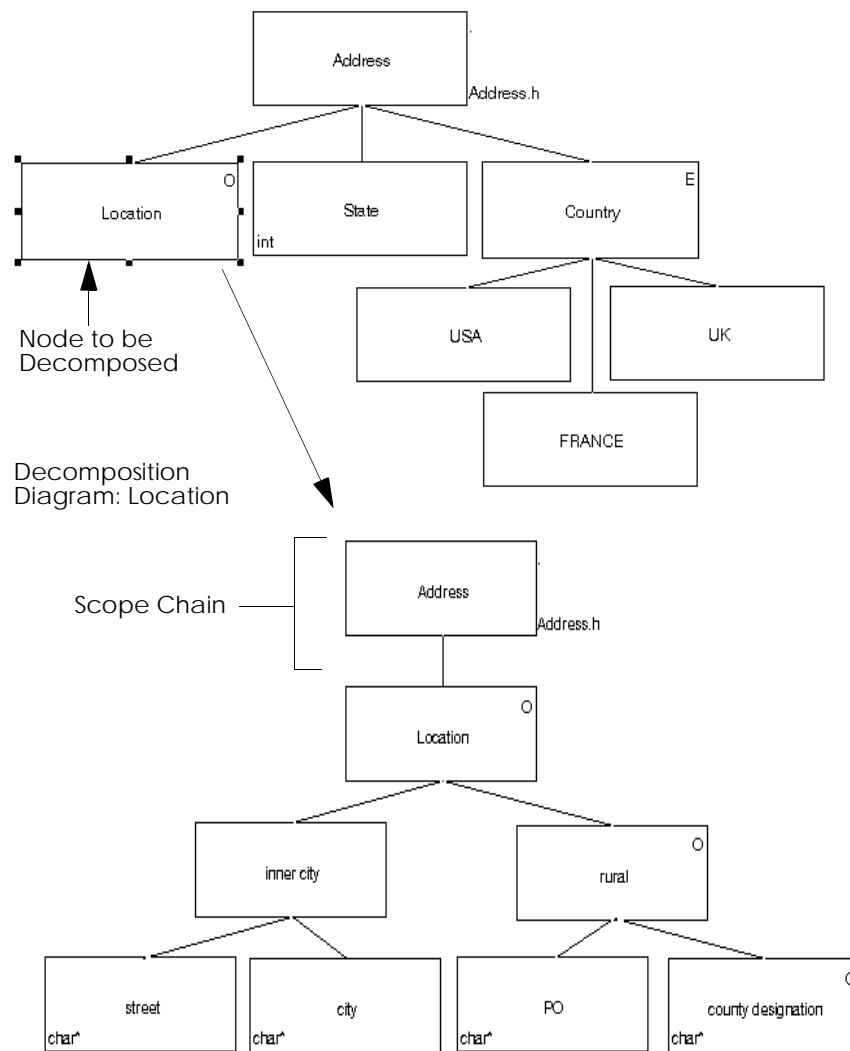
5. From the **File** menu, choose **Save** to store the new diagram.

Decomposing an Object

If your diagram becomes too crowded, you can alleviate some of the congestion by continuing the data structure in another diagram. One way to do this is to decompose an object in order to further define it in a lower-level diagram. Decomposing objects creates a hierarchy of data structure diagrams, in which each lower-level diagram further defines the structure of an object in the parent diagram. The decomposed node in the parent diagram appears as an object to which you add substructure in the decomposition diagram. The decomposed node and its entire scope chain in the parent diagram is copied to the new decomposition diagram, as shown in Figure 23.

Note: In this example, SEDirectory and SEFile display marks have been turned on so that the scope specifications are visible.

Figure 23: Decomposition of a Leaf Node



Only sequence and selection objects that are undefined leaf nodes in the current diagram can be decomposed. The object to be decomposed should not have a data type. If it does, you must remove the data type

annotation either before or after decomposing the object in order for the model to be semantically correct.

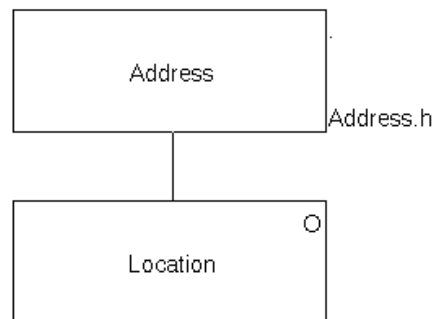
Note: If you need to rename the parent object, use the **Rename Object Systemwide** command to do so, as described in “Renaming Objects Systemwide” on page 13-8. This preserves the relationship between the parent and its decomposition.

To create a decomposition diagram:

1. Select an undefined leaf node without a data type, which you will decompose.
2. From the **GoTo** menu, choose **Scoped Decomposition**.
3. In the confirmation dialog, click **Yes** to confirm that you want to create the diagram.

StP/SE creates the decomposition diagram, which contains a copy of the entire scope chain of the selected object in the parent diagram, as shown in Figure 24.

Figure 24: Decomposition with Original Object's Scope Chain



4. Add detail to the structure in the decomposition diagram to further define the object.
5. From the **File** menu, choose **Save**.

Navigating to a Parent

To navigate from an abstract data type definition to a data structure containing an object assigned that data type:

1. Select the root object in the abstract data type definition diagram.
2. From the **GoTo** menu, choose **Parent**.

A list of diagrams in which the data type is used appears.

3. Select the diagram to which you want to navigate and click **OK**.

The diagram appears, with the object selected whose data type is defined in the data type definition diagram.

To navigate to the parent data structure from a decomposition of a data object:

1. Select a decomposed object in the decomposition diagram.
2. From the **GoTo** menu, choose **Parent**.

The higher-level object that is connected by a component link to the selected object appears in the same or a higher-level diagram.

Showing All Children

You can use the **Show All Children** command on the **DSE** menu to:

- Show all children of a decomposed parent symbol on the same diagram as the parent
- Combine on one diagram all children of a symbol that is the root node in more than one diagram

If you select a symbol that has decompositions, the **Show All Children** command draws on the current diagram all children of the selected symbol. The decompositions can be from any diagram or diagrams in the system.

If the selected node is a root node that also appears as the root node on any other diagrams, the **Show All Children** command draws on the current diagram all first-level children of the matching root nodes. The root nodes must have the same directory and file specification.

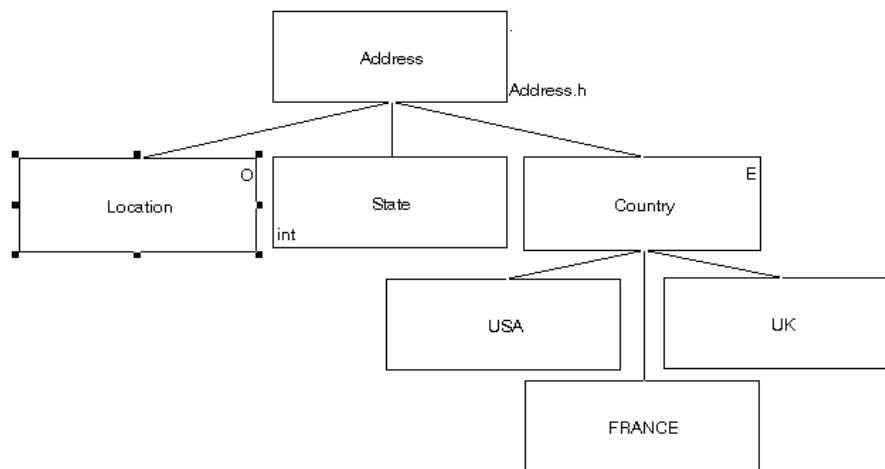
Show All Children draws only the first level children of the selected object. For nodes whose children have children, you can repeatedly choose **Show All Children** to draw each child node's children in the parent diagram as well.

To show all children on the current diagram:

1. Select a symbol whose child nodes are defined in other diagrams, and whose definitions you want to appear on the current diagram.

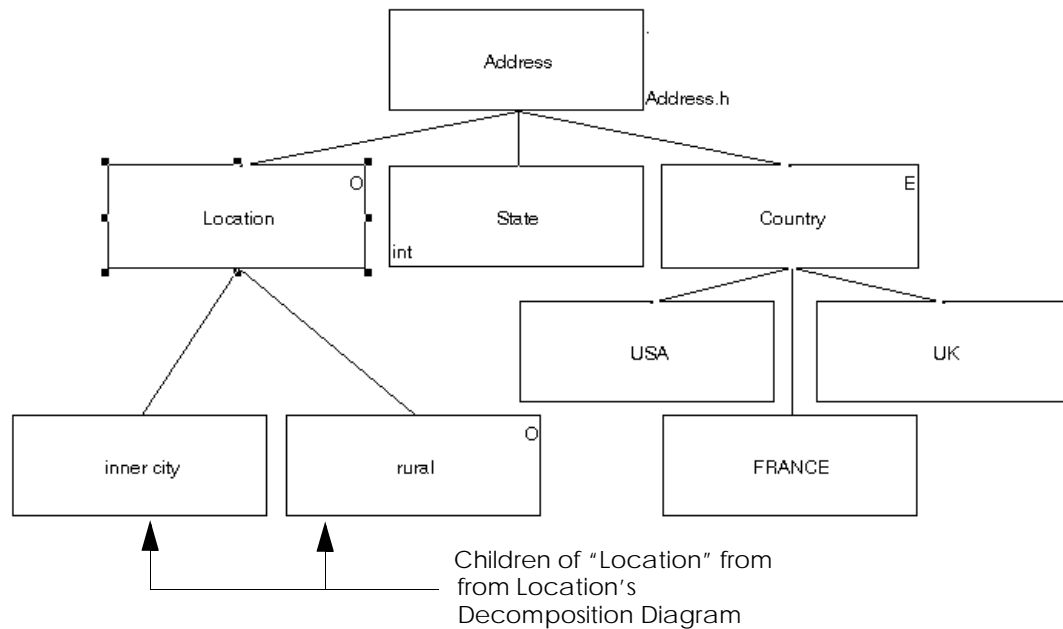
In this example, the symbol *Location* is selected on the *Address* diagram, as shown in Figure 25. *Location*'s children are defined in the decomposition diagram shown in Figure 23 on page 4-34.

Figure 25: Showing All Children



2. From the **DSE** menu, choose **Show All Children**.
All children of the symbol *Location* are drawn on the *Address* diagram, as shown in Figure 26.

Figure 26: Children Shown on Decomposition's Parent



Generating a BNF File

You can generate a Backus-Naur format (BNF) file for:

- The current diagram in the Data Structure Editor
- One or more selected diagrams in the Desktop list
- All data structure diagrams in your entire system model at once

The default output location for the generated files is the *dse_files* directory in the current project and system. The filename for each generated BNF file is *<diagram_name>.bnf*.

Generating a BNF File for the Current Diagram

To generate a BNF file for the current diagram in the Data Structure Editor, choose **Generate BNF for Diagram** from the **DSE** menu. The file is created in the default directory, which is set in the **Generate BNF** dialog box (see Figure 27).

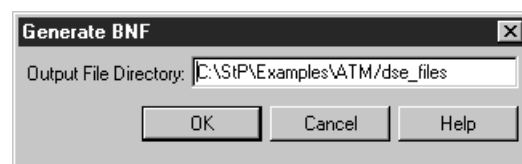
Creating BNF Files for Selected Diagrams or Entire Model

To generate BNF files for selected diagrams or the entire model from the Desktop:

1. In the Model pane, open the **Diagrams** category and select **Data Structure**.
2. If you want to create BNF files for certain diagrams only, select the diagrams in the objects pane.
3. From the **Code** menu, choose **BNF**.
4. From the **BNF** menu, choose either:
Generate BNF—To create BNF files for selected diagrams
Generate BNF for All Diagrams—To create BNF files for all diagrams in your entire model

The **Generate BNF** or **Generate BNF for All Diagrams** dialog box appears.

Figure 27: Generate BNF Dialog Box



5. If you want to change the default output directory, edit the contents of the **Output File Directory** field.
6. Click **OK** or **Apply** to generate the BNF file(s).

Generating BNF Files with Script Manager

If you prefer, you can use the StP Script Manager to generate a BNF report in a selected publishing format, such as RTF or FrameMaker. You can execute one of the StP-provided scripts, *all_bnf* or *bnf*, or define a new script using StP's Query and Reporting Language (QRL).

See *Query and Reporting System* for more information on running scripts in the Script Manager, or on modifying scripts using QRL.

Validating a Data Structure Diagram

StP/SE editors provide two kinds of validation checks for the diagrams that comprise your system model:

- Syntax checks
- Semantic checks

This section describes how to check syntax and semantics for the current diagram in the Data Structure Editor. You can also check semantics for the entire model or for one or more selected diagrams from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, "Checking SE Models."

Syntax Checks

A data structure diagram is syntactically complete and correctly drawn if:

- All sequence, selection, enumeration, and typedef symbols are labeled.
- All link connection rules are followed.
- All enumerations have at least one sequence child.
- Children of an enumeration do not have children or data types.

Checking diagram syntax does not check the contents of the repository.

Syntax checks are applied automatically when you save the current diagram, or you can choose the **Check Syntax** command on the **Tools** menu.

Semantic Checks

A diagram is semantically correct if all objects are properly defined in the repository and all elements are properly balanced. For a complete list of data structure diagram semantic checks, see “Data Structure Editor Checks” on page 9-6.

To invoke semantic checking for the current diagram, choose either of the following commands from the **Tools** menu:

- **Check Semantics**—In addition to displaying results in the Message Log, optionally allows you to save results to a file in a selected format
- **Check Semantics Selectively**—Allows you to apply only selected semantic checks

5 **Creating State Transition Diagrams**

In support of real-time extensions to structured analysis, StP/SE provides the State Transition Editor (STE) for drawing state transition diagrams. State transition diagrams are widely used in hardware and software design to show the relationships between states, events, and actions. State transition diagrams show the events that cause a transition from one state to another, and the actions performed in connection with the transitions.

The State Transition Editor can be used either as an independent tool or in connection with the Control Specification Editor and control flows in the Data Flow Editor. The Control Specification Editor uses the information in state transition diagrams to produce tabular output for several control specification tables.

This chapter describes:

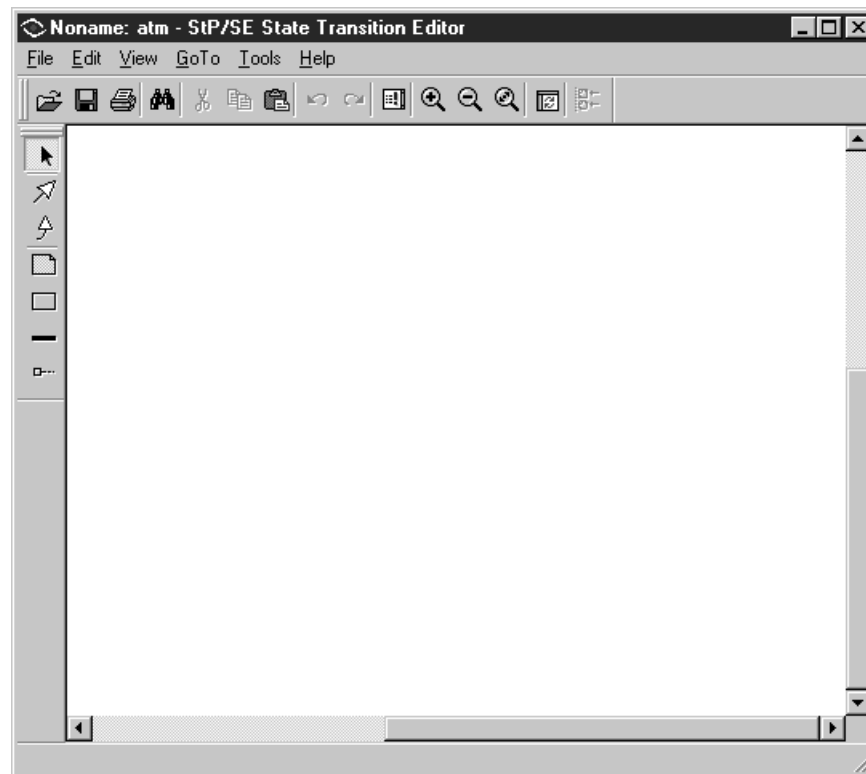
- “Using the State Transition Editor” on page 5-2
- “Creating a State Transition Diagram” on page 5-7
- “Using Filters to Hide or Show Event/Action Bars” on page 5-13
- “Creating Associated Cspec Tables” on page 5-14
- “Validating State Transition Diagrams” on page 5-15

This chapter provides a brief explanation of state transition notation. For complete details on the notation of this diagramming technique, refer to *Strategies for Real-Time System Specification* by Derek J. Hatley and Imtiaz A. Pirbhai (Dorset House, New York, NY, 1988).

Using the State Transition Editor

The State Transition Editor (Figure 1) is an interactive graphical tool for drawing state transition diagrams in support of real-time extensions to structured analysis.

Figure 1: State Transition Editor window



The State Transition Editor provides the standard StP diagram editor functions and menu options. For general information about using diagram editors, see *Fundamentals of StP*.

In addition to the standard StP diagram editor features, the State Transition Editor provides:

- Symbols for drawing state transition diagrams
- Navigations to related diagrams and tables

These features are described briefly in this section. For more information about using each of these features to create or edit a state transition diagram, see “Creating a State Transition Diagram” on page 5-7.

Starting the State Transition Editor

StP/SE provides access to the State Transition Editor from:

- The StP Desktop
- Control Specification Editor
- Data Flow Editor

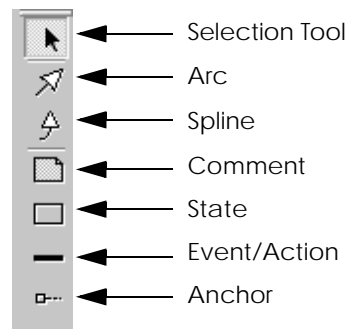
For information about starting the State Transition Editor from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

You can start the State Transition Editor from the Data Flow Editor or Control Specification Editor by using an appropriate navigation command from that editor’s **GoTo** menu. If the State Transition Editor has already been started, the navigation command uses the current session, rather than starting another copy of the editor. For information about starting the State Transition Editor from the Data Flow Editor or Control Specification Editor, see the chapter describing that editor.

Using the State Transition Editor Symbols

You select symbols for drawing state transition diagrams from the Symbols toolbar (Figure 2).

Figure 2: State Transition Editor Symbols Toolbar



Procedures for using the symbols are described in this chapter. For a summary description of each symbol, see Appendix B, “StP/SE Symbol Reference.” For general instructions on using the Symbols toolbar, see *Fundamentals of StP*.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object whose symbol is selected on the current diagram. Use the **GoTo** menu to navigate. In the State Transition Editor, the commands on the **GoTo** menu allow you to:

- Display the related data flow diagram
- Create or display one of the following Cspec tables in the Control Specification Editor:
 - Action Logic Table (ALT)
 - Event Logic Table (ELT)
 - State Event Matrix (SEM)
 - State Transition Table (STT)

Navigation starts another editor from a current editor session. When you navigate to another StP/SE editor, the current session continues. You can navigate to these StP editors from the State Transition Editor:

- Control Specification Editor
- Data Flow Editor
- Requirements Table Editor

The **GoTo** menu provides context-sensitive choices for the selected symbol.

To navigate to a target:

1. Select a symbol on the current diagram.
2. Choose a command from the **GoTo** menu.

The navigation target appears.

If there is more than one possible target, a selection list appears from which you can choose the appropriate target. If a Cspec table or matrix does not exist, you are given the option to create it. The Cspec table or matrix is automatically populated with information from the state transition diagram.

Table 1 describes the navigation targets and commands available for each State Transition Editor symbol.

Table 1: GoTo Menu Commands

Navigate From	Navigate To	Command
Any STE symbol	The related diagram in the Data Flow Editor	Data Flow Diagram
	The related Cspec bar in a Data Flow Editor diagram	Cspec Bar
	Displays the state event matrix in the Control Specification Editor that defines the selected symbol	State Event Matrix
	Displays the state transition table in the Control Specification Editor that defines the selected symbol	State Transition Table
State	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements
Event/Action bar	Displays the action logic table in the Control Specification Editor that defines the selected action	Action in Action Logic Table
	Displays the event logic table in the Control Specification Editor that defines the selected event	Event in Event Logic Table
	Requirements table (see “Allocating Requirements to a Model” on page 1-10)	Allocate Requirements

Creating a State Transition Diagram

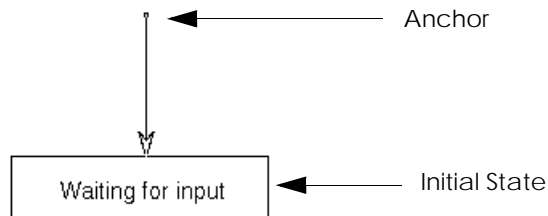
A state transition diagram consists of an anchor point, state symbols, and event/action bars. The basic rules of state transition diagrams are:

- An event (input signal) causes a transition from one state to another.
- An action (output signal) is performed in connection with the transition.

Representing the Initial State

The initial state describes the state of your system before it is affected by events or actions. It consists of a state symbol connected to an anchor symbol, as shown in Figure 3. There can be only one initial state in each diagram.

Figure 3: Initial State Symbol



Creating the Initial State by Navigating

If you create the state transition diagram by navigating from the Data Flow Editor, StP draws the anchor and initial state symbol for you. This establishes an automatic link between the data flow diagram and the state transition diagram.

To create the state transition diagram and the initial state by navigating from the Data Flow Editor:

1. In the Data Flow Editor, select the Cspec bar in a data flow diagram.
2. From the **GoTo** menu, choose **State Transition Diagram**.

3. In the confirmation box, click **Yes** to confirm that you want to create the diagram.

The State Transition Editor starts. An anchor connected to an unlabeled initial state symbol appears in the drawing area (see Figure 3).

4. Label the initial state symbol.

Drawing the Initial State Manually

Alternatively, you can begin a state transition diagram by drawing the initial state manually in the State Transition Editor, as follows:

1. In the State Transition Editor, insert an anchor point into the blank drawing area.
2. Insert a state symbol below the anchor point.
3. Draw an arc from the anchor point to the state symbol.

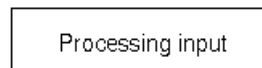
An arc with an anchor point as the origin designates the symbol to which it points as the initial state.

4. Label the initial state symbol.

Representing States and Transitions

States represented by the state symbol, describe the status of the system before, after, and between real-time events. Figure 4 contains an example of a state symbol. Its label is a string of any length that describes the status of the system. State symbols must be unique to a single diagram; StP/SE does not allow other references to the state in another state transition diagram.

Figure 4: State Symbol



When the system is in any given state, events (combinations of input signal values) cause the system to do either or both of the following:

- Change its state (a transition)
- Produce actions (combinations of output signal values)

For each transition, there is an event that causes the transition and, optionally, an action associated with that transition. Events and actions are described in “Representing Events and Actions” on page 5-10.

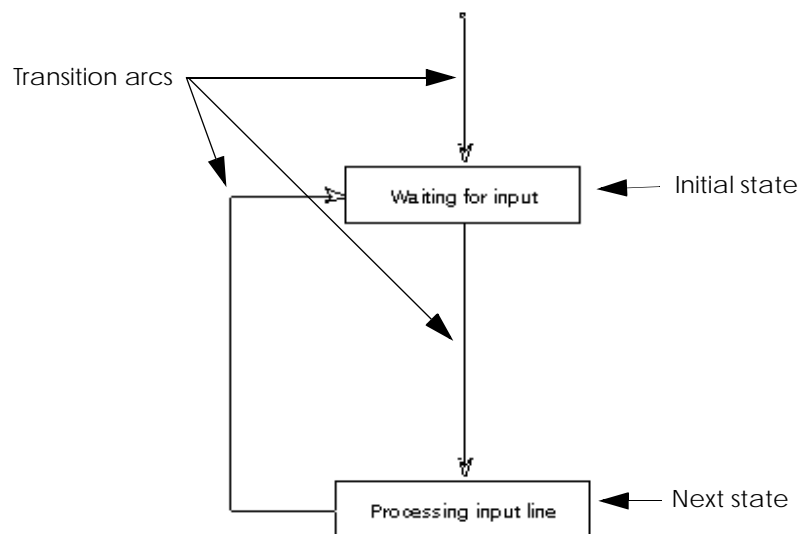
In the State Transition Editor, an arc represents a transition between states, as shown in Figure 5. The arcs are directional and must link the states in the order in which they occur, from the current state to the next state. No two transitions can have the same start state and no transition can be referenced more than once.

To draw states and transitions:

1. Insert state symbols into the diagram as needed.
2. Label the state symbols.
3. Draw connecting transition arcs from each state to the next state.

Figure 5 illustrates state symbols and transition arcs.

Figure 5: Drawing States and Transitions



Representing Events and Actions

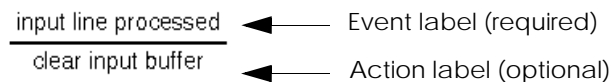
Events are combinations of input signal values that cause the system to:

- Change its state (a transition)
- Produce actions

Actions are combinations of output signal values that activate the next state. For each transition, there is an event that causes the transition and, optionally, an action associated with that transition.

Events and actions are represented by horizontal event/action bars, as shown in Figure 6. The text above the bar describes the event. The text below the bar describes the action. An event label is required; an action label is optional.

Figure 6: Event/Action Bar



Events and actions are associated with transitions between states. No two transitions can have the same event.

To associate events and actions with transitions:

1. Insert an event/action bar and drop it on a transition arc.
This establishes the link between the event/action bar and the arc.
2. Insert the event/action bar off the arc.
3. Place the cursor slightly above the event/action bar, and label the event (required).
4. Place the cursor slightly below the event/action bar, and label the action (optional).
5. Repeat this process for each transition arc.

Figure 7 illustrates event/action bars.

Figure 7: Adding Event/Action Bars

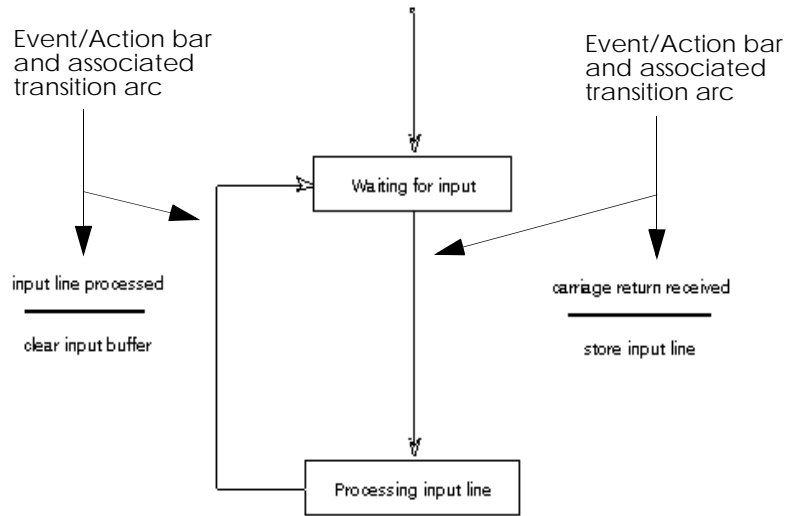
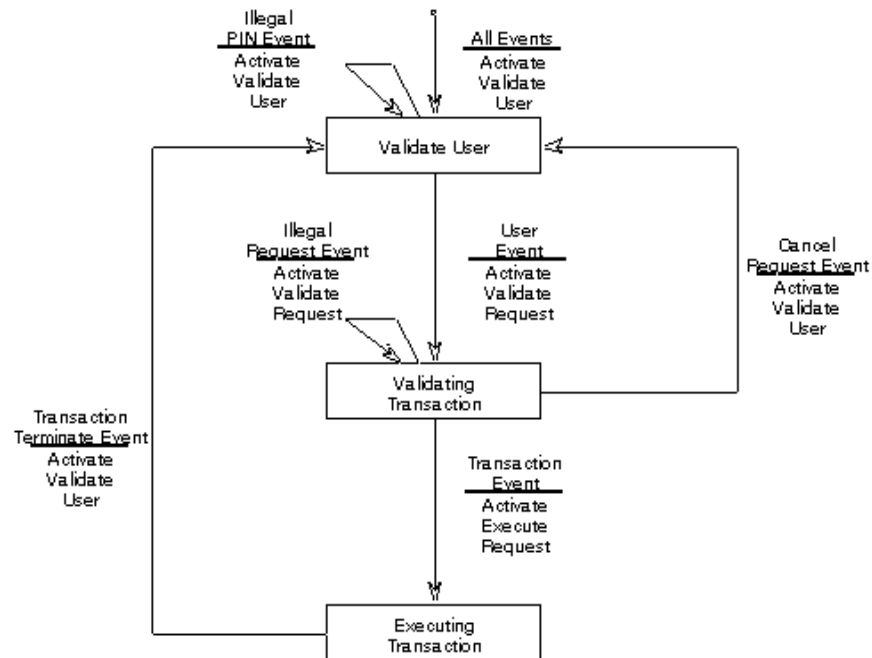


Figure 8 shows an example of a completed state transition diagram, with several event/action bars and transitions.

Figure 8: Completed State Transition Diagram



Moving Event/Action Bars

If you move a transition arc to which an event/action bar is attached, the event/action bar and its labels move with the arc. If the bar doesn't move with the arc, it is not attached.

To attach an event/action bar to a new transition arc:

1. Select the event/action bar to be moved.
2. Insert the event/action bar to the new arc.
3. Select both the event/action bar and the new arc (select each object with the left mouse button while pressing the Shift key).
4. From the **Tools** menu, choose **Attach to Arc**.

Annotating the Diagram

You can use the Object Annotation Editor (OAE) to add a single global annotation to each diagram and to annotate the states and transitions in the diagram. Use the **On Selection** option on the OAE **Help** menu to provide a description of available annotations. For information about the OAE, see *Fundamentals of StP*.

Using Filters to Hide or Show Event/Action Bars

The State Transition Editor allows you to display a state transition diagram with or without the event/action bars, using the following filters:

- HideEventAction
- ShowEventAction

To hide or show event/action bars:

1. From the **View** menu, choose **Apply Filter**.
2. On the **Apply Filter** dialog:
 - Make sure the **Location** group is set to either **StP Installation** or **Both**.
 - Select **Editor Specific Filters** to display a list of filters specific to data flow diagrams.
3. Select one of the editor-specific filters.
4. Click **OK**.

StP redisplay the diagram with the event/action bars either hidden or visible, accordingly.

For more information about using filters, see *Fundamentals of StP*.

Creating Associated Cspec Tables

StP/SE uses the information in state transition diagrams to create the following control specification (Cspec) tables:

- Action logic table (ALT)—required for actions
- Event logic table (ELT)—required for events
- State event matrix (SEM)
- State transition table (STT)

In order for a state transition diagram to be semantically correct, you must create at least the action logic table and the event logic table for actions and events, respectively. The other tables are optional.

To create an associated Cspec table:

1. Select a state, event/action bar, or transition arc on the state transition diagram.
2. From the **GoTo** menu, choose a command for the type of table you want to create (for descriptions of navigation commands, see “Navigating to Object References” on page 5-4).
3. In the confirmation box, click **Yes** to confirm that you want to create the table.

StP/SE starts the Control Specification Editor (CSE) and creates and automatically populates the table with information from the state transition diagram. The table assumes the same name as the state transition diagram.

4. From the **File** menu, choose **Save** to save the new table.

For more information about Cspec tables, see Chapter 6, “Creating Control Specifications.”

Validating State Transition Diagrams

StP/SE editors provide two kinds of validation checks for the diagrams that comprise your system model:

- Syntax checks
- Semantic checks

This section describes how to check syntax and semantics for the current diagram in the State Transition Editor. You can also check semantics for the entire model or for one or more selected diagrams from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, “Checking SE Models.”

Syntax Checks

A state transition diagram is syntactically complete and correctly drawn if:

- All states are labeled.
- All event/action bars are labeled with events.
- Each state is the target of (reachable by) at least one transition.
- There are no unconnected anchor points.

Checking diagram syntax does *not* check the contents of the repository.

Syntax checks are applied automatically when you save the current diagram, or you can choose the **Check Syntax** command on the **Tools** menu.

Semantic Checks

A state transition diagram is semantically correct if symbols are properly used, transitions are correctly connected to states and associated with events and actions, and events and actions are defined in the related Cspec tables. For a complete list of state transition diagram semantic checks, see “State Transition Editor Checks” on page 9-6.

Creating State Transition Diagrams

For information on checking Cspec semantics, see “Validating a Cspec” on page 6-30.

To invoke semantic checking for the current diagram, choose either of the following commands from the **Tools** menu:

- **Check Semantics**—In addition to displaying results in the Message Log, optionally allows you to save results to a file in a selected format
- **Check Semantics Selectively**—Allows you to apply only selected semantic checks

6 Creating Control Specifications

StP/SE provides the Control Specification Editor (CSE) for creating and editing control specification (Cspec) tables associated with the real-time extensions for structured analysis. Cspec tables define values associated with the control flows in data flow diagrams and the states and events in state transition diagrams.

This chapter describes:

- “Anatomy of a Cspec Table” on page 6-2
- “Using the Control Specification Editor” on page 6-8
- “Creating and Editing Cspec Tables” on page 6-15
- “Switching Between Cspec Tables” on page 6-21
- “Table Descriptions” on page 6-22
- “Validating a Cspec” on page 6-30

This chapter provides a brief explanation of control specifications. For complete details on this specification technique, refer to *Strategies for Real-Time System Specification* by Derek J. Hatley and Imtiaz A. Pirbhai (Dorset House, New York, NY, 1988).

Anatomy of a Cspec Table

A logical control specification (Cspec) consists of a state transition diagram and multiple physical tables and matrixes that specify values for the elements of the control specification. Each Cspec table or matrix contains information about the control flows in the parent control flow diagram and the states and events in the state transition diagram.

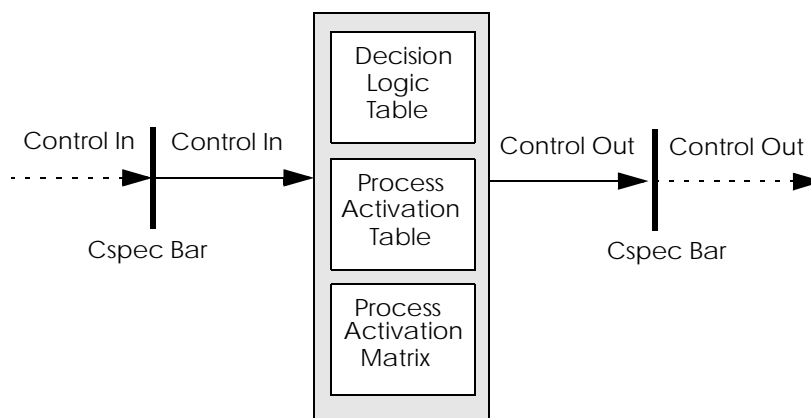
A Cspec consists of one or more of the following diagrams and individual physical tables and/or matrixes:

- State Transition Editor (STE) diagram
- Decision table (DET)
- Process activation table (PAT)
- Process activation matrix (PAM)
- State transition table (STT)
- State event matrix (SEM)
- Event logic table (ELT)
- Action logic table (ALT)

Flow within the Cspec

Figure 1 and Figure 2 show the general flow of control signals in a Cspec. These examples are provided as a basis for understanding this chapter. For methodological details, consult the reference suggested in the introduction to this chapter.

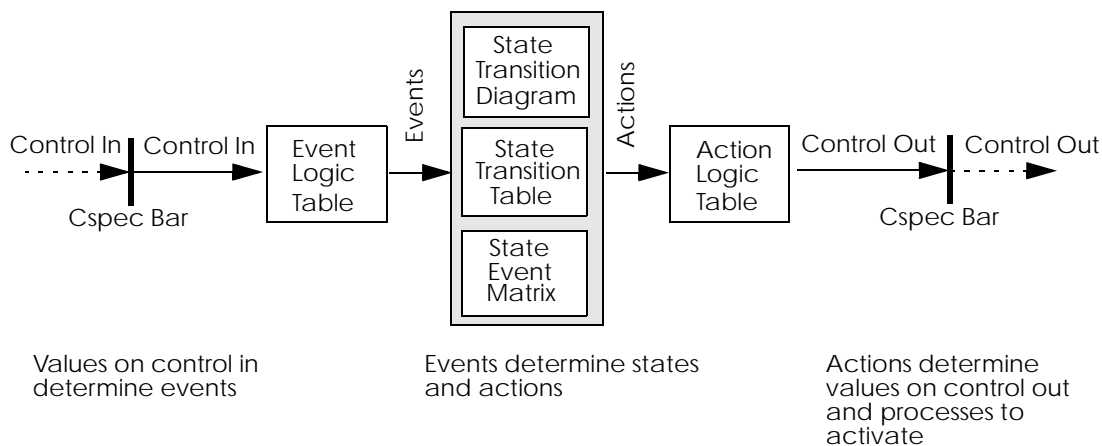
Figure 1: Control Signal Flow for Combinational Logic



For the decision table, the value on control in determines the value of control out.

For the process activation table, the value on control in determines the process activation.

Figure 2: Control Signal Flow for Sequential Logic



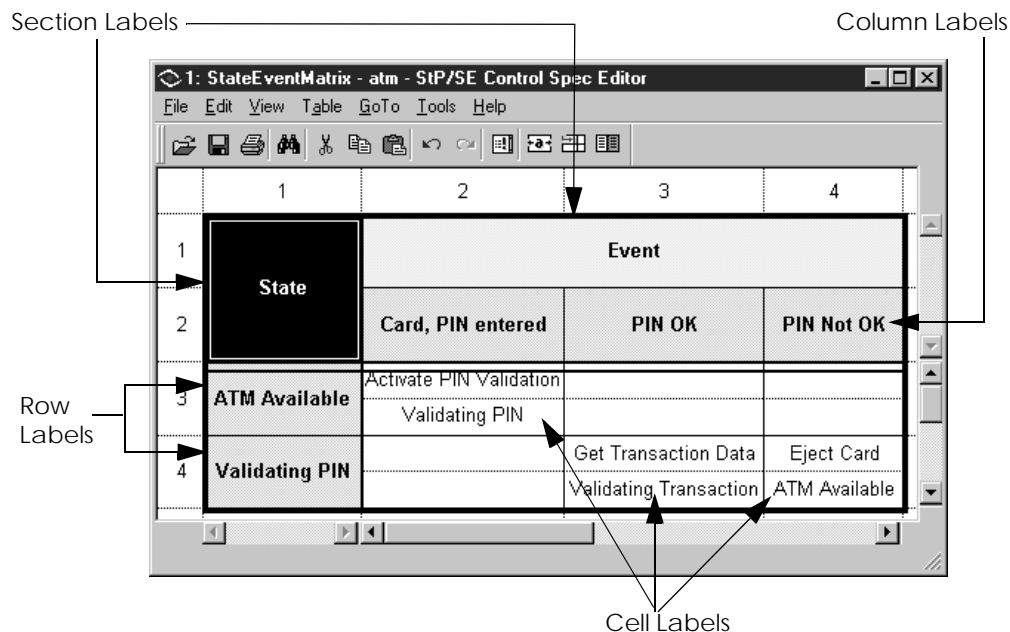
Overview of Table Elements

Each table is divided into a certain number of vertical and horizontal sections containing cells into which information is entered either manually by the user or automatically by StP/SE. The exact arrangement of cells and their allowed contents vary, according to the table type. The basic components of a Cspec table are:

- Section labels, which organize the table cells into groups of similar elements
- Column and row labels, which contain the names of control flows, processes, actions, events, and states related to the Cspec
- Cell labels, which contain control values, names of structured analysis objects such as actions, events, and states, or data such as process activations and event definitions

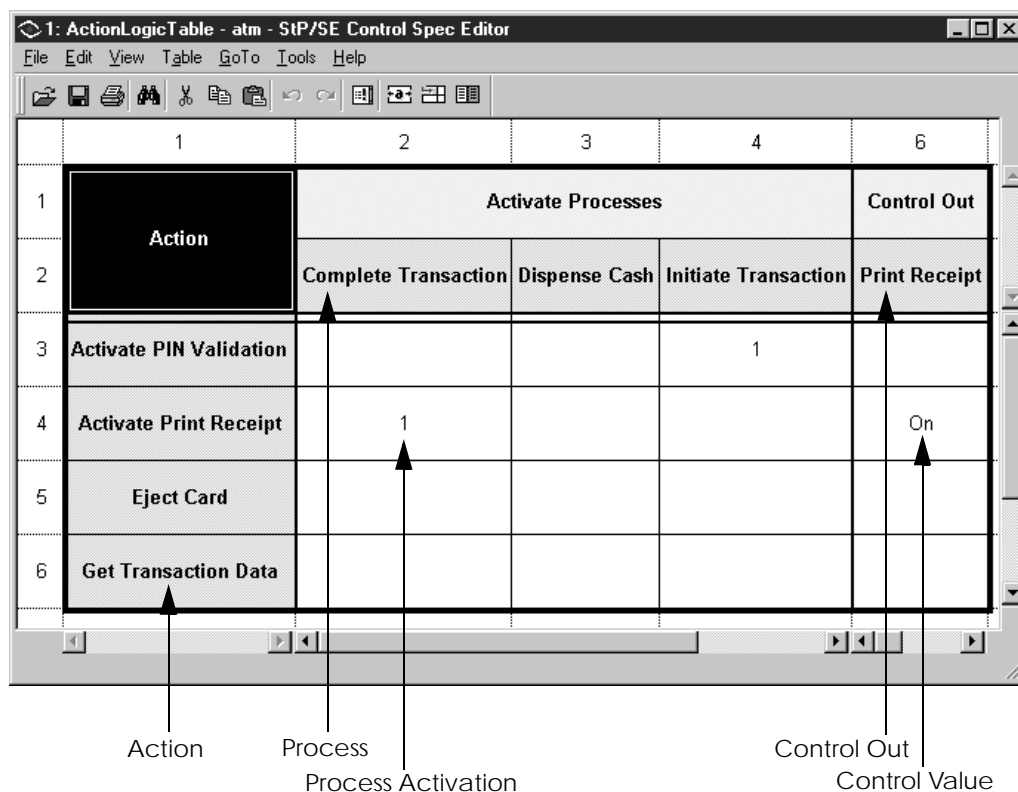
Figure 3 illustrates the basic table components in a state event matrix.

Figure 3: Elements of a Cspec Table



Column, row, and cell elements are specific to each table. For example, Figure 4 shows the column label, row label, and table cell elements for an action logic table.

Figure 4: Control Specification Editor Window - Action Logic Table



	1	2	3	4	6
1	Action	Complete Transaction	Dispense Cash	Initiate Transaction	Print Receipt
2	Activate PIN Validation			1	
3	Activate Print Receipt	1			On
4	Eject Card				
5	Get Transaction Data				
6					

Labels below the table with arrows pointing to specific cells:

- Action: points to cell (1,1)
- Process: points to cell (2,1)
- Process Activation: points to cell (3,1)
- Control Out: points to cell (5,1)
- Control Value: points to cell (6,1)

Table 1 describes the table elements that appear in each table type. For an example of each table, see “Table Descriptions” on page 6-22.

Creating Control Specifications

Table 1: Control Specification Table Elements

Element	Description	Control Specification Table						
		A L T	D E T	E L T	P A M	P A T	S E M	S T T
Action	Name of an action relating to this Cspec that activates the next state	✓					✓	✓
Control Combination	Combination of two sets of these elements comprising an entire row of values: - control in or out values + process activations - control in values + control out values - control in values + event definitions	✓	✓	✓		✓		
Control In	Name of a control input signal flowing into the Cspec bar on the related data flow diagram		✓	✓	✓	✓		
Control Out	Name of a control output signal flowing out of the Cspec bar on the related data flow diagram	✓	✓					
Control Value	User-specified value in <i>Allowed Value</i> item on Data Definition note for the control in or control out flow definition in a data structure diagram	✓	✓	✓		✓		
Event	Name of an event relating to this Cspec that initiates a transition between states			✓			✓	✓
Event Definition	User-specified value indicating whether or not the event occurs: 1 = Event recognized (no entry) = Event not recognized			✓				
Process	Name of a process on the related data flow diagram that is activated	✓			✓	✓		

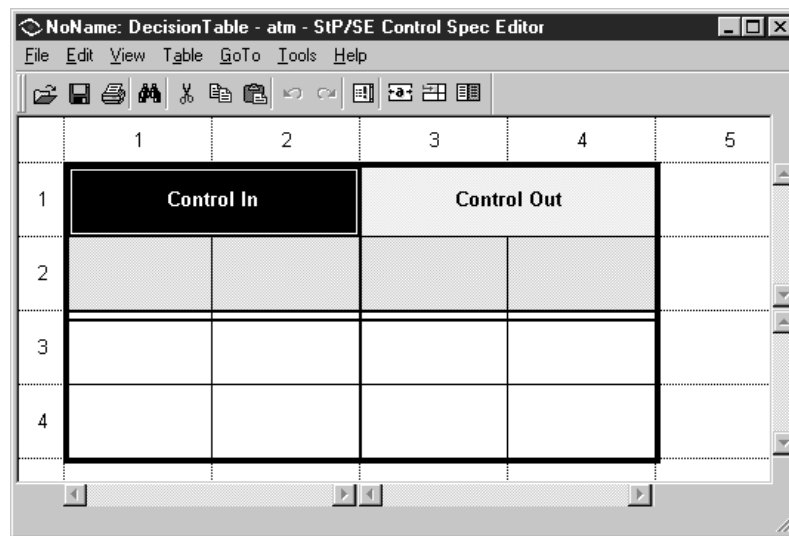
Table 1: Control Specification Table Elements (Continued)

Element	Description	Control Specification Table						
		A L T	D E T	E L T	P A M	P A T	S E M	S T T
Process Activation	User-specified value indicating whether, and in what sequence, process is activated: 0 = Disable the process T = Trigger the process 1, 2 ... n = Enable process in order indicated (no entry) = Do nothing	✓			✓	✓		
Process Combination	Set of processes corresponding to an entire table row	✓			✓			
State	Name of a state relating to this Cspec that describes the status of the system						✓	✓

Using the Control Specification Editor

The Control Specification Editor (shown in Figure 5 with a decision table), is a table editor used for creating and editing the various Cspec tables.

Figure 5: Control Specification Editor Window - Decision Table



Although the Control Specification Editor window is similar to that of other StP table editors, the exact appearance and functions available depend on the type of table you are editing (see "Table Descriptions" on page 6-22 for examples of each Cspec table). The Control Specification Editor provides the standard StP table editor menu options. For general information about table editors, see *Fundamentals of StP*.

The component physical tables and matrixes of a Cspec are treated as one entity when you copy, rename, or delete an entire Cspec from the StP Desktop **File** menu. To apply these commands to an entire Cspec, select the **All Control Spec** category in the Model pane and a Cspec in the objects pane before invoking the command. All of the Cspec's individual physical tables and matrixes are copied, renamed, or deleted at once. You

can also copy, rename, or delete individual Cspec tables by selecting the table's name on the StP Desktop objects pane for a specific table tool, such as Decision Tables, and choosing a command from the **File** menu.

In addition to the standard StP table editor features, the Control Specification Editor provides:

- Navigations to related tables and diagrams
- Commands on the **Edit** and **Tools** menus that are specific to the Control Specification Editor

These features are described briefly in this section. For information about using these features, see “Creating and Editing Cspec Tables” on page 6-15.

Starting the Control Specification Editor

StP/SE provides access to the Control Specification Editor from the:

- StP Desktop
- Data Flow Editor
- State Transition Editor

For information about starting the Control Specification Editor from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

You can start the Control Specification Editor from the Data Flow Editor or State Transition Editor by using an appropriate navigation command from that editor's **GoTo** menu. If the Control Specification Editor has already been started, the navigation command uses the current session, rather than starting another copy of the editor. For information about starting the Control Specification Editor from the Data Flow Editor or State Transition Editor, see the chapter describing that editor.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object in the repository. Use the **GoTo** menu to navigate. In CSE, the commands on the **GoTo** menu allow you to display diagrams, or symbols on diagrams, that are related to the selected cell in a Cspec table.

Navigation may start another editor from a current editor session. When you navigate to another StP/SE editor, the current session continues.

You can navigate to these editors from the Control Specification Editor:

- Data Flow Editor
- Data Structure Editor
- State Transition Editor

The **GoTo** menu provides context-sensitive choices for the selected table cell.

To navigate to a target:

1. Select an appropriate cell in the current table.
2. Choose a command from the **GoTo** menu.

The navigation target appears.

Table 2 through Table 8 provide a summary of navigations for the Control Specification Editor. Each navigation table describes the navigations available from a particular type of Cspec table.

When navigating from the Control Specification Editor to the Data Structure Editor, if the target diagram does not exist, you are given the option to create it. The new data structure diagram has the same name as the existing control in or control out from which you are navigating.

Table 2: Navigations from Decision Table

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
Control In Control Out	Related symbol on a data structure diagram	Data Definition

Table 3: Navigations from Process Activation Table

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
Process	Reference to process on the data flow diagram	Process on Data Flow Diagram
Control In Control Out	Related symbol on a data structure diagram	Data Definition

Table 4: Navigations from Process Activation Matrix

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
Process	Reference to process on data flow diagram	Process on Data Flow Diagram
Control In	Related symbol on a data structure diagram	Data Definition

Table 5: Navigations from State Transition Table

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
	Related state transition diagram	State Transition Diagram
Current State	Reference to the state on a state transition diagram	State on State Transition Diagram

Table 6: Navigations from State Event Matrix

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
	Related state transition diagram	State Transition Diagram
State	Reference to the state on a state transition diagram	State on State Transition Diagram

Table 7: Navigations from Event Logic Table

Navigate From	Navigate To	GoTo Menu Command
Any table cell	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
Event	Related state transition diagram	State Transition Diagram
Control In	Displays the related symbol on a data structure diagram	Data Definition

Table 8: Navigations from Action Logic Table

Navigate From	Navigate To	GoTo Menu Command
Any	Cspec bar on the data flow diagram	Cspec Bar on Data Flow Diagram
Action	Related state transition diagram	State Transition Diagram
Process Activation	Reference to the process on a data flow diagram	Process on Data Flow Diagram
Control Out	Related symbol on a data structure diagram	Data Definition

Using Editor-Specific Menu Commands

In addition to the standard table menu commands described in *Fundamentals of StP*, the Control Specification Editor provides editor-specific commands on the **Edit** and **Tools** menus. Table 9 describes these special commands.

Table 9: Editor-Specific Commands

Menu	Command	Description	For Details, See
Edit	Edit Cspec Group Annotation	Activates the Object Annotation Editor for creating or editing an annotation for the Cspec.	<i>Fundamentals of StP</i>
	Table Type	Provides a means of selecting and displaying the other tables for the current Cspec in the current Control Specification Editor session.	“Switching Between Cspec Tables” on page 6-21
Tools	Set Column Values from Data Definition	Automatically populates the control value cells in a control in or control out column with values from all Allowed Value annotations on the data structure object that defines the control flow in the selected column label.	“Setting Control Values” on page 6-20

Creating and Editing Cspec Tables

You can navigate from the Data Flow Editor or State Transition Editor to create these Cspec tables from information in related diagrams:

- From a data flow diagram you can create any Cspec table or matrix.
- From a state transition diagram you can create an:
 - Action logic table (ALT)
 - Event logic table (ELT)
 - State event matrix (SEM)
 - State transition table (STT)

When you create a table by navigating from a data flow or state transition diagram, the Control Specification Editor automatically populates the table with information from those diagrams. For example, StP/SE populates an event logic table (Figure 6) with the following information:

- Control in column labels are automatically derived from the names of the control flows entering the Cspec bar on the data flow diagram.
- Event column labels are automatically derived from events on the state transition diagram.

Figure 6: Event Logic Table Created by Navigation

	1	2	3	4	6
1	Control In				Event
2	PIN Validated	Transaction Validated	Receipt Printed	Cash Dispensed	Card, PIN entered
3	1				
4	0				
5		1			
6		0			

Control In Control Value Event

To complete the table, you may need to add to, delete, or modify the information manually or with a command, as described in “Editing an Existing Table” on page 6-18.

Alternatively, you can create tables by entering all of the information explicitly or with one of the commands described in this chapter.

Creating a Cspec Table by Navigation

To create a new table from a Cspec bar in a data flow diagram or from an object in a state transition diagram:

1. Select the Cspec bar in the data flow diagram, or an appropriate object in the state transition diagram.

2. From the **GoTo** menu, choose a table type.
3. In the confirmation dialog, click **Yes** to confirm that you want to create the table.

StP/SE creates the new table and automatically populates the table cells with labels derived from the relevant elements in the related diagrams. Figure 7 shows an example of a decision table, populated with column labels in Row 2, derived from a data flow diagram.

Figure 7: New Decision Table

	1	2	3	4	5	6
1	Control In					
2	Transaction Valid	Cash Dispensed	Receipt Printed	PIN Validated	Transaction Req	Eject Card
3						
4						
6						

4. Edit the table, as follows:
 - Adjust column widths and display other rows and columns in the table, as desired.
 - Complete the table by entering any needed data, control values, or names of structured analysis objects in the unpopulated cells.

For more information, see “Editing an Existing Table,” which follows.
5. From the **File** menu, choose **Save**.

Editing an Existing Table

Although a table may be populated with some of the relevant information when you create it using navigation, you may need to edit the table to:

- Populate the control value cells
- Add information that can only be entered manually, such as process activations and event definitions
- Remove unneeded elements from the table
- Update the table after modifying the related data flow or state transition diagram

Note: Changes made while editing the Cspec table are not reflected in the related data flow or state transition diagrams.

The following table briefly describes methods for editing the various elements in a Cspec table. For a definition of each table element, see “Table Descriptions” on page 6-22. In addition to the automated methods, you can also edit the table cells by typing the entries manually.

Table 10: Cspec Table Editing Methods

Table Element	Editing Method	For Details, See
Column, row, or cell labels containing structured analysis objects	Use Edit > Set Label to add or change a column, row, or cell label by selecting an entry from a list.	“Setting Column, Row, and Cell Labels” on page 6-19
Control value cells	Use Edit > Set Label to populate a single control value cell with a value you select from a list.	
	Use Tools > Set Column Values from Data Definition to automatically populate an entire column for a control flow.	“Setting Control Values” on page 6-20

Table 10: Cspec Table Editing Methods (Continued)

Table Element	Editing Method	For Details, See
Process activation cells	Type the information manually.	
Event definition cells		

You can display and edit an existing table by opening it from the **File** menu or by navigating to it from a Cspec bar on a data flow diagram or from an object in an state transition diagram.

Setting Column, Row, and Cell Labels

You can use **Set Label** on the **Edit** menu to add or modify column, row, and cell labels representing the following objects:

- Processes and control flows on associated data flow diagrams
- Allowed Value annotation items for control flows defined in associated data structure diagrams
- Actions, events, and states on associated state transition diagrams

This command allows you to choose a label for the selected cell from a list of relevant structured analysis objects or control value annotations on the related data flow, data structure, or state transition diagram.

To set labels in a table or matrix:

1. Select a column, row, or other table cell that represents a structured analysis object (for table cell definitions, see “Table Descriptions” on page 6-22).
2. From the **Edit** menu, choose **Set Label**.
3. Select one of the objects on the **Set Label** submenu.

A matching label is applied to the selected cell. The source of the labels for each structured analysis table element is described in Table 1 on page 6-6.

Setting Control Values

You can enter control values in a decision table, process activation table, event logic table, or action logic table.

Each control value cell in a control column represents one of any number of user-defined allowed values for the control flow named in the column label. For example, in Figure 8 the allowed values for the *PIN Validated* control flow are 1 and 0.

Figure 8: Control Values in an Event Logic Table

The screenshot shows a software window titled "1: EventLogicTable - atm - StP/SE Control Spec Editor". It contains a table with 6 columns and 4 rows. The first column is labeled "Control In" and the last column is labeled "Event". The second column is labeled "PIN Validated", the third "Transaction Validated", the fourth "Receipt Printed", and the fifth "Cash Dispensed". The sixth column is labeled "Card, PIN entered". The first row is a header row. The second row contains the control flow names. The third and fourth rows contain control values: "1" and "0" in the "PIN Validated" column. Arrows point from the text "Column label containing control flow name" to the "PIN Validated" column header and from "Allowed values in control value cells" to the "1" and "0" values.

	1	2	3	4	6
1	Control In				Event
2	PIN Validated	Transaction Validated	Receipt Printed	Cash Dispensed	Card, PIN entered
3	1				
4	0				

Column label containing control flow name

Allowed values in control value cells

The Control Specification Editor can automatically populate control in or control out value cells for a control flow, if the values exist as Allowed Value items on a Data Definition annotation note for the flow's definition in a data structure diagram (see "Defining Control Information" on page 4-15). Each Allowed Value annotation item represents one of the possible control values for that control flow. The Control Specification Editor

populates as many control value cells in a column as there are Allowed Value annotation items for that control flow.

If the appropriate Allowed Value annotation items exist, you can use any of the following methods to populate a table with control values:

- Use **Tools > Set Column Values from Data Definition** to populate a selected column, as described in this section.
- Use **Edit > Set Label** to populate a single control value cell with a value you select from a list, as described in “Setting Column, Row, and Cell Labels” on page 6-19.
- Manually type the control values into the control value cells.

To set control values in a table using the **Set Column Values from Data Definition** command:

1. Select a control in or a control out column label cell.
2. From the **Tools** menu, choose **Set Column Values from Data Definition**.

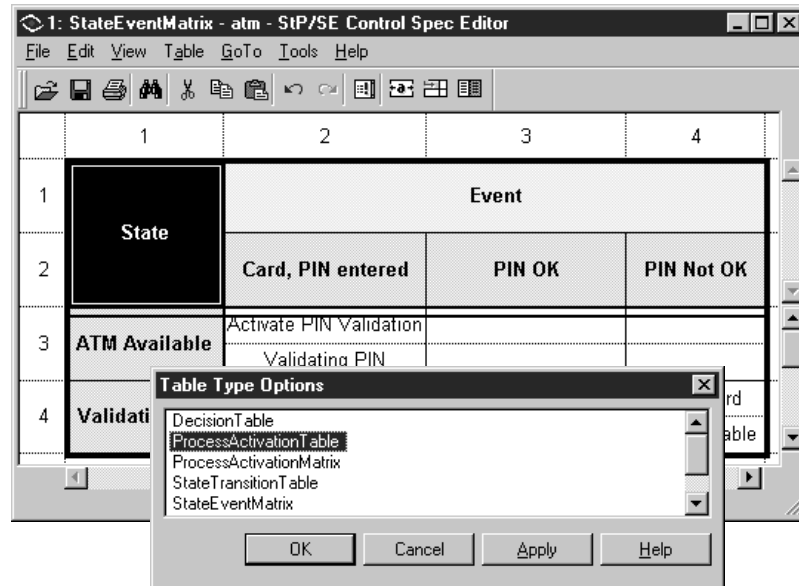
The command populates the control value cells for the selected flow.

Switching Between Cspec Tables

To switch between different table types for the current Cspec:

1. From the **Edit** menu, choose **Table Types**.
2. In the **Table Type Options** dialog (Figure 9), select the type of table you want to open.

Figure 9: Switching to Another Table



3. Click **OK** or **Apply**.
If the table exists, it is displayed. If the table does not yet exist, a confirmation box appears, asking if you want to create one.
4. Edit or create the table, as needed.

Table Descriptions

This section shows the Control Specification Editor window for each type of table or matrix that composes the Cspec. It notes the table elements for each table and describes how the table is named and the directory in which it is stored.

For a description of each table element, see "Overview of Table Elements" on page 6-4.

Action Logic Table (ALT)

Figure 10 shows the types of column, row, and cell labels that can be entered in an action logic table.

Action logic tables are automatically given the same name as the parent data flow diagram and an *.alt* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 10: Action Logic Table

The screenshot shows a software window titled "1: ActionLogicTable - atm - STP/SE Control Spec Editor". The window contains a table with 6 columns and 6 rows. The columns are labeled 1 through 6. The rows are labeled 1 through 6. The table content is as follows:

	1	2	3	4	6
1	Action	Activate Processes			Control Out
2		Complete Transaction	Dispense Cash	Initiate Transaction	Print Receipt
3	Activate PIN Validation			1	
4	Activate Print Receipt	1			On
5	Eject Card				
6	Get Transaction Data				

Below the table, there are labels with arrows pointing to specific cells:

- Action: points to cell (3,1)
- Process: points to cell (2,2)
- Process Activation: points to cell (4,2)
- Control Out: points to cell (2,6)
- Control Value: points to cell (4,6)

Decision Table (DET)

Figure 11 shows the types of column, row, and cell labels that can be entered in a decision table.

Decision tables are automatically given the same name as the parent data flow diagram and a *.det* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 11: Decision Table

The screenshot shows a software window titled "1: DecisionTable - atm - STP/SE Control Spec Editor". It contains a decision table with 7 columns and 4 rows. The columns are labeled 1 through 7. The rows are labeled 1 through 4. The table is divided into two main sections: "Control In" (columns 1-4) and "Control Out" (columns 5-7). The cells in the "Control In" section contain the following text: Row 1: "Control In"; Row 2: "Transaction Validated", "Cash Dispensed", "Receipt Printed", "PIN Validated"; Row 3: (empty); Row 4: (empty). The cells in the "Control Out" section contain the following text: Row 1: "Control Out"; Row 2: "Eject Card", "Print Receipt"; Row 3: (empty); Row 4: (empty). Arrows point from labels below the table to specific parts: "Control In" points to the first column of the "Control In" section; "Control Out" points to the first column of the "Control Out" section; "Control Value" points to the first column of the "Control In" section.

	1	2	3	4	6	7
1	Control In				Control Out	
2	Transaction Validated	Cash Dispensed	Receipt Printed	PIN Validated	Eject Card	Print Receipt
3						
4						

Control In

Control Out

Control Value

Event Logic Table (ELT)

Figure 12 shows the types of column and cell labels that can be entered in an event logic table.

Event logic tables are automatically given the same name as the parent data flow diagram and an *.elt* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 12: Event Logic Table

The screenshot shows a software window titled "1: EventLogicTable - atm - StP/SE Control Spec Editor". The window contains a table with the following structure:

	1	2	3	4	6
1	Control In				Event
2	PIN Validated	Transaction Validated	Receipt Printed	Cash Dispensed	Card, PIN entered
3	1				
4	0				

Arrows point from labels below the table to specific cells:

- "Control In" points to the header cell in column 1.
- "Control Value" points to the cell containing "0" in row 4, column 1.
- "Event" points to the header cell in column 6.
- "Event Definition" points to the cell containing "Card, PIN entered" in row 2, column 6.

Process Activation Matrix (PAM)

Figure 13 shows the types of column, row, and cell labels that can be entered in a process activation matrix.

Process activation matrices are automatically given the same name as the parent data flow diagram and a *.pam* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 13: Process Activation Matrix

The screenshot shows a software window titled "1: ProcessActivationMatrix - atm - StP/SE Control Spec Editor". The window contains a 4x4 matrix with the following content:

	1	2	3	4
1	Control In	Activate Processes		
2		Complete Transaction	Specify Transaction Data	Dispense Cash
3		Transaction Validated		
4		Cash Dispensed		

Below the matrix, three labels with arrows point to specific cells:

- Control In**: Points to the cell at row 1, column 1.
- Process**: Points to the cell at row 2, column 2.
- Process Activation**: Points to the cell at row 3, column 2.

Process Activation Table (PAT)

Figure 14 shows the types of column and cell labels that can be entered in a process activation table.

Process activation tables are automatically given the same name as the parent data flow diagram and a *.pat* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 14: Process Activation Table

The screenshot shows a software window titled "1: ProcessActivationTable - atm - StP/SE Control Spec Editor". The window contains a table with 6 columns and 4 rows. The columns are labeled 1 through 6 at the top. The rows are labeled 1 through 4 on the left. The table content is as follows:

	1	2	3	4	6
1	Control In				Activate Processes
2	Transaction Validated	Cash Dispensed	Receipt Printed	PIN Validated	Complete Transaction
3					
4					

Below the table, there are labels with arrows pointing to specific cells:

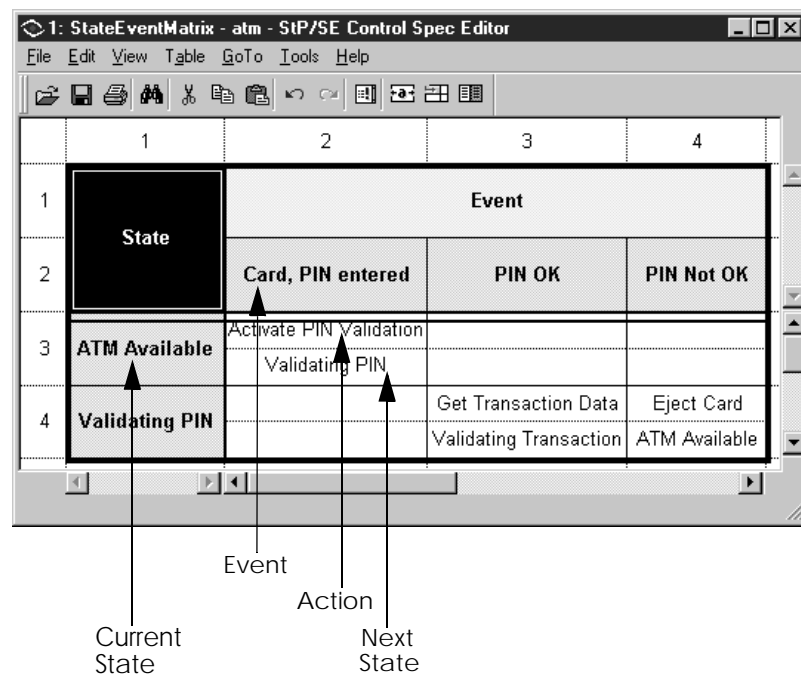
- "Control In" points to the cell at row 1, column 1.
- "Control Value" points to the cell at row 2, column 2.
- "Process" points to the cell at row 2, column 5.
- "Process Activation" points to the cell at row 2, column 6.

State Event Matrix (SEM)

Figure 15 shows types of column, row, and cell labels that can be entered in a state event matrix.

State event matrices are automatically given the same name as the parent data flow diagram and an *.sem* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 15: State Event Matrix



State Transition Table (STT)

Figure 16 shows the types of column and cell labels that can be entered in a state transition table.

State transition tables are automatically given the same name as the parent data flow diagram and an *.stt* file extension. They are stored in the *cse_files* directory in the path defined by the project and system names.

Figure 16: State Transition Table

	1	2	3	4
1	Current State	Event	Action	Next State
2	ATM Available	Card, PIN entered	Activate PIN Validation	Validating PIN
3	Validating PIN	PIN OK	Get Transaction Data	Validating Transaction
4	Validating PIN	PIN Not OK	Eject Card	ATM Available

Current State Event Action Next State

Validating a Cspec

The Control Specification Editor is linked to the Data Flow Editor and the State Transition Editor, so that information entered in each table can be compared and checked for consistency within the model.

StP/SE editors provide two kinds of validation checks for control specifications:

- Syntax checks
- Semantic checks

This section describes how to check syntax and semantics for the current Cspec in the Control Specification Editor. You can also check semantics for the entire model, an entire Cspec, or for one or more selected Cspec tables from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, “Checking SE Models.”

Syntax Checks

A Cspec table is syntactically correct if all row and column labels within each section (such as control in or control out sections) are unique. Checking the table syntax does not check the contents of the repository.

Syntax checks are applied automatically when you save the current table, or you can choose the **Check Syntax** command on the **Tools** menu.

Semantic Checks

A table is semantically correct if the control specifications in its tables balance completely with the control flows to and from a Cspec bar on a data flow diagram and with the states, events, and actions on the related state transition diagram. For a complete list of control specification semantic checks, see “Control Specification Editor Checks” on page 9-7.

To invoke semantic checking within the Control Specification Editor, choose either of the following commands from the **Tools** menu:

- **Check Table Semantics**—Checks current table only; in addition to displaying results in the Message Log, optionally allows you to save results to a file in a selected format
- **Check Cspec Semantics**—Checks semantics for the entire Cspec

In support of the Yourdon/Constantine structured design methodology, StP/SE provides the Structure Chart Editor (SCE) for:

- Drawing structure charts
- Creating Program Design Language (PDL) module specifications

Structure charts show how a system is intended to function. Using symbols to represent identifiable program modules, they illustrate the system hierarchy and organization, as well as the data and control information that is passed between the modules. These modules are presented from an external, “black box” point of view; that is, the symbols represent what the modules do rather than how they do it. Structure charts also provide a text description of each program module in the system.

This chapter describes:

- “Using the Structure Chart Editor” on page 7-2
 - “Creating a Structure Chart” on page 7-11
 - “Setting Properties of Structure Chart Objects” on page 7-22
 - “Creating a Formal Definition for a Module” on page 7-37
 - “Defining an Object’s Data or Return Type” on page 7-40
 - “Generating Module PDL Files” on page 7-41
 - “Using Filters” on page 7-47
 - “Validating a Structure Chart” on page 7-48
-

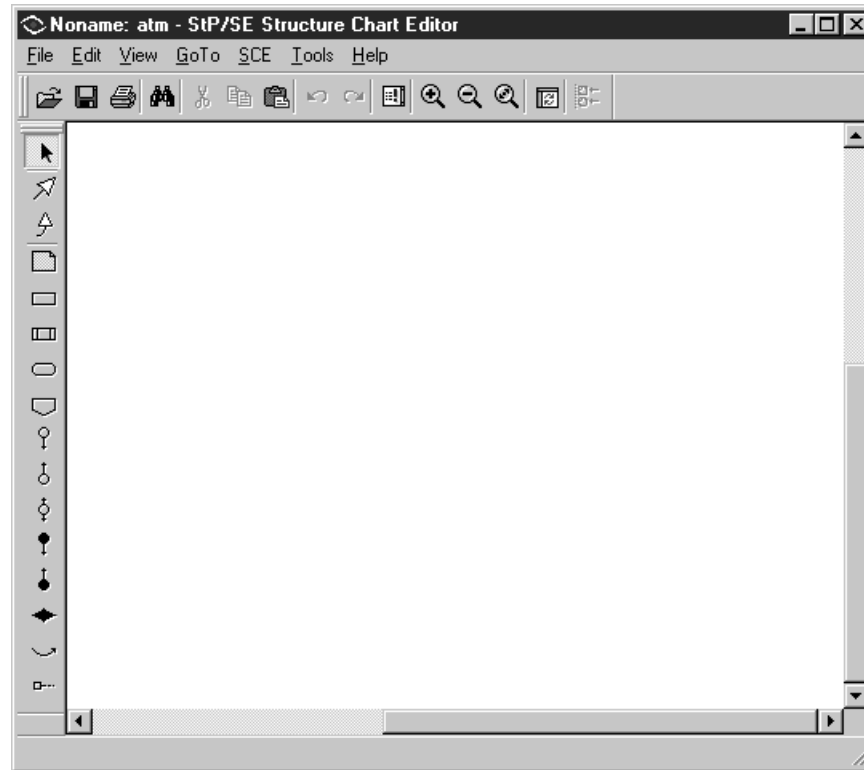
This chapter provides a brief explanation of structure chart notation. For complete details on the notation of this diagramming technique, refer to *Structured Design: Fundamentals of a Discipline of Computer Program and System Design* by Edward Yourdon and Larry L. Constantine (Prentice Hall, 1986).

For instructions on automatically generating structure charts from existing C code, see Chapter 11, “Reverse Engineering.”

Using the Structure Chart Editor

The Structure Chart Editor (Figure 1) is an interactive graphical editor for drawing structure charts and creating Program Design Language (PDL) module specifications. It also provides access to the Data Structure Editor for defining module return types and data types for global data modules and parameters.

Figure 1: Structure Chart Editor Window



The Structure Chart Editor window provides the standard StP diagram editor functions and menu options. For general information about diagram editors, see *Fundamentals of StP*.

In addition to the standard StP diagram editor features, the Structure Chart Editor provides:

- Symbols for drawing structure chart diagrams
- Navigations to related diagrams and tables
- The **SCE** menu, providing commands specific to the Structure Chart Editor
- Display marks representing additional information about structure chart objects

These features are described briefly in this section. For more information about using these features, see “Creating a Structure Chart” on page 7-11.

Starting the Structure Chart Editor

StP/SE provides access to the Structure Chart Editor from the StP Desktop.

For information about starting the Structure Chart Editor from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

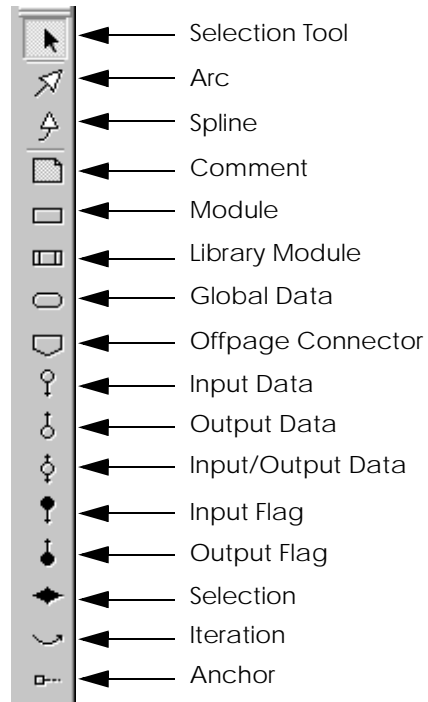
Using the Structure Chart Editor Symbols

The Structure Chart Editor provides the following types of symbols:

- Program modules
- Global data modules
- Library modules
- Data and control parameters (such as Input Data and Input Flag)
- Procedural symbols (selection and iteration)
- Offpage connectors and anchors

You select symbols for drawing structure chart diagrams from the Symbols toolbar (Figure 2).

Figure 2: Structure Chart Editor Symbols Toolbar



Procedures for using the symbols are described in this chapter. For a summary description of each symbol, see Appendix B, “StP/SE Symbol Reference.” For general instructions on using the Symbols toolbar, see *Fundamentals of StP*.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object whose symbol is selected on the current diagram. Use the **GoTo** menu to navigate. In the Structure Chart Editor, the commands on the **GoTo** menu allow you to:

- Display lower-level or other related structure charts
- Create or display data structure diagrams in the Data Structure Editor that define return types for modules and data types for data objects in structure charts
- Create or display module definitions
- Display the flow chart in the Flow Chart Editor that describes the code elements of a function in the structure chart
- Access a selected object’s C source code in a code viewer or editor

In some cases, navigation starts another editor from a current editor session. When you navigate to another StP/SE editor, the current session continues. You can navigate to these StP editors from the Structure Chart Editor:

- Data Structure Editor
- Flow Chart Editor
- Requirements Table Editor

The **GoTo** menu provides context-sensitive choices for the selected symbol.

To navigate to a target:

1. Select a symbol on the current diagram.
2. Choose a command from the **GoTo** menu.

The navigation target appears.

If there is more than one possible target, a selection list appears from which you can choose the appropriate target. In some cases, if a navigation target does not exist, you are given the option to create it.

Table 1 describes the navigation targets and commands available for each the Structure Chart Editor symbol.

Table 1: GoTo Menu Commands

Navigate From	Navigate To	Command
Program module	A module that is connected to an anchor and is the defining point for the selected program module; that is, it contains the substructure that further defines the module's signature.	Definition
	Object Selector with all references for the module, from you can choose a target.	All References
	Flow chart description of the code elements for the function represented by the module.	Flow Chart
	Object Selector with list of all diagrams that call the module, from which you can choose a target.	Caller
	Displays the definition of the module's return type value in a data structure diagram.	Return Type
	Object's source code definition in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code Definition
	Object's references in source code in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code References
	Requirements table (see "Allocating Requirements to a Model" on page 1-10).	Allocate Requirements
Parameter	Definition of the parameter's data type value in a data structure diagram.	Type Definition
	Requirements table (see "Allocating Requirements to a Model" on page 1-10).	Allocate Requirements

Table 1: GoTo Menu Commands (Continued)

Navigate From	Navigate To	Command
Global data	Definition of the global data module's data type value in a data structure diagram.	Global Type
	Object Selector with all references for the global data module, from which you can choose a target.	All References
	Object's source code definition in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code Definition
	Object's references in source code in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code References
	Requirements table (see "Allocating Requirements to a Model" on page 1-10).	Allocate Requirements
Library module	Object's references in source code in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code References
	Requirements table (see "Allocating Requirements to a Model" on page 1-10).	Allocate Requirements
Offpage connector	Identically named offpage connector on the current diagram that is the logical couple of the selected connector.	Offpage on This Diagram
	Identically named offpage connector on a related structure chart diagram that is the logical couple of the selected connector.	Offpage on Other Diagram

Using the SCE Menu

In addition to the standard diagram menus described in *Fundamentals of StP*, the Structure Chart Editor provides the **SCE** menu. This menu allows you to perform a variety of functions that modify structure chart diagrams.

Table 2 describes the commands available from the **SCE** menu.

Table 2: SCE Menu Commands

Command	Description	For Details, See
Edit PDL Note	Starts the Object Annotation Editor, adds a Module PDL Note to the selected module, and allows you to enter or edit the note's description.	"Generating Module PDL Files" on page 7-41
Generate PDL	Generates a program design language (PDL) file for the selected module. If no module is selected, PDL files are generated for all modules on the current diagram.	"Generating Module PDL Files" on page 7-41
View Generated PDL	Displays the generated PDL in a View Text window.	"Viewing a Module PDL" on page 7-46
Generate C Code	Generates C code files from structure chart symbols and annotations.	Chapter 10, "Generating C Code"

Using Display Marks

A display mark is a symbol or string that appears on or near an object and conveys additional information about that object. Several structure chart annotations cause display marks to appear in the diagram.

You can control the behavior of these display marks using the **Display Marks** tab of the **Options** dialog box. To display this dialog box, choose **Options** on the **Tools** menu. For details about using the dialog box, see *Fundamentals of StP*.

Table 3 describes the display marks that can appear on a structure chart. For more information and an example of each mark, see the section referenced in the table.

Table 3: Structure Chart Display Marks

Display Mark For	Name	Description	For Details, See
Access mode to global data from a module	AccessMode	A hollow or solid circle or a solid square appearing at the end of a link between a module and a global data object, indicating the mode used to access the global data.	“Specifying an Access Mode to Global Data” on page 7-34
Array size for global data	ArraySize	Identifies the size of a global data array.	“Specifying Array Size for Global Data” on page 7-36
Data type for global data	GlobalReturnType	Indicates the data type for global data. Default is int.	“Specifying Data and Return Types” on page 7-31
Lexically included program module	LexicalModule	Indicates that this program module is lexically included within its parent module.	“Adding a Lexical Include Annotation to a Module” on page 7-32

Table 3: Structure Chart Display Marks (Continued)

Display Mark For	Name	Description	For Details, See
Return type of a program module	ModuleReturnType	Indicates the return type for a program module. Default is int.	“Specifying Data and Return Types” on page 7-31
Data type of a parameter	ParamType	Indicates the data type of a parameter, if one has been explicitly specified.	“Specifying Data and Return Types” on page 7-31
Existence of a PDL specification file	Pdl	Indicates a PDL file exists for this program module.	“Generating Module PDL Files” on page 7-41
Directory scoping object	SEDirectory	Identifies a directory to which a program module or global data is scoped.	“Changing Scope Values for an Object Reference” on page 7-27
File scoping object	SEFile	Identifies a file to which a program module or global data is scoped.	“Changing Scope Values for an Object Reference” on page 7-27

Creating a Structure Chart

Creating structure charts to define a system’s architecture in the Structure Chart Editor involves four basic tasks:

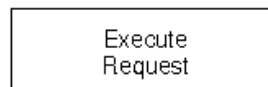
- Drawing modules and arcs to illustrate the hierarchical structure of the system you are modeling
- Setting properties of structure chart objects
- Defining the structure chart objects’ data and return types in the Data Structure Editor
- Generating the module specifications that define what each module is intended to do

This section discusses the initial task of using modules and arcs to represent the hierarchical structure of your system model. The other major sections in this chapter discuss the remaining tasks.

Representing Program Modules

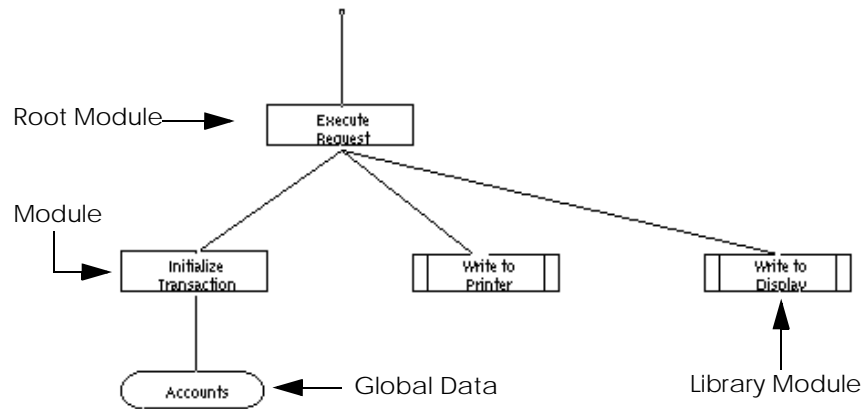
A module symbol, shown in Figure 3, represents a set of lexically contiguous statements in the code that compose an identifiable program unit of the modeled system. Some examples of program modules are a subprogram, subroutine, or function. Each module has an associated name, which is a text string appearing as a label on the module symbol.

Figure 3: Module Symbol Example



Module calls are represented by arcs that connect the modules in a structure chart. Modules can call or be called by other program modules. They can have “in” arcs from other modules or from an anchor or offpage connector. If a module has no in arcs from other modules in the current diagram, it is a root module (see Figure 4). That is, the root module in a diagram is the module at the highest level of the calling hierarchy in that diagram. A module can have “out” arcs that link it to another module, a library module, a global data module, or an offpage connector.

Figure 4: Root Module Connected to Other Symbols



Note: A root module in one diagram may appear in another diagram as a called module.

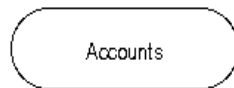
A module can have a lexical include annotation, which indicates that methodologically it is lexically included in the module to which its “in” arc is connected. Lexically included modules represent parts of a procedure that are logically separate from but lexically contained within another module. As with other program modules, you can create “out” links from an included module to any other program module, library module, or global data module. For more information, see “Adding a Lexical Include Annotation to a Module” on page 7-32.

A module can also have a PDL text specification associated with it that describes the module’s intended function. For more information on PDL specifications, see “Generating Module PDL Files” on page 7-41.

Representing Global Data

You use the global data symbol (Figure 5) to represent a part of the system that consists solely of global data elements.

Figure 5: Global Data Symbol

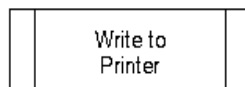


In structured design, only program modules call global data. Calls are represented by “in” links (arcs) to the global data module. Figure 4 on page 7-12 shows an example in which the Initialize Transaction module calls a global data module containing account data. Global data objects do not have out links.

Representing Library Modules

A library module exists outside the modeled system. Library modules generally perform a task that is relatively self-contained and uncoupled from the system, such as a library module called *Write to Printer*.

Figure 6: Library Module Example



A library module can be called by any program module, as represented by “in” links (arcs) to the library module (see Figure 4 on page 7-12). Library modules cannot have “out” links, as they do not call other modules in an SE model.

Library modules do not have associated PDL files because their functionality is already defined outside of the system being modeled.

Drawing Modules and Arcs

Generally, you order the objects in structure charts to be read from left to right and top to bottom. However, you can work on your diagram in any order, making changes to it as you develop the system model in greater detail. For example, you don’t have to start with the highest-level

module; you can start by representing any part of your system. You can then add a controlling, higher-level module to the diagram that is linked to each of the lower-level modules. The left-to-right order does not imply any particular calling order in a program.

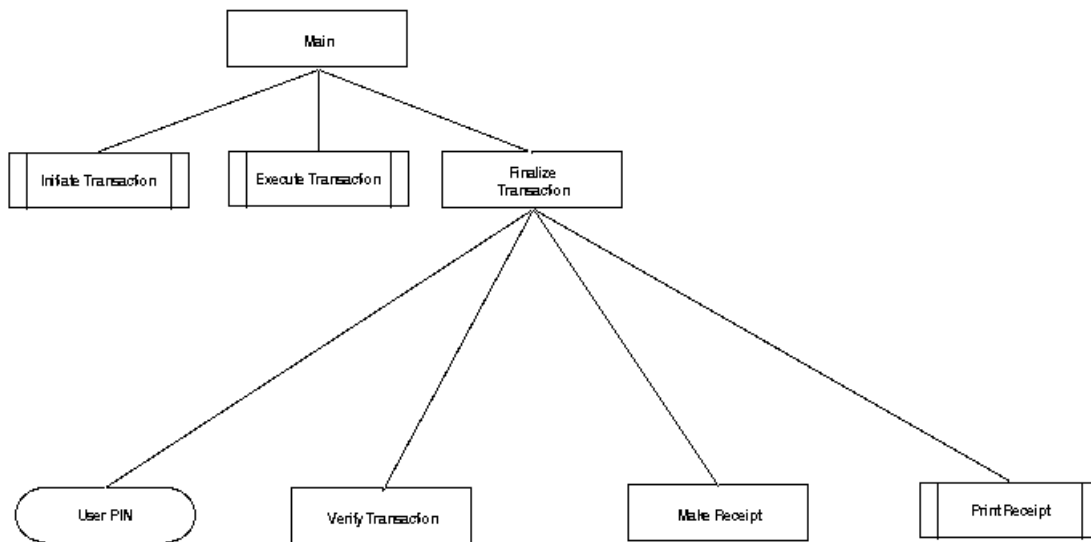
To draw the modules and arcs in a structure chart diagram:

1. Insert a program module symbol into the drawing area to create the root module for this structure chart.
2. Insert other program module, global data, and library module symbols into the drawing area, as needed.
3. Label the modules.
4. Link the module symbols with arcs to show the hierarchical structure, starting at the root module.

Always draw a link from a higher-level module to a lower-level module.

Figure 7 shows the beginning of the highest-level structure chart in the functional hierarchy of the ATM system.

Figure 7: Beginning a Structure Chart

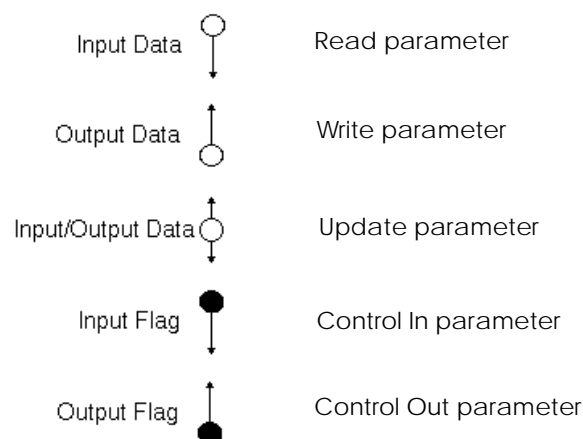


Showing the Transfer of Data and Control Information

Parameter symbols such as Input Data and Output Flag, represent data or control information that is transferred between modules. Data parameters show data transfer, and control signals (or flags) show the transfer of control information.

The Structure Chart Editor provides three types of data parameter symbols and two types of control signals (flags), as shown in Figure 8.

Figure 8: Parameter Symbols



Using Formal versus Actual Parameters

Parameters are categorized as either “formal” or “actual”:

- A formal parameter is used in a function definition, and is placed on a link between a root module and an anchor point in a module’s formal definition.
- An actual parameter is used in a function call, and is placed on a link between two modules.

Only formal parameters are generated to Program Design Language (PDL) specification files.

The StP/SE C code generator uses formal parameters to generate a function signature that comprises the parameter's definition, including its data type. The code generator evaluates the parameters from left to right and top to bottom.

For more information on formal parameters, see "Creating a Formal Definition for a Module" on page 7-37. For more information on PDL files, see "Generating Module PDL Files" on page 7-41.

Drawing Data and Control Parameters

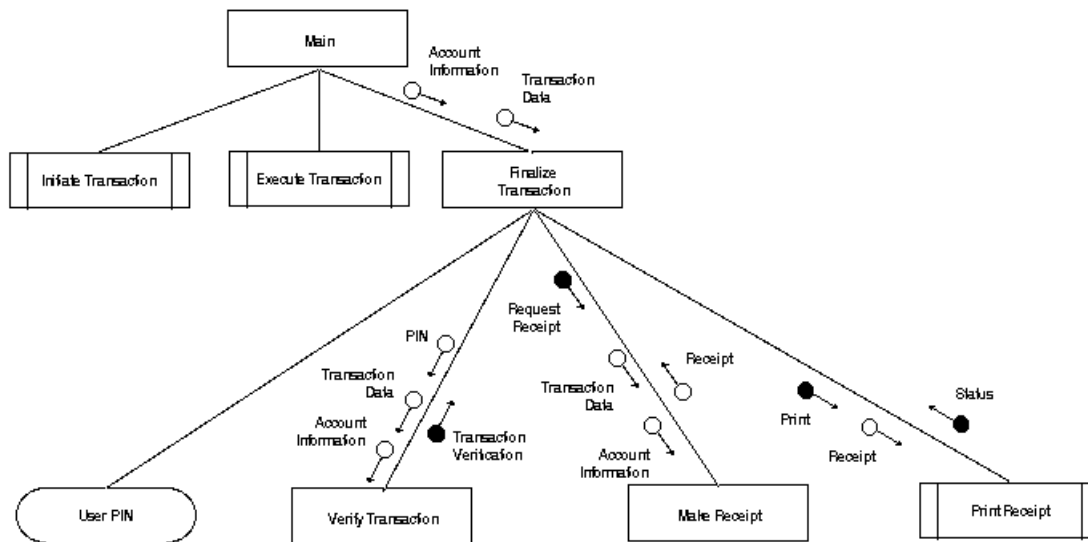
To draw data and control parameters on your structure chart:

1. Select the appropriate data or control parameter symbol from the Symbols toolbar and insert it into the drawing area, near a link between:
 - Program modules (for actual parameters)
 - An anchor and the root module in a module's formal definition (for formal parameters)
2. Label each data or control symbol.

Depending on the placement of a parameter symbol and the label associated with it, you may find that the label is hard to read or that it overlaps the tail of the parameter symbol. To eliminate this overlap, you can add a blank space or blank line at the beginning or end of the label. Such blanks are ignored by the StP/SE checking and generating programs.
3. From the **File** menu, choose **Save**.

Figure 9 shows an example of the highest-level structure chart for a sample system, on which all data and control parameters have been drawn.

Figure 9: ATM System



Moving Data and Control Parameters

If you move or delete an arc, the associated parameters are moved or deleted with it. If you are dissatisfied with the placement of the parameters after moving a module, you can move the parameter symbols individually.

If you move a parameter to a different arc, you must associate the parameter with the new arc by choosing the **Attach to Arc** command on the **Tools** menu.

To move a parameter to a new arc:

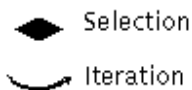
1. Select the parameter to be moved.
2. Drag the parameter to the new arc.
3. Select both the new arc and the parameter, using the Shift key plus left mouse button.
4. From the **Tools** menu, choose **Attach to Arc**.

Using Procedural Symbols

Structure charts use special notations called procedural symbols to indicate a calling relationship between modules that involve selective or iterative calls. The Structure Chart Editor supports two types of procedural symbols, shown in Figure 10:

- The selection symbol usually represents a relationship that contains an “if” statement or “case” statement in the calling module.
- The iteration symbol represents a calling relationship within a loop construct.

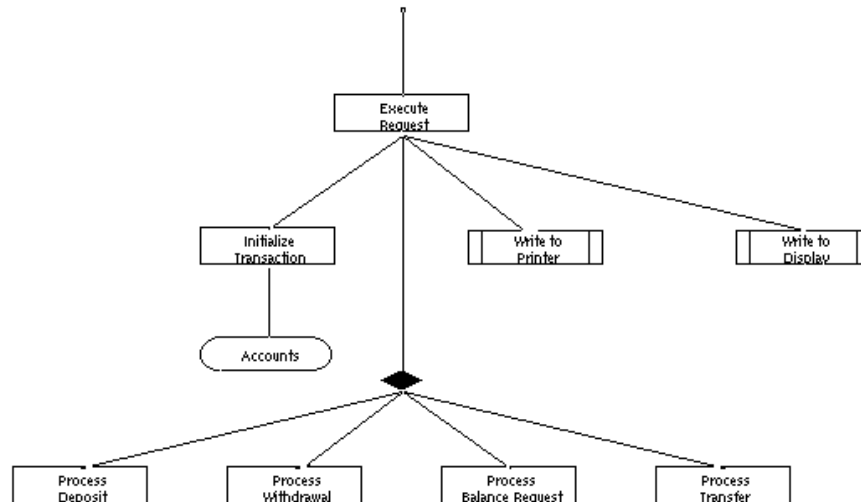
Figure 10: Selection and Iteration Symbols



Adding a Selection Symbol

Use the selection symbol to indicate that a module calls another module only when certain conditions specified in the calling module are met. The selection symbol in Figure 11 shows that the *Execute Request* module can call any of the modules at the bottom of the diagram, depending on the results of an “if” or “case” statement in the calling module.

Figure 11: Adding a Selection Symbol



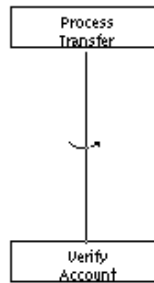
To add a selection symbol:

1. Insert the called modules into the drawing area.
2. Draw a link from the calling module to a called module with a vertex where you want to put the selection symbol.
To create the vertex, before connecting the arc to one of the called modules, click the left mouse button where you want the vertex to appear; then complete the connection to the module.
3. Connect the vertex to each called module.
4. Insert a selection symbol on the vertex or arc.

Adding an Iteration Symbol

To indicate repetition through a sequence of module calls, insert an iteration symbol onto a connection arc, as shown in Figure 12. The iteration symbol shows that the *Process Transfer* module can call the *Verify Account* module multiple times.

Figure 12: Adding the Iteration Symbol



Using Offpage Connectors

You can use an offpage connector symbol, shown in Figure 13, to simplify the appearance and improve readability of a cluttered diagram.

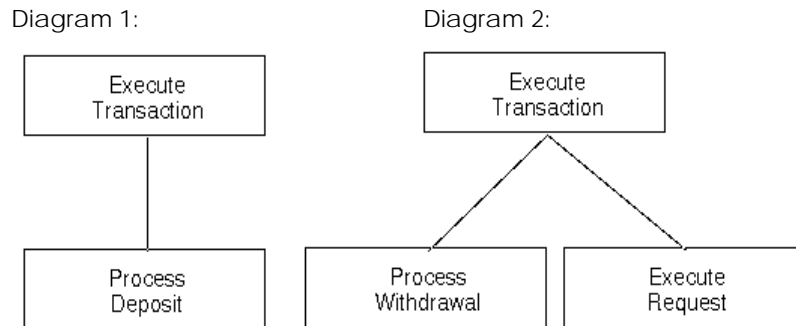
Figure 13: Offpage Connector Symbol



An offpage connector does not represent a system function or procedure. Its only purpose is to link physically separate but logically related parts of a structure chart. The parts you connect can be in the same diagram or in different diagrams.

For example, suppose you have two diagrams, each of which defines a portion of the same structure, as shown in Figure 14.

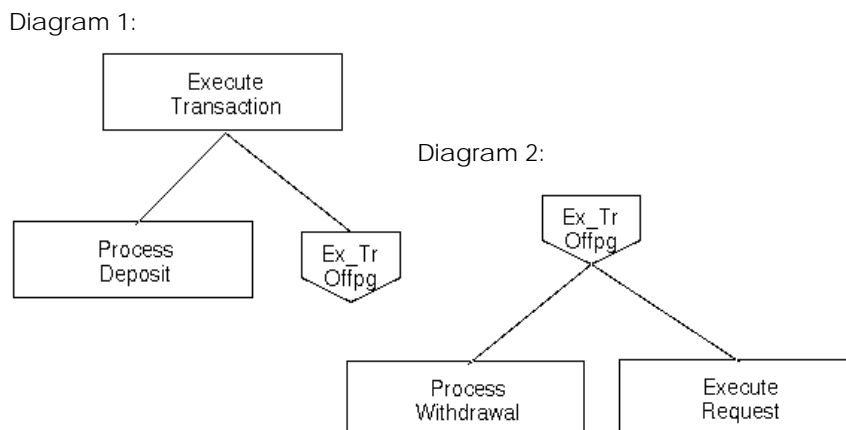
Figure 14: Separate but Logically Related Diagrams



You can use offpage connectors to logically connect the separate structure chart diagrams. Or if a diagram is too large for the editor window, you can use a pair of offpage connectors to link the currently visible portion of the diagram with a part that does not concurrently fit into the viewing window. Diagrams linked by offpage connectors can represent functions at parallel levels within the hierarchical structure, as well as hierarchical relationships.

Figure 15 shows how offpage connectors link separate diagrams to create a single logical structure.

Figure 15: Offpage Connector Example



The identical labels on the offpage connectors serve as the link between the two structures.

Note: Although it is not strictly methodologically correct, StP/SE allows you can connect offpage connectors to other offpage connectors, with logically expected results.

Using offpage connectors to link the separate diagrams or different parts of the structure chart allows you to easily navigate from one to the other by selecting the offpage connector and choosing one of the following navigation commands:

- **Offpage on This Diagram**
- **Offpage on Other Diagram**

Setting Properties of Structure Chart Objects

Modules, global data modules, and parameters can have properties, such as:

- File and directory information
- Data type
- Array size
- Lexical include designation
- Read/Write access to global data
- Storage class

These properties represent some of the most common relationships and annotations for structure chart objects. You can assign these properties to structure chart objects by editing the **Properties** dialog box, as explained in “Using the Properties Dialog Box” on page 7-23. Most assigned properties are stored as annotations of the object in the repository. The directory and file properties are stored as SEDirectory and SEFile objects.

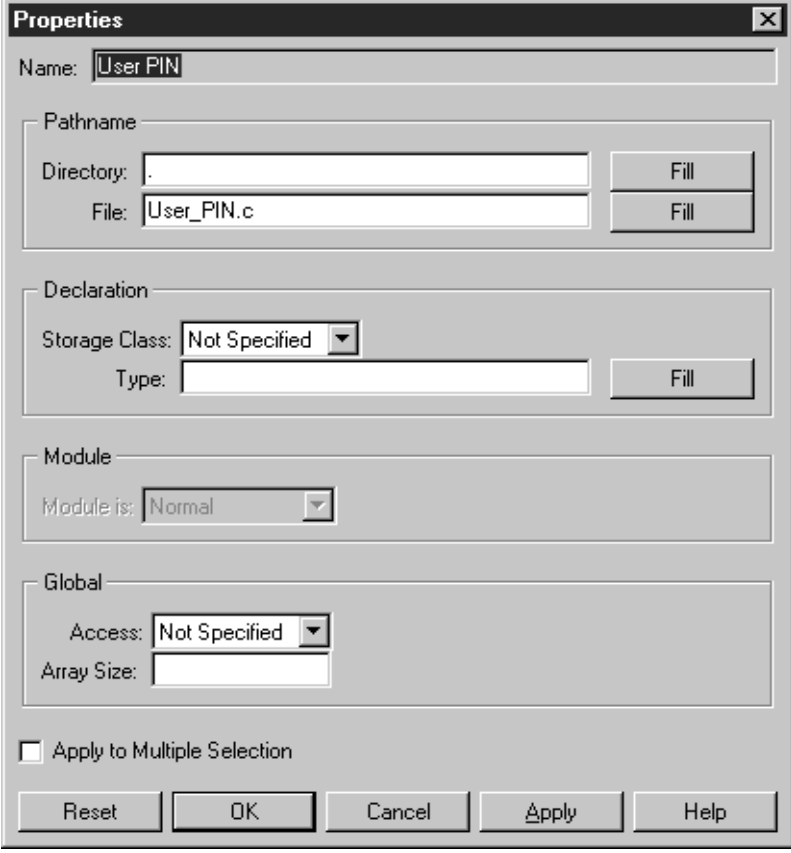
Alternatively, you can use the Object Annotation Editor (OAE), described in *Fundamentals of StP*, to add these or other annotations to structure chart objects. The OAE **Help** menu provides descriptions of all available annotation notes and items.

Some properties are specifically used for C code generation from structure chart diagrams. For information about these annotations, see “Structure Chart Annotations Used in Code Generation” on page 10-3.

Using the Properties Dialog Box

The **Properties** dialog box (Figure 16) enables you to specify or change any or all of the characteristics of a selected object at one time. The dialog box always displays the properties for the currently selected object. If you select a different object, the properties shown in the dialog box change to those of the newly selected object.

Figure 16: Properties Dialog Box



The Properties dialog box is a standard Windows-style window with a title bar labeled "Properties" and a close button (X). It contains several sections for configuring object properties:

- Name:** A text field containing "User PIN".
- Pathname:** A section containing two text fields: "Directory:" (empty) and "File:" (containing "User_PIN.c"). To the right of each field is a "Fill" button.
- Declaration:** A section containing a "Storage Class:" dropdown menu (set to "Not Specified") and a "Type:" text field (empty). To the right of the "Type:" field is a "Fill" button.
- Module:** A section containing a "Module is:" dropdown menu (set to "Normal").
- Global:** A section containing an "Access:" dropdown menu (set to "Not Specified") and an "Array Size:" text field (empty).
- Apply to Multiple Selection:** A checkbox that is currently unchecked.
- Buttons:** At the bottom are five buttons: "Reset", "OK", "Cancel", "Apply", and "Help".

To use the **Properties** dialog box:

1. Select a single object or multiple objects of the same type.
2. From the **Edit** menu, choose **Properties**.
The **Properties** dialog box appears with current settings for the selected object. If multiple objects are selected, properties for the first object appear, but the dialog fields are initially dimmed.
3. To apply properties to multiple selected objects, select **Apply to Multiple Selection** to make the previously dimmed fields editable.
4. Enter or change the values for the selected object(s), as needed.
For details about each setting, see “Summary of Object Properties and Dialog Options” on page 7-25 and other subsequent sections.
5. Click **OK** or **Apply**.
6. From the **File** menu, choose **Save** to save the object’s new properties in the repository.

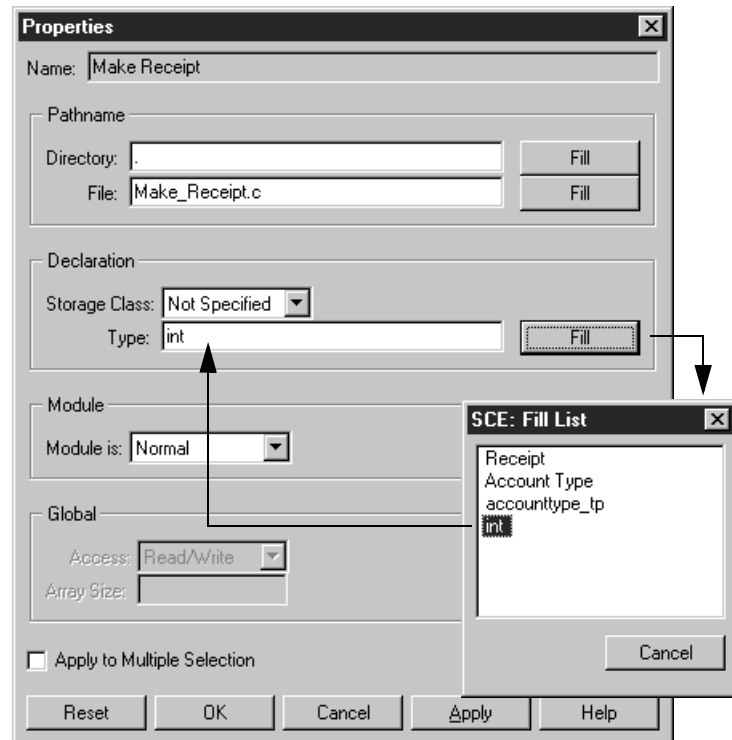
Selecting Values from a Fill List

The **Fill** buttons on the **Properties** dialog allows you to choose an existing value for the adjacent field from a list.

To select a property value from the **Fill** list:

1. On the **Properties** dialog, click the **Fill** button to the right of the **Directory**, **File**, or **Type** field.
2. Select one of the values in the list as shown in Figure 17; then close the **Fill** window.
The selected value appears in the appropriate **Properties** dialog field.

Figure 17: Choosing a Value from the Fill List



Summary of Object Properties and Dialog Options

Table 4 provides a summary of the various parts of the **Properties** dialog box and their possible settings.

Setting Properties of Structure Chart Objects

Table 4: Object Properties Summary

Property or Option	Description	Settings	For Details, See
Name	Name of the object to which the properties will be applied.	(Read-only field)	
Fill button	Displays a list of choices for the adjacent property specification.	(not applicable)	“Selecting Values from a Fill List” on page 7-24
Directory, File	SEDirectory and SEFile scope object values, respectively, which define the directory and filename for the C code for this program module or global data.	Directory and file pathnames	“Changing Scope Values for an Object Reference” on page 7-27
Storage Class	Used by the C code generator to assign an appropriate qualifier to the definition for a module, global data object, or parameter.	Unset by default. Accepts values of const, static, static const, register, or register const.	“Choosing a C Storage Class” on page 7-29
Type	Return type for a module, or data type for global data or a parameter, as represented by the ModuleReturnType, GlobalReturnType, or ParamType display mark.	Predefined C type (char, double, float, int, long, short, or void) or the name of a data object defined in a data structure diagram. No default value.	“Specifying Data and Return Types” on page 7-31
Module Is	Specifies whether or not the module is lexically included within another module.	Lexical Include or Normal. Default is Normal.	“Adding a Lexical Include Annotation to a Module” on page 7-32
Access	Access mode used by a module to access global data, indicated by the AccessMode display mark on an arc (“Don’t Care” default has no display mark).	Read Write Read/Write Don’t Care (default)	“Specifying an Access Mode to Global Data” on page 7-34

Table 4: Object Properties Summary (Continued)

Property or Option	Description	Settings	For Details, See
Array Size	Defines global data object as an array with a limited number of occurrences in your system.	A string that specifies the array bounds; for example, 1000 or 1-10.	“Specifying Array Size for Global Data” on page 7-36
Apply to Multiple Selection	Option that applies the property values to multiple selected objects.	Selected or unselected	“Using the Properties Dialog Box” on page 7-23

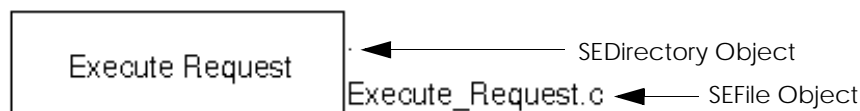
Changing Scope Values for an Object Reference

Program module and global data names are not unique within a system model. These objects are distinguished from identically-named objects of the same type by mapping them to SEFile and/or SEDirectory objects. In StP, object identification through this mapping process is called scoping. Modules and global data are scoped to files, which in turn are scoped to directories. They do not use links to other parent modules as scope nodes.

The SEDirectory and SEFile objects indicate the directory and file location for any C code for the structure chart object. Modules and global data are assigned default SEDirectory and SEFile values when they are created by a user. The SEDirectory default is a period (.). At code generation time, StP/SE interprets the period as the current directory from which you started the C code generator. The SEFile default is *<module_name>.c*. Identically named objects with default file and directory scope always refer to the same repository object.

The directory and file scope information appears as display marks in the outside upper-right and lower-right corners of the symbol when the SEDirectory and SEFile display marks are turned on. The default directory appears as a small dot, as shown in Figure 18.

Figure 18: Default SEDirectory and SEFile Display Marks



You may want to change the scope values for a module or global data in order to:

- Specify a different directory or filename for this object's C code
- Differentiate between this module or global and one with the same name in the repository

To change the scope values for a particular reference to an object or set of objects, you edit the Directory and/or File properties on the **Properties** dialog box.

Changing the scope of a program module or global effectively creates a new program module or global object in the repository. All annotations for the original object are copied to the newly scoped object. The original object with the default scope remains in the repository, along with the newly scoped one. Other references to the original object remain unchanged. If you want some of the other references to point to the newly scoped object, you must use the same procedure to change their scope properties on the **Properties** dialog.

Note: To change the value of an SEFile or SEDirectory object for all instances of its use in the current system, see "Renaming SEFile and SEDirectory Objects" on page 13-12.

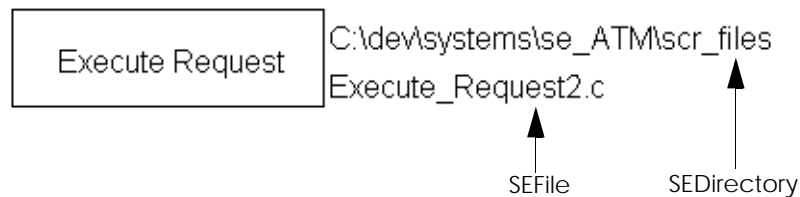
To change scope values for program modules or global data:

1. Select one or more program modules or global data objects for which you want to change the directory and/or file scope.
2. From the **Edit** menu, choose **Properties**.
3. On the **Properties** dialog, edit the specifications in the **Directory** and/or **File** fields, either by typing them manually or by using the **Fill** button to select from a list (see "Selecting Values from a Fill List" on page 7-24).

4. Click **OK** or **Apply**.

If the display marks are turned on, the changed directory and/or file scoping information appears, as shown in Figure 19.

Figure 19: Module with User-Specified Directory and File Scope



Choosing a C Storage Class

If you plan to use the C Code Generator to produce C code from your structure chart, you can assign a C storage class to every program module, global data object, and parameter in the chart. The code generator uses this information to add the appropriate qualifier to the definition for the function or global data.

Accepted storage class values are:

- `const`
- `static`
- `static const`
- `register`
- `register const`

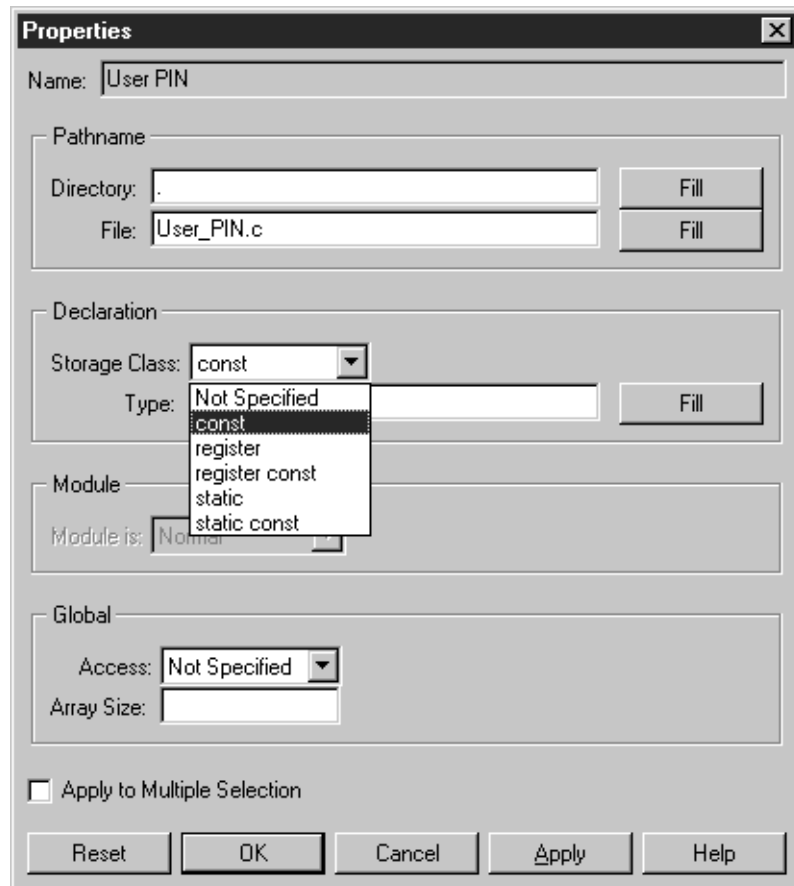
Note: It is left to the user's discretion to assign appropriate storage class values to structure chart objects, according to the C dialect in use. Not all of these values are correct for all structure chart objects.

To add storage class information:

1. Select a module or global data object on the structure chart diagram.
2. From the **Edit** menu, choose **Properties**.

3. In the **Storage Class** field on the **Properties** dialog, display the options list and select an appropriate storage class for this object (see Figure 20).

Figure 20: Selecting a Storage Class



4. Click **OK**.
This adds a Storage Class item to a Module Definition, Global Definition, or Parameter Definition annotation note. The storage class information does not appear on the diagram.

Specifying Data and Return Types

All formal data and control parameters (those on an arc between an anchor and the root module) must have a data type annotation. Additionally, program modules may have type annotations that specify a return type and global data objects may have type annotations that specify a data type. StP/SE ignores any type annotations on actual parameters.

If the required type information is not specified, the default is `int`. To specify a different data type, edit the contents of the **Type** field on the **Properties** dialog box.

You can assign the return type or data type for a structure chart object as either:

- A predefined C type (`char`, `double`, `float`, `int`, `long`, `short`, or `void`)
- The name of a data object that is defined in a data structure diagram

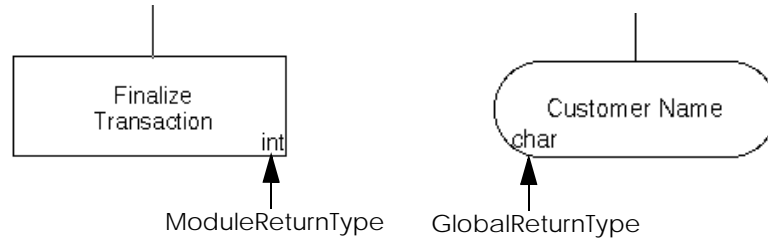
If the specified type does not exist, you can define it by navigating to the Data Structure Editor. For details, see “Defining an Object’s Data or Return Type” on page 7-40.

To add type information to an structure chart object:

1. Select the object to which you want to add type information.
2. From the **Edit** menu, choose **Properties**.
3. On the **Properties** dialog, enter a predefined or other type specification in the **Type** field, either by typing it manually or by using the **Fill** button to select from a list (see “Selecting Values from a Fill List” on page 7-24).
4. Click **OK**.

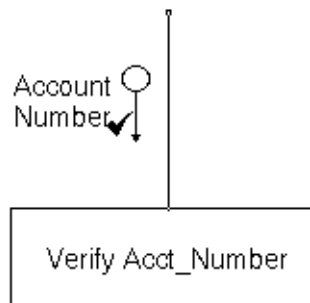
The type annotation appears as a `ModuleReturnType` or `GlobalReturnType` display mark, as shown in Figure 21, or as a `ParamType` display mark, as shown in Figure 22.

Figure 21: Type Display Marks for Modules and Global Data



The ParamType display mark appears as a check mark next to the parameter. It indicates only that a data type has been specified for the parameter. The data type itself appears on the **Properties** dialog box.

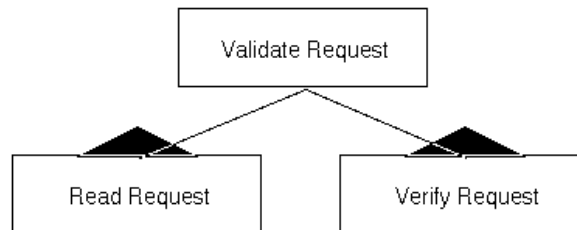
Figure 22: Parameter with ParamType Display Mark



Adding a Lexical Include Annotation to a Module

A lexically included program module represents a part of a subroutine or function that is logically separate from but lexically contained within the module to which it points. For example, you can structure your code to perform two different logical operations, such as to read a line of input and verify it, in the same module. In your structure chart diagram, you can represent both operations as lexical includes into another module, as shown in Figure 23.

Figure 23: Lexical Module Example



Another example of a lexical include is a module that physically contains several other modules that are local to the containing module. Each of the component modules is represented by a lexically included module. This construct is supported by high-level programming languages such as Pascal and Ada.

Specifying a module as a lexical include creates a Lexical Module annotation note with an **Is Lexical Module** item. A lexically included module is indicated by a LexicalModule display mark—a solid triangle at the top of the included module. Lexically included modules do not have associated PDL specifications. Instead, the PDL is associated with the including module.

To indicate that a module is lexically included within another module:

1. Select a module you want to include in the parent module.
2. From the **Edit** menu, choose **Properties**.
3. On the **Properties** dialog, select **Lexical Include** in the **Module Is:** field.
4. Click **OK**.
5. From the **View** menu, choose **Refresh Display Marks** to see the LexicalModule display mark.

Specifying an Access Mode to Global Data

You can add an access mode to the arcs connected to any global data module. The access mode indicates the kind of access modules can have to the global data: Read, Write, Read/Write, or Don't Care. The default is Don't Care.

You can assign the access mode by editing the contents of the **Global Access** field on the global's **Properties** dialog box. Specifying an access mode for a global data module creates a Global Usage annotation note for its "calling" arc, with an item called Access Mode. The resulting AccessMode display mark appears on the arc(s) or spline(s) connected to that global. If the global has multiple arcs, the display mark for the access mode appears on all of them. To assign different access modes to the same global data from different modules, draw a copy of the global data symbol for each different access mode, including the global's connecting arc, and set the access mode property separately for each of the duplicate global data symbols.

To specify a module's access mode to the global data:

1. Select a global data module in the structure chart diagram.
2. From the **Edit** menu, choose **Properties**.
3. In the **Global Access** field on the **Properties** dialog, display the options list and select one of the access modes.

Figure 24: Choosing a Global Access Mode

The image shows a 'Properties' dialog box with the following sections and controls:

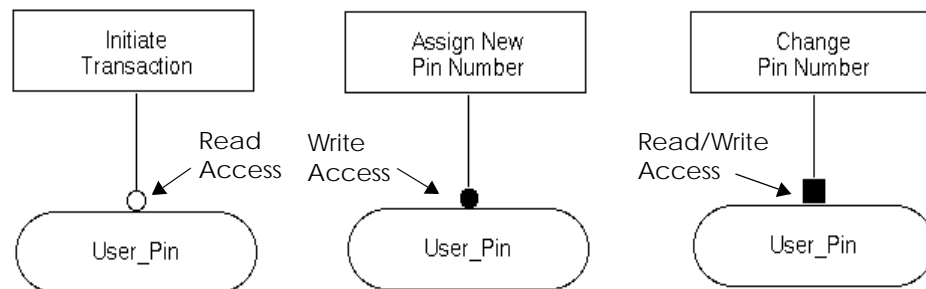
- Name:** A text field containing 'User PIN'.
- Pathname:** A group box containing:
 - Directory:** A text field with a dot '.'.
 - File:** A text field containing 'User_PIN.c'.
 - Two 'Fill' buttons to the right of the Directory and File fields.
- Declaration:** A group box containing:
 - Storage Class:** A dropdown menu set to 'const'.
 - Type:** A text field containing 'int'.
 - A 'Fill' button to the right of the Type field.
- Module:** A group box containing:
 - Module is:** A dropdown menu set to 'Normal'.
- Global:** A group box containing:
 - Access:** A dropdown menu set to 'Not Specified'.
 - Array Size:** A dropdown menu set to 'Not Specified'.
 - A list box below the Array Size dropdown showing 'Read', 'Write', and 'Read/Write'.
 - An unchecked checkbox labeled 'Apply to Multiple Selection'.
- Buttons:** At the bottom are 'Reset', 'OK', 'Cancel', 'Apply', and 'Help' buttons.

4. Click **OK**.

An AccessMode display mark appears between the global data symbol and the end of its “calling” arc, as shown by the examples in Figure 25.

If you choose **Not Specified** as the access mode, no display mark appears on the arc.

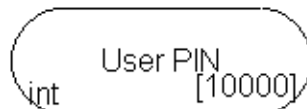
Figure 25: Global Data with Access Mode Display Marks



Specifying Array Size for Global Data

You can assign an array size to a global data object to indicate that it represents an array of a limited set of data objects that all have the same data type. Figure 26 shows an example of a global that represents an array—in this case, of 500 bank addresses, all of which have the data structure defined by the *Address* data type.

Figure 26: Global Data Object with Array Size Display Mark



To specify array size on the **Properties** dialog, enter the array limits as text in the **Array Size** field. For example, you could enter 1000 for an upper limit only, or 1-10 to specify both lower and upper limits. The text you enter appears enclosed in square brackets, as an ArraySize display mark in the lower-right corner of the symbol. If you include the brackets in your text entry, the extra pair of enclosing brackets are omitted.

Specifying array size creates a Data Definition annotation note with an item called Array Size.

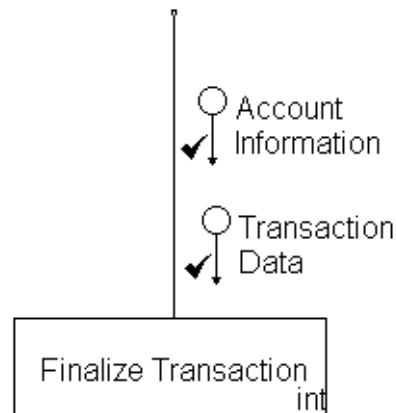
Creating a Formal Definition for a Module

A well-defined structure chart should have a formal definition, also known as a formal “signature,” for each program module that appears in the diagram.

A formal definition of a module is a separate structure containing:

- The module to be defined, linked to an anchor point
- Formal parameters for the module on the anchor-to-module link
- A return type declaration for the module
- A data type declaration (indicated by a check display mark) for each of the module’s formal parameters
- Optionally, additional substructure showing the module’s more detailed subfunctions

Figure 27: Definition of a Module (without substructure)



You can create the formal definition of a program module in either a:

- Separate, lower-level structure chart diagram (named the same as the module it defines)
- Separate structure within the same diagram containing the module to be defined

Defining modules in separate diagrams allows you to physically partition the system into logical subsystems, showing specific functions on separate diagrams. This helps to simplify the overall representation of your system on higher-level diagrams.

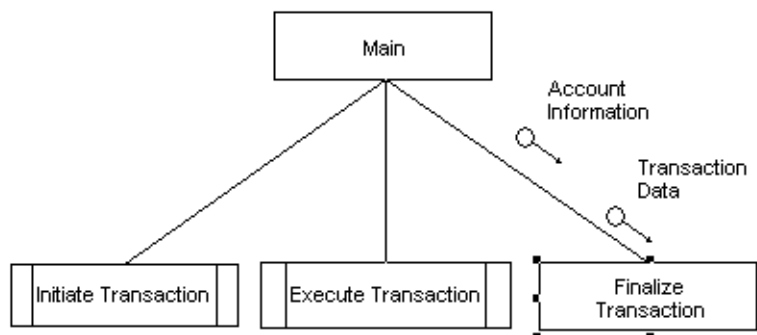
You don't have to complete all the details of a structure chart before you start defining modules and adding lower-level structure charts. You can define a module at any time during a Structure Chart Editor session.

Defining a Module in a Separate Diagram

Use the **Definition** command to define a module in a separate lower-level diagram:

1. Select the module to be defined.

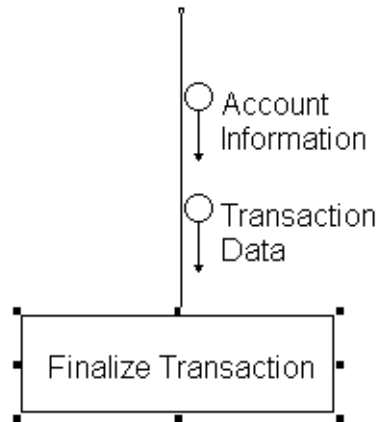
Figure 28: Module to Be Defined



2. From the **GoTo** menu, choose **Definition**.
3. In the confirmation box, click **Yes** to confirm that you want to create the diagram.

The definition diagram appears, as shown in Figure 29. Actual parameters from the module's in-link in the original structure become formal parameters for the module in the definition.

Figure 29: Formal Definition Diagram



4. Specify the module's return type and the formal parameters' data types on the **Properties** dialog, as described in "Specifying Data and Return Types" on page 7-31.
5. If necessary, define the data and return types in data structure diagrams, as described in "Defining an Object's Data or Return Type" on page 7-40 (predefined C types do not require data definitions).
6. Optionally add substructure to the module to further define its functionality.
7. From the **File** menu, choose **Save** to link the new definition diagram to the parent diagram and save it in the repository.

To return to the parent diagram:

1. Select the root module in the subdiagram.
2. From the **GoTo** menu, choose **Caller**.
The diagram that contains the parent module is displayed, and the calling module is selected.

Defining a Module in the Same Diagram

To define a module's substructure in the same diagram:

1. Insert an anchor symbol onto another part of the diagram containing the module to be defined.
2. Insert a module symbol onto the diagram, label it with the name of the module to be defined, connect it to the anchor
3. Add formal parameters to the anchor-to-module link.
4. In the formal definition, specify the module's return type and the formal parameters' data types on the **Properties** dialog, as described in "Specifying Data and Return Types" on page 7-31.
5. If necessary, define the data and return types in data structure diagrams, as described in "Defining an Object's Data or Return Type" on page 7-40 (predefined C types do not require data definitions).

Defining an Object's Data or Return Type

The data or return type assigned to a parameter, global data, or program module must be defined as either a:

- Predefined C type (char, double, float, int, long, short, or void)
- Data structure in a data structure diagram

Predefined C types do not require any further definition.

To create a data structure that defines a data or return type:

1. On the structure chart, select a module, data parameter, or global data symbol for which a type has been specified.
2. From the **GoTo** menu, choose the appropriate type definition command (**Return Type** for modules, **Global Type** for global data modules, or **Type Definition** for parameters).

3. In the confirmation dialog, click **Yes** to confirm that you want to create the data structure diagram.
This starts the Data Structure Editor and creates a new data structure diagram containing a symbol whose label corresponds to the type for the selected structure chart object.
4. Follow the instructions in “Creating a Data Structure Diagram” on page 4-8 to define the data or return type.

Generating Module PDL Files

Each program module in a structure chart should have a Program Design Language (PDL) specification, which serves as a textual description of a module. The PDL specification is generated to a file and can contain a variety of information, including:

- Callers of the module
- Module inputs and outputs
- Module description
- Programming language statements

Note: Only parameters on an anchor-to-module link in a module’s formal definition are generated to PDL files as formal parameters in function headers. Parameters on module-to-module links represent actual parameters in function calls and do not appear in the PDL file. For more information, see “Using Formal versus Actual Parameters” on page 7-15.

The following is an example of a PDL file for the *Compare PINs* module:

```
module Compare PINs
PDL generated
    06/15/99 15:35:03 by user@host
This module has 3 formal parameters:
    Valid_User, User_Pin, Correct_Pin
input data params
    User_Pin, Correct_Pin
input flags
output data params
output flags
    Valid_User
calls
called by
    Validate User
description
    if User_Pin = Correct_Pin then
        Valid_User = True
    Else
        Valid_User = False
    endif
end module
```

The PDL file is generated to the *sce_files* directory in the path defined by the project and system names. The name of the generated file is *<module_name>.pdl*.

Creating a PDL file involves two separate procedures:

- Adding a Module PDL annotation to each module for which you want to generate a PDL file
- Generating the PDL files for all PDL-annotated modules

The following sections describe these procedures in detail.

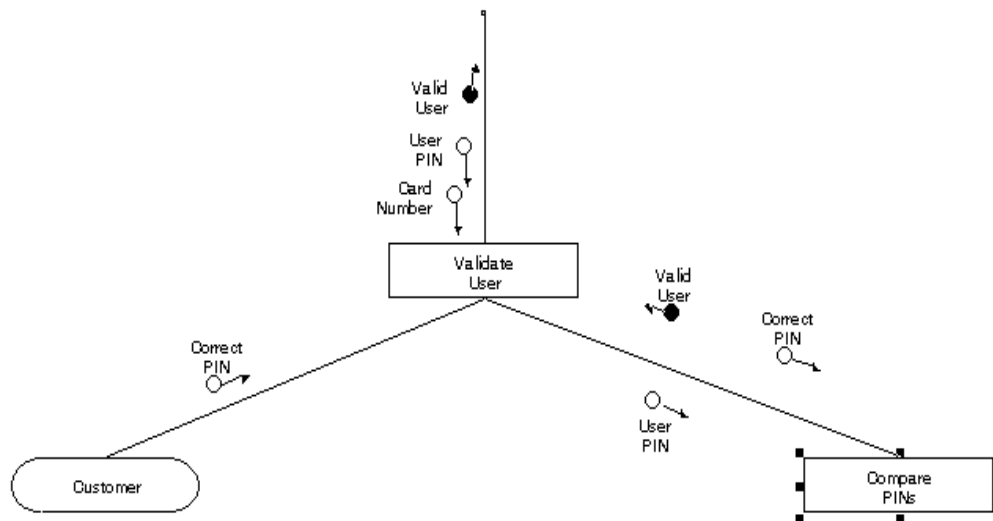
Creating a Module PDL Annotation

Before you can generate a PDL file for a module, the module must have a Module PDL annotation note.

To create the annotation note:

1. Select a module for which a PDL file is to be generated, as shown in Figure 30.

Figure 30: Generating a PDL File for a Module



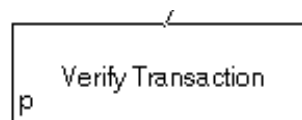
2. From the **SCE** menu, choose **Edit PDL Note**.

This creates a Module PDL annotation note with a **Has PDL?** item for the selected module and displays a **Description** dialog box for the note description entry.

3. In the **Description** dialog, type a textual description or pseudo code explaining what the module does.
4. Click **OK** to save the note description.

A **p** display mark appears in the lower-left corner of the module(s), as shown in Figure 31.

Figure 31: Pdl Display Mark



Generating the PDL File

You can generate PDL files for:

- One or more modules for the current diagram in the Structure Chart Editor
- All modules on selected diagrams in the Desktop objects pane
- All modules in your entire system model

Each module for which a PDL is to be generated must have a PDL annotation, as described in “Creating a Module PDL Annotation” on page 7-42.

Generating PDLs for the Current Diagram

To generate PDL files for any or all PDL-annotated modules in the current diagram:

1. In the Structure Chart Editor, select each module for which a PDL file is to be generated.
If no module is selected, a PDL file will be generated for every PDL-annotated module in the current diagram.
2. From the **SCE** menu, choose **Generate PDL**.
StP/SE generates the PDL file(s).

Creating PDL Files for Selected Diagrams or Entire Model

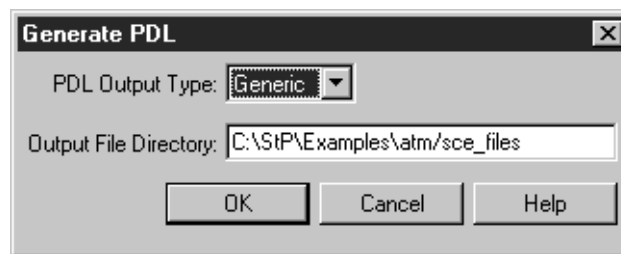
To create PDL files for all PDL-annotated modules in selected diagrams or for the entire model:

1. In the Model pane on the StP Desktop, open the **Diagrams** category and select **Structure Chart**.
2. To generate PDL files for modules in particular diagrams only, select the diagrams in the objects pane.

3. From the **Code** menu, choose **PDL**; from the **PDL** submenu, choose either:
 - **Generate PDL**—For one or more selected diagrams in the objects pane
 - **Generate PDL for All Diagrams**—For all structure chart diagrams in your system model

A dialog box similar to the one in Figure 32 appears.

Figure 32: Generate PDL Dialog Box



4. From the options list in the **PDL Output Type** field, select either **Generic** or **ADA**.
5. Optionally, edit the contents of the **Output File Directory** field (the default output location for the generated files is the *sce_files* directory in the current project and system).
6. Click **OK** to generate the PDL files.

Generating PDLs with Script Manager

If you prefer, you can use the StP Script Manager to generate a PDL report in a selected publishing format, such as RTF or FrameMaker. You can execute one of the StP-provided scripts, *all_pdl* or *one_pdl*, or define a new script using StP's Query and Reporting Language (QRL).

See *Query and Reporting System* for more information on running scripts in the Script Manager, or on modifying scripts using QRL.

Viewing a Module PDL

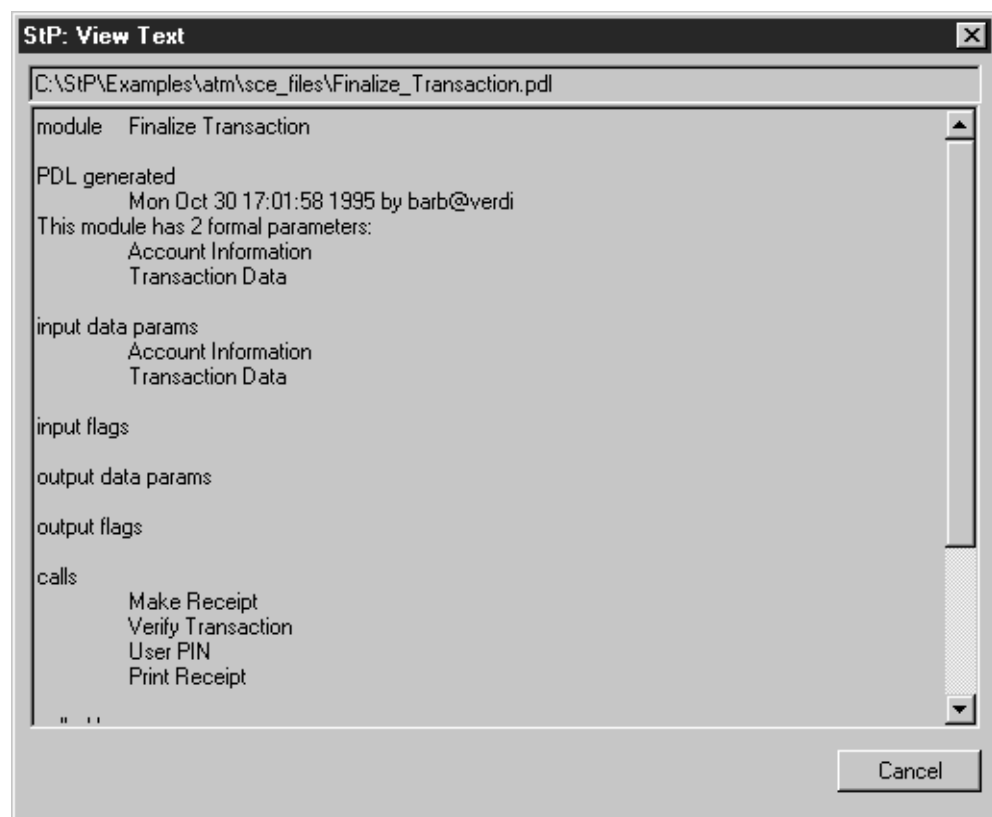
After generating a PDL file, you can view its contents from the Structure Chart Editor using the **View Generated PDL** command. This command opens a read-only window containing the PDL file contents. You can also use a standard text file editor to view or edit the PDL.

To view a generated PDL file from the Structure Chart Editor:

1. Select the module.
2. From the **SCE** menu, choose **View Generated PDL**.

The **View Text** window appears, as shown in Figure 33.

Figure 33: PDL File in the View Text Window



Using Filters

If your structure chart becomes too crowded to read, you can use the **Apply Filter** command to suppress the display of parameter labels or selected module types and their call links. In addition to a standard set of StP filters, StP/SE provides these editor-specific filters for structure charts:

- **Hide/Show Libraries**—Hides or shows all library modules and associated call links
- **Hide/Show IncludedModules**—Hides or shows all included program modules and associated call links
- **Hide/Show ParameterLabels**—Hides or shows all parameter labels

To apply an editor-specific filter to a structure chart:

1. From the **View** menu, choose **Apply Filter**.
2. On the **Apply Filter** dialog:
 - Make sure the **Location** group is set to either **StP Installation** or **Both**.
 - Select **Editor Specific Filters** to display a list of filters specific to structure charts
3. Select one or more filter(s) in the list.
4. Click **OK**.

The diagram reappears with the appropriate objects hidden or visible, depending on your selection(s).

To redisplay the diagram with all objects visible:

1. From the **View** menu, choose **Apply Filter**.
2. On the **Apply Filter** dialog, make sure the:
 - **Location** group is set to either **StP Installation** or **Both**
 - **Editor Specific Filters** option is not selected
3. In the **Filters** list, select **Show All**.
4. Click **OK**.

The diagram reappears with all objects visible.

For more information about using filters, see *Fundamentals of StP*.

Validating a Structure Chart

StP/SE editors provide two kinds of validation checks for the diagrams that comprise your system model:

- Syntax checks
- Semantic checks

This section describes how to check syntax and semantics for the current diagram in the Structure Chart Editor. You can also check semantics for the entire model or for one or more selected diagrams from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, “Checking SE Models.”

Syntax Checks

A structure chart diagram is syntactically complete and correctly drawn if:

- All module, library module, global data, offpage connector, and parameter symbols are labeled.
- All link connection rules are followed.

Checking diagram syntax does not check the contents of the repository.

Syntax checks are applied automatically when you save the current diagram, or you can choose the **Check Syntax** command on the **Tools** menu.

Semantic Checks

A diagram is semantically correct if all objects are properly defined in the repository and all elements are properly balanced. For a complete list of structure chart semantic checks, see “Structure Chart Editor Checks” on page 9-9.

To invoke semantic checking for the current diagram, choose either of the following commands from the **Tools** menu:

- **Check Semantics**—In addition to displaying results in the Message Log, optionally allows you to save results to a file in a selected format
- **Check Semantics Selectively**—Allows you to apply only selected semantic checks

8 Creating Flow Charts

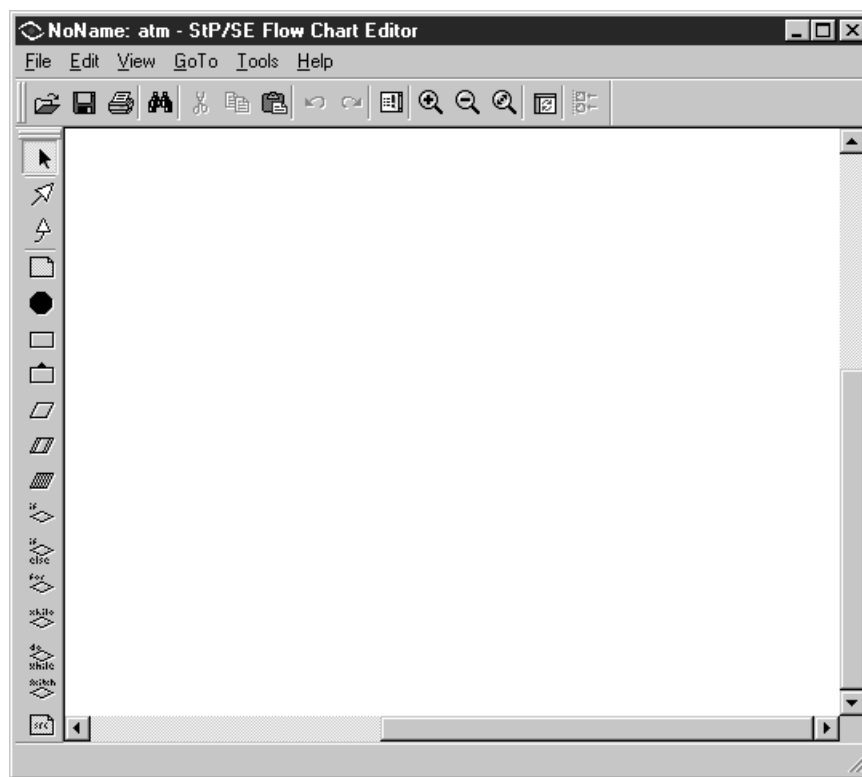
In support of structured design, StP/SE provides the Flow Chart Editor (FCE) for creating a graphical representation of the program design for individual functional modules in your system. These flow charts contain symbols representing several program elements, including library and function calls, conditional statements, and actual blocks of code.

This chapter describes:

- “Starting the Flow Chart Editor” on page 8-3
- “Using the Flow Chart Editor Symbols” on page 8-3
- “Navigating to Object References” on page 8-5
- “Associating a Flow Chart with a Function” on page 8-7
- “Creating an Entry Point” on page 8-7
- “Representing Function and Library Calls” on page 8-8
- “Representing Code Blocks and Macros” on page 8-9
- “Representing Conditional Statements” on page 8-10
- “Creating Exit Points” on page 8-12
- “Adding Source Comments” on page 8-15

This chapter includes a brief explanation of how to use the flow chart symbols available in the Flow Chart Editor. While some of these are standard symbols, others have been developed specifically for use with the StP/SE Flow Chart Editor and have no equivalent in standard flow chart notation. For general information on flow charting, refer to *Software Engineering—A Practitioner’s Approach*, by Roger S. Pressman (Fourth Edition, McGraw-Hill, 1996).

For instructions on automatically generating flow charts from existing C code, see Chapter 11, “Reverse Engineering.”



In addition to the standard StP diagram editor features, the Flow Chart Editor provides:

- Symbols for drawing flow charts
- Navigations to related diagrams

These features are described briefly in this section. For more information, see “Creating a Flow Chart” on page 8-7.

Starting the Flow Chart Editor

StP/SE provides access to the Flow Chart Editor from the:

- StP Desktop
- Structure Chart Editor

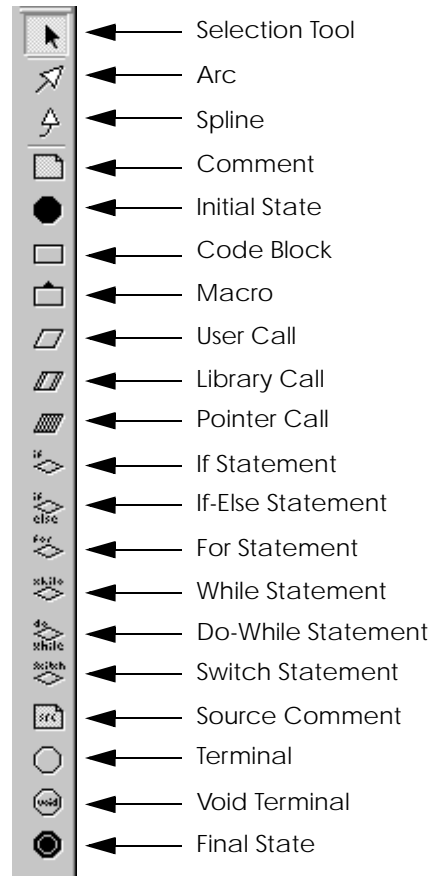
For information about starting the Flow Chart Editor from the Desktop, see “Invoking Other Desktop Commands” on page 2-6.

You can start the Flow Chart Editor from the Structure Chart Editor by using an appropriate navigation command from that editor’s **GoTo** menu. If the Flow Chart Editor has already been started, the navigation command uses the current session, rather than starting another copy of the editor. For information about starting the Flow Chart Editor from the Structure Chart Editor, see “Navigating to Object References” on page 7-6.

Using the Flow Chart Editor Symbols

You select symbols for drawing flow charts from the Symbols toolbar (Figure 2).

Figure 2: Flow Chart Editor Symbols Toolbar



Procedures for using the symbols are described in this chapter. For a summary description of each symbol, see Appendix B, “StP/SE Symbol Reference.” For general instructions on using the Symbols toolbar, see *Fundamentals of StP*.

Navigating to Object References

“Navigation” is the means by which you can display a different reference to an object whose symbol is selected on the current diagram. Use the **GoTo** menu to navigate. In the Flow Chart Editor, the commands on the **GoTo** menu allow you to display and edit:

- Another related flow chart
- The structure chart diagram containing the flow-charted function
- Existing C source code for the flow-charted function in a code viewer or editor

When you navigate to the Structure Chart Editor, the current Flow Chart Editor session continues, as well.

The **GoTo** menu provides context-sensitive choices for the selected symbol.

To navigate to a target:

1. Select a symbol on the current diagram.
2. Choose a command from the **GoTo** menu.

The navigation target appears.

If the selected symbol has more than one possible target, a selection list appears from which you choose the appropriate target. Navigations to C source code can occur only if the flow chart or an associated structure chart was reverse engineered from the C code, or if the C code was generated from an associated structure chart by the C code generator.

Table 1 describes the navigation targets and their associated menu commands that are available for each Flow Chart Editor symbol.

Table 1: GoTo Menu Commands

Navigate From	Navigate To	Command
Entire diagram or any symbol	Flow chart that calls this flow chart's modeled function.	Flow Chart's Caller
	Flow chart that models the function for a call appearing in this flow chart.	Flow Chart's Callee
	Formal definition of the SCE module that corresponds to the function modeled in the chart.	Flow Chart's SCE Module
	Source code for modeled function in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Flow Chart's Source Code Definition
User Call	Formal definition of the structure chart module that corresponds to the currently selected call.	Structure Chart Module
	Flow chart that models a function that calls this call.	Call's Caller
	Flow chart that models a function called by this call.	Call's Callee
	Defining flow chart for the currently selected call.	Call's Flow Chart
	Source code for selected user call in a code viewer or editor (see "Navigating from Model to Source Code" on page 12-38).	Source Code Definition
Library Call	A reference to the corresponding library module in the Structure Chart Editor.	Structure Chart Library Module
	Flow chart that models a function that calls this call.	Call's Caller
Pointer Call	Formal definition of the structure chart module that corresponds to the currently selected pointer call.	Flow Chart's SCE Module

Creating a Flow Chart

Although the Flow Chart Editor is intended primarily for flow-charting functions already defined in a structure chart, you can use this editor to create a flow chart for any purpose.

A flow chart consists of:

- An entry point, represented by an initial state symbol
- Other flow chart symbols, representing various program design elements
- One or more exit points, represented by terminal, void terminal, and final state symbols
- Arcs that link objects in the flow chart (optionally labeled; for example, to indicate true/false logic paths)

Associating a Flow Chart with a Function

In order to navigate between the structure chart defining a function and the flow chart describing the code elements for that function, you must associate the flow chart with a function defined in a structure chart diagram.

To associate the flow chart with a function in a structure chart, save the flow chart with the same name as the defined function.

This also enables you to navigate from the flow chart to the related C source code that was either created by the C code generator or used to reverse engineer the flow chart or related structure chart.

Creating an Entry Point

The entry point for a flow chart is an initial state symbol, shown in Figure 3. The initial state indicates the beginning of the program flow for a function.

Figure 3: Initial State Symbol



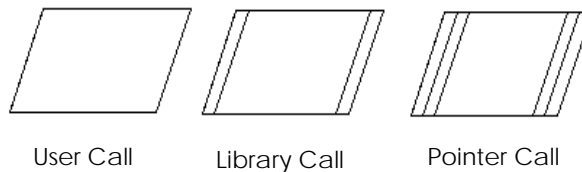
If you navigate to the Flow Chart Editor from the Structure Chart Editor to create a new flow chart, the flow chart automatically contains an initial state. Alternatively, you can drag and drop an initial state from the symbol list into the diagram. There should be only one initial state per function.

Figure 5 on page 8-9 shows an initial state symbol as an entry point connected to a library call.

Representing Function and Library Calls

Calls to user functions, library functions, and pointers to functions are represented by different types of parallelogram symbols in the Flow Chart Editor, as shown in Figure 4.

Figure 4: Call Symbols



A call symbol is only a reference that indicates where the call takes place within the program code. A call must be connected to a code block, which contains all information about call invocation, including parameter information.

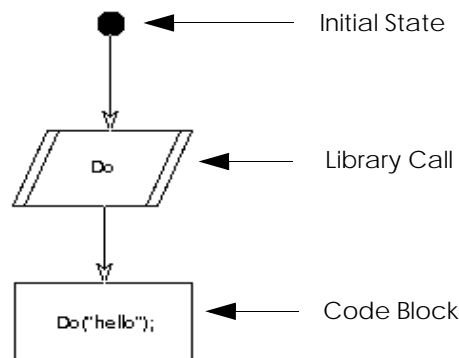
The code block can be located either above or below the call symbol, as described in “Representing Code Blocks and Macros” on page 8-9.

To draw a call:

1. Insert a call symbol into the drawing area.
2. Label the call.
3. Connect the call to a code block above or below the call with an arc.

Figure 5 illustrates a library call.

Figure 5: Initial State and Call Statement Example



Representing Code Blocks and Macros

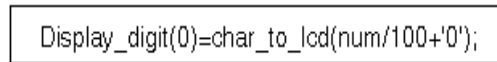
You can incorporate actual segments of code into your flow chart, using symbols for:

- Code blocks
- Macros

Code Blocks

A code block, shown in Figure 6, is represented by a simple rectangular symbol whose label contains a few lines of actual code. The code is generally context-specific. Typically, this code may consist of the parameter and other information associated with a corresponding call symbol that appears just before or after the code block.

Figure 6: Code Block Example

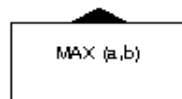


```
Display_digit(0)=char_to_lcd(num/100+'0');
```

Macros

A macro is represented by a rectangle symbol with a triangle on top, as shown in Figure 7. A macro usually represents a frequently used routine. A macro's label includes its name and parameters.

Figure 7: Macro Symbol



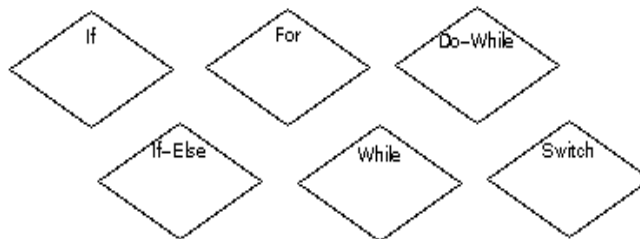
To draw a code block or a macro:

1. Insert a code block or macro symbol into the drawing area.
2. Label the symbol with the actual code to be executed.

Representing Conditional Statements

Diamond symbols represent conditional tests within your flow chart, as shown in Figure 8.

Figure 8: Conditional Symbols



Conditional statements have true or false logic paths, represented by the direction in which the arc leaves the conditional symbol:

- True—Indicated by an arc leaving the left side of the symbol
- False—Indicated by an arc leaving the right side of the symbol

You can optionally label the arcs with a user-chosen designation, such as “True” or “False.” However, the label has no effect on the true/false logic path.

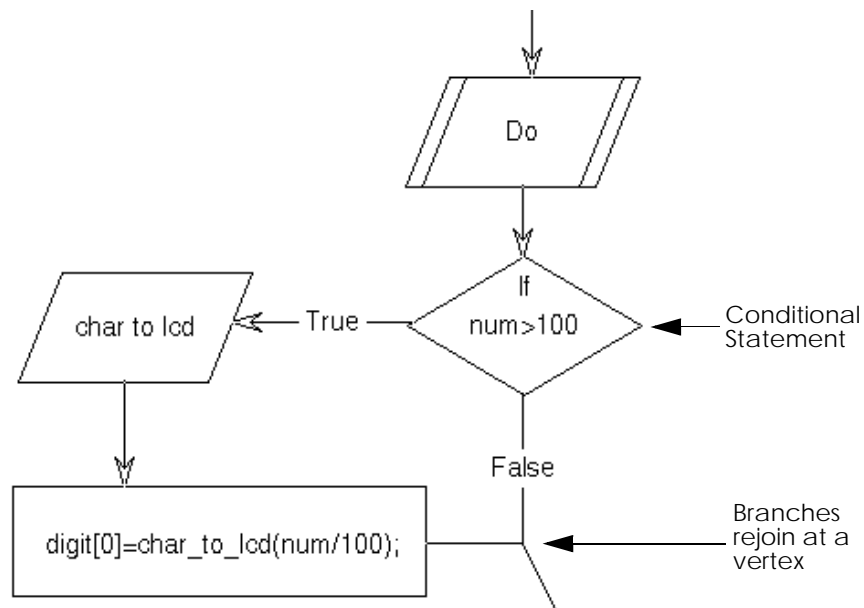
The symbol’s label contains code representing the condition to be tested, as shown in Figure 9.

To draw a conditional statement:

1. Insert a conditional statement from the Symbols toolbar into the drawing area.
2. Label the conditional statement.
3. Draw an arc from another symbol into the conditional statement.
4. Draw an arc from the conditional statement to indicate either a true or false logic path.
5. Optionally, label the True/False arcs.

Figure 9 shows an if conditional statement.

Figure 9: Conditional Statement Example



Creating Exit Points

Exit points, shown in Figure 10, indicate the logical end of a function in a flow chart. They may also indicate whether or not a value is returned from a function, depending on which symbol you use.

Figure 10: Exit Symbols



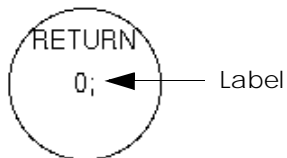
You can use the following types of exit symbols in your flow charts:

- Terminal symbol, which indicates the return of a value from within a particular branch of a function
- Void terminal, which indicates that no value is returned from that particular branch of a function
- Final state, which indicates the logical end of a function

To draw an exit symbol on a flow chart:

1. Insert an exit symbol from the Symbols toolbar into the drawing area.
2. If it is a terminal symbol, label it, as shown in Figure 11.

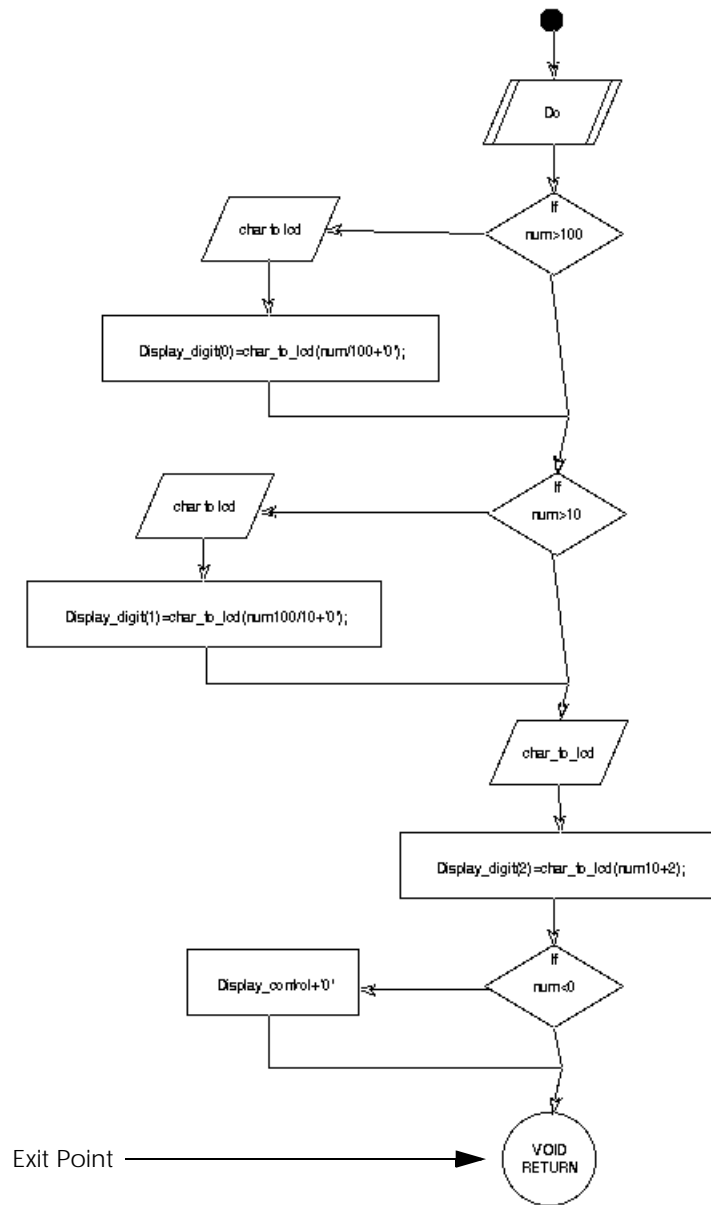
Figure 11: Labeling a Terminal Symbol



3. Draw a connection arc from another symbol to the exit symbol.

Figure 12 illustrates a simple flow chart with a void terminal exit point.

Figure 12: Flow Chart with Exit Point

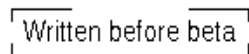


Adding Source Comments

In addition to the standard StP comment symbol for adding general comments to diagrams, the Flow Chart Editor provides a source comment symbol for representing source comments that appear in code. While the standard StP comment appears as text only, the Flow Chart Editor source comment contains comment text surrounded by brackets, as shown in Figure 13.

You can use an arc to connect a source comment to any Flow Chart Editor symbol.

Figure 13: Source Comment Symbol



[Written before beta]

Validating Flow Charts

StP/SE provides syntax checks for validating flow charts created with the Flow Chart Editor.

A flow chart is syntactically complete and correctly drawn if:

- All symbols are labeled.
- There is one start symbol and one end symbol.

Checking diagram syntax does *not* check the contents of the repository.

Syntax checks are applied automatically when you save the current diagram, or you can choose the **Check Syntax** command from the **Tools** menu.

There are no semantic checks for flow charts. However, you can check semantics for the entire model from the StP Desktop. For more information on semantic checking and on navigating to the source of a syntax or semantic error, see Chapter 9, “Checking SE Models.”

9 **Checking SE Models**

This chapter lists the semantic checks for each editor, describes the semantic checking commands available from the editors and the StP Desktop, and explains how to selectively limit the semantic checks to be applied. Additionally, it describes commands and options available for checking the structured model prior to C code generation and for checking the validity of the reverse engineered semantic model.

This chapter includes the following topics:

- “Syntax versus Semantic Checks” on page 9-1
- “Types of Semantic Checks” on page 9-2
- “Using Semantic Checking Commands” on page 9-9
- “Checking the Model for C Code Generation” on page 9-14
- “Checking the Semantic Model” on page 9-14

Syntax versus Semantic Checks

In StP/SE, you can apply the following validation checks to the diagrams and tables that comprise your system model:

- Syntax checks—A diagram is syntactically complete and correctly drawn if all objects are properly labelled and all connection rules are followed. A table is syntactically correct if it contains no duplicate names and cell values can be validated.
-

- Semantic checks—A diagram is semantically correct if all objects are properly defined in the repository according to methodological constraints and all elements are properly balanced. Semantic checks of tables are product-specific.

Syntax checks are applied automatically when you save the current diagram in each StP/SE editor. You can also apply syntax checks by choosing the **Check Syntax** command on the editor's **Tools** menu. For more information on syntax checks, see the chapters describing each editor.

Semantic checks are specific to each editor and can be applied to individual diagrams or tables or to your entire structured model at once. You can select certain categories of checks you want to run on diagrams. All semantic checks for each Cspec table or matrix are always executed.

Types of Semantic Checks

The tables in the following sections describe the semantic checks that are performed for each editor. You can select and apply certain semantic checks independently, using the **Check Semantics Selectively** command. The **Selectable?** column in each table indicates whether the semantic check can be selectively applied.

Data Flow Editor Checks

Table 1 lists the semantic checks StP applies to data flow diagrams.

Table 1: Data Flow Diagram Semantic Checks

Semantic Check	Selectable?	For More Details, See
All stores or flows in a given diagram are defined in an StP/SE data structure diagram	No	
Offpage flows balance with parent process	Yes	“Decomposition Balance Check” on page 9-4
Flows balance in and out of stores	Yes	“Store Balance Check” on page 9-4
Flows balance in and out of split flows and merged flows	Yes	“Split/Merge Flow Balance Check” on page 9-5
Flows balance in and out of each process	Yes	“Process Balance Check” on page 9-5
All processes have either a decomposition or Pspec	Yes	“Pspec or Decomposition Check” on page 9-5
Processes with Pspecs have no incoming control flows		
All Cspec bars have an associated table or diagram	Yes	“Cspec Implemented Check” on page 9-5

The following sections describe these semantic checks in more detail.

Decomposition Balance Check

This balance check examines all the flows into and out of a decomposition diagram to make sure they balance with all flows into and out of the diagram's parent process. For the decomposition diagram, the list of flows comprises everything flowing in from an offpage symbol (anchor, offpage process, or offpage external). Stores are not considered in this balance check, except as noted below. For the parent process, the list of flows comprises all flows into and out of that process, without regard to whether they come from another process, an offpage symbol, an anchor, or a store.

Two exceptions regarding stores are as follows:

- If the parent process has a flow into it from a store, the presence of the store is used to remove ambiguity, if needed.
- If a parent and child diagram both use a particular store, a special case checks for this and does not report an error. This allows modeling a store in a parent diagram, and, in a decomposition diagram, modeling the same store as either a store or as a flow in from an anchor.

You can add a store to a decomposition that is not in the parent diagram, without generating a balance error.

Store Balance Check

This balance check examines stores on the current diagram that are not utilized in the parent diagram to determine that:

- Any flow going into a store has a corresponding flow out of the store, and any flow out of a store has a corresponding flow into the store
- All flows going into and out of a store are actually part of the store

Note: Placing a store on a diagram is a legal way of introducing a new flow into a model; no balance check looks for the same store in a parent process diagram.

Split/Merge Flow Balance Check

This balance check examines split flows and merged flows to verify that complete preservation of data exists across the split or merged flows. It is possible to split a flow into its component pieces, but all parts of the parent data must be used in the segments of the split flow.

Process Balance Check

This semantic check works on all processes in the child diagram, individually, performing a balance check across the process. It considers only direct input flows to the process and direct output flows from the process.

If flows cannot be resolved from the inputs to the outputs of the process, an error is reported. This occurs even if a wider search of the entire diagram could resolve the ambiguity, because the ambiguity indicates a modeling error.

By default this semantic check is not applied automatically. It must be specifically selected on the **Check Semantics Selectively** dialog. It is especially useful to modelers who use processes to transform data to sub-parts of a structure. Modelers who treat processes strictly as data transformers, where the input data and output data are not necessarily parts of each other, may prefer not to use this check, since it reports semantic errors that are not actually errors in this circumstance.

Pspec or Decomposition Check

This semantic check verifies that all processes have either a Pspec or a decomposition diagram, but not both. If a process has a Pspec, it checks to make sure that the process has no incoming control flows.

Cspec Implemented Check

This semantic check verifies that if there is a Cspec bar in the diagram, it has at least one associated control specification table or state transition diagram. It does not check the associated table or diagram.

Data Structure Editor Checks

Table 2 lists the semantic checks StP applies to data structure diagrams.

Table 2: Semantic Checks for Data Structure Diagrams

Semantic Check	Selectable?
All typedefs and primitive sequences/selections have valid Data Type annotations	Yes
No non-primitive elements have Data Type annotations	Yes
All data types are either defined on a data structure diagram or are predefined C types (void, char, int, float, double, short, long)	No
Selections have at least one child on either the parent diagram or in a decomposition diagram	No

State Transition Editor Checks

Table 3 lists the semantic checks StP applies to state transition diagrams.

Table 3: Semantic Checks for State Transition Diagrams

Semantic Check	Selectable?
There is exactly one initial state per diagram	Yes
Each transition has an associated Event/Action	Yes
No two transitions have the same start node and event, and no transition is referenced more than once	Yes
No state node is referenced in another diagram	Yes
Each event is defined in a Cspec Event Logic Table	Yes
Each action is defined in a Cspec Action Logic Table	Yes

Control Specification Editor Checks

The semantic checks for control specifications include a separate set of mandatory checks for each of the different table types. You cannot apply control specification checks selectively.

To check all tables in a specified Cspec, from the StP Desktop select the **All Control Spec** category in the Model pane and a Cspec in the objects pane; then choose **Check Semantics for Selected Objects** from the **Tools > Check** menu.

Table 4: Semantic Checks for Control Specifications

Semantic Check	C s p e c	Table Type						
		A L T	D E T	E L T	P A M	P A T	S E M	S T T
Each process must have a reference on the corresponding data flow diagram.	✓	✓			✓	✓		
Each control in cell's data elements must have a matching control flow entering the Cspec bar.	✓		✓	✓	✓	✓		
Each control out cell's data elements must have a matching control flow leaving the Cspec bar.	✓	✓	✓					
Control value links must be valid values for the control in or control out cell to which they correspond. The set of valid values is defined by the Allowed Value annotation items on any data structure diagram symbol that has the same name as the control in or control out cell.	✓	✓	✓	✓		✓		
Each control combination cell can cause any given event only once.	✓			✓				

Table 4: Semantic Checks for Control Specifications (Continued)

Semantic Check	C s p e c	Table Type						
		A L T	D E T	E L T	P A M	P A T	S E M	S T T
Each control combination cell can include any given control in or control out cell only once.	✓	✓	✓	✓		✓		
Process activation links must have either non-negative integer values or “T” (for triggered processes).	✓	✓			✓	✓		
Each control in cell can activate a given process combination only once.	✓				✓			
Each action can activate a given process combination only once.	✓	✓						
Each action can create a given control combination only once.	✓	✓						
Each process combination can include a given process cell only once.	✓	✓			✓			
Each state must be the target of at least one transition (the state must be reachable).	✓						✓	✓
Each event must be defined in an ELT.	✓						✓	✓
Each action must be defined in an ALT.	✓						✓	✓
There must be no ambiguous transitions (multiple transitions from a state cannot be triggered by the same event).	✓						✓	✓
All incoming control flows must be used within the Cspec.	✓							
All outgoing control flows must be generated by the Cspec.	✓							

Structure Chart Editor Checks

Table 5 lists the semantic checks StP applies to structure charts.

Table 5: Semantic Checks for Structure Charts

Semantic Check	Selectable?
Modules have a formal SCE definition—that is, each program module should either be connected directly from a formal caller node (anchor) or have no in links	Yes
Modules not formally defined have a PDL definition	Yes
Actual parameters do not have data type annotations	Yes
Formal parameters have valid data type annotations	Yes
Modules have same number of formal and actual parameters—that is, the number of parameters on formal calls and corresponding actual calls match	Yes
Modules and globals have valid data type annotations	Yes
Offpage connectors with outgoing links have corresponding ones with ingoing links	Yes

Flow Chart Editor Checks

There are no semantic checks for flow charts.

Using Semantic Checking Commands

Semantic checking commands appear in almost every SE editor and also on the StP/SE Desktop. Some commands operate on the current or selected diagram or table only, applying semantic checks specific to that editor. Other commands operate on your entire structured model.

Table 6 describes the StP/SE semantic checking commands.

Table 6: Semantic Checking Commands

From	Command	Description
StP Desktop Tools > Check menu	Check Semantics for Selected Objects	Checks semantics defined for the selected StP/SE editor on the selected diagram(s) or table(s) in the objects pane. Optionally allows you to save results to a file in a selected format.
	Check Semantics for Whole Model	Checks semantics for all diagrams and associated Cspec tables in your system (available from the StP Desktop only).
	Check Semantics Selectively	Displays a dialog box for choosing semantic checks to be run on the selected diagram(s) in the objects pane, then executes the selected semantic checks. For details, see “Checking Semantics Selectively” on page 9-12.
Diagram editor Tools menu	Check Semantics	Checks semantics for the current diagram only. Optionally allows you to save results to a file in a selected format.
	Check Semantics Selectively	Displays a dialog box for choosing semantic checks to be run on the current diagram.
Table editor Tools menu	Check Table Semantics	Checks semantics for the current table only. Optionally allows you to save results to a file in a selected format.
	Check Cspec Semantics	Checks semantics for all tables associated with this Cspec. Optionally allows you to save results to a file in a selected format.

Checking Semantics from the StP Desktop

From the Desktop, you can check the semantics of:

- One or more selected diagrams or tables
- The entire model

To apply semantic checks from the StP Desktop:

1. In the Model pane, select or open a category.
2. To limit semantic checking to certain types of diagrams, tables, or model elements, select a subcategory in the Model pane.
3. To limit semantic checking to specific diagrams, tables, or model elements, select them in the objects pane.
4. Choose the appropriate semantic checking command from the **Tools > Check** menu (for the **Check Semantics Selectively** command, see “Checking Semantics Selectively” on page 9-12).
5. If the **Check Semantics for...** dialog box appears, optionally specify a file format and file in which to store the semantic checking results; then click **OK** (see “Specifying a File and Format for Error Messages” on page 9-12).

The appropriate semantic checks are applied and potential errors appear in the Message Log and, optionally, in the user-specified file.

Checking Semantics from Within an Editor

To apply semantic checks from within an StP/SE editor:

1. Open the diagram or table you want to check.
2. Choose the appropriate semantic checking command from the **Tools** menu (for the **Check Semantics Selectively** command, see “Checking Semantics Selectively” on page 9-12).

The appropriate semantic checks are applied and potential errors appear in the Message Log and, optionally, in the user-specified file.

Specifying a File and Format for Error Messages

All semantic checking commands display potential semantic errors in the StP Message Log by default. Additionally, each command (except **Checking Semantics Selectively**) brings up the **Check Semantics** dialog, which allows you to save the results to a table in a user-specified file in one of the following formats:

- MIF (FrameMaker)
- RTF (Microsoft Word)
- HTML

Note: If you do not specify a file and format in the **Check Semantics** dialog, you can still save the error messages to a file from the Message Log and then print the file in ASCII format.

To use the **Check Semantics** dialog:

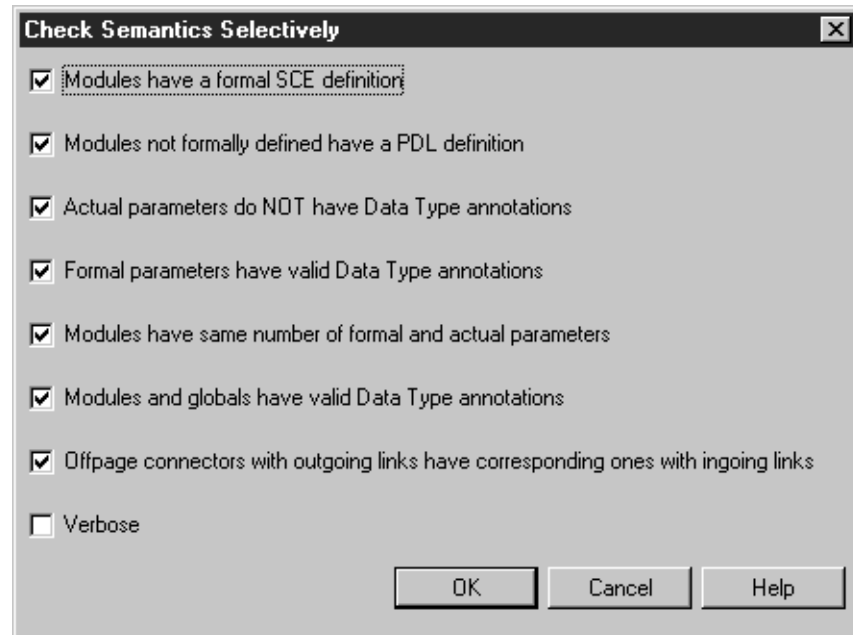
1. In the **Output Format** field, optionally specify the format for any generated error messages. By default, semantic error messages appear in the StP Message Log.
2. In the **Output File** field, optionally specify the file to contain the error messages, if desired.
3. Click **OK**.
The appropriate semantic checks are applied and potential errors appear in the Message Log and, optionally, in the user-specified file.

Checking Semantics Selectively

When you are checking diagrams from within an editor, you can select the checks you want to run using the **Check Semantics Selectively** command. This command displays the **Check Semantics Selectively** dialog box for the current editor, which contains editor-specific choices. Figure 1 shows the **Check Semantics Selectively** dialog box for the Structure Chart Editor.

Each **Check Semantics Selectively** dialog box contains a **Verbose** option. If selected, status messages about the checking process are displayed in the Message Log, in addition to any semantic errors. If not selected, only the semantic errors appear in the Message Log.

Figure 1: Example of Check Semantics Selectively Dialog Box



To use the dialog box:

1. Do either of the following:
 - From the Desktop **Tools** menu, choose **Check > Check Semantics Selectively**
 - From the editor **Tools** menu, choose **Check Semantics Selectively**
2. In the **Check Semantics Selectively** dialog, select semantic check options, as desired.
3. Click **OK**.

The selected semantic checks are applied. Error messages and warnings appear in the editor's message log.

Navigating to the Source of an Error

When running syntax or semantic checks on a single diagram or Cspec table, you can navigate from most errors in the message log to the affected part of the diagram to make corrections.

To navigate to the affected part of the diagram or table, do either of these:

- Double click on the semantic error message
- Select the error message and click the **Go to Error Origin** button.

The affected diagram or table appears in the appropriate editor, with one or more symbols or table cells containing errors highlighted.

Note: If the **Go to Error Origin** button is dimmed, navigation from the selected message is not provided.

Checking the Model for C Code Generation

To ensure that structure chart and data structure diagrams are complete enough to generate useful C code, StP/SE automatically executes certain validation checks on these models during code generation. Failed checks are reported as warnings. You can also run the checks from the **Generate C Code** options dialog box without generating code. For details, see “Checking the Model for Code Generation” on page 10-26.

Checking the Semantic Model

To ensure that the semantic model created by Reverse Engineering is usable and has not been corrupted, use the **Check RE Semantic Model Consistency** command. This command checks the semantic model that was reverse engineered from C code in order to determine if its indices and data fields are valid or not. The command appears on the StP Desktop **Code > Reverse Engineering** menu. For more information, see “Checking the Semantic Model” on page 11-27.

10 Generating C Code

StP/SE provides the ability to automatically generate C code from structure charts and data structure diagrams created with the Structure Chart Editor and the Data Structure Editor.

The C Code Generator provides the following capabilities:

- Support for both ANSI and Kernighan & Ritchie (K&R) C
- Generation of code requiring minimal editing before compiling
- Support for partial code generation for a single file, a set of files, or a complete system
- Incremental code generation, which updates generated code from modified diagrams while retaining unchanged code
- Ability to check model for completeness prior to generating code
- User-configurable script for generating code that conforms to local coding conventions

This chapter includes the following topics:

- “What Code is Generated?” on page 10-2
 - “How is C Code Generated?” on page 10-2
 - “Assigning Annotations for Code Generation” on page 10-3
 - “SCE Support for Code Generation” on page 10-6
 - “DSE Support for Code Generation” on page 10-14
 - “Include File Generation” on page 10-20
 - “Illegal Variable Names” on page 10-21
 - “Generating Code” on page 10-22
 - “Updating and Editing Code” on page 10-27
-

What Code is Generated?

StP/SE generates C code that conforms to either ANSI C or Kernighan and Ritchie (K&R) C standards. StP/SE generates code to:

- Source files (.c files)
- Header files (.h files)

For more information, see “Output Files and Directories” on page 10-22.

How is C Code Generated?

StP/SE generates C code from objects in the repository, using a Query and Reporting Language (QRL) script. You can customize the QRL script to conform to local coding conventions. For information about QRL scripts, see *Query and Reporting System*. For more information about the objects in the repository, see *Object Management System*.

The QRL script searches the repository and extracts certain information about the objects in structure charts and data structure diagrams, including specific object annotations. The extracted information is then used to produce the C code.

The quality and completeness of the generated C code depends to a large extent on the annotations you apply to the objects in your diagrams. Results also depend on the options you choose during code generation.

Assigning Annotations for Code Generation

Some objects require specific annotations in order to generate correct C code. You can assign most of these as properties on the object property sheet, using the editor's **Edit Object Properties** dialog box as described in the relevant editor chapter. Others can only be assigned as annotations, using the Object Annotation Editor (OAE) described in *Fundamentals of StP*.

These properties and annotations are described briefly in "Structure Chart Annotations Used in Code Generation" on page 10-3 and "Data Structure Annotations Used in Code Generation" on page 10-5.

For more detailed information about object properties, annotations, and other factors affecting code generation, see "SCE Support for Code Generation" on page 10-6 and "DSE Support for Code Generation" on page 10-14.

Structure Chart Annotations Used in Code Generation

The following table briefly describes object annotations that are used for or affect C code generation from structure charts. Annotations that you can assign as properties on the object property sheet are indicated by the word "Property" in the Property or Annotation column. For more information on each annotation, see the section referenced in the table.

Table 1: Structure Chart Annotations for Code Generation

Object	Property or Annotation	Description	For Details, See
Modules, Globals, Parameters	Type (property)	Return type of a module, or data type of a global variable or parameter. Default is int.	“Function Return Types” on page 10-11 “Global Data” on page 10-13 “Function Pointers” on page 10-10
	Storage Class (property)	C storage class for a function, global, or parameter. It is the user’s responsibility to assign appropriate values according to the object and C dialect. Accepted values are: const , static , static const , register , and register const . Unset by default.	“Storage Class for Functions, Globals, and Parameters” on page 10-12
Modules	C Code Body note	String representing the body of a function, optionally placed in the generated code for the module.	“Function Code Bodies” on page 10-13
	Module Comment note	String representing a user-entered comment about a function, optionally placed in the “Comment” portion of the function header comment.	“Descriptive Function Headers” on page 10-8
	Module Definition note/ Variable Arguments item	Annotation indicating whether a function has a variable number and type of parameters. Default value for the Variable Arguments item is False.	“Variable Argument Functions” on page 10-9
Globals	Array Size (property)	String representing the size of the array. Unset by default.	“Arrays” on page 10-13
	Global Comment note	String representing a global comment.	

Table 1: Structure Chart Annotations for Code Generation

Object	Property or Annotation	Description	For Details, See
SE File	SE File Definition note/ Included File item	String identifying files to be #included into the .c file.	"Include File Generation" on page 10-20

Data Structure Annotations Used in Code Generation

The following table briefly describes object properties and annotations that are used for or affect C code generation from data structure diagrams. Annotations that you can assign as properties on the object property sheet are indicated by the word "Property" in the Property or Annotation column. For more information on each annotation, see the section referenced in the table.

Table 2: Data Structure Annotations for Code Generation

Object	Property or Annotation	Description	For Details, See
All	Data Structure Comment note	String representing a user-entered comment about a data structure, optionally placed directly before the structure definition in the generated code.	"Data Structure Comments" on page 10-14

Table 2: Data Structure Annotations for Code Generation

Object	Property or Annotation	Description	For Details, See
Sequences Selections Typedefs	Array Size (property)	String representing the size of the array. Unset by default.	“Arrays” on page 10-19
	Type (property)	Data type of a data structure element. Default is int.	“Creation of Data Definitions” on page 10-15
	System Type (property)	Selectable option that tells StP to interpret the data element as a system type, for which it generates no definition.	“System Types” on page 10-19
Intermediate nodes (of any type)	Structure Tag (property)	String that specifies the name of the C structure tag for the struct, union, or enum data object. Unset by default.	“Abstract Data Types” on page 10-16 and “Structure Tags” on page 10-17
SE File	SE File Definition note/ Included File item	String identifying files to be #included into the .h file.	“Include File Generation” on page 10-20

SCE Support for Code Generation

Code generation from Structure Chart Editor (SCE) diagrams includes support for:

- Both K&R and ANSI C
- Descriptive function headers
- Function definitions with formal parameters
- Variable argument functions
- Function pointers

- Function return types
- Function calls with actual parameters
- Storage class for functions, globals, and parameters
- Function code bodies
- Arrays
- Global data
- Include file generation

The following sections discuss how object annotations and other factors affect code generation from SCE diagrams with respect to these issues. For information on included files, see “Include File Generation” on page 10-20.

ANSI versus K&R C

StP/SE enables you to generate either ANSI or K&R C from SCE diagrams. The three main differences between ANSI and K&R C that affect the generated code are:

- Function definitions
- Function prototypes
- Functions that accept variable arguments

Function Definitions

The ANSI standard differs from K&R C in the way a function’s formal parameters are defined. In the ANSI form, the names and types of each parameter appear in the parenthesized list.

```
float  
func1 (int a, float b)  
{  
/* some code here */  
}
```

If you elect to generate K&R C, formal parameters are generated in the traditional form:

```
float
func1 (a, b)
int a;
float b;
{
}
```

Function Prototypes and Forward Declarations

The ANSI standard allows the capability for function prototyping, which requires the types of the arguments as well as the return type of a function to be declared in advance. This helps to enforce strong type checking. Using the example from “Function Definitions” on page 10-7 for `func1`, the following prototype is generated:

```
float func1(int, float) ;
```

Since K&R C does not support function prototypes, forward declarations are generated instead. Using the same example for `func1`, the following forward declaration is generated when creating K&R C code:

```
float func1() ;
```

The C code generator creates a corresponding prototype file for each C source file. For example, file *main.c* has a corresponding file named *main_protos.h*, which contains function prototypes or forward declarations, depending on whether you are generating ANSI or K&R C code. All functions defined in file *main.c* have a prototype or forward declaration in *main_protos.h*, except for statically defined functions, which are defined at the top of the C source file. Prototypes for statically defined functions are enclosed within tags to facilitate incremental code generation.

Functions Accepting Variable Arguments

Functions that accept a variable number of arguments are declared differently in ANSI C than they are in K&R C. The difference between these styles is described in “Variable Argument Functions” on page 10-9.

Descriptive Function Headers

Whether or not a module has a description, the code generator automatically generates one, as shown in the following example.

Using the OAE, you can add a Module Comment annotation note to a module. If you select the **Generate Comments** option during code generation, the C code generator uses the text entered in the note's description field to generate a function header comment for the object. If an object has no Module Comment annotation, the code generator creates a function header comment line with no comment text.

The comment appears directly preceding the function definition. For example:

```
/****** StP/SE Function *****/
* (321:14)
*
* Function: main
*
* Calls:
*   module1
*   module2
*
* Description:
*   Module created by the SE Code Generator.
*
* Comment:
*
*****/

/* StP/SE module 321:14 */
int
main(int argc, char ** argv)
{
```

Function Definitions with Formal Parameters

Code is generated only for SCE modules that are root nodes. A root node is either linked to an anchor point (that is, a formal caller) or it has no in links. Parameters attached to the link between an anchor and a module symbol are generated as the formal parameters in the function header. Parameters are evaluated from left to right, top to bottom.

If there is more than one root node for a module, then a warning appears and no formal parameters are generated; a concatenation of all child nodes are used for the definition of the module. That is, all of the children for the root node are taken into account, whether or not they are drawn on the same diagram.

Variable Argument Functions

Using a module annotation, you can indicate that the function represented by the module accepts a variable number and type of arguments. This annotation affects the way the function is declared in the generated code. To indicate that a module represents a variable argument function, set its return type to boolean and use the OAE to add a Module Definition annotation note with a Variable Arguments item set to a value of True.

The way a variable argument function is declared in the generated code differs in ANSI and K&R C.

For ANSI C

The function declaration for variable arguments contains an ellipsis. For example:

```
func (int a, int b, ...)
```

Any function that uses a variable number of arguments must have at least one non-variable argument. For example, the following is incorrect:

```
func (...)
```

The ANSI C code generator checks the SCE diagram to ensure that functions defined as having variable arguments have at least one non-variable parameter that is passed to it with each call.

The code generator also adds a `#include` directive to each generated file containing a function with a variable number of parameters. For ANSI the `#include` directive is:

```
#include <stdarg.h>
```

For K&R C

In place of formal parameters, the function declaration contains the macro `va_alist`. Also, the macro `va_dcl` appears in place of parameter type declarations. For example, if function `print_results` is declared as having a variable number of arguments, the function declaration looks like this:

```
print_results(va_alist)
va_dcl
{
```

The code generator also adds a `#include` directive to each generated file containing a function with a variable number of parameters. For K&R C the `#include` directive is:

```
#include <varargs.h>
```

Function Pointers

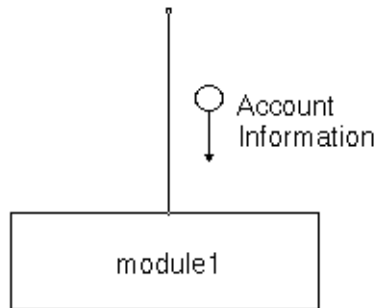
To facilitate code generation of function pointers, the data type for an SCE parameter representing a function pointer must be specified in the following format, in which you use an At sign (@) in place of the object name:

```
<datatype> (*@)()
```

You can specify the data type of an SCE parameter in the **Type** field on the object property sheet or by annotating the object using the OAE. This creates a Parameter Definition annotation note with an item called Data Type.

Figure 1 provides an example containing a function *module1* with a formal parameter *Account Information* that is a pointer to a function.

Figure 1: SCE Parameter as a Function Pointer



The data type for parameter *Account Information* should be specified as:

```
int (*@)()
```

The following code is generated from this data type specification:

```
module1(int(*Account Information)())  
{  
  ...  
}
```

Function Return Types

You can specify the return type of an SCE module in the **Type** field on the object property sheet or by annotating the object using the OAE. This creates a Module Definition annotation note with an item called Module Return Type.

If a return type other than a basic C type has been specified for a module in an SCE diagram, the C code generator checks to ensure that a data structure diagram exists for the specified module return type. If a data structure diagram exists, the type definition is generated into the appropriate **.h* file.

If the module does not have an assigned return type, `int` is assumed and the code generator issues a warning.

Function Calls with Actual Parameters

Calls to functions are generated as part of the function code body and are enclosed within comments.

The arguments to the functions being called are taken directly from the labels on the parameters attached to the link between the calling module and the called module on an SCE diagram. The parameters are defined to be the actual parameters. Parameters are evaluated from left to right, top to bottom.

Storage Class for Functions, Globals, and Parameters

You can specify a C storage class for a module, global, or parameter in the **Storage Class** field on the object property sheet or by annotating the object using the OAE. This creates one of the following annotations:

- Module Definition note with an item called Module Storage Class
- Global Definition note with an item called Storage Class
- Parameter Definition note with an item called Storage Class

During code generation, an appropriate qualifier is added to the definition for the function, global variable, or parameter, based on the value of the storage class annotation. For more information about storage classes, see “Choosing a C Storage Class” on page 7-29.

Function Code Bodies

Using the OAE, you can add a C Code Body annotation note to a module. The note description accepts a text string representing the body of a function.

If you select the **Generate C Code Body** option during code generation, the C code generator inserts the text in the note’s description field directly into the function body. No checking is performed to validate that the text string is valid C code.

Arrays

You can specify an array size for a global data object in the **Array Size** field on the object property sheet or by annotating the object using the OAE. This creates a Global Definition annotation note with an item called Array Size. You enter the array size as a text string, optionally enclosed in square brackets.

During code generation, the string is used, as is, for the size of an array. For example, if the Array Size for a global data object is defined as [20][30][40], the generated code is:

```
int array_var [20][30][40];
```

If the text you enter does not include the beginning and ending square brackets, the brackets are automatically added. For example, if the Array Size is entered as 1000, the brackets are automatically added and the generated code is:

```
int array_var [1000];
```

Global Data

If a module accesses a global, the global variable appears as part of the function code body in the generated code. The global variable usage is enclosed within comments. If the global is modeled as being defined in the same file as the module (that is, they have the same file scope), then a definition for the global appears at the top of the file where it is accessed. Otherwise, the global is declared as extern at the top of the file, and the definition is generated to the file indicated by the file scoping object.

You can specify the data type of a global in the **Type** field on the object property sheet or by annotating the object using the OAE. This creates a Global Definition annotation note with an item called Data Type. The data type is used in the global variable definition. For example, if a module accesses a global named *global_var* of type char *, the definition for the global looks like this, where <storage class> is a C storage class specified for the global:

```
<storage class> char * global_var;
```


You specify the C storage class for a global in the **Storage Class** field on the object property sheet, or by using the OAE to add a Global Definition annotation note with an item called Storage Class.

DSE Support for Code Generation

Code generation from Data Structure Editor (DSE) diagrams includes support for:

- Data structure comments
- Creation of data definitions, including abstract data types and typedefs
- System types
- Arrays
- Ifndef statements
- Include file generation

The following sections discuss how object annotations and other factors affect code generation from data structure diagrams with respect to these issues. For information on included files, see “Include File Generation” on page 10-20.

Data Structure Comments

Using the OAE, you can add a Data Structure Comment annotation note to a data structure object. If you select the **Generate Comments** option during code generation, the C code generator uses the text entered in the note’s description field to generate a data structure comment for the object. If an object has no Data Structure Comment annotation, the code generator creates a data structure comment line with no comment text.

The comment appears directly preceding the data structure definition, enclosed within comment tags. For example:

```
/***** StP/SE Data Element *****/
* (320:18)
*
```

```
* Data Element: class
*
* Description:
*   Created by the SE Code Generator.
*
* Comment:
*
*
***** /

/* StP/SE data definition 320:18 */
struct class
{
...
}
```

Creation of Data Definitions

As code is generated from structure chart models, a list of data types is kept for each global, parameter, or return type encountered. The code generator then generates a definition for each of these types from the data structure diagram in which it is defined. It puts the generated definition into the appropriate header file to be included in the source file.

In StP/SE, each data element is assigned one of the following data types:

- Predefined, basic C type (for example: int, float, or char *)
- C abstract data type (ADT) whose structure is defined in a Data Structure Editor type definition diagram
- System type

If a data type is not assigned, int is assumed.

For details about code generation of abstract data types, see “Abstract Data Types,” which follows. For details about code generation of system types, see “System Types” on page 10-19.

Abstract Data Types

Each time the code generator encounters a C abstract data type (ADT), it must determine whether the C ADT is a struct, union, enum, or typedef. After locating the data structure object representing the object's type, it analyzes the type's data structure. The data structure symbol

representing the root of the type's data structure determines the C ADT. The following table shows the corresponding C ADT for each data structure root object. For example, if the root object for the type's data structure is a sequence symbol, the code generator interprets the C ADT for the object as a struct.

Table 3: C Abstract Data Types for Data Structure Objects

Data Structure Root Object	Corresponding C Abstract Data Type
Sequence	struct
Selection	union
Enumeration	enum
Typedef	typedef

Sequences, selections, and enumerations may have children, which represent the components of the ADT. The children are evaluated from left to right, top to bottom.

A child of a sequence or selection can either have an assigned data type or children of its own, but not both. The data type can be either a basic C type or another abstract data type. A child that has children represents a nested struct or union.

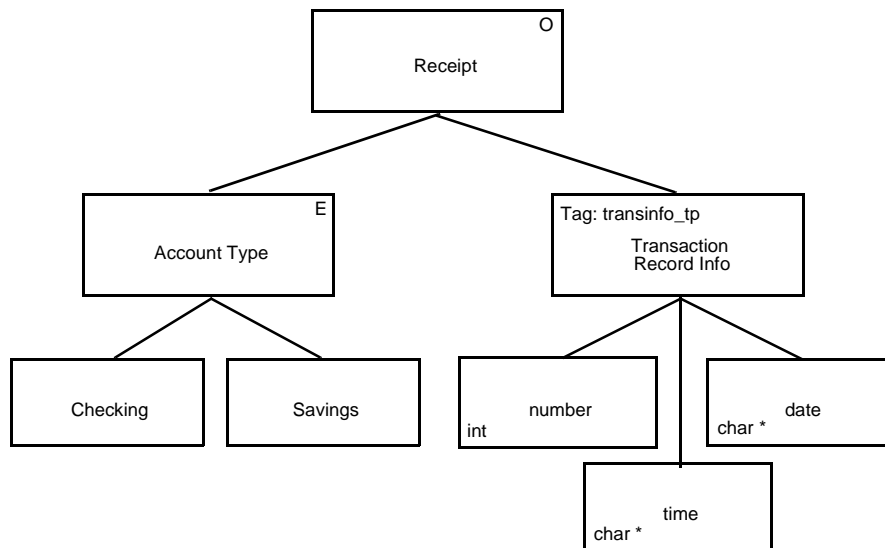
Structure Tags

You can assign a structure tag to any sequence, selection, or enumeration object that is an intermediate node in a data structure diagram. You do this by entering a string representing a tag name in the **Structure Tag** field on the object property sheet, or by using the OAE to add a Tag annotation item for the Data Definition note. The string contains the name for the C structure tag to be used for the struct, union, or enum.

Example

This example illustrates an abstract data type (Figure 2), followed by the code that is generated for it.

Figure 2: Abstract Data Type Code Generation Example



The generated code for the structure shown in Figure 2 is:

```
union Receipt
{
    enum
    {
        Checking;
        Savings;
    } Account Type;
    struct transinfo_tp
    {
```

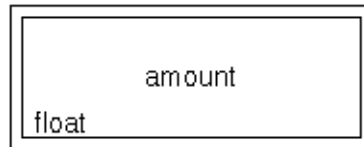
```
int number;  
char * date;  
char * time;  
} Transaction_Record_Info;  
};
```

Typedefs

A typedef symbol defines a new name for an existing data type. The typedef's data type is a string representing the real type that the typedef redefines. You can assign the data type by entering it in the **Type** field on the object property sheet, or by using the OAE to add a Data Type annotation item for the Typedef Definition note.

The following example shows a typedef that defines a new type called "amount" for an existing data type, float.

Figure 3: Typedef with a Type Annotation



The resulting generated code is:

```
typedef float amount;
```

System Types

Using the object property sheet or the OAE, you can define a data element (a sequence or selection object that has no children) as a system type (for example, pid or time_t), rather than assigning it a data type. This creates a Data Definition annotation note with a System Type item of type boolean, set to True.

The C code generator does not generate a data definition for an object defined as a system type. Instead, the object's file scope determines the appropriate #include directive to be included into files referencing the data element.

Arrays

You can specify an array size for a sequence, selection, or typedef object in the **Array Size** field on the object property sheet or by annotating the object using the OAE. This creates a Data Definition annotation note with an item called Array Size. You enter the array size as a text string, optionally enclosed in square brackets.

During code generation, the string is used, as is, for the size of an array. For example, if the Array Size for a data element is defined as [20][30][40], the generated code is:

```
int array_var [20][30][40];
```

If the text you enter does not include the beginning and ending square brackets, the brackets are automatically added. For example, if the Array Size is entered as 1000, the brackets are automatically added and the generated code is:

```
int array_var [1000];
```

Ifndef Statements

Include files contain the appropriate #ifndef statements to prevent compiler errors from occurring due to a file being included more than once, as shown in the following example include file, *module_tp.h*:

```
/****** StP/SE File *****/
* File: module_tp.h
*
*   created on Thu Aug 03 16:49:28 1995 for
*   mcmann@woodstock from system mcmann
*
*   modified on Thu Aug 03 16:57:36 1995 for
*   mcmann@woodstock from system mcmann
*
* Components:
*   union_field_tp
*   module_tp
*
*****/
```

```
#ifndef _MODULE_TP_H
#define _MODULE_TP_H

[data type definitions go here]
#endif          /* _MODULE_TP_H */
```

Include File Generation

Using the OAE, you can add to an SEFile object an SEFile Definition annotation note with multiple Included File items, one for each file to be included. The value of the Included File item is a text string representing the name of the file to be included. Each Included File item should contain only one file name.

To start the OAE to create or edit an SEFile object annotation:

1. From the **Model Elements** category on the StP/SE Desktop, choose **File Objects**.
2. Select an SEFile object from the objects pane.
3. From the **Tools** menu, choose **Edit Annotation**.
4. In the OAE, add the SEFile Definition note and an Included File item for each file to be included (for details on using the OAE, see *Fundamentals of StP*).

The C code generator takes the Included File text string as is, encloses it within double quotes, and generates it as `#includes` into the `.c` or `.h` file. For system include files, enclose the text string within angled brackets (`< >`) in the annotation, for example, as in `<stdio.h>`. This tells the code generator to generate the string without the double quotes.

For example, if file *main.c* has two Included File annotations, *module_tp.h* and *<stdio.h>*, the following is generated at the top of the file where *main* is defined:

```
/* Start of main.c */
#include "module_tp.h"
#include <stdio.h>
```

If *module_tp.h* has two Included File annotations, *<stdlib.h>* and *file_list.h*, the following is generated at the top of the file where *module_tp.h* is defined:

```
/* Start of module_tp.h */
#include <stdlib.h>
#include "file_list.h"
```

Illegal Variable Names

The code generator replaces spaces and other characters that are not legal for C identifiers with underscores (_).

The following are illegal characters for C identifiers:

!@#\$%^&*()+= | \ \{ } , . / ? ~ ' ; : " \ n \ t

Generating Code

Table 4 lists the code generation commands available from the StP Desktop and from the Structure Chart Editor. Additionally, you can execute code generation directly from the command line, using Query Reporting System (QRS), as described in the *Query and Reporting System* manual.

Table 4: C Code Generation Commands

Command	Menu	For		Description
Generate C for Entire Model	Desktop Code menu	(All Desktop categories and subcategories)		Generates C code for all structure charts and associated data structures in the entire system model.
Generate C		Diagrams category	Structure Chart	Generates C code for selected structure charts and associated data structures.
		Model Elements category	Directory Objects	Generates C code for a selected directory or set of directories (includes associated structure charts and data structures).
			File Objects	Generates C code for a selected file or set of files (includes associated structure charts and data structures).
Generate C Code	SCE menu	Structure Chart Editor		Generates C code for the currently loaded diagram and its associated data structures.

Output Files and Directories

Each structure chart program module, global data object, and data structure root object in an StP/SE model is scoped to a default or user-specified directory and file defined by their assigned SEDirectory and SEFile objects. SEDirectory objects determine the output directories for the generated code. SEFile scope objects determine the names of the code output files. By default, structure chart program modules, global data, and data structure root objects are initially scoped to a default directory represented by a period (.) in the object's SEDirectory scope qualification. Users can change the default scope for individual objects to a user-specified directory.

At code generation time, StP/SE interprets the default (.) directory as the current directory from which you started the C code generator. By default, it generates code for objects with default scope to the current directory and code for objects with non-default scope to the user-specified directories, as indicated by the SEDirectory object.

Overriding Default Directories or Files

Optionally, you can override the default and user-specified SEDirectory and SEFile specifications and generate all code into one area. To override the defaults, specify an alternate directory location and/or file name in the **Generate C Code** dialog box, as follows:

- Edit the contents of the **Directory** field.
- Select **Use Alternate File Name**, and enter the filename in the **File Name** field.

Overriding Default File Extensions

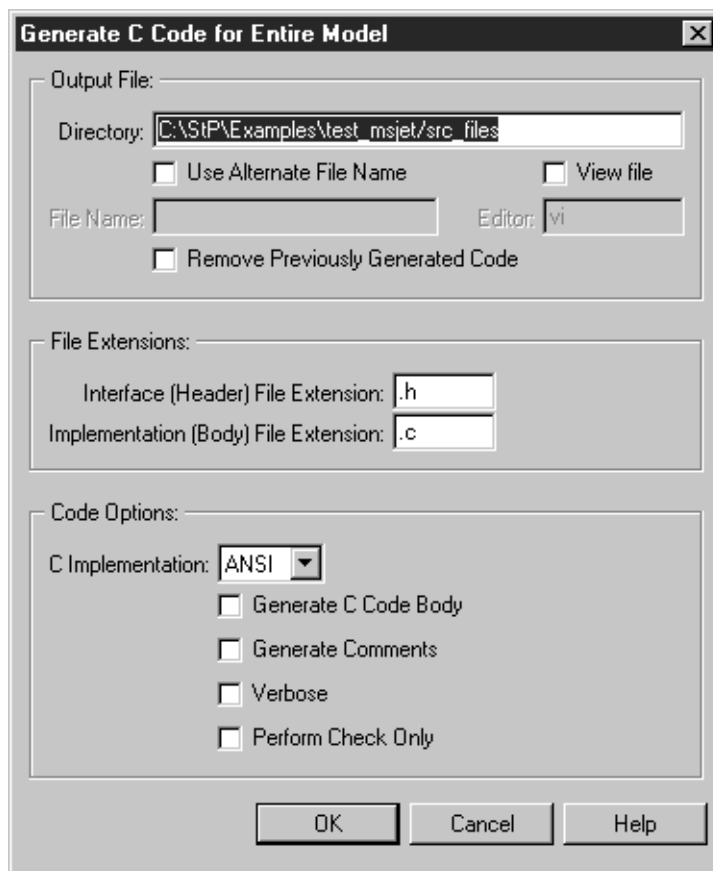
By default, structure chart objects are generated to *.c files and data structure objects are generated to *.h files. Optionally, you can specify different extensions in the following File Extension fields in the **Generate C Code** dialog box:

- **Interface (Header) File Extension**
- **Implementation (Body) File Extension**

Using the Generate C Code Dialog Box

Choosing any C code generation command described in Table 4 causes the **Generate C Code** dialog box (Figure 4) to appear. The **Generate C Code** dialog box enables you to select various options for the generated code.

Figure 4: Generate C Code Dialog Box



To use the **Generate C Code** dialog box:

1. Select a code generation command, as described in Table 4.
2. In the **Generate C Code** dialog box, select options, as described in Table 5 on page 10-25.
3. Click **OK**.

The generated code is written to the specified files, as described in “Output Files and Directories” on page 10-22.

Summary of Code Generation Options.

Table 5 is a summary of the options for code generation.

Table 5: Generate C Code Options

Group	Option	Description
Output File	Directory	Output directory. You can edit this field to override the default directory or the directory specified by the SEDirectory scope qualification. For details, see “Output Files and Directories” on page 10-22.
	Use Alternate File Name	If selected, overrides both default and user-specified SEFile scope qualification and places code output in the specified alternate file.
	File Name	
	View File	If selected, loads the generated files into the specified text editor.
	Editor	Specifies the text editor for viewing files. Valid values are vi, textedit, emacs, or any other text editor you want to use.
	Remove Previously Generated Code	If selected, overrides the incremental code generation mode; removes all previously generated code in the default or alternate output directory.
File Extensions	Interface (Header) File Extension	Overrides the default file extension (.h) for data structure objects with the specified extension. For details, see “Overriding Default File Extensions” on page 10-23.
	Implementation (Body) File Extension	Overrides the default file extension (.c) for structure chart objects with the specified extension. For details, see “Overriding Default File Extensions” on page 10-23.

Table 5: Generate C Code Options (Continued)

Group	Option	Description
Code Options	C Implementation	Specifies ANSI or K&R C output.
	Generate C Code Body	Generates information included in the C Code Body module annotation note (if any) into the code. If not selected, note information is not included in the code.
	Generate Comments	Includes information from the Module Comment or Data Structure Comment annotation notes (if any) in the generated code. If not selected, no comments are generated.
	Verbose	Displays messages about code generation in the StP Desktop Execution Window.
	Perform Check Only	Applies code generation checks to structure charts and data structure diagrams, to determine if they are complete enough to generate useful C code. Does not generate any code. For details, see “Checking the Model for Code Generation” on page 10-26.

Checking the Model for Code Generation

To ensure that structure charts and data structure diagrams are complete enough to generate useful C code, the code generator automatically executes the following validation checks on these models during code generation. Any checks that fail are reported as warnings, although the code is still generated.

Optionally, you can run the checks without generating code, by clicking **Perform Check Only** in the **Generate C Code** dialog box.

When performing code generation checks, StP/SE checks:

- That a module has a formal definition for it (see “Function Definitions with Formal Parameters” on page 10-9).
- That functions defined as having variable arguments have at least one non-variable parameter and at least one argument being passed to it with each call (ANSI only).
- To ensure that data structure diagrams exists for all non-basetypes encountered in structure charts or data structure diagrams (that is, module return types, parameter types, global types, and structure field types). If a data object has no assigned data type, int is assumed and a warning is generated.
- If there is more than one definition for a data element, checks whether the correct data element can be determined based on scope. If not, generates a warning that the definition for the data element is assumed (see “Creation of Data Definitions” on page 10-15).

Updating and Editing Code

StP/SE allows you to update or modify generated code in the following ways:

- Generate code incrementally
- Replace all existing code with new code
- Edit code outside of tagged fields directly

StP/SE allows you to edit certain parts of the generated code, and preserves the user-modified code during subsequent incremental code generation.

Incremental Updates versus All New Code

By default, StP/SE generates code incrementally. That is, if code is generated to a file that already exists, only new code and code modifications are added to the file. Previously-generated, user-written, and user-modified code is preserved.

To generate all new code, without saving any of the previous code, use the **Remove Previously Generated Code** option on the **Generate C Code** dialog box (Figure 4 on page 10-24).

Note: This option removes both user-entered and previously generated code before generating new code from the diagrams.

Editing Generated Code

During code generation, all information generated into implementation or interface files is placed within tagged fields. Each tagged field is identified by a unique ID code (for example, 314:27), which is placed in the function or data definition header, as shown in this example of generated code:

```
/* ***** StP/SE Function ***** */
* (314:27)
*
* Function: inc_test2
*
* Calls:
*
* Description:
*   Module created by the SE Code Generator.
*
* Comment:
*
***** /

/* StP/SE module 314:27 */
void
inc_test2()
{
    /* StP/SE code */

    /* StP/SE code end */
}
/* StP/SE module end */
```

You can directly modify areas of code outside of the tagged fields. If code is generated to the same file more than once, any code outside of the tagged fields is retained and any additions or changes to the SE model cause new or changed code to be added to the tagged fields.

Do not modify any of the code within the tagged fields or any StP/SE-generated comments; that is, comments that are identified with:

```
StP/SE <File|Function|Data Element>.
```

Each of the following sections contains a list of tagged fields for implementation or interface files. Each table is followed by a simple example in which the tagged fields are highlighted.

Tagged Fields for Implementation Files

Table 6 lists the tagged fields for implementation (source) files.

Table 6: Tagged Fields for Implementation Files

Section	Start Tag	End Tag
Include Section	/* StP/SE include files needed by type references within */	/* StP/SE include files end */
Global Definitions	/* StP/SE globals used in this file */	/* StP/SE globals end */
Function Signature	/* StP/SE module nnn */	{
Code	/* StP/SE code */	/* StP/SE code end */
Code Close	}	/* StP/SE module end */

The following is an implementation (source) file example.

Note: Do not edit any of the tagged fields, shown in bold in this example.


```

/***** StP/SE File *****/
* File: main.c
*
*   created on Thu Aug 03 13:27:51 1995 for
*   mcmann@woodstock from system mcmann
*
*   modified on Thu Aug 03 15:03:18 1995 for
*   mcmann@woodstock from system mcmann
*
* Functions:
*   main
*   inc_test2
*
*****/

/* StP/SE include files needed by type references within */
#include <stdio.h>
#include "protos.h"
/* StP/SE include files end */

/* StP/SE globals used in this file */
extern int global1;
extern struct FILE global2;
/* StP/SE globals end */

/***** StP/SE Function *****/
* (314:27)
*
*   Function: inc_test2
*
*   Calls:
*
*   Description:
*   Module created by the SE Code Generator.
*
*   Comment:
*
*****/

/* StP/SE module 314:27 */
void
inc_test2()
```

```
{
    /* StP/SE code */

    /* StP/SE code end */
}
/* StP/SE module end */
/***** StP/SE Function *****/
* (321:32)
*
*   Function: main
*
*   Calls:
*       inc_test2
*       new_module
*
*   Description:
*       Module created by the SE Code Generator.
*
*   Comment:
*
*****/
/* StP/SE module 321:32 */
int
main(int argc, char ** argv)
{
    /* StP/SE code */

    /* inc_test2(); */
    /* new_module( a, b); */

    /* global1; (read) */
    /* global2; (read) */

    /* StP/SE code end */
}
/* StP/SE module end */
```

Tagged Fields for Interface Files

Table 7 lists the tagged fields for interface (header) files.

Table 7: Editing Guidelines for Interface Files

Section	Start Tag	End Tag
type reference	/* StP/SE include files needed by type references within */	/* StP/SE include files end */
data definition	/* StP/SE data definition n*/	/* StP/SE data definition end */

The following is an interface (header) file example.

Note: Do not edit any of the tagged fields, shown in bold in this example.

```

/***** StP/SE File *****/
* File: module_tp.h
*
*   created on Thu Aug 03 16:49:28 1995 for
*   mcmann@woodstock from system mcmann
*
*   modified on Thu Aug 03 16:57:36 1995 for
*   mcmann@woodstock from system mcmann
*
* Components:
*   union_field_tp
*   module_tp
*
*****/

#ifndef _MODULE_TP_H
#define _MODULE_TP_H

/* StP/SE include files needed by type references within */
#include "name_tp.h"
#include "new_tp.h"

```

Generating C Code

```
#include "enum_field_tp.h"
#include "address_tp.h"
#include "new_tp.h"
/* StP/SE include files end */

/***** StP/SE Data Element *****/
* (2:19)
*
* Data Element: union_field_tp
*
* Description:
*   Created by the SE Code Generator.
*
* Comment:
*
*****/

/* StP/SE data definition 2:19 */
union union_field_tp
{
    struct name_tp name;
    int value;
};
/* StP/SE data definition end */

/***** StP/SE Data Element *****/
* (320:83)
*
* Data Element: module_tp
*
* Description:
*   Created by the SE Code Generator.
*
* Comment:
*
*****/

/* StP/SE data definition 320:83 */
struct module_tp
{
    new_tp2 dd;
    enum enum_field_tp enum_field;
    struct
    {
        address_tp address;
    }
}
```

```
        struct name_tp name;
    } person;
    new_tp submodule_list;
    union union_field_tp union_field;
};
/* StP/SE data definition end */

#endif          /* _MODULE_TP_H */
```


11 Reverse Engineering

This chapter describes how to use StP/SE Reverse Engineering (RE) to automatically generate a comprehensive graphical model of your software from C source code. Topics covered in this chapter are as follows:

- “What is Generated?” on page 11-1
- “Overview of Reverse Engineering” on page 11-2
- “Parsing the Source Files” on page 11-6
- “Extracting Comments” on page 11-27
- “Generating a Model from Parsed Files” on page 11-34
- “Maintaining and Updating Your Model” on page 11-44
- “Generated Structure Charts” on page 11-56
- “Generated Data Structure Charts” on page 11-66
- “Generated Flow Charts” on page 11-71
- “Generated Annotations” on page 11-81
- “Reverse Engineering Directory Structure” on page 11-83

What is Generated?

The Reverse Engineering process generates both a semantic and a graphical model of your software, as well as analytical reports.

The semantic model contains information extracted from C code. The extracted information is then used to generate the graphical model, whose components and sources of information are described in Table 1.

Table 1: Components of a Reverse Engineered Model

Model Component	Source Code Used
Structure charts	Information about functions, function calls, and parameters
Data structure diagrams	Information about struct, union, and enum definitions
Flow charts	Internal description of each parsed function body
Annotations	Extracted C comments

The diagrams and annotations created by Reverse Engineering are stored in the repository for the current project and system. All other Reverse Engineering files are stored in the `\revc_files` directory for the current project and system, as described in “Reverse Engineering Directory Structure” on page 11-83.

Overview of Reverse Engineering

StP Reverse Engineering is a collection of utilities, each of which performs a different task related to generating semantic and graphical models of your system from C source code:

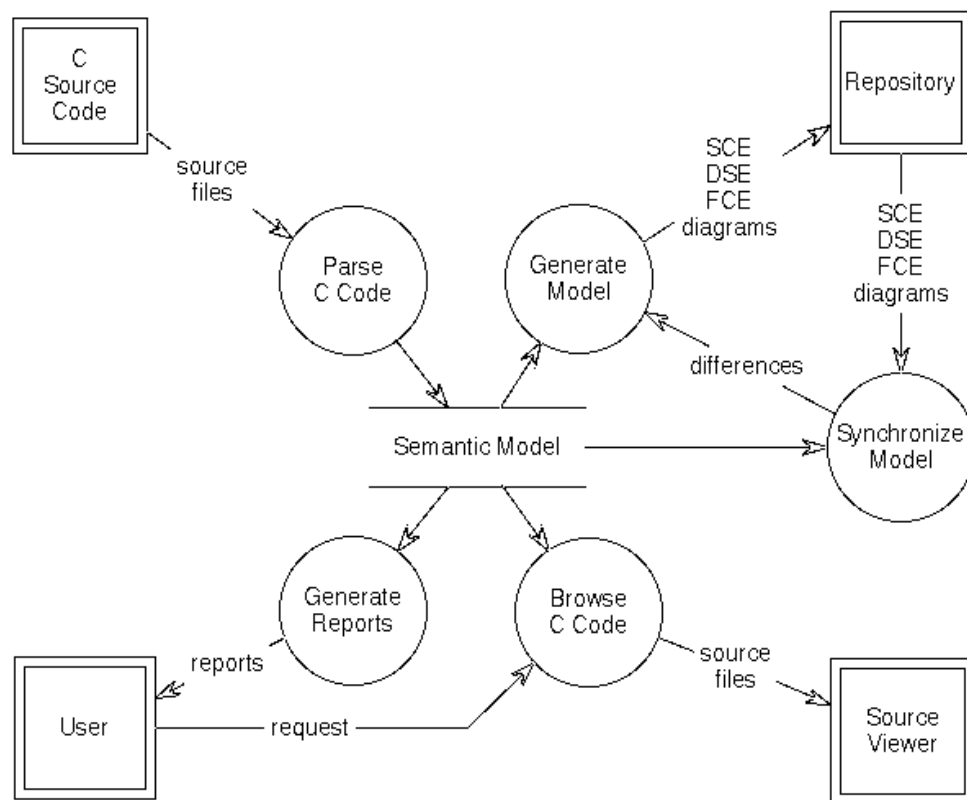
- Source code parser—Extracts code from multiple C source files to produce a comprehensive semantic model of your software.
- Comment extractor—Associates comments in the source code, extracted by the parser, with data structures, functions, and global variables in the semantic model.
- Model generator—Uses the semantic model to automatically create the diagrams and annotations that compose a graphical model of your software.
- Synchronizer—Compares the generated model to the source code, enables you to examine any differences introduced by modifying the code or model, and updates the model accordingly.

Additionally, you can generate C Code Metrics reports from the reverse engineered semantic model, which help you analyze the results of the reverse engineering process. For more information, see “C Metrics Reports” on page 14-22.

You can also navigate to the source code from your reverse engineered model (see “Navigating from Model to Source Code” on page 12-38 for more information).

Figure 1 depicts the StP/SE reverse engineering process and its results.

Figure 1: Overview of Reverse Engineering



Using Reverse Engineering

Creating a reverse engineered model from source code involves three phases, each corresponding to a command on the **Reverse Engineering** menu. The following procedure describes the order in which you use these commands. Each command is discussed in detail in a separate section of this chapter.

To use Reverse Engineering:

1. Create and open a new system into which you will reverse engineer the code.
2. From the Desktop **Code** menu, choose **Reverse Engineering**; then choose the following commands from the **Reverse Engineering** submenu, in this order:
 - **Parse Source Code** (see “Parsing the Source Files” on page 11-6).
 - **Extract Source Code Comments** (see “Extracting Comments” on page 11-27).
 - **Generate Model from Parsed Source Files** (see “Generating a Model from Parsed Files” on page 11-34).

After generating the semantic and graphical models of your system, you can use other Reverse Engineering commands to update the model, generate reports, or navigate to source code.

Reverse Engineering Commands

Table 2 describes the commands on the **Reverse Engineering** submenu.

Table 2: Reverse Engineering Commands

Command	Description	For Details, See
Generate Model from Parsed Source Code	Automatically creates, and saves to the repository, a graphical model of the reverse engineered software.	“Generating a Model from Parsed Files” on page 11-34

Table 2: Reverse Engineering Commands (Continued)

Command	Description	For Details, See
Parse Source Code	Extracts all information needed to generate design documentation from the source code and stores it in a semantic model that is separate from the object repository.	“Parsing the Source Files” on page 11-6
Extract Source Code Comments	Associates comments extracted by the parser with functions, global variables and data structure definitions.	“Extracting Comments” on page 11-27
Remove File from RE Semantic Model	Removes the contents of the specified files from the semantic model.	“Removing File Contents from the Semantic Model” on page 11-25
Remove Directory from RE Semantic Model	Removes the contents of the specified directories from the semantic model.	
Synchronize Model	Compares the semantic model with the objects in the repository that comprise the design model, identifies differences, and optionally updates the design model according to your specifications.	“Maintaining and Updating Your Model” on page 11-44
Browse C Code	Searches for specified C constructs in a semantic model.	“Listing the Files in the Semantic Model” on page 11-24, and “Browsing C Code” on page 12-13
Generate C Code Metrics Reports	Allows you to generate or view reports that help you interpret and evaluate the reverse engineering results.	“C Metrics Reports” on page 14-22

Table 2: Reverse Engineering Commands (Continued)

Command	Description	For Details, See
.Check RE Semantic Model Locks	Reports the lock status of a semantic model and removes any redundant locks.	“Semantic Model Locks” on page 11-26
Check RE Semantic Model Consistency	Checks whether the semantic model has been corrupted.	“Checking the Semantic Model” on page 11-27

Parsing the Source Files

The Source Code Parser extracts code from multiple C source files and produces a semantic model of your software. The semantic model is tightly bound to the source code and emphasizes, at a low level, the information flow through the software. All other Reverse Engineering commands require access to the semantic model.

The Source Code Parser parses C source code files. You can parse all or selected C source code files for your software. You can also parse the files in incremental mode to update an existing semantic model with any source code changes made since the semantic model was last generated.

Using the Parse Source Code Dialog Box

The **Parse Source Code** dialog box (Figure 2) enables you to specify details about the files for parsing.

Figure 2: Parse Source Code Dialog Box

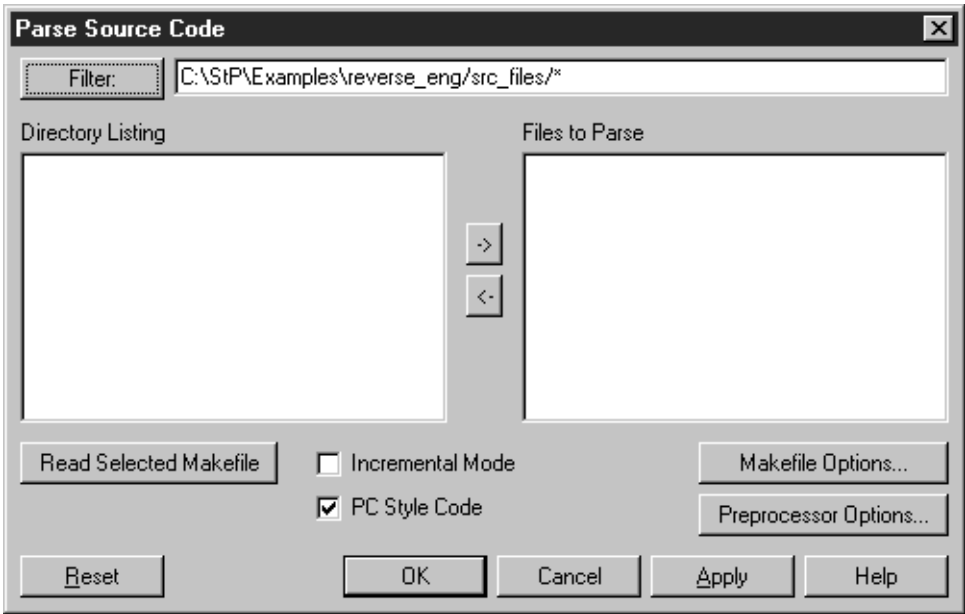


Table 3 summarizes the parser specifications and options.

Table 3: Parse Source Code Dialog Box Options

Specification	Description	For Details, See
Filter button	Applies the filter pattern in the filter text field to the Directory Listing contents.	“Deciding Which Files to Parse” on page 11-10
Filter text field	Displays current directory or user-specified pattern for determining the Directory Listing contents.	
Directory Listing	Displays directories and files from which you can choose files to be parsed.	

Table 3: Parse Source Code Dialog Box Options (Continued)

Specification	Description	For Details, See
Files to Parse	Displays source files to be parsed.	“Deciding Which Files to Parse” on page 11-10
Read Selected Makefile button	Reads the selected makefile and displays the appropriate information in the Parse Source Code dialog box.	“Using the Makefile Reader” on page 11-12
Makefile Options button	Displays the Makefile Reader Options dialog box, where you can change the default makefile options.	
Incremental Mode	Parses only those specified files that have been modified since the last time they were parsed.	“Using Incremental Mode” on page 11-11
PC Style Code	If selected, causes the parser to look for PC-style path names and other common PC features in code.	“Parsing PC Source Code” on page 11-11
Preprocessor Options button	Displays the Parser Preprocessor Options dialog box, where you can change the default preprocessor options.	“Using the Parser Preprocessor Options Dialog Box” on page 11-15
Reset button	Redisplays the default values in the dialog box.	

Parsing the Files

Parser processing time depends on the amount of code being parsed. For information about which files are needed, see “Deciding Which Files to Parse” on page 11-10.

To parse the source code files and create the semantic model:

1. Make sure your current system is the one into which you want to generate the model.
2. From the Desktop **Code** menu, choose **Reverse Engineering > Parse Source Code**.

The **Parse Source Code** dialog box appears.

3. In the **Filter** text field, edit the directory path to point to the location of the source code or the path for the makefile you want to use.
4. Click the **Filter** button.

The directory's contents appear in the **Directory Listing** pane.

To display the contents of a subdirectory, double click a directory name in the list.

To display the contents of the parent directory, double click *Parent Directory*, which appears at the top of the list.

5. Take one of the following actions:
 - Select a makefile in the **Directory Listing** box and go to Step 6.
 - Select the source file(s), and use the right arrow button to move them into the **Files to Parse** list. Go to Step 9.
6. To change the default makefile options, click the **Makefile Options** button (see “Using the Makefile Reader” on page 11-12).
7. Click Read Selected Makefile button.

The Makefile Reader reads the makefile, finds the first target and its prerequisites, and displays them in the **Files to Parse** list. It also finds the user include and system include directories to be searched during preprocessing.
8. To change the default preprocessor options, click **Preprocessor Options** (see “Using the Parser Preprocessor Options Dialog Box” on page 11-15). When done, click **OK** on the **Preprocessor Options** dialog to return to the **Parse Source Code** dialog.
9. To run the parser in incremental mode, select **Incremental Mode** (see “Using Incremental Mode” on page 11-11).
10. If you are parsing PC source code files, ensure that the **PC Style Code** option is selected (see “Parsing PC Source Code” on page 11-11).

11. Click **OK** or **Apply** to run the parser.

Note: Once StP has begun to parse the files, do not attempt to abort the parser by killing the process directly, as this can corrupt the semantic model (see “Checking the Semantic Model” on page 11-27). If necessary, you can terminate the parser by clicking **OK** in the parser status window.

If you encounter problems when parsing files, see “Common Parsing Problems” on page 11-22. After the source files parse without error, you can execute any of the other Reverse Engineering commands.

Deciding Which Files to Parse

Your choice of which source files to parse affects the type and content of the diagrams produced by StP/SE. If you plan to generate only data structure charts and no other diagrams, specify only header files when parsing the source code. When generating the model, as described in “Generating a Model from Parsed Files” on page 11-34, select Data Structures as the only type of diagram to create.

You can parse:

- An individual source file
- Several selected source files
- All source files used to build the executables for an entire system

Processing several or all executables at once allows you to create a model for a particular part or all of the software system. Generally, you should parse as a group all related executables that share common data definitions in include (header) files. This causes identical information for different executables to be diagrammed only once, rather than reproduced in identical diagrams and definitions for each executable. Parsing executables as a group also enables the C Code Browser to locate items across the boundaries of individual executables.

The source files to be parsed can reside anywhere on the file system. They are not necessarily all in the same directory. Although you can parse files that are incomplete or have syntax errors, you may receive warning messages. Occasionally you may inadvertently omit necessary source files from the **Files to Parse** list. If StP/SE finds a call to a function for which it has no definition, it draws the callee as a library function.

StP does not extract the source code from version control systems. If your code resides within a version control system, you must manually extract readable copies of the source into a directory where StP can process them. If you choose to run reverse engineering incrementally with such code, make sure only changed files are copied over their previous instances.

Using Incremental Mode

Select **Incremental Mode** to parse only those specified files that have been modified since the last time they were parsed. The parser uses the file system modification date stamp to determine when the files were last modified.

When **Incremental Mode** is selected, a file is parsed only if it:

- Has been modified since last being parsed
- Failed to pass prior parsing correctly
- #includes a file that has been re-parsed incrementally
- Has just been added to the source file list

Generally, you should use the **Incremental Mode** option for parsing files except when:

- Many files have changed
- A frequently included file has changed

In these cases, parsing in non-incremental mode is usually faster. However, the results of parsing incrementally and non-incrementally are the same.

Parsing PC Source Code

If you are parsing PC source code, select the **PC Style Code** option in the **Parse Source Code** dialog box.

For PC C Code to parse correctly, reserved words that specify memory size (for example, far, huge, and near) should be handled as defines. Enter these reserved words as declarations in the **Defines** field in the **Parser Preprocessor Options** dialog box, as described in “Specifying Compiler Directives as Defines” on page 11-18.

Using the Makefile Reader

StP Reverse Engineering provides a feature called the Makefile Reader, which automatically finds the files needed to model your software program using the information contained in a Nmake or GNU makefile.

The Makefile Reader uses the same rules as the Microsoft **Nmake** or Free Software Foundation (FSF) GNU **make** command to read the makefile and determine all the prerequisites for the primary target or, alternatively, for any subordinate targets that you specify in the **Makefile Options** dialog box. It then displays the:

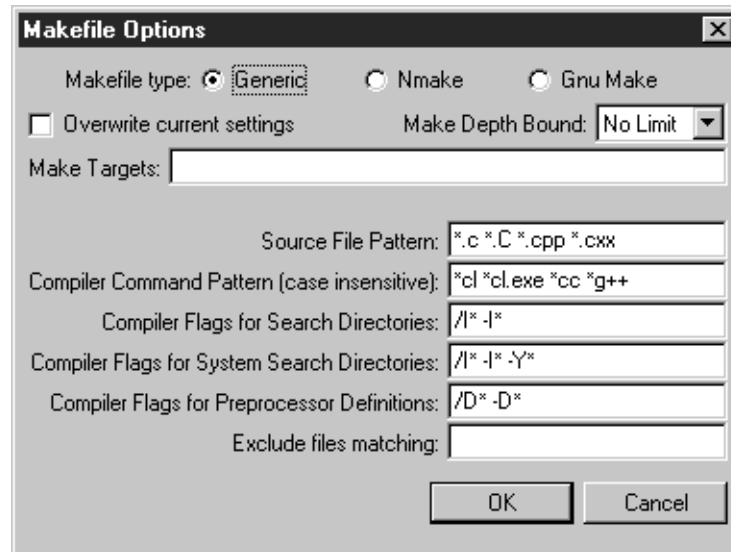
- Prerequisite files in the **Files to Parse** list on the **Parse Source Code** dialog box
- Directories containing user-defined and system-defined header files in the **User Include Search Directories** and **System Include Search Directories** lists on the **Parser Preprocessor Options** dialog box (see “Using the Parser Preprocessor Options Dialog Box” on page 11-15)

You can preprocess and parse the files as displayed or edit the information before parsing.

Using the Makefile Options Dialog Box

If you want to change the default makefile options, you use the **Makefile Options** dialog box, shown in Figure 3.

Figure 3: Makefile Options Dialog Box



The **Makefile Options** dialog box options are described in Table 4.

Table 4: Makefile Options Dialog Box Specifications

Specification	Description
Makefile type	Specifies the type of makefile and the rules to use to read it—generic, Nmake (Microsoft), or GNU (Free Software Foundation)
Overwrite current settings	Clears the existing files from the Files to Parse window and all fields on the Parser Preprocessor Options dialog box. If you do not select this option, the Makefile Reader concatenates new information with existing information.
Make Depth Bound	Number of directory levels that the Makefile Reader searches to find prerequisite files. For more information, see “Reading Recursive Makes” on page 11-14.

Table 4: Makefile Options Dialog Box Specifications (Continued)

Specification	Description
Make Targets	Subordinate targets for the makefile. For more information, see “Selecting Subordinate Targets” on page 11-15.
Source File Pattern	Pattern that identifies a source file contained in the compiler command. If a source file has an odd suffix, add it to this list to ensure that the Makefile Reader extracts the correct source files from the Makefile.
Compiler Command Pattern (case insensitive)	Pattern that identifies the name of the executable that runs the compiler.
Compiler Flags for Search Directories	Flags that identify search directories for header files that are included in the source files with #include directives.
Compiler Flags for System Search Directories	Flags that identify search directories for system-defined header files that are included in the source files with #include directives.
Compiler Flags for Preprocessor Definitions	Flags that identify preprocessor definitions.
Exclude files matching	Pattern that identifies files that you want the Makefile reader to ignore. For example, to exclude yacc generated files from the Makefile run, type <code>*Y* .c</code> in this field.

Reading Recursive Makes

The Makefile Reader can trace prerequisite files through a hierarchy of directories when **make** is invoked recursively. By default, there is no limit on the number of directory levels the Makefile Reader searches; however, you can specify a limit using the **Depth Bound** option. Select **1**, **2**, **3**, **4**, or **5** directory levels, or use the default setting of **No Limit**.

Selecting Subordinate Targets

By default, the MakeFile Reader finds the files needed to model the primary target or executable identified in the makefile. You can override the default by specifying one or more subordinate targets in the **Make Targets** text entry field. Whenever this field has a value, the MakeFile Reader generates prerequisites for the specified targets only.

A subordinate target can be any object in the makefile that has a dependency line. If you specify more than one target, be sure to insert a blank space between names.

In addition to specifying targets, you can also define macros in the **Make Targets** field. Type macros in the form <NAME> = <value>. For example:

```
OBJECTS="db_ant.o ant_files.o ant_main.o ant_ids.o"
```

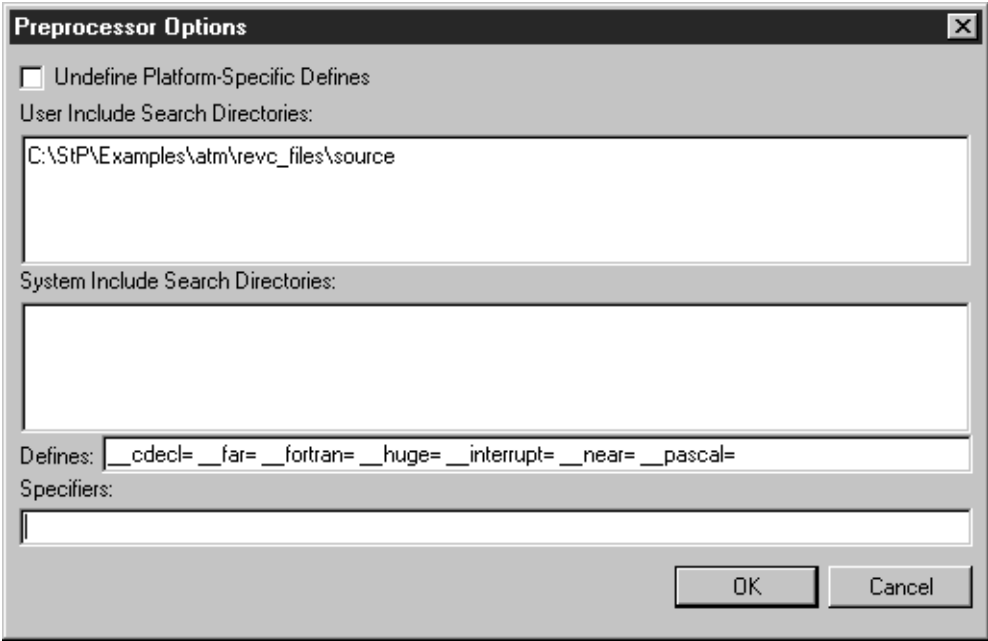
or

```
BIN=$(HOME)\bin
```

Using the Parser Preprocessor Options Dialog Box

You use the **Parser Preprocessor Options** dialog box, shown in Figure 4, to change the default preprocessor options.

Figure 4: Parser Preprocessor Options Dialog Box



The **Parser Preprocessor Options** dialog box is described in Table 5.

Table 5: Parser Preprocessor Dialog Box Summary

Element	Description	For Details, See
Undefine Platform-Specific Defines option	If selected, Reverse Engineering ignores the standard compiler define directives set by default for the platform you are using.	“Undefining Platform-Specific Defines” on page 11-17

Table 5: Parser Preprocessor Dialog Box Summary (Continued)

Element	Description	For Details, See
User Include Search Directories list	Directories containing user-defined header files, which are included with the #include "header.h" notation.	"Specifying Include File Search Directories" on page 11-18
System Include Search Directories list	Directories containing system-defined header files, which are included with the #include <header.h> notation.	
Defines field	Preprocessor flags that direct the compiler to create a semantic model for the exact version of the software built by the compiler.	"Specifying Compiler Directives as Defines" on page 11-18
Specifiers field	User-specified list, entered as a string, containing name translations for parsing dialects of C code that are not 100% ANSI or K&R C-compliant.	"Entering C Dialects as Specifiers" on page 11-19

Undefining Platform-Specific Defines

By default, preprocessor predefined constants are set to different values for each platform so that the standard user include files are parsed correctly. You may need to undefine these directives and define an entirely new set in the **Defines** field, if you are parsing code that comes from a compiler or architecture that differs from the standard for the platform you are using (see "Specifying Compiler Directives as Defines" on page 11-18).

To find out which directives are defined by default for your platform, enter `'rmpp -list'` at a command prompt; the directives are echoed back.

To disable these directives, select the **Undefine Platform-Specific Defines** option on the **Parse Source Code** dialog box.

Specifying Include File Search Directories

You can enter as many of the following include file search directories for user and system-defined header files as you want:

- Directories containing any user-defined header files, which are included with the `#include "header.h"` notation.
- Directories containing any system-defined header files, which are included with the `#include <header.h>` notation.

When you click the **Read Makefile** button on the **Parse Source Code** dialog box, the Makefile Reader fills the **User Include Search Directories** and **System Include Search Directories** lists for you. You can edit, delete, or add to directories in these lists.

For StP on Windows NT, you do not need to specify any files contained in the Include environment variable, because StP automatically adds their paths to the end of the **System Include Search Directories** list.

Generally, you should specify directories as absolute paths, although you can specify relative paths or a combination of both types. Relative paths are evaluated from the *revc_files* directory for the current project and system.

StP searches the specified directories in top to bottom order.

Specifying Compiler Directives as Defines

If the source code was compiled with preprocessor flags specifying the compilation of a particular version of the software, you may want to specify compiler directives for parsing the code. Specifying the directives tells the Reverse Engineering code parser to create a semantic model for the exact version of the software built by the compiler. These flags are usually passed to the compiler from the Makefile, which holds all the preprocessor flags needed to build each version of the software.

To specify the preprocessor flags, enter a string containing the necessary compiler directives, separated by spaces, into the **Defines** field. This creates a file called *Defines* in the workspace directory. If the directive includes spaces or punctuation, enclose it within double quotes. For example:

```
VERSION="6.0 Beta" MACHINE=DEBUG
```


Normally, you do not need to use the **Defines** field. Enter compiler directives only if necessary—for example, if you need to add to the standard defines or are parsing code from a compiler or architecture that differs from the default defines for the platform you are using.

Entering C Dialects as Specifiers

StP/SE is designed to parse C source code written for any hardware platform. However, there are sometimes dialectic differences that must be accommodated by the parser. For example, C source code for PCs may include extra reserved words such as “cdecl packed.”

To enable the parser to handle dialectic differences, enter additional declaration specifiers, separated by spaces, into the **Specifiers** field in the dialog box. This creates a file called *specifiers* in the system directory. The parser reads this file and extends its knowledge of the language accordingly.

For example, you can enter the following declaration specifiers in the **Specifiers** field to allow PC C to parse correctly:

```
cdecl packed
```

Other dialects of C may use completely new reserved words. You can usually get these to parse by using both the **Defines** and the **Specifiers** fields.

Parsing C Code Containing Embedded Code

If the dialect of C code used in your organization supports the embedding of SQL, assembler, or other languages, you can configure the Reverse Engineering parser to search for and ignore the embedded code. This allows you to reverse engineer the original C source code without special preprocessing that would create extra C code you may not want in your source code.

To instruct StP Reverse Engineering to ignore embedded code, you specify in a file the starting and ending tokens that surround the code you want to ignore. The tokens you choose should be tokens used by the embedded language to signify sections of embedded code, such as

declaration sections and SQL commands. The Reverse Engineering parser then looks for the user-specified tokens in the source code and ignores all code found between the matching starting and ending tokens.

For example, when embedded in C code, some dialects of SQL use the token EXEC to indicate the beginning of an embedded SQL command, and a semicolon (;) to indicate the end of the embedded command. By specifying EXEC as the start token and a semicolon as the end token, you instruct Reverse Engineering to search for and ignore each embedded SQL command.

To specify tokens you add them to a file to which the *re_token_file* ToolInfo variable points. You can set this variable in the master ToolInfo file so the parser uses the specified tokens whenever anyone parses C code in any StP/SE system. See *StP Administration* for more information on ToolInfo variables and files.

Token Search Patterns

The token strings you choose and the order in which you enter them in the file affect the results of the parser search. Precedence of start tokens is by the order in which the token pairs occur in the file.

For example, suppose your file contains these token specifications:

```
"EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"  
"EXEC" ";"
```

When parsing the following SQL pseudocode code embedded within C pseudocode, the Reverse Engineering parser first finds and ignores all of the SQL declarations in the declaration section; then finds and ignores the embedded SQL command.

```
some C code here...  
EXEC SQL BEGIN DECLARE SECTION;  
    some SQL declarations here...  
EXEC SQL END DECLARE SECTION;  
more C code here...  
EXEC embedded SQL command;  
additional C code here...
```

However, suppose you entered the tokens in the file in the reverse order:

```
"EXEC" ";" "  
"EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"
```

In this case, the Reverse Engineering parser finds the first EXEC token at the beginning of the BEGIN DECLARE SECTION in the preceding code and ignores code only until it encounters the semicolon token at the end of the BEGIN DECLARE SECTION on the same line. It then stops ignoring code and erroneously attempts to parse the SQL declarations between the first end token and the next EXEC token in the EXEC SQL END DECLARE SECTION.

Likewise, the Reverse Engineering parser would not be able to identify a nested token pattern, such as this:

```
some C code here...  
EXEC embedded SQL command  
    EXEC embedded SQL command;  
    more SQL code  
    ;  
more C code here...
```

The parser would ignore all code between the first EXEC token and the first found semicolon (;) end token, thus ignoring the nested EXEC as well. Having ignored the second EXEC token, the parser would not look for its paired semicolon end token either. Thus, the parser would erroneously attempt to parse the SQL code between the two semicolons.

It is the user's responsibility to choose appropriate tokens and enter them in the token file in an appropriate order that will allow correct parsing of the C code while ignoring the embedded language code.

Specifying the Embedded Tokens

You can specify as many pairs of tokens as are necessary to identify all of the embedded code to be ignored.

Token matching is case-sensitive. User-specified tokens must match exactly the tokens used by the embedded language, or the parser will not be able to identify the code to ignore.

To specify the start and end tokens:

1. Create a file (for instance, *embedded_tokens*) to contain the token specifications.
2. Add a new line to the file for each pair of start and end tokens you want the Reverse Engineering parser to use.
Enclose each token within double quotes and separate the start token from the end token with a single space. For example:

```
"EXEC SQL BEGIN DECLARE SECTION;" "EXEC SQL END DECLARE SECTION;"  
"EXEC" " " ;"
```
3. In the ToolInfo file, set the *re_token_file* ToolInfo variable to point to the file containing the token specifications (see *StP Administration*).
For example, if your specifications are in the *embedded_tokens* file, set the value of the *re_token_file* ToolInfo variable as follows:

```
re_token_file=\usr\srp\template\embedded_tokens
```

When you run the Reverse Engineering parser, it looks for the file and the tokens you have specified.

Common Parsing Problems

You will have problems when parsing source code if:

- There are no locks on the semantic model
- Some files fail preprocessing
- The parser cannot locate required source or header files
- Case and spelling of source filenames is incorrect
- Directory of source files is not specified as a path
- Paths to #include<> files are specified incorrectly

There are also some coding scenarios that can cause problems during parsing, discussed in the following sections.

Locks on the Semantic Model

If there are read or write locks on the semantic model, the parser cannot update the database. For details about obtaining information on locks, see “Semantic Model Locks” on page 11-26.

Preprocessing Problems

By default, files are preprocessed before they are parsed. Sometimes preprocessing produces the problem that prevents the files from parsing correctly. To see the preprocessed source code, look in the *revc_files\code_dir* directory in the current system. A copy of all files that fail parsing is in this directory.

Only the first syntax error found in each source file is reported, so bugs must be corrected one at a time. Make sure all source files compile correctly before attempting to process them through Reverse Engineering.

Missing Source and Header Files

If the parser cannot find the required source files, error messages appear in the StP message log. If it cannot find the required header files, warning messages appear.

To correct these problems:

1. Specify the locations of the missing files in the **Parser Preprocessor Options** dialog box.
See “Using the Parser Preprocessor Options Dialog Box” on page 11-15 for information about this dialog box.
2. Reparse the files.
If only source files are missing, you can reparse incrementally. If both source and header files were missing, you must reparse in non-incremental mode.

Floating Typedef

Suppose a header file called *any_name.h* contains the following:

```
typedef struct any_name anything;
```

However, the data structure *struct any_name* has not been previously defined. It is actually defined in a C source file:

```
#include "any_name.h"
struct any_name { char a, b, c; };

anything this, that, *the_other;
```

This scenario is handled as expected by StP/SE. However, problems arise if more than one C source file includes *any_name.h* and each file defines *struct any_name* differently. StP/SE is confused by this programming style. The files will parse; however, a single data structure chart called *any_name* will hold all of the fields that comprise the different versions of the structure definition.

Typedef and Variable Name Clashes

C permits the use of the same name for a typedef and a variable.

```
typedef unsigned int byte;
byte byte;
```

This in itself is not a problem, but it can be confusing when these variables appear in expressions that also contain casts to the identically named type.

```
byte x = (byte)1 + (byte);
```

Some compilers seem to be able to handle this, but in StP/SE it generates a syntax error. Avoid using a typedef name as a variable name.

Listing the Files in the Semantic Model

To display a list of files in the semantic model, using the C Code Browser:

1. From the StP Desktop **Code** menu, choose **Reverse Engineering > Browse C Code**.
2. In the **Browse C Code** dialog box, display the options list in the **C Identifier Category** field, and select **File** as the construct category you want to search for.
3. Type an asterisk (*) in the adjacent text entry field for the object name.
4. Select **Use Focus** and click **Set/Change Focus**.

5. In the **Select Files to Search** dialog box, select any combination of the file type options (**Srcs**, **User Hdrs**, **Sys Hdrs**) and click the **Populate Candidate Files With** button.

A list of prospective files matching your selected file categories appears in the **Candidate Files** list.

6. To determine if a file is empty or not, move the file to the **Selected Files** list and click **OK**; then click **OK** on the **Browse C Code** dialog box.
7. Click **Cancel** when done viewing files or file contents.

Removing File Contents from the Semantic Model

You can selectively remove the contents of source files and/or directories from the semantic model to:

- Remove a parsed file that you did not intend to parse
- Trim the size of a semantic model to concentrate on a small portion of a large model

You can remove the contents of files or directories from the semantic model with one of these Reverse Engineering commands, provided the graphical model also exists:

- **Remove File from RE Semantic Model**
- **Remove Directory from RE Semantic Model**

Removing contents from the semantic model does not affect the contents of your existing repository, until you regenerate or update the repository from your semantic model. The empty file remains, in case you want to reparse it at a later date.

To remove the contents of files and directories from the semantic model:

1. From the **Model Elements** category on the Desktop, select **File Objects** or **Directory Objects**.

A list of files and directories read in from the repository appears in the objects pane.

2. Select one or more file or directory objects in the list, whose contents you want to remove from the semantic model.

3. From the Desktop **Code** menu, choose **Reverse Engineering**; from the **Reverse Engineering** submenu, choose either:

- **Remove File from RE Semantic Model**
- **Remove Directory from RE Semantic Model**

The contents of the specified files and/or directories are removed from the semantic model.

Semantic Model Locks

Reverse Engineering uses locking to prevent users from simultaneously performing operations on the same data in the database. For example, you cannot have two parsers executing at the same time, updating the same semantic model.

Due to the massive number of records stored in the semantic model, and the large numbers of contiguous records that are removed and replaced during parsing, it is not appropriate to lock individual records. Instead, complete tables are locked exclusively by all StP Reverse Engineering processes.

While the C Code Browser is in use, it holds read-only locks on the reverse engineered semantic model. Since the Reverse Engineering parser and the diagram generators require both read and write locks in order to update the database, neither can be run while the C Code Browser is accessing the same system.

Other than C Code Browser users, semantic model locks do not affect users of the StP repository, which is always available to multiple readers and writers.

To find out the lock status of a semantic model, from the StP Desktop **Code** menu on the, choose **Reverse Engineering > Check RE Semantic Model Locks**. StP removes any redundant locks and reports active locks by listing the name of the person and the machine holding the lock, as well as the time the lock was applied.

Checking the Semantic Model

It is possible for the semantic model's indices to become corrupted if a process that is writing it is killed before completing the task. Also, if the data on the disk is corrupted due to a disk crash, for example, the data fields in the semantic model may be invalid or out of range.

To check whether the semantic model has been corrupted, from the StP Desktop **Code** menu, choose **Reverse Engineering > Check RE Semantic Model Consistency**.

After checking the semantic model, the command displays a summary line stating whether the semantic model can be used or if it has been corrupted. If it is corrupted, you should destroy the semantic model by rerunning the parser on all the source files in non-incremental mode. This removes the current semantic model and rebuilds an entirely new one.

Extracting Comments

Comments represent an important source of information that can enhance the final design documentation. The C code parser alone does not sufficiently handle comments; it stores comments, but it does not associate them with objects in your model. In order to appropriately place C code comments in your semantic model, you must run the C comment extractor.

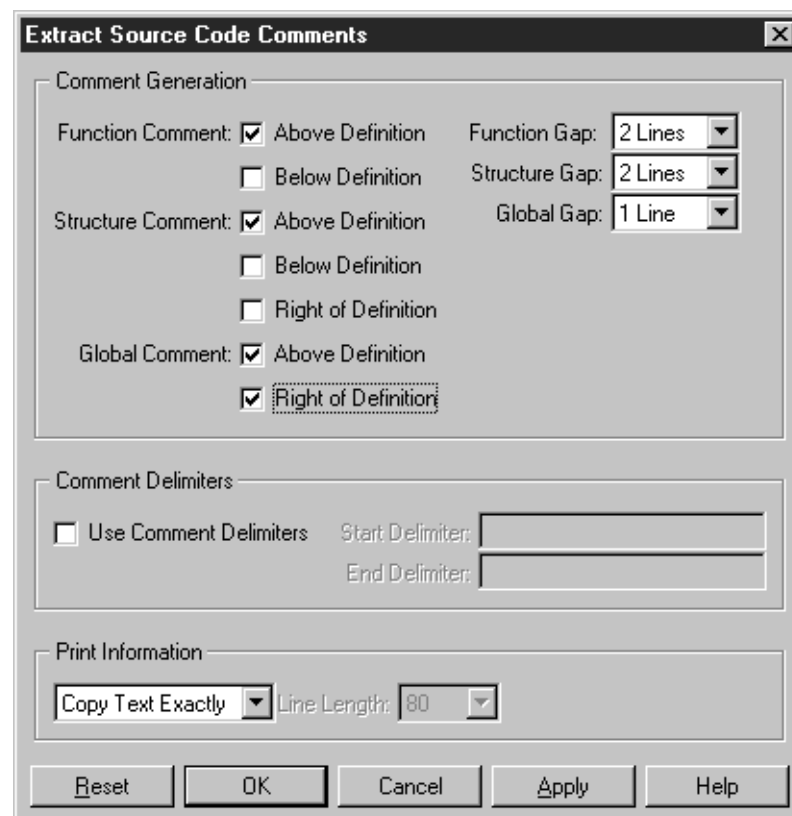
The comment extractor attempts to associate source code comments with data structures, functions, and global variables, by using search parameters that you provide. It then concatenates the found comments and writes them to the semantic model. The model generator uses the comments in the semantic model to create annotations for diagrams in the graphical model.

The comment extraction phase is optional. If you do not extract comments from the source code, they do not appear as annotations in your model.

Using the Extract Source Code Comments Dialog

The **Extract Source Code Comments** dialog box (Figure 5) enables you to specify search parameters for finding and extracting comments from source code.

Figure 5: Extract Source Code Comments Dialog Box



You specify comment extraction parameters by setting non-exclusive options for each of the three C code object categories in the dialog box:

- **Function Comment** settings—options for extracting comments related to functions
- **Structure Comment** settings—options for extracting comments related to compound (abstract) data types
- **Global Comment** settings—options for extracting comments related to global variables

These options are described in Table 6.

Table 6: Extract Source Code Comments Dialog Box Options

Option	Description	For Details, See
Comment Generation group:		
Above Definition	Extracts comments found above the definition of the source code object.	“Comment Position Settings” on page 11-31
Below Definition	Extracts comments found below the definition of the source code object.	
Right of Definition	Extracts comments found to the right of the definition of the source code object.	
Function Gap Structure Gap Global Gap	Specifies within how many non-comment lines of a source code object a comment must occur, in order to be associated with that object. Legal values are 1 to 4.	“Gap Settings” on page 11-31

Table 6: Extract Source Code Comments Dialog Box Options (Continued)

Option	Description	For Details, See
Comment Delimiters group:		
Use Comment Delimiters	Allows you to specify start and end delimiters (as strings) for extraction of formatted comments.	
Start Delimiter		
End Delimiter		
Print Information group:		“Formatting Options” on page 11-32
Copy Text Exactly Left Justify Text Fully Justify Text	Provides options for formatting comment text.	
Line Length	(Available only with Fully Justify Text option) Maximum line length allowed in reformatted comment text, which determines where line breaks occur.	

Extracting Comments from Source Code

To extract C comments from your source code:

1. From the Desktop **Code** menu, choose **Reverse Engineering > Extract Source Code Comments**.
2. In the **Extract Source Code** dialog box, make selections for the source code object categories and fill in the fields, as needed (for details, see Table 6 on page 11-29).
3. Click **OK** or **Apply**.
This runs the comment extractor and automatically saves the current dialog box option settings.

Comment Position Settings

The position settings indicate where you think the comments for that schematic are most likely to occur within the source code (above, below, or to the right of the schematic's definition in the code).

The position specifications are cumulative. If you click all three position options for a comment (above, below, and to the right of an object definition), the comment extractor extracts for that object all comments appearing within the specified gap, no matter where they are positioned relative to the definition.

For example, the dialog box in Figure 5 on page 11-28 causes the comment extractor to:

- Look for all comments positioned above a global definition, as well as all comments positioned to the right of the definition.
- Ignore any comments found if there is a gap of more than one blank line between the comment and the object definition.

Note: If a struct, union, or enum has its fields declared on separate lines, then any comment to the right of each line is associated with the element declared on that line, regardless of the comment position setting.

For example:

```
struct xxx (  
    int    first_field;    // An unused field at this time  
    char    name[400];    // A textual field with a  
                        // multi-line comment for the  
                        // field  
    float    difference    // this comment is for this field
```

Gap Settings

The **Gap Setting** option determines the number of non-comment lines (containing either code or white space) that constitute a legal gap between a comment and its associated object. For example, a comment separated from a function by two lines would be associated with that function only if the gap setting is two lines or more; a gap setting of one line would not associate the comment with this function. If you set too large a gap, chances are you will extract irrelevant comments. This will not be obvious until you see the comments in the StP annotation editor.

The gap setting determines spacing for comments both below and above the target object, as indicated by the position setting. Different gaps can be set for functions, structures, and globals.

The gap for functions and globals is measured from the line on which the function or global name appears. For example, the gap is measured from the second line in the following code fragment:

```
int
main( int argc, char **argv )
```

The gap for structures is measured from the line containing the curly bracket that occurs after the structure name. For example, the gap is measured from the third line in the following code fragment:

```
struct
    stock_item
{
    int code;
```

Formatting Options

You can select one of the following formatting options for comment text:

- **Copy Text Exactly**—Copies comment text exactly as it appears in the code, without reformatting.
- **Left Justify Text**—Reformats comments by left-justifying the text with a ragged right margin.
- **Fully Justify Text**—Reformats comments by justifying both left and right margins.

If you select this option, enter the maximum line length allowed in the **Line Length** field. This number determines where line breaks occur.

Evaluating and Improving Comment Extraction

Unless you have written the source code, you cannot know exactly where comments will be found. You may need to experiment with the Extract C Comment option settings to see what combination yields the best results.

Each time you run the comment extractor, diagnostic statistics are displayed in the StP Message Log. These statistics indicate the percentage of user-defined functions, structures, and globals that have been found to have associated comments, given your search specifications. If the percentages are not satisfactory, you can rerun the comment extractor with new options to increase the percentage of comments found. Comment extraction replaces any comments in the semantic model with the latest ones found.

Note: The percentage of structures and identifiers that are commented is based only on the structures and identifiers found in the source files and in header files that are included using the `#include "filename.h"` notation. Comments extracted from system files are not included in the diagnostic calculation.

The comment extraction program can be used to check comment usage standards within an organization, also. For example, if every function should have a comment above its definition, make sure that you achieve a 100% rate of extracting comments for functions.

Detection of Inappropriate Language

If inappropriate language, such as swear words, is found in any of the comments, StP/SE replaces it with a random collection of `#@!&*` characters and informs you of the number of words it has replaced. Otherwise, this feature is silent.

Generating a Model from Parsed Files

The **Generate New Model** command creates a graphical model of your software from the semantic model produced by the C parser. The graphical model consists of structure charts, data structure diagrams, flow charts, and their related annotations, which are stored in the repository for the current StP project and system.

It is assumed that you are generating these diagrams and annotations into an empty system. If the system is not empty, repository contents will be removed during generation of the new model (for details, see “Regenerating an Entirely New Model” on page 11-45).

You use these dialog boxes to specify which diagrams you want to generate and how they should be drawn:

- **Generate New Model** dialog box
- **Set Diagram Generation Options** dialog box

The following sections provide an overview of each dialog box and explain how to use them.

Using the Generate New Model Dialog Box

The **Generate New Model** dialog box (Figure 6) enables you to specify the diagrams to be generated and to optionally back up existing diagrams.

Figure 6: Generate New Model Dialog Box

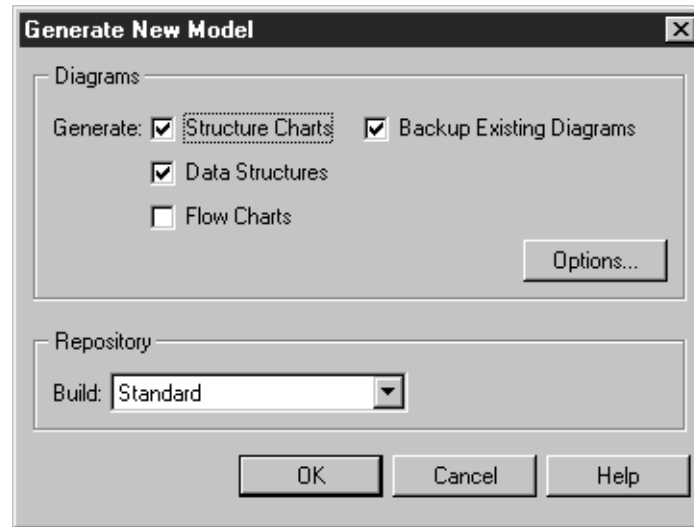


Table 7 describes the **Generate New Model** dialog box options.

Table 7: Generate New Model Dialog Box Options

Option	Description
Diagrams group:	
Generate options list: Structure Charts Data Structures Flow Charts	Allows non-exclusive choice of the types of diagrams to be created (the default is Structure Charts and Data Structures). Annotations are automatically generated as needed for the specified diagram types.
Backup Existing Diagrams	Copies the existing DSE, FCE, and SCE diagrams and their annotations, and places the copies in the <i>backup</i> directory in the current project/system directory.
Options button	Displays the Set Diagram Generation Options dialog box, which allows you to specify options for generating Structure Charts and Data Structure Charts. For option descriptions, see Table 8 on page 11-37.

Table 7: Generate New Model Dialog Box Options (Continued)

Option	Description
Repository group:	
Build options list: Standard Privileged	If Privileged is selected, decreases the time required for the initial StP repository build if the current user is the system owner. Standard is the default.

To use the **Generate New Model** dialog box:

1. From the StP Desktop **Tools** menu, choose **Reverse Engineering > Generate New Model**.
2. In the **Generate New Model** dialog box, select one or more types of diagrams you want to generate.
You do not need to generate structure charts in order to generate data structure charts.
3. Select **Backup Existing Diagrams** to save copies of the current data structure, structure chart, and flow chart diagrams and their annotations in a backup directory.
4. Click **Options** to display the **Set Diagram Generation Options** dialog box; set options as desired, and click **OK** (see “Using the Set Diagram Generation Options Dialog” on page 11-36).
5. In the **Generate New Model** dialog box, click **OK** to generate the model.
Diagrams and their related annotations are generated and stored in the repository for the current project and system. Results are reported in the StP Message Log.

Using the Set Diagram Generation Options Dialog

The **Set Diagram Generation Options** dialog box (Figure 7) enables you to set options that control how the generator draws the specified diagrams.

This dialog is accessed from either of these dialog boxes:

- **Generate New Model** dialog box, described in “Using the Generate New Model Dialog Box” on page 11-34
- **Synchronize Model** dialog box, described in “Using the Synchronize Model Dialog Box” on page 11-46

Figure 7: Set Diagram Generation Options Dialog Box

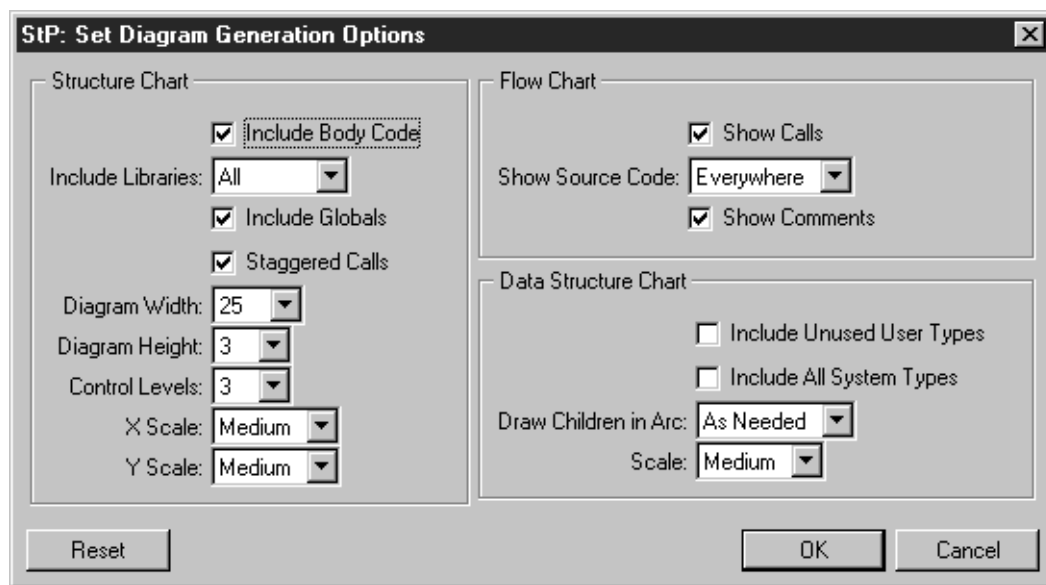


Table 8 describes the options for the **Set Diagram Generation Options** dialog box.

Table 8: Set Diagram Generation Options

Option	Description
Structure Chart group:	
Include Body Code	Creates C Code Body annotation notes on the structure chart for all functions in the source code. Each note contains the code for a single function.

Table 8: Set Diagram Generation Options (Continued)

Option	Description
Include Libraries	Specifies which library calls appear on the generated structure chart. Choices are All , None , or Selected . If you choose Selected , library calls enclosed in curly braces ({ }) in the appropriate library template file do not appear in the structure chart. For details, see “Suppressing Library Calls in Structure Charts” on page 11-42.
Include Globals	Creates diagrams showing accesses on global variables. This allows you to see the extent of the software’s reliance on global data. However, too many globals can make the diagrams harder to read.
Staggered Calls	Creates diagrams in which functions called by the same parent function are drawn as modules placed at slightly different, alternating heights on the chart, rather than as adjacent modules. This reduces the width of the diagram.
Diagram Width	Limits the maximum number of modules allowed in any one row of a structure chart, which limits the overall width of the structure chart. Choices available on the menu range from 5 to 100 . Suggested settings are: 25 (for viewing diagrams on screen) 15 (for printing on an 8.5 x 11 page)
Diagram Height	Limits the maximum number of levels of calls shown on any structure chart, which limits the overall height of the structure chart. Choices range from 1 to 10 . Suggested initial setting is 3 .
Control Levels	Determines the presence or absence of diamond and arrow symbols representing how modules are called, and determines the levels of nesting shown. Choices range from 0 to 10 . Suggested initial setting is 2 or 3 . For more information, see “Setting Control Levels” on page 11-43.
X Scale	Determines how much horizontal space to leave between adjacent symbols in the diagrams.
Y Scale	Determines how much vertical space to leave between adjacent symbols in the diagrams.

Table 8: Set Diagram Generation Options (Continued)

Option	Description
Flow Chart group:	
Show Calls	Adds extra symbols to flow charts to depict functions, library functions, and pointers to functions. Default setting is on .
Show Source Code	Controls the display of relevant source code in symbols containing code on flow charts. Choice are Nowhere , In Tests , and Everywhere . The In Tests option displays source code in test cases (conditional symbols) only. Default is Everywhere .
Show Comments	Displays relevant source code comments in comment boxes on flow charts. Default setting is on .
Data Structure Chart group:	
Include Unused User Types	Forces the production of data structure diagrams for all structures, whether or not the structures are actually used in the structure charts.
Include All System Types	Creates data structure charts for data structures defined in the system header files.
Draw Children in Arc	Provides control over whether child symbols are drawn in a semi-circular arc beneath a parent. The arc provides maximum horizontal compression for wide diagrams. Options are Never , As Needed , or Always . The As Needed option draws arcs only for 12 or more children.
Scale	Determines how much horizontal and vertical space to leave between adjacent symbols in the diagrams.

Specifying Library Functions

StP/SE processes flow information as well as calling information and data information. StP/SE examines all of the functions in the source code it parses and builds a flow model for each one. In order to build a comprehensive flow model for the entire software system, StP/SE needs

to know exactly how each library function uses its parameters. The following template files tell StP/SE about the functions and how each function accesses its parameters:

- *templates\se\re\re_stdfuncs* file in the default installation contains templates for the C standard library functions
- *templates\se\re\user_re_funs* file contains user-created templates for library functions that do not have templates in the *re_stdfuncs* file

If a library function has no template in either file, the missing information can cause misleading information for flows on generated structure charts. For example, a global may be shown as being read when it is actually being written or updated. Also, some parameters' flows may be drawn in the incorrect direction.

When generating structure charts, a warning appears if any library templates are missing. The warning lists each function for which a library template could not be found.

Adding Templates for Library Functions

To produce the highest-quality diagrams, we recommend that you add library templates to the *user_re_funs* file for any required library function that does not have a template in the *re_stdfuncs* file.

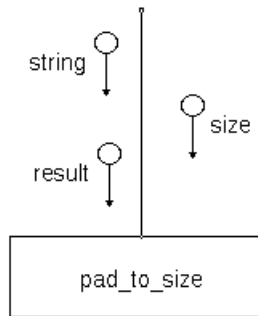
To add a template to the *templates\se\re\user_re_funs* file, follow the instructions in the comments at the top of the file.

To illustrate the importance of providing library templates, consider this function definition:

```
void pad_to_size( char *string, int size, char *result )
{
    int spaces = size - strlen( string );
    if ( spaces < 0 )
        return;
    strcpy( result + spaces / 2, string );
}
```

If no library templates are provided, a call to *pad_to_size* appears as shown in Figure 8.

Figure 8: Call to `pad_to_size` - No Library Template Available

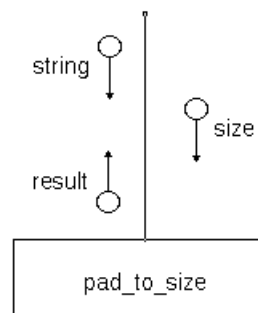


However, by examining the following lines in the *re_stdfuncs* library template file, StP/SE determines that `strcpy` takes two parameters, that the first is a write and the second is read, and that `strlen` takes a single parameter, which is read:

```
strcpy w r
strlen r
```

StP/SE can use this information to draw calls to *pad_to_size* as shown in Figure 9.

Figure 9: Call to `pad_to_size` - Library Template Available



Suppressing Library Calls in Structure Charts

If you do not want a library call to appear on the generated SCE diagram, enclose the entry in curly braces ({ }) in the appropriate library template file. The braces take effect when the **Include Libraries** option in the **Set Diagram Generation Options** dialog box is set to **Selected**.

Note: You should still provide a template for the library call's parameter accesses.

For example, if you only want structure charts to show calls to the `system()` function and the `exec` family of functions, you could change the file as shown in the following example.

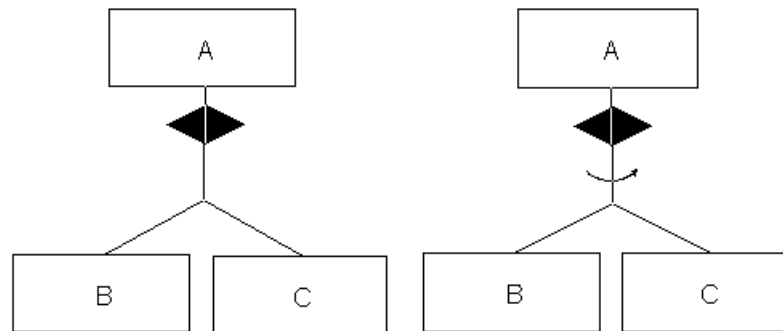
```
{
...           (other files before included in braces
strtoul      r w r
}
system      r
{
wcstombs     w r r
wctomb       w r

#           SYS\UNISTD.h
_exit        r
access       r r
alarm        r
chdir        r
chmod        r r
chown        r r r
close        r
ctermid      w
cuserid      w
dup          r
dup2         r r
}
execl        r r r r r
execle       r r r r r
execlp       r r r r r
execv        r r
execve       r r r
execvp       r r
{
... (other functions in braces to end of file)
```


Setting Control Levels

If the **Control Levels** field is set to zero, the procedural symbols shown in Figure 10 are not incorporated at all. Increasing the control level value increases the levels of nesting described by the symbols. For example, suppose a function A calls its subordinates B and C from within a while loop that is inside an if statement. In Figure 10, the structure chart on the left would be generated if the control level was set to 1; the structure chart on the right would be generated if the control level was set to 2.

Figure 10: Effect of Different Control Level Settings



Including Unused and System Types

The **Include Unused User Types** option forces the production of data structure diagrams for all structures, whether or not the structures are actually used in the structure charts. If this option is turned off (the default), diagrams are generated only for structures used to define the types of functions, parameters, and globals that actually appear in the structure charts. If only header files (and no source files) have been parsed, turn this option on; otherwise, no diagrams will be produced.

The **Include All System Types** option creates data structure charts for data structures defined in the system header files. In C header files, many structures are declared that implement low-level system concerns. Typically, this involves only a few file-handling data structures. However, Reverse Engineering processes approximately 90 data structures for

XView applications. Turn this option off (the default setting) to ignore anything defined in a header file that has been #included with the <> notation.

Maintaining and Updating Your Model

Once you have generated an initial model of your software, you can maintain it, using one or more of the following methods:

- Edit the diagrams with the StP/SE editors
- Modify the C source code and regenerate an entirely new model
- Modify the C source code, update the semantic model, and execute the Reverse Engineering **Synchronize Model** command to find differences and optionally update the graphical model

For instructions on editing your model directly with the StP/SE editors, refer to the chapters that discuss each editor. For instructions on using the other two methods, see the sections that follow.

Modifying the Source Code

StP/SE provides the following tools to help you identify parts of the source code you may want to modify:

- C Code Browser—Allows you to browse the reverse engineered semantic model and navigate to the corresponding C source code prior to editing (see “Browsing C Code” on page 12-13).
- Reverse Engineering **Synchronize Model** command—Identifies any differences between the current graphical model and the most recently generated semantic model, and allows you to navigate to the corresponding C source code for editing. For more information, see “Synchronizing the Model with Modified Code” on page 11-45.

After modifying the source code, you can rerun Reverse Engineering to generate an entirely new model or to update your model incrementally with the code changes only (synchronization).

Regenerating an Entirely New Model

To replace an existing generated model with an entirely new one, rerun all phases of Reverse Engineering on the modified source code:

1. Execute **Parse Source Code** in incremental mode to update the semantic model (if there are many code changes, use non-incremental mode to create an entirely new semantic model).
2. Execute **Extract Source Code Comments** to replace the comments in the semantic model with current ones.
3. Execute **Generate New Model** to generate an entirely new model from the current semantic model.

WARNING: If the current system repository contains any existing diagrams or annotations, the following contents will be removed during generation of the new model:

- Any structure charts, data structure diagrams, or flow charts previously generated with Reverse Engineering
- Any structure charts, data structure diagrams, or flow charts created or edited with StP/SE editors
- All annotations, including those created with other editors (such as the Data Flow Editor or the State Transition Editor)

To simplify rerunning Reverse Engineering, the dialog box options for each command are saved from the previous execution of that command and appear as the current defaults. You can change the option settings or use the same ones when you re-execute the command.

Synchronizing the Model with Modified Code

To synchronize the graphical model of your software with modified source code, you update the semantic model and run the model synchronizer, as follows:

1. Re-parse the source code in incremental mode to update the semantic model (if there are many code changes, use non-incremental mode to create an entirely new semantic model). For details, see “Parsing the Source Files” on page 11-6.

2. Rerun comment extraction to remove all existing comments from the semantic model, gather current comments from the source code and add them to the semantic model. For details, see “Extracting Comments” on page 11-27.
3. Run the model synchronizer to compare the semantic and graphical models and optionally update the graphical model.

The **Synchronize Model** command allows you to select synchronization options in the **Synchronize Model** dialog box and runs the model synchronizer.

The model synchronizer:

- Finds all differences between the graphical model in the repository and the corresponding information in the semantic model created from the source code
- Allows you to navigate to differences in the graphical model and the code, and to automatically update the graphical model with selected or all changes in the code

Using the Synchronize Model Dialog Box

The **Synchronize Model** dialog box (Figure 11) enables you to specify the behavior of the synchronization and the files to synchronize.

Figure 11: Synchronize Model Dialog Box

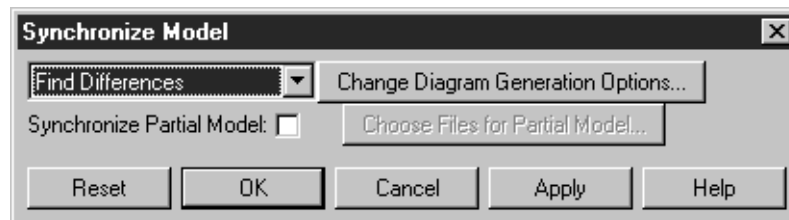


Table 9 describes the **Synchronize Model** dialog box options.

Table 9: Synchronize Model Dialog Box Options

Option	Description	For Details, See
Find Differences; Load Existing Differences	Synchronization options that determine the contents of the Synchronization Differences dialog: Find Differences —Finds and stores differences between semantic and graphical models. Load Existing Differences —Displays stored differences from the previous Synchronize Model execution.	
Change Diagram Generation Options button	Displays the Set Diagram Generation Options dialog box, which allows you to specify options for generating the diagrams.	“Using the Set Diagram Generation Options Dialog” on page 11-36
Synchronize Partial Model	Allows you to search for differences only in specified source files, and optionally apply them to the model. If selected, click Choose Files for Partial Model to specify the files.	“Partial Synchronization” on page 11-48
Choose Files for Partial Model button	Displays the Select Files to Search dialog box, which allows you to select which files to search for differences. Available only when Synchronize Partial Model is selected.	

To use the **Synchronize Model** dialog box:

1. From the Desktop **Code** menu, choose **Reverse Engineering > Synchronize Model**.
2. In the **Synchronize Model** dialog box, choose either of these synchronization options, as described in Table 9:
 - **Find Differences**
 - **Load Existing Differences**
3. Click **Change Diagram Generation Options**.
The **Set Diagram Generation Options** dialog box appears. For instructions on using this dialog box, see “Using the Set Diagram Generation Options Dialog” on page 11-36.
When done, the **Synchronize Model** dialog box reappears.
4. Optionally, to synchronize the model with changes from specified files or directories only, select the **Synchronize Partial Model** option and click **Choose Files for Partial Model**.
The **Select Files to Search** dialog box appears. For instructions on using this dialog box, see “Partial Synchronization” on page 11-48.
5. When the **Synchronize Model** dialog box reappears, click **OK**.
The **Synchronization Differences** dialog box appears with a list of differences.

For instructions on updating your model with the displayed differences, see “Applying Synchronization Results” on page 11-52.

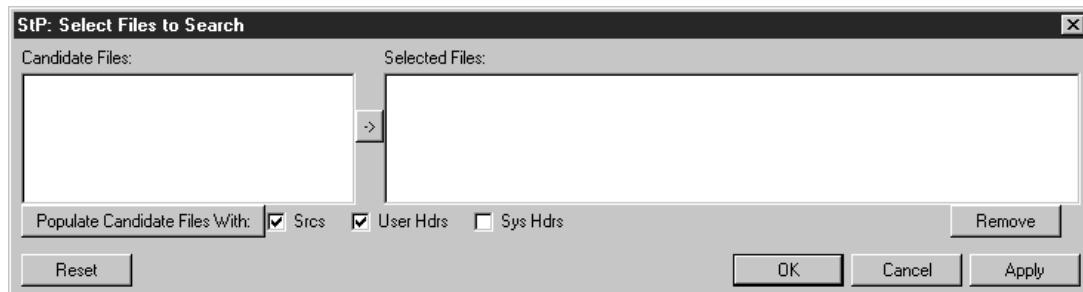
Partial Synchronization

Partial synchronization allows you to search for differences only in specified source files, and optionally apply them to the model.

To specify the source files for partial synchronization:

1. In the **Synchronize Model** dialog box (Figure 11 on page 11-46), choose a synchronization option and change diagram generation options (see “Using the Synchronize Model Dialog Box” on page 11-46).
2. Select the **Synchronize Partial Model** option and click **Choose Files for Partial Model**.
The **Select Files to Search** dialog box appears (Figure 12).

Figure 12: Select Files to Search Dialog Box



3. Select any combination of the file type options (**Srcs**, **User Hdrs**, **Sys Hdrs**) and click the **Populate Candidate Files With** button (see Table 10 for details).
A list of prospective files matching your selected file categories appears in the **Candidate Files** list.
4. From the **Candidate Files** list, select the files you want to search; then click the arrow button (->) to move them into the **Selected Files** list.
5. Click **OK** to retain your selections and exit this dialog box.
6. In the **Synchronize Model** dialog box, click **OK**.
The **Synchronization Differences** dialog box appears with a list of differences (see Figure 13 on page 11-52).

Table 10 lists the **Select Files to Search** dialog box options.

s

Table 10: Select Files to Search Dialog Box Options

Option	Description
Candidate Files list	A list of potential files.
Selected Files list	The final list of selected files with which you want to compare the graphical model.
Populate Candidate Files With button and options	A button and non-exclusive set of choices that determine which files appear in the Candidate Files list. You can select any combination of Srcs (source files), User Hdrs (user header files) and Sys Hdrs (system header files).

Table 10: Select Files to Search Dialog Box Options (Continued)

Option	Description
Arrow button (->)	A button to put highlighted files from the candidate files list into the selected files list.
Remove button	A button to remove highlighted files from the selected files list.

Detected Differences

The model synchronizer detects and reports the following differences between the graphical model and C source code.

Table 11: Detected Differences

Difference Type	What is Detected
Function Existence	Function exists in design but not in code. Function exists in code but not in design.
Function Signature	Function parameters differ in code and design (in type, number, or both). Function return type differs in code and design.
Function Calls	Where <i>x</i> can be either a regular or library module: - Function calls <i>x</i> in design but not in code. - Function calls <i>x</i> in code but not in design. - Function <i>x</i> uses global <i>y</i> in design but not in code. - Function <i>x</i> uses global <i>y</i> in code but not in design. Function symbol (library, include module, or module) differs in design and code.

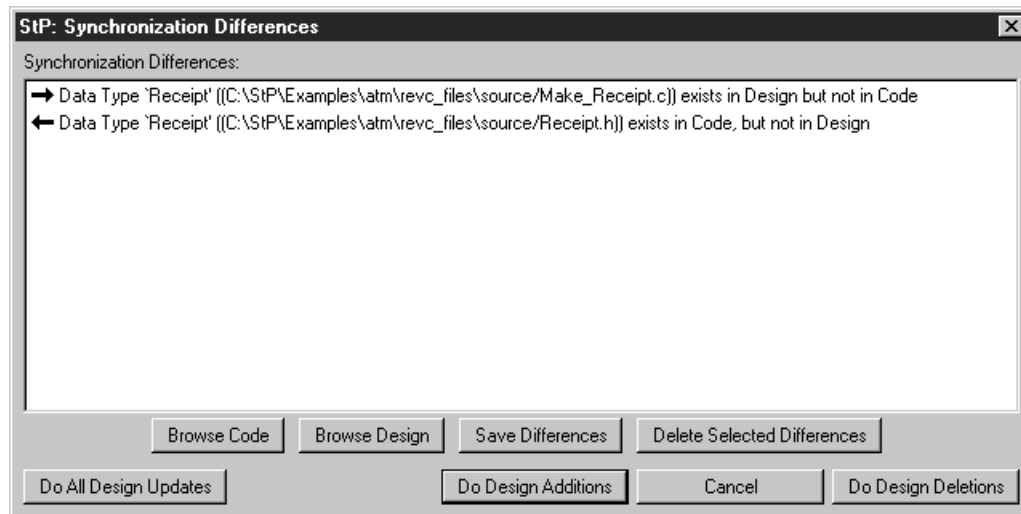
Table 11: Detected Differences (Continued)

Difference Type	What is Detected
Data Type Existence	Where the data type is a structure, union, or enumeration: - Data type exists in design but not in code. - Data type exists in code but not in design. Data type symbol (struct, union, or enumeration) differs in design and code.
Data Type Fields	Data type has child <i>x</i> in design but not in code. Data type has child <i>x</i> in code but not in design. Field <i>x</i> has a different type in the code and design.
Global Variable Existence	Global defined in design but not in code. Global defined in code but not in design. Global variable type (including storage class) differs in code and design. Global variable access mode differs in a certain function between design and code.
Comment Differences	Comment for function differs in design and code. Comment for global differs in design and code. Comment for data type differs in design and code. Comment for field differs in design and code.
Typedef Existence	Typedef exists in code but not in design. Typedef exists in design but not in code. Typedef has a different type in the code and design.

Applying Synchronization Results

The **Synchronize Model** command stores any detected differences between the design and new semantic models in the Synchronize Differences list. This list automatically appears in the **Synchronization Differences** dialog box, shown in Figure 13, when you execute the **Synchronize Model** command.

Figure 13: Synchronization Differences Dialog Box



Arrow icons (-> <-) preceding differences indicate whether a difference needs to be inserted into or removed from the design in order to match the code. Right arrows indicate elements that need to be inserted into the design; left arrows indicate elements to be removed from the design.

You can browse to the corresponding source code for any difference or apply all or selected changes to the graphical model from this dialog box. Table 12 describes the buttons on the dialog box that allow you to apply differences.

Table 12: Synchronization Differences Dialog Buttons

Button	Description
Browse Code	Displays the corresponding piece of source code for the selected difference in a window for viewing.
Browse Design	Displays the element corresponding to the selected difference in the graphical model in the appropriate StP/SE editor.
Save Differences	Saves all differences to a file that you can open later for viewing, browsing, or updating your model.
Delete Selected Differences	Removes selected differences in the Synchronization Differences list from the graphical model if not represented in the semantic model.
Do All Design Updates	Redraws the model according to all detected differences (includes additions, deletions, and changes to the model).
Do Design Additions	Adds all entirely new features to the graphical model (does not remove any features or redraw diagrams for changed features).
Do Design Deletions	Removes all features from the graphical model that are not represented in the semantic model (does not add or change any features).

Applying Differences to the Model

To apply differences to the model, click the button for an appropriate option, as described in Table 12. The diagrams are updated and the new files are committed to the repository.

To delete only selected differences from the graphical model:

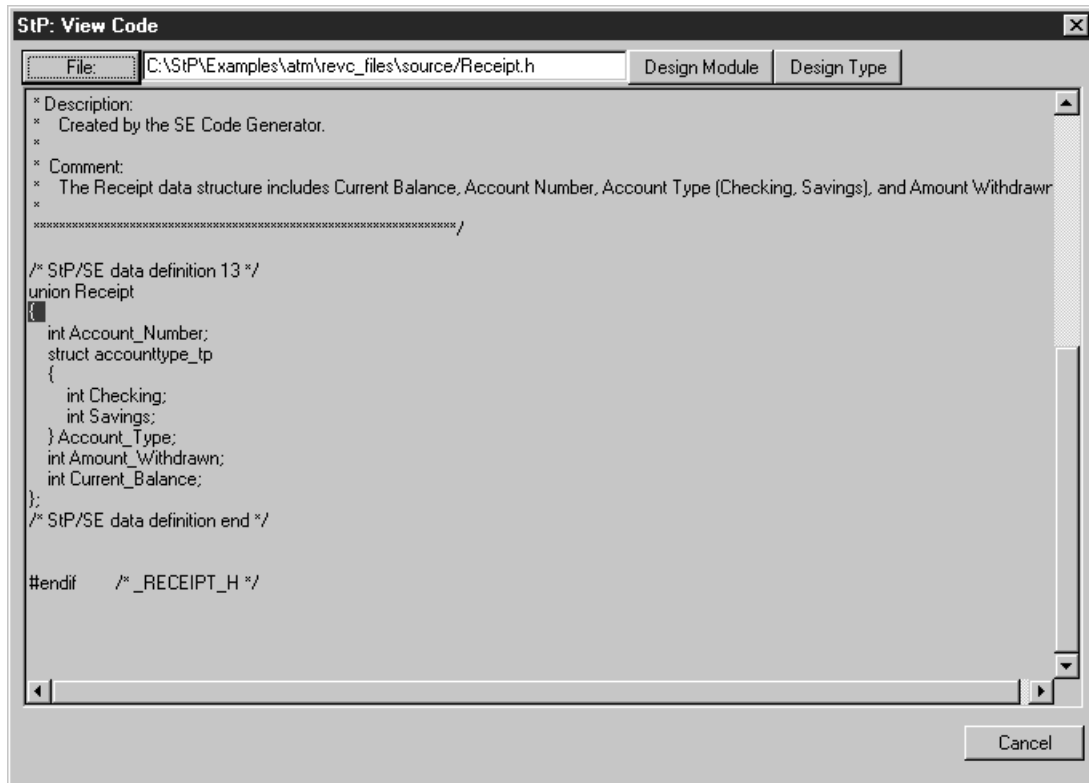
1. Highlight one or more differences in the **Synchronization Differences** dialog that are preceded by a left arrow (for removal).
2. Click **Delete Selected Differences**.
The affected diagrams are updated and the changes committed to the repository.

Browsing to Corresponding Code or Design Elements

To browse to a specific piece of code or element in a model from a selected difference, do one of the following:

- Select a difference and click **Browse Design** to browse the model.
The browser displays the corresponding element in the graphical model in the appropriate StP/SE editor. One or more affected elements in the model are highlighted.
- Select a difference and click **Browse Code** to browse the source code.
The **View Code** dialog box appears, as shown in Figure 14, displaying a corresponding segment of source code, in which the modified code is highlighted. The name of the file containing the code is displayed in the **File** text field.

Figure 14: The View Code Dialog Box



The following table describes the **View Code** dialog box options.

Table 13: View Code Dialog Box Options

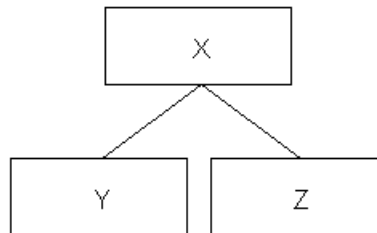
Option	Description
File button and field	Entering a directory and filename and pressing the File button displays the contents of the specified file in the view window.

Table 13: View Code Dialog Box Options (Continued)

Option	Description
Design Module button	If you select the name of a function in the displayed code, allows you to navigate to any SCE diagram in which the selected function is defined or referenced, by choosing from a displayed list.
Design Type button	If you select the name of a data structure element in the displayed code, allows you to navigate to any DSE diagram in which the selected element is defined or referenced, by choosing from a displayed list.

Generated Structure Charts

Structure charts depict the hierarchical relationship between functions and modules. Figure 15 describes the hierarchical relationships that exists between modules X, Y, and Z. We can see that module X calls Y and Z. The ordering of subordinates from left to right is based on the order in which they are called. Module X first calls Y, then Z.

Figure 15: Depicting Hierarchical Relationships

If X calls module Z in two different places, only one of the calls is represented on the diagram. StP/SE always chooses the least nested of the calls. If both calls are nested at the same depth within the code, then the call that appears first in the code is chosen. The left to right ordering

of subordinates is therefore based on the order in which the module calls appear within the code.

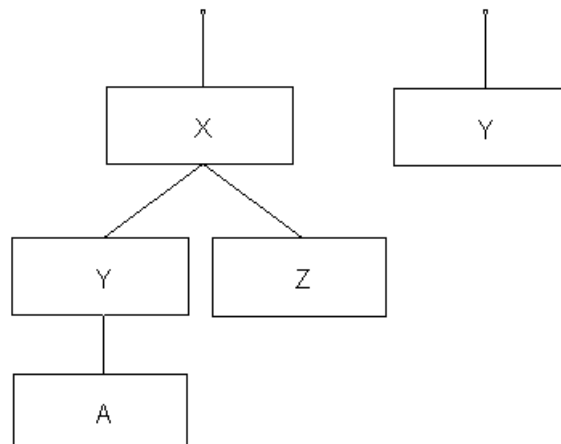
Defining Points

Each module is defined in only one place, although many calls to it may exist on different diagrams. StP/SE can be made to generate a separate defining diagram for every single source code function, but it can also generate several defining points on a single diagram, thus reducing the total number of diagrams that are produced and increasing the amount of information on individual diagrams.

StP/SE defines a module on another module's diagram only if there is room for it and if there is only one call to it in the entire system.

By using the diagram height and width settings on the **Set Diagram Generation Options** dialog box, it is possible to vary the number of levels of call information that are depicted on each diagram.

Figure 16: Showing Multiple Defining Points



In Figure 16, Y is called only by X; there is no other module in the system that calls Y. It is not necessary to produce a separate diagram for Y, because putting it on this diagram does not break the maximum diagram height and width settings.

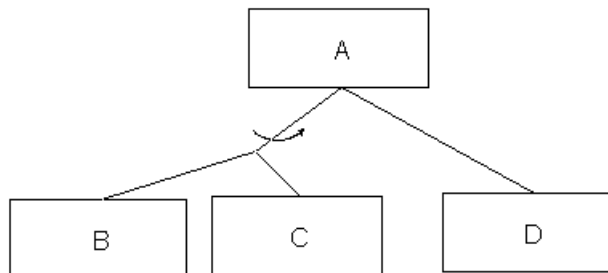
If *Y* was also called by another function not depicted on this diagram, then *Y* would be defined on its own diagram. The call to *A* would be drawn on *Y*'s diagram and would not be depicted in *X*'s diagram.

Control Levels for Procedural Symbols

Structure charts can incorporate procedural symbols (also known as control symbols) to show when calls to subordinate modules are made within iterative or conditional control (selection) constructs, such as “while” and “if” constructs. You can vary the usage level, and therefore the number, of procedural symbols drawn in the structure chart by setting a control level between 0 and 10 on the **Set Diagram Generation Options** dialog box.

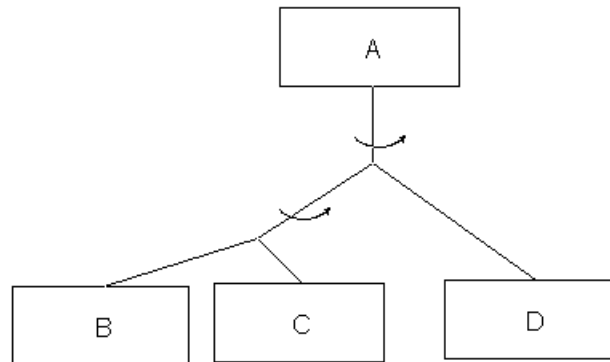
The structure chart in Figure 17 indicates that module *A* executes a loop construct containing calls to modules *B* and *C* before calling module *D*.

Figure 17: Loop Construct Example 1



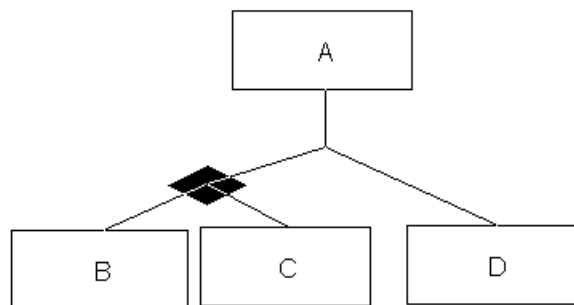
The structure chart in Figure 18 indicates that module *A* executes a loop construct that contains another loop (containing calls to *B* and *C*) and a call to module *D*.

Figure 18: Loop Construct Example 2



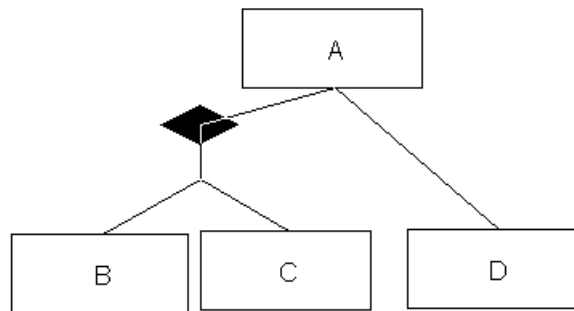
The structure chart in Figure 19 indicates that module *A* calls either *B* or *C* (but not both) before it calls module *D*.

Figure 19: Loop Construct Example 3



The conditional arrangement shown in Figure 19, represented by a diamond selection symbol, illustrates the classic “if...then...else” control construct. The next diagram models an “if...then” construct. The structure chart in Figure 20 indicates that if the test condition in the branch construct is satisfied, then both modules *B* and *C* are called. Module *D* is always called.

Figure 20: Loop Construct Example 4



Varying the control levels setting on the **Set Diagram Generation Options** dialog box determines the number of layers of diamonds, arrows, and vertices that are shown between calls on structure charts generated by StP/SE. The choice of control levels is a matter of personal preference. A diagram with a large number of symbols is harder to manipulate within the structure chart editor.

Note: When interpreting the control constructs governing the calls, keep in mind that no matter how many calls to a function actually exist in the code, only the call that occurs at the outermost lexical scope is drawn on the structure chart. For example, a function *D* may also be called conditionally, or a function *C* may be called whenever *B* is called. However, only a single call to each function is drawn on the diagram.

The following code and figures provide an example of how varying the control levels setting can affect the generated structure chart.

```
int main()
{
    int ok;
    ok = check_parameters();
    if ( !ok )
    {
        read_parameter_file();
        if ( check_parameters() != ok )
            return 8;
    }
    while( !eof( data_file ) )
    {
        entries = read_line( data_file );
        for( loop = 0; loop < entries; loop++ )
        {
            int type;
            fscanf( data_file, "%d", &type );
            if ( type == 1 )
            {
                if ( stderr != NULL )
                    fprintf( "warning, type 1 record should not be in the data
                        file.\n" );
                else
                    printf( "warning, type 1 record unexpected here." );
            }
        }
    }
    return 0;
}
```

If you set the control levels to 0 for the preceding code, the generated structure chart appears as in Figure 21.

Figure 21: Structure Chart for Control Levels Set to 0

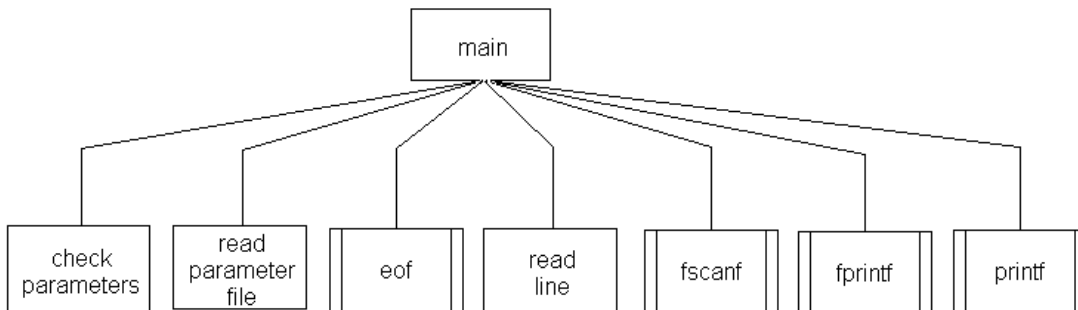


Figure 22 shows the result of setting the control levels to 1.

Figure 22: Structure Chart for Control Levels Set to 1

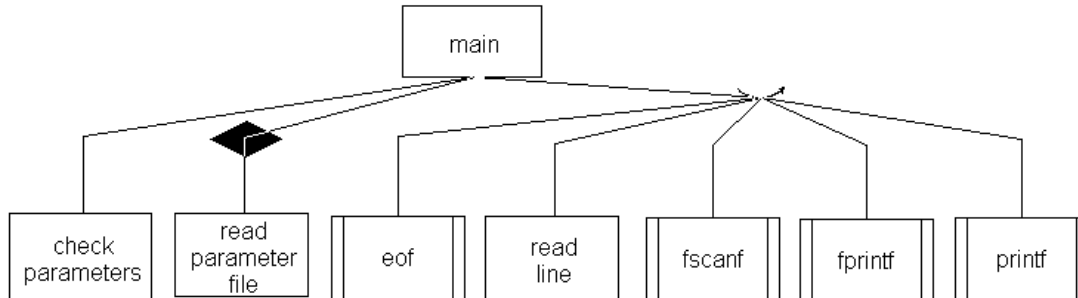
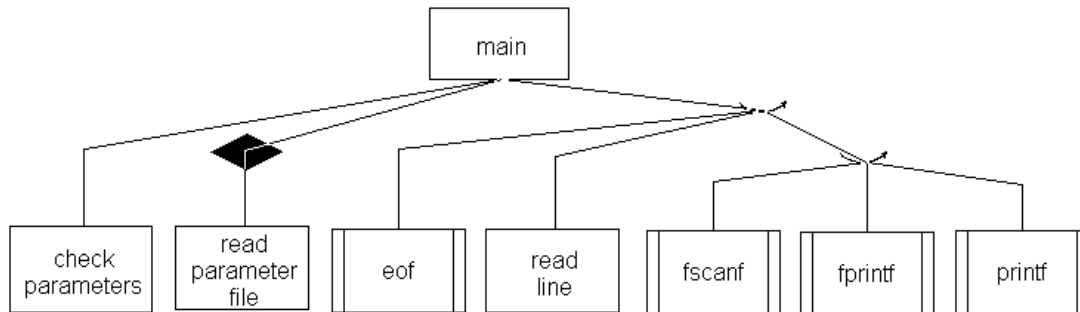


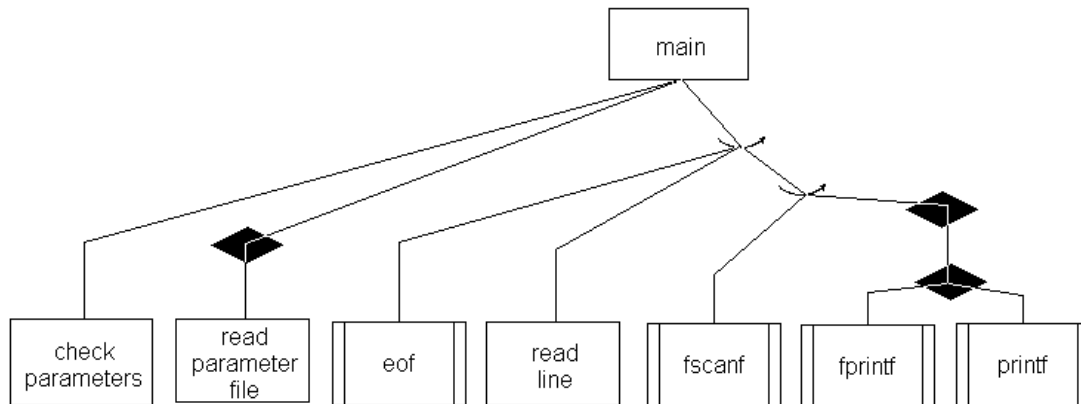
Figure 23 shows the result of setting the control levels to 2.

Figure 23: Structure Chart for Control Levels Set to 2



Control Levels settings of 4 or more produce the structure chart shown in Figure 24.

Figure 24: Structure Chart for Control Levels Set to 4 or More

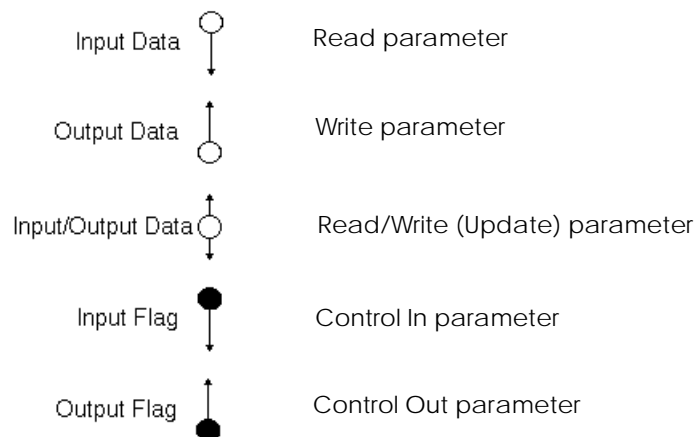


Parameters

StP/SE supports both formal and actual parameters on diagrams. The Reverse Engineering diagram generator draws only formal parameters on the diagrams to represent both actual and formal parameters in the code. Instead of drawing the actual parameters, it draws formal parameters on the appropriate arc-to-module link for the module's defining point.

The direction of the arrow symbol indicates whether the parameter is read, written, or both read and written. If a formal parameter is of a simple type, such as an integer, and it is only read within the test part of control constructs, then the diagram generator models it as a control parameter (or "flag"). Figure 25 shows all of the parameter symbols.

Figure 25: Parameter Symbols for Structure Charts



The diagram generator draws a parameter as an input/output data symbol if it is sometimes read and sometimes written by a function, whether or not an actual update takes place.

To classify accesses on a parameter, StP/SE applies C semantic rules of data access. Consider the following code:

```
int main( int argc, char **argv )
{
    int arg_count = argc;
    if ( argv[0] && strcmp( argv[0], "my_prog" ) == 0 )
        argv = NULL;
}
```

In this piece of code, `main` first reads `argc`, then reads `argv`, and finally writes `argv`. However, when StP/SE models the parameter accesses, it views what the function is doing to its parameters, as seen by a module that calls it. Taking this view, `argv` is not actually written (`*argv = "something"` would indicate that the parameter is being written). Both parameters are analyzed as reads.

The diagram generator draws unused parameters reads, although it warns you about them.

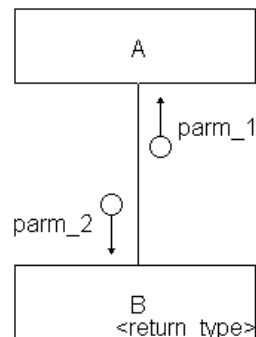
Parameter accesses are filtered up from lower calls to higher ones. For example,

```
int main( int argc, char **argv )
{
    if ( check_parms( argc, argv ))
        printf( "Bad arguments\n" );
}
```

The accesses on `argc` and `argv` are inherited by `main` from `check_parms`. If `argc` is evaluated as a control flow in `check_parms`, then it is also drawn as a control flow on `main`.

When drawing parameters, StP/SE plots them alternately on either side of the call, as seen in Figure 26. The return type appears in the lower right corner of the module symbol.

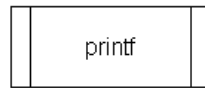
Figure 26: Drawing Parameters on Structure Charts



Library Modules

Function declarations in the source code hold information about return types and, optionally, parameter types. Function definitions contain the same information as a declaration, but also hold the body of the function. A function definition that has been parsed by StP/SE is considered a user-defined function. Any function that is called but has not been defined is considered a library function. StP/SE draws calls to library functions using the notation shown in Figure 27.

Figure 27: Depicting Calls to Library Functions



If a source file has not been parsed because it was intentionally left out, could not be found, or contained syntax errors, then the functions it defines will be drawn as libraries.

Global Data

The representation of global data on structure charts is optional when using StP/SE. However, if this option is used, external variables and static variables with file scope (that is, variables that are defined outside of C functions) are modeled as global data. Global data accesses are always shown to the right of function calls and do not contain any extra information through their positioning.

Filtering of global accesses is not performed. A function that calls another inherits the accesses on the parameters but does not inherit the global accesses.

Generated Data Structure Charts

C structure declarations are drawn as sequence symbols on DSE diagrams. C union declarations are shown using the selection symbol. In both cases, the members of the declaration are drawn as children (leaf nodes) of the parent, using sequence symbols.

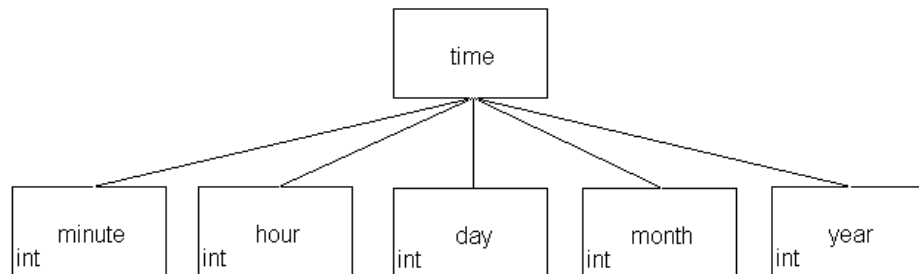
Each child appears with its data or return type shown as a display mark in the lower-left corner of the symbol. For example, the following code produces the diagram shown in Figure 28.

```
struct time
{
    int minute;
    int hour;
```



```
int day;  
int month;  
int year;  
};
```

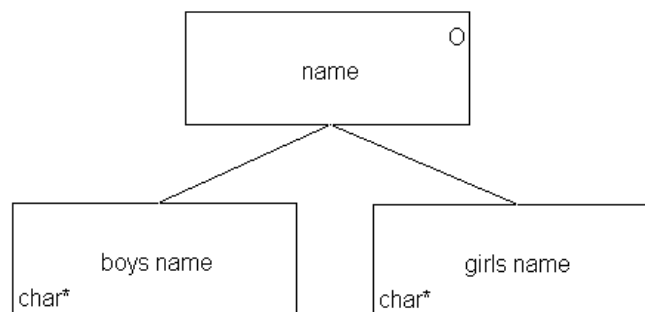
Figure 28: Data Structure Example for Struct



As another example, this code produces the diagram shown in Figure 29.

```
union name  
{  
    char *boys_name, *girls_name;  
};
```

Figure 29: Common Declaration Example



Nested Structures

StP/SE tries to avoid drawing nested structures on diagrams. If one structure is declared inside another, then generally two diagrams are created—one to hold the declaration of each structure. Only unnamed structures appear on the same diagram with their parent.

For example, the following code is generated as two diagrams, as shown in Figure 30.

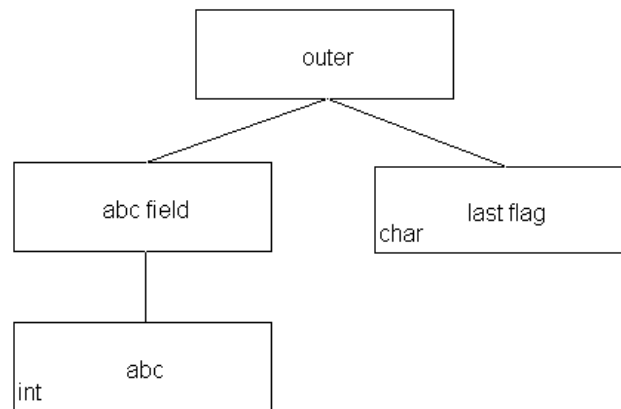
```
struct outer
{
    struct inner
    {
        int abc;
    } abc_field;
    char last_flag;
};
```

Figure 30: Example of Nested Structures



Alternatively, if the structure was defined as follows, it would appear on a diagram called *outer*, as shown in Figure 31.

```
struct outer
{
    struct { int abc; } abc_field;
    char last_flag;
};
```

Figure 31: Alternative to Nested Structures

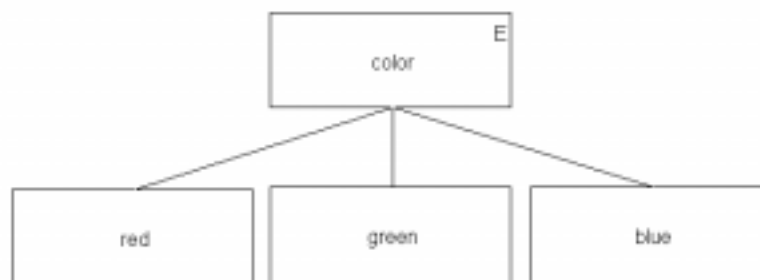
Enumeration

Each enumeration appears on its own diagram. The name of the diagram is the same as the name of the enumeration.

The root node for an enumeration appears as an enumeration symbol. All of the enumeration's children appear as sequence symbols.

For example, the following code generates the diagram shown in Figure 32.

```
enum color { red, blue, green };
```

Figure 32: Enumeration Example

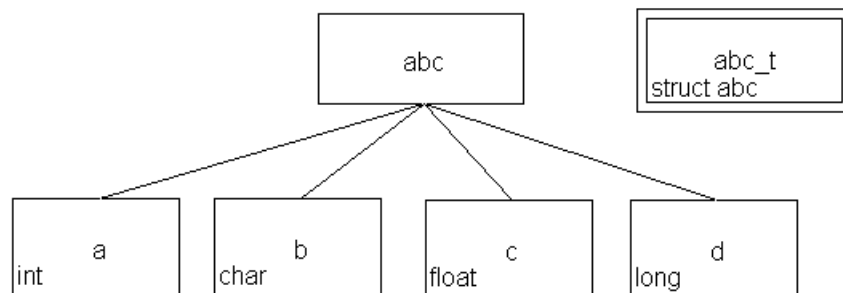
Typedefs

Typedefs that are new names for existing primitive types are all drawn on a special data structure diagram called *Typedefs*.

A typedef that provides an alias for a struct, union, or enum is shown on the appropriate struct, union, or enum's diagram as a solitary typedef symbol next to the root node. The following code generates a diagram called *abc*, as shown in Figure 33.

```
typedef struct abc{ int a; char b; float c; long d; } abc_t;
```

Figure 33: Typedef Providing Alias for a Struct



A structure may not have a name at all; for example:

```
typedef struct { int a; char b; float c; long d; } abc_t;
```

The unnamed structure is given the same name as the name of the typedef. Thus, the preceding code would result in a generated diagram named *abc_t* for a structure called *abc_t* with a typedef object called *abc_t*.

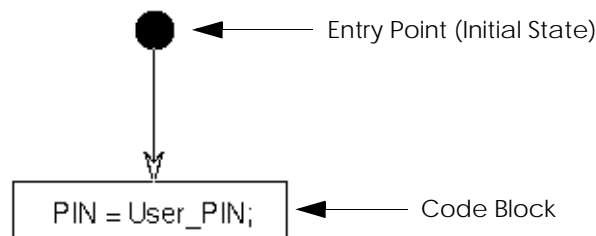
Generated Flow Charts

Flow charts provide a graphical representation of the program design for individual functional modules in your system. They contain symbols representing library and function calls, conditional statements, and actual blocks of code. Each chart represents a particular functional module.

Entry Point

Every flow chart has an entry point, which is represented by an initial state symbol connected to the symbol representing what is to be executed first when the function is called. When you are investigating a flow chart that has been reverse engineered, begin by looking for the entry point, which is located at the top of the diagram.

Figure 34: Entry Point in a Flow Chart



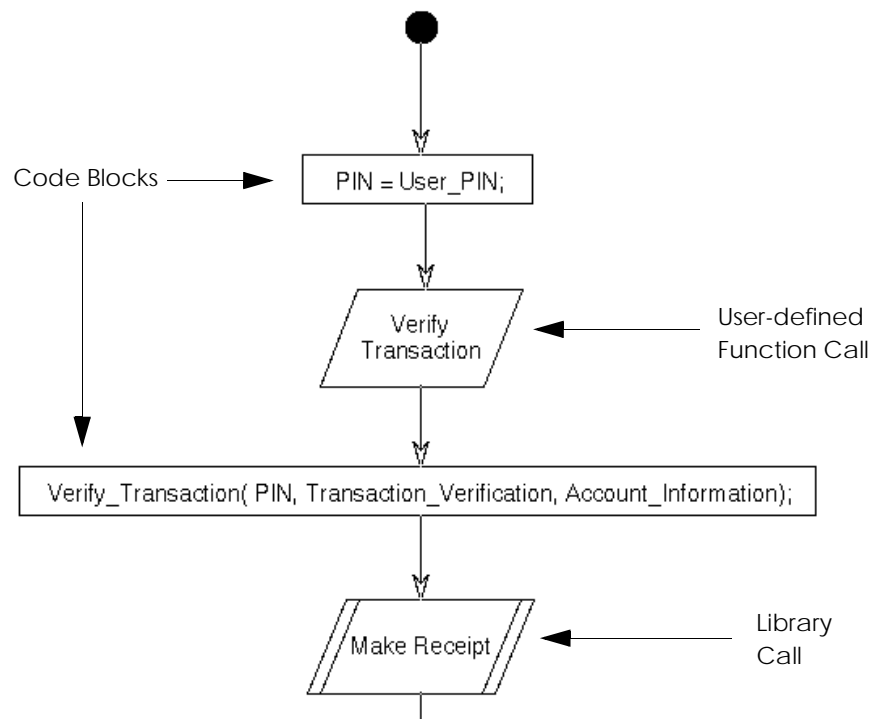
Code Blocks

Actual segments of source code appear in the code block symbol, as shown in Figure 34. The copied source code in the label appears exactly as found in the original C source code, except leading white space is removed. For legibility, no more than 5 lines of source code appear on any single symbol.

Calls

Calls are drawn with labels showing the name of the function that is being called. Different symbols indicate whether the function being called is a library, a user-defined call, or a pointer call.

Figure 35: Calls on a Flow Chart



Only the name of the function is labeled on a call symbol. The full details of the call appear in a code block or in a conditional diamond symbol. For example, in Figure 35 the actual call to *Verify Transaction* happens inside the code block following the user-defined function call.

Call symbols emphasize the calls in the code and provide a navigation mechanism between flow charts and structure charts.

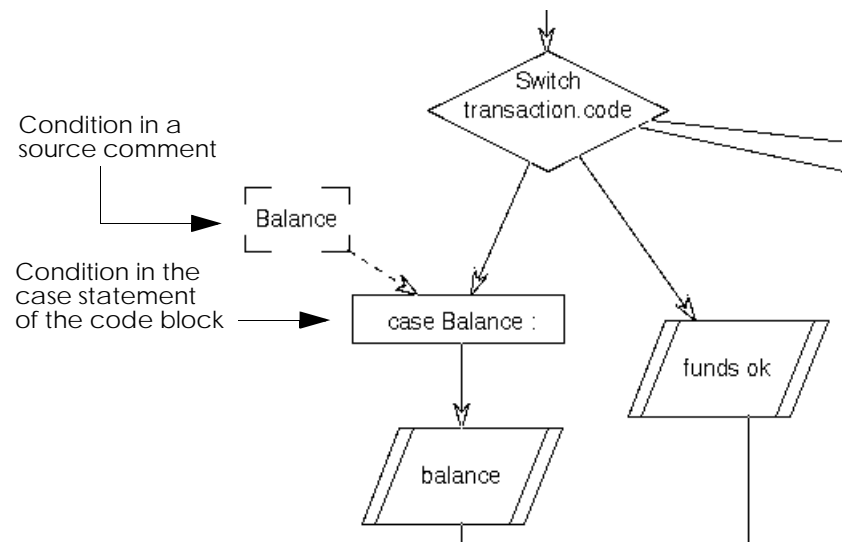
Decision Constructs

StP/SE provides three types of conditional test symbols for decision constructs: if, if-else, and switch symbols. Each appears as a diamond, with the type of the symbol indicated at the top of the diamond.

On if and if-else symbols, the true branch always appears to the left and the false branch to the right of the conditional symbol (see Figure 38 on page 11-75 for an example).

On switch symbols, the case statement in the code block indicates the condition. To make it more visible, Reverse Engineering also places the condition within a comment pointing to the code block, as shown in Figure 36.

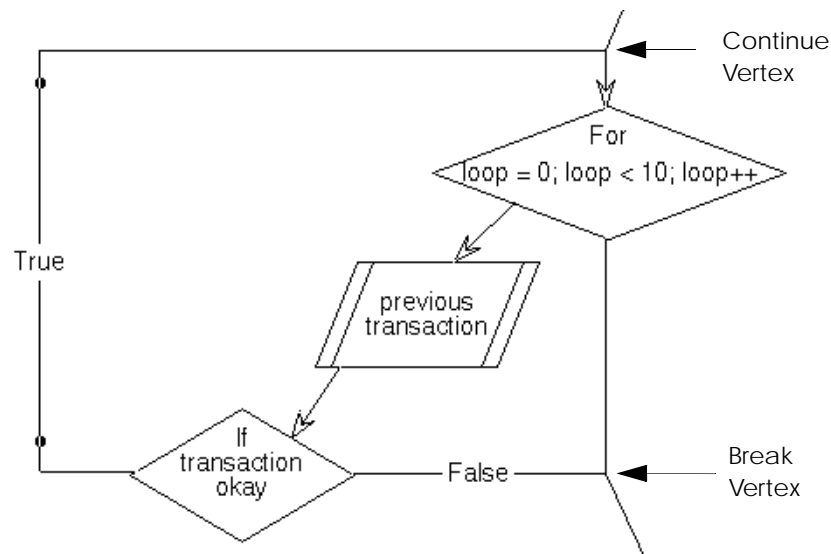
Figure 36: Decision Constructs in a Flow Chart



Loops

StP/SE supports three conditional test symbols for loop constructs: for, while, and do-while symbols. Each loop symbol appears as a diamond that represents the loop control condition.

Figure 37: Loop Constructs in a Flow Chart

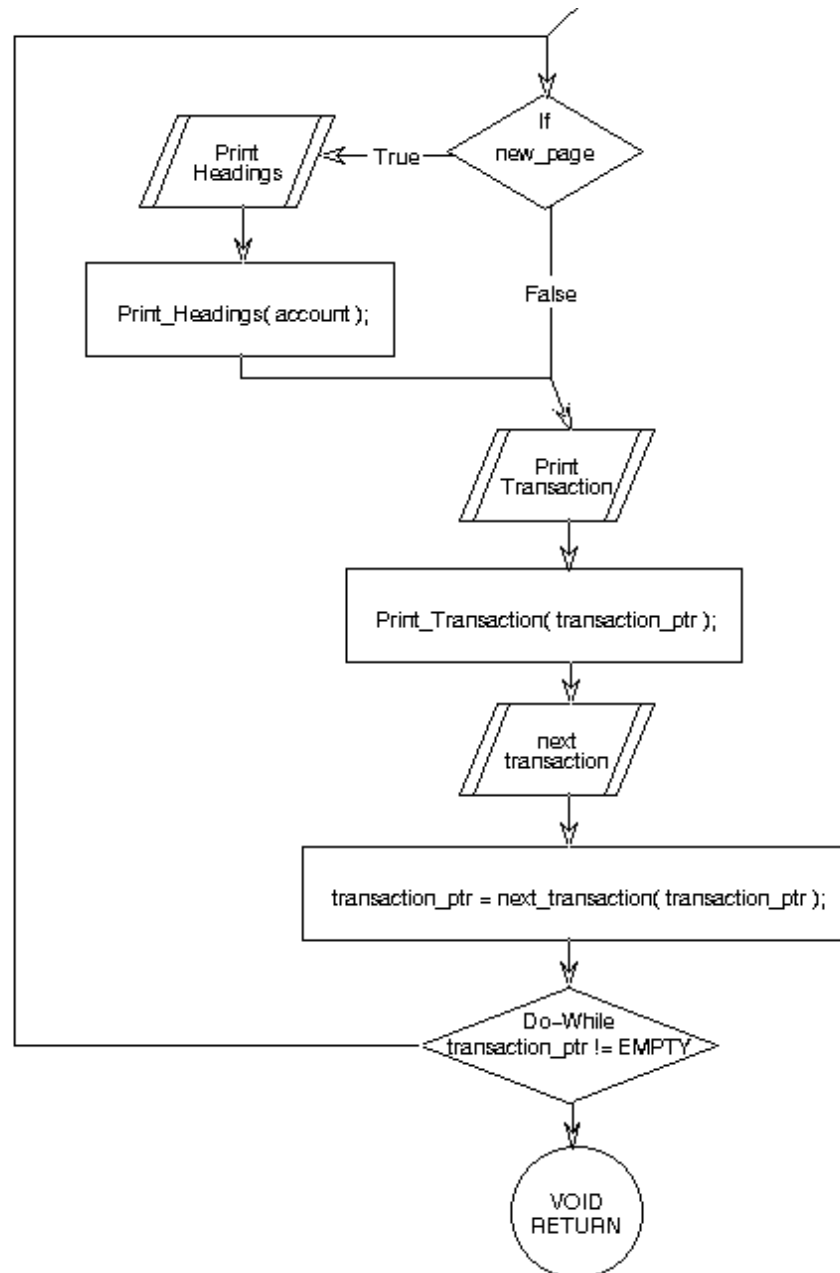


The vertex on the right-exiting link of the for loop symbol is known as the “break vertex.” If there is a break statement in the code, it appears as a link connected to this vertex. The vertex above the diamond is known as the “continue vertex.” Continue statements are linked to this vertex.

For loop constructs, the true branch always exits the loop symbol to the left and the false branch exits to the right. When the loop condition is true, the left link is taken and the body of the loop statement is executed. After each execution, control returns to the vertex just above the diamond. When the test within the diamond fails, the right branch is executed.

Figure 38 contains a do-while loop. The test appears at the end of the body of the loop to ensure that the loop body is always executed at least once.

Figure 38: Do-While Loop Construct



Exits and Aborts

C programs may contain function calls that:

- Have more than one exit point
- Return a value
- Return no value
- Do not return

Exit Points

An exit point is the point at which execution returns to the caller of the function. Reverse Engineering generates one of the following symbols to represent a function's exit point (for exceptions, see “Calls to the exit() Library Function” on page 11-79).

Figure 39: Exit Points



Figure 40 illustrates a call that returns a value.

Figure 40: Terminal Symbol Example

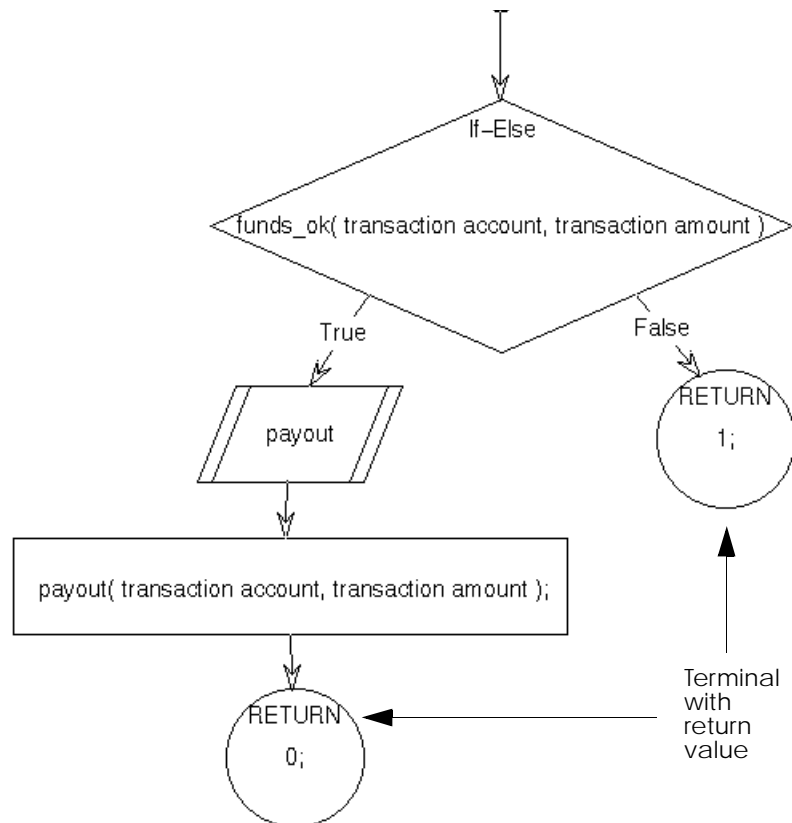


Figure 41 illustrates a void return. Void Terminal symbols are not labeled.

Figure 41: Void Terminal Symbol Example

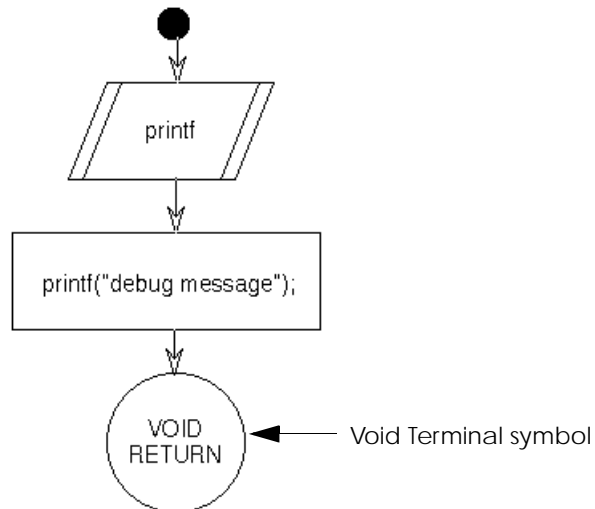
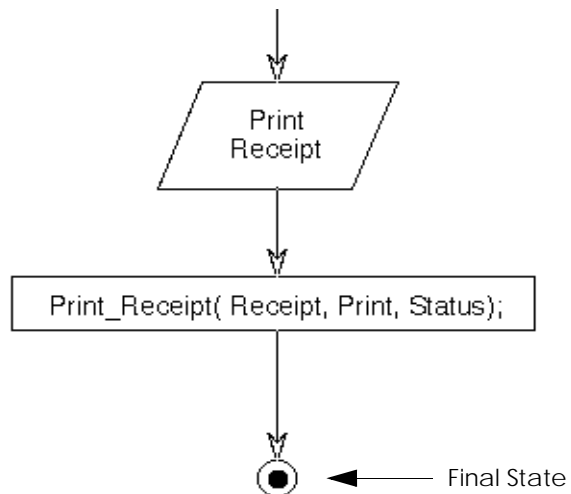


Figure 42 illustrates a function with no return statement, using a Final State symbol. Final State symbols are not labeled.

Figure 42: Flow Chart with Final State Symbol



Calls to the exit() Library Function

Program-terminating function calls are functions that ultimately call the *exit()*, *_exit()*, or *abort()* library functions. These library functions exit the program immediately rather than returning program execution to the caller.

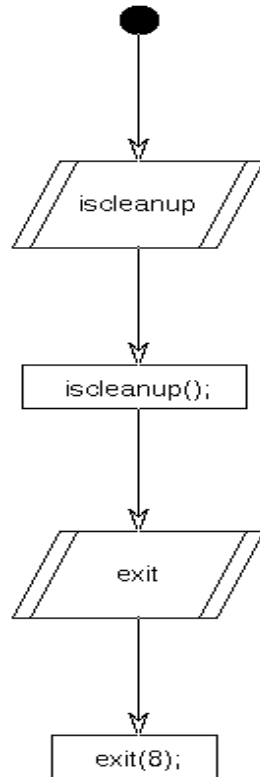
Reverse Engineering draws program-terminating function calls without a link to a Terminal, Void Terminal, or Final State symbol.

For example, the source code for function *tidy_up* is:

```
void tidy_up( n )
{
    int n;
    {
        iscleanup();
        exit(8);
    }
}
```

The final call in this function is to the system library function *exit()*, which forces the program to exit immediately. Reverse Engineering draws this function as shown in Figure 43.

Figure 43: Reverse-engineered Flow Chart for *tidy_up* Function

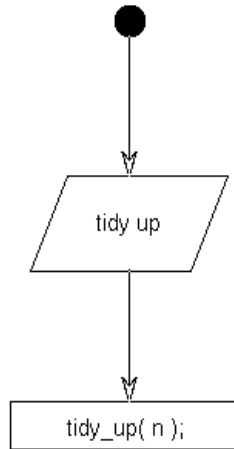


Another example contains a function, *finish()*, that calls *tidy_up*:

```
static void finish( int n )
{
    tidy_up( n );
}
```

Because *finish()* calls a function (*tidy_up*) that terminates program execution, the flow chart for *finish()* is also drawn as a program-terminating function, without a Terminal, Void Terminal, or Final State symbol (see Figure 44).

Figure 44: Reverse-Engineered Flow chart for *finish()* Function



Generated Annotations

During the reverse engineering process, when StP finds the appropriate information, it creates the annotations described in the following tables.

Table 14: Structure Chart Annotations

Object	Note	Item	Value
Module	ModuleDefinition	StorageClass	Function's C storage class
		ModuleReturnType	Function's return type
		VariableArguments	True/False
	CCodeBody	Description	Function's source code
	ModuleComment	Description	Associated source code comments
Parameter	ParameterDefinition	ArraySize	Array information
		DataType	Parameter's type declaration
Global data node	GlobalDefinition	ArraySize	Array declaration
		DataType	Global's type declaration
		StorageClass	Global's storage class
	GlobalComment	Description	Associated source code comments

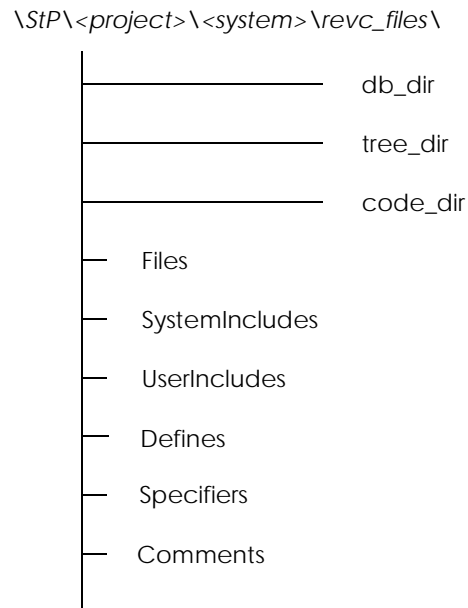
Table 15: Data Structure Annotations

Object	Note	Item	Value
Sequence, Selection, Enumeration	DataDefinition	SystemType	True/False
		ArraySize	Array declaration
		DataType	Data element's data type
Typedef	TypedefDefinition	ArraySize	Array declaration
		DataType	Typedef's data type
All	DataStructureComment	Description	Associated source code comments

Reverse Engineering Directory Structure

The `\revc_files` directory for the current project and system contains the reverse engineering files, as shown in Figure 45. Diagrams and annotations created by StP/SE are stored in the repository.

Figure 45: Reverse Engineering Directory Contents



Db_dir Directory

The *db_dir* directory holds the database files for the current project and system. In exceptional circumstances, you may need to enter this directory to clear database locks. To remove database locks, type `rm *.lock` at the command prompt in the *db_dir* directory. Lock files are text files and can be viewed; they show the owner and mode of each lock held on the file.

Tree_dir Directory

The *tree_dir* directory holds ascii files that correspond to a syntactic and semantic parse tree for every function defined in the source code. You should never need to access this directory. The information held in these

trees is used to perform flow analysis during generation of structure charts. It is used predominantly by the C Code Browser and the flow chart generation program.

Code_dir Directory

The *code_dir* directory holds temporary copies of each source file as it is being preprocessed and parsed by Reverse Engineering. If a source file fails parsing, a copy of the fully expanded file can be found in this directory, which may help to determine why the file did not parse.

Files File

The *Files* file contains the name of each C source file that is part of this system.

SystemIncludes File

The *SystemIncludes* file holds the paths for system-included files. These are files that are included with the `#include <...>` notation. Paths can be either absolute or relative to the current project and system. The default path `\usr\include` is added automatically to the end of this list.

UserIncludes File

The *UserIncludes* file holds the paths for user-included files. These are files that are included with the `#include "..."` notation. Paths can be either absolute or relative.

Defines File

The *Defines* file contains each of the defines entered in the **Defines** field in the **Parse C Code** dialog box. The spaces separating the defines in the dialog box are converted to newline characters in the *Defines* file.

Specifiers File

The *Specifiers* file contains each of the specifiers entered in the **Specifiers** field in the **Parse C Code** dialog box. The spaces separating the specifiers in the dialog box are converted to newline characters in the *Specifiers* file.

Comment_template File

The *comment_template* file holds an encoded description of the comment template for this system. It is read and written by the comment gathering process and is not meant to be accessed manually.

12 Browsing Models and Code

StP/SE provides various facilities for searching the repository or associated source code for specific StP objects or C constructs:

Topics covered in this chapter are as follows:

- “Browsing SE Models” on page 12-2
- “Browsing C Code” on page 12-13
- “Navigating from Model to Source Code” on page 12-38

The SE Browser searches the repository for specified SE objects. The C Code Browser searches for specified C constructs in a semantic model created by the Reverse Engineering tool. Navigation commands in the Structure Chart Editor and Data Structure Editor allow you to navigate from an SE model to associated source code for viewing or editing.

StP also provides the standard OMS Repository Browser, which allows you to query the repository in terms of Persistent Data Model types. For more information about the OMS Repository Browser, see *Fundamentals of StP*.

Browsing SE Models

The SE Browser searches the repository for SE objects that match user-specified search parameters. It then lists the matching objects and allows you to view their definitions and other related objects. The SE Browser shares many similar features with the OMS Repository Browser, but differs in its focus on structured model-specific constructs, rather than StP Persistent Data Model types.

Using the SE browser, you can construct queries based on either StP/SE constructs or the Object Management System (OMS) query language. Knowledge of the Object Management System query language is not necessary for using the SE Browser, but it can help increase the precision of queries. For details about the OMS query language, see *Object Management System*.

Query results appear in a table where you can perform a variety of functions with them, such as sorting, printing, and saving to a file. You can also use the query results as a point of departure for navigating to StP editors or browsing the repository.

SE Browser Overview

The SE Browser shares features common to all StP table editors. For information about using the table editors, see *Fundamentals of StP*. Commands that are specific to the SE Browser are covered in “SE Browser Menus” on page 12-5.

The SE Browser table is divided into sections corresponding to StP/SE constructs, with one section per construct. Figure 1 shows an empty SE Browser table with section headings.

Figure 1: SE Browser Table with Section Headings

	2	3	4	5
1	Diagram	Type		
2	*	*		
3				
4	Module	Parent Modules		
5	*	*		
6				
7	Parameter	Type	Kind	
8	*	*	*	
9				
10	Process	Index	Parent Proc	Children Processes
11	*	*		
12				
13	Flow	Type	Kind	
14	*	*	*	
15				
16	Abstract Data Type			
17	*			

SE Browser Sections

Table 1 lists and describes horizontal sections of the Browser. For descriptions of section rows, see “Section Row Descriptions” on page 12-4.

Table 1: StP/SE Browser Sections

Section	Description
Diagram, Type	Accepts queries on data flow and structure chart diagrams

Table 1: StP/SE Browser Sections (Continued)

Section	Description
Module, Parent Modules	Accepts queries on program modules in structure chart diagrams
Parameter, Type, Kind	Accepts queries on parameters in structure chart diagrams
Process, Index, Parent Process	Accepts queries on processes in data flow diagrams
Flow, Type, Kind	Accepts queries on data flows and control flows in data flow/control flow diagrams
Abstract Data Type	Accepts queries on data structure type definitions that have children

Section Row Descriptions

Each object model construct in the SE Browser table is described in a group of three rows per section:

- A header row—Read-only column headers
- A Query-By-Example (QBE) row—Rows that accept queries based on object model constructs
- One or more body rows—Displays results of queries

Figure 2: StP SE Browser Object Description Group

Header Row	Parameter	Type	Kind
QBE Row	*	*	*
Body Row			

Object Ids

Each object in the repository has a unique object id that is assigned when the object is created. Object ids appear in a vertical section to the left of the initially visible columns, and is hidden by default. For more

information about object ids, see *Object Management System*. For instructions on hiding or showing browser sections, see “Hiding and Showing Table Sections” on page 12-5.

Hiding and Showing Table Sections

You can hide from view or show each of its sections. To hide or show a section, from the **View** menu, point to **Hide/Show Groups** and choose one of the **Hide** or **Show** commands from the **Hide/Show Groups** menu.

Starting the SE Browser

To start the SE Browser from the StP Desktop, choose **Browse Structured Models** from the **Tools** menu.

Initially, the SE Browser window contains only section headings, as shown in Figure 1 on page 12-3. You must query to populate the table with data. Query Execution is discussed in “Executing OMS Queries” on page 12-6.

SE Browser Menus

The SE Browser provides the standard table editor menus (**File**, **Edit View**, **Table**, **Tools**, and **Help**). For descriptions of these menus, see *Fundamentals of StP*.

Additionally, these menus provide commands specific to querying and browsing the repository for object-model constructs:

- **GoTo**
- **SE**
- **Sort**
- **Browse**
- **Query**

The commands that appear on these menus are context sensitive. They depend on the part of the browser selected. Table 2 describes the types of commands available from these menus.

Table 2: SE Browser Menus

Menu	Description
GoTo	Provides navigation options for displaying an StP/SE diagram or table from the query results in the selected body row. See “Navigating from Query Results” on page 12-10.
SE	Provides options for manipulating the contents of the SE Browser. See “Using the SE Menu” on page 12-12.
Sort	Provides options for sorting the query results in the section containing the selected body row(s).
Browse	Finds repository objects related to the selected object and displays the results. See “Browsing from Query Results to Related Objects” on page 12-10.
Query	Provides options for querying the repository based on the information in the selected Query-By-Example (QBE) row. Each menu command corresponds to a different query. See “Executing OMS Queries” on page 12-6.

Executing OMS Queries

There are three ways to execute OMS queries with the SE Browser:

- Using a Query-By-Example row in the table
- Editing a Query-By-Example as an OMS query
- Using Object Management System query language

The OMS query language queries the repository in terms of the Persistent Data Model (PDM). The PDM is the conceptual scheme that defines the data in the repository and its interrelationships. Each StP/SE construct maps to a PDM type. For a description of the OMS query language and the PDM, see *Object Management System*. For specific StP/SE-to-repository mappings, see Appendix A, “Application Types and PDM Types.”

The number of objects your query returns appears in the Browser Message Area. If your query contains a syntactical error, the error appears in the Message Log. The contents of the Browser remain unchanged.

Using Query-By-Example Rows

Query-By-Example (QBE) rows correspond to body rows, but can accept regular expressions or literal values. This enables you to use pattern matching for querying the repository.

To execute a query using a QBE row:

1. Select a QBE row or cell.
2. Type an object name or a regular expression.

You can type QBE entries in multiple columns in a row.

Figure 3: Query-By-Example Entry

Diagram	Type
*	*

3. From the **Query** menu, choose **Execute QBE**.

The query appears in OMS query language in the Browser message area. The results appear in the body rows.

Figure 4: Query-By-Example Results

Diagram	Type
*	*
top	DfeDiagram
1.1	DfeDiagram
3	DfeDiagram
1	DfeDiagram
0	DfeDiagram
jm	SoeDiagram
jm2	SoeDiagram

Using Cut and Paste

You can construct a query by cutting text from a body row and pasting it in a QBE row.

Editing a Query-By-Example as an OMS Query

In some instances, a query based on pattern matching may not contain enough detail to return the desired results from the repository. The Browser enables you to add detail to a restrictor clause in an OMS query to get the desired result.

To display a QBE in OMS query language for editing or execution:

1. Select a QBE row or cell in the SE Browser.
2. Type an object name or a regular expression.

Figure 5: Entering a Query-By-Example for Editing



3. From the **Query** menu, choose **Edit QBE as OMS Query**.
The QBE query appears in OMS query language in the **Execute Query** dialog box.
4. Optionally edit the query and select desired options, as described in “The Execute Query Dialog Box” on page 12-9.
5. Click **OK** or **Apply**.

Using Object Management System Query Language

If you are familiar with StP/SE to PDM mappings and the OMS query language, you can construct and execute OMS queries.

1. From the **Query** menu, choose **Execute OMS Query**.
2. In the **Query** field on the **Execute Query** dialog, type the OMS query.
3. Select desired options, as described in “The Execute Query Dialog Box” on page 12-9.
4. Click **OK** or **Apply**.

The Execute Query Dialog Box

When you use the OMS query language or edit a regular expression as an OMS query, the **Execute Query** dialog box appears.

Figure 6: Execute Query Dialog Box

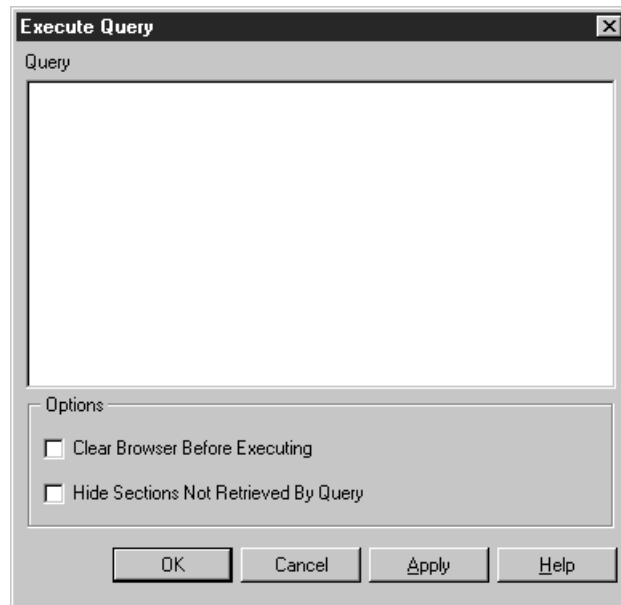


Table 3 lists the parts of the **Execute Query** dialog box.

Table 3: Execute Query Dialog Box Summary

Element	Description
Query text box	Allows you to enter or edit an OMS query.
Clear Browser Before Executing	If selected, clears all other query results from the table before loading the results of the current query. If not selected, the results of the current query appear along with the data already in the table, appended to the appropriate section.

Table 3: Execute Query Dialog Box Summary (Continued)

Element	Description
Hide Sections Not Retrieved By Query	If selected, hides all sections in the report that are not retrieved by the query. If not selected, all sections are displayed.

Navigating from Query Results

You can navigate from an SE Browser query result to these editors:

- Data Flow Editor
- Structure Chart Editor
- Data Structure Editor

The **GoTo** menu provides navigation choices specific to the selected row.

To navigate:

1. Select a body row that contains query results by clicking its row number.
2. Choose a command from the **GoTo** menu.
3. If multiple navigation targets exist, select one from the list on the **Object Selector** dialog box and click **OK** or **Apply**.

The appropriate editor appears with the selected construct.

Browsing from Query Results to Related Objects

Once you have executed queries and populated the table with objects, the **Browse** menu commands allow you to explore the relationships between those objects. Browsing is a hypertext-like method for navigating from one or more selected objects to related objects in the repository.

For example, you might select a data flow diagram, then browse to that diagram's processes. From the processes, you could browse to the process's parent or children.

Browse targets are available from the **Browse** menu. Browse targets vary according to the selected StP/SE construct. Table 4 describes the browse targets for each construct.

Table 4: Browse Targets

StP/SE Construct	Browse Target
Diagrams	SE Diagram's Modules
	SE Diagram's Processes
	SE Diagram's Entire Contents
Module	Module's Callers
	Module's Callees
	Module's Formal Parameters
	Module's Actual Parameters
Parameters	Parameter's Diagrams
Process	Process's Parents
	Process's Children
Flow	Flow's Diagrams
Abstract Data Type	Abstract Data Type's Children
	Abstract Data Type's Parent
	Parameters of Abstract Data Type

To browse to a target:

1. Select the row(s) containing the browsing source by clicking the row number(s), using Shift-click for multiple rows.
2. Choose a command from the **Browse** menu.

The **Browse** command's results appear in the appropriate body rows. For example, if you selected diagram 1 and chose **SE Diagram's Processes** from the **Browse** menu, the results would appear in the **Process** section of the **Browse** window, as shown in Figure 7.

Figure 7: Browse Command Results

Process	Index	Parent Proc
*	*	
Perform ATM Transactio	0	

Using the SE Menu

The **SE** menu contains commands that allow you to manipulate the contents in the SE Browser. These commands are described in Table 5.

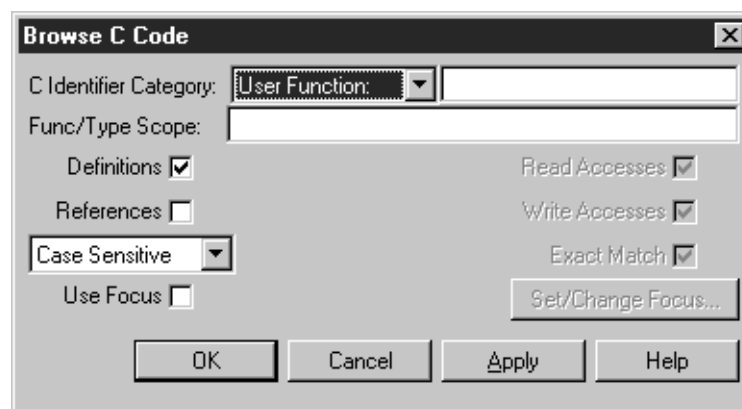
Table 5: SE Menu Command

Command	Description
Reduce to Selected Rows	Clears all unselected rows from the browser. This command does not clear any other sections of the SE Browser.
Fill Browser	Fills the entire browser with the current QBE queries.
Clear Browser	Deletes all but one body row from the browser. This command does not clear header rows or QBE rows.
Recompile Relationships	If you make changes to a process or module hierarchy while you are using the SE Browser, you need to use the this command to make them available to the browser.

Browsing C Code

The C Code Browser provides very detailed static analysis-oriented browsing capabilities of a collection of C source files that have been parsed into the semantic model by the Reverse Engineering tool.

Figure 8: Browse C Code Dialog Box



Using the C Code Browser, you can choose an object category or specific object, specify all the descriptive attributes of the object(s), then search for code references that match your criteria. Matches for the search appear in a selector dialog box, from which the user can browse to the associated source code. In cases where there is a design counterpart, browsing to design is also supported.

Alternatively, you can use the **GoTo** menu commands from within an StP editor to navigate to source code. For more information on navigating to source code from an editor, see "Navigating from Model to Source Code" on page 12-38.

Starting the C Code Browser

To start the **C Code Browser**, from the StP Desktop **Code** menu choose **Reverse Engineering > Browse C Code**.

Using the C Code Browser

The C Code Browser enables you to search source files, user header files, and source header files for various C programming constructs, which you select from an options list of C identifiers:

- User Function
- Global
- Library Function
- Constant
- Typedef
- Literal Value
- Data Structure
- Data Member
- #define
- Comment
- File
- Identifier (for other identifiers or user-specified combinations)

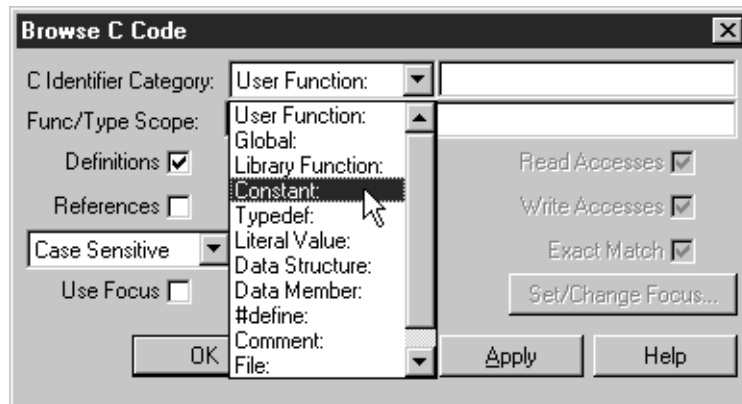
The user's choice implies certain limitations on the code search, and provides some user-selectable options to define the search, as well.

To use the **C Code Browser**:

1. In the C Code Browser, display the options list in the **C Identifier Category** field and select the C construct you want to search for. Other options on the Browser are activated by the construct category you select in this field.

Note: Selecting the **Identifier** category displays the **Identifier Properties** dialog box; for details see "Using the Identifier Properties Dialog Box" on page 12-19).

Figure 9: C Identifier Category Options List



2. In the adjacent text entry field, enter the name of a specific construct or an expression, including optional wild card characters.
3. Select from among the activated options in the **C Code Browser** window (see “Summary of C Code Browser Options” on page 12-16).
4. To optionally limit the search to specified files, select the **Use Focus** option.

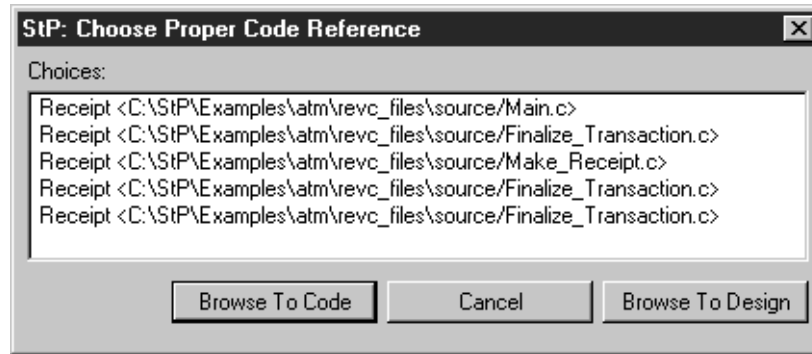
Additionally, to specify a new set of files to search, click **Set/Change Focus**. The **Select Files to Search** dialog box appears. For instructions on using this dialog box, see “Selecting Files” on page 12-22.

5. Click **OK** or **Apply**.

One of the following occurs:

- Code that meets the search conditions appears in the **View Code** window (skip the remaining steps here and refer to “Viewing Source Code” on page 12-24)
- If multiple targets match the search criteria, a list of possible targets appears in the **Choose Proper Code Reference** selection dialog box.

Figure 10: Choose Proper Code Reference Dialog Box



6. Select a code reference from the dialog box.
7. Browse to the code or graphical model by clicking one of these buttons:
 - **Browse to Code**—Displays the related source code in the **View Code** window (see “Viewing Source Code” on page 12-24)
 - **Browse to Design**—Displays the object in your graphical model

Summary of C Code Browser Options

Table 6 lists C Code Browser options.

Table 6: C Code Browser Options Summary

Element	Description
C Identifier Category (options list selections)	User function —A function defined in the source code
	Global —A global variable defined or used in the source code
	Library function —A function not defined in the source code
	Constant —A constant value defined or used in the source code
	Typedef —A typedef defined or used in the source code
	Literal Value —A literal value used somewhere in the source code. For example, “2” would be a literal value for the line of code <code>if (a == 2)</code> .
	Data Structure —A data structure defined or used in the source code
	Data Member —A member of a data structure defined or used in the source code
	#define —A macro defined or used in the source code
	Comment —A comment, or fragment of a comment, appearing somewhere in the source code
	File —The name of a source code file that the semantic model has encountered. Can either be a source file or an include file
	Identifier —Used to specify any identifier defined or used in the source code that is not on the category menu, or to combine identifiers for a single search. See “Using the Identifier Properties Dialog Box” on page 12-19.

Table 6: C Code Browser Options Summary (Continued)

Element	Description
C identifier text field	Accepts a name or regular expression, including optional wild card characters, for the object of the search. See “Using Pattern Matching in the Text Field” on page 12-21.
Func/Type Scope	Allows specification of an identifier’s C scope. This field is automatically filled in when you select an item from the results of a previous query. See “Specifying an Identifier’s C Scope” on page 12-21.
Definitions	When selected, looks for the definition of the specified object. This field does not apply to Comment or Literal Value.
References	When selected, looks for references to the specified object.
Case Sensitive/Insensitive	Performs the search either with or without consideration of case.
Read Accesses	Allows a search for references used in read-only mode (not changed by the code). Does not apply to File, Comment, Literal Value, #define, or function.
Write Accesses	Allows a search for references used in write mode. Does not apply to File, Comment, Literal Value, #define, or function.
Exact Match	When selected for Literal Value or Comment categories, searches for an exact match to the characters in the text field, rather than treating the text entry as a regular expression. This option is ignored for all other category choices.
Use Focus	Allows you to focus a query by limiting it to the current file settings, if any, on the Select Files to Search dialog box. To specify a new set of files, also click Set/Change Focus to display the dialog box. See “Selecting Files” on page 12-22.

Table 6: C Code Browser Options Summary (Continued)

Element	Description
Set/Change Focus button	Displays a dialog box for specifying the particular files to be used for the search. Available only when Use Focus is selected. See “Selecting Files” on page 12-22.

Using the Identifier Properties Dialog Box

Choosing a single identifier from the C identifier options list on the **Browse C Code** dialog automatically limits the scope of a search to certain predefined criteria for each identifier, as shown in Table 7.

Table 7: Predefined Search Criteria for C Identifier Categories

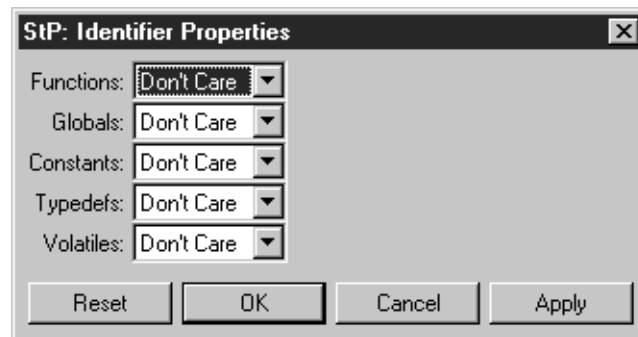
C Identifier Category	Searches For				
	Functions	Globals	Typedefs	Constants	Volatiles
User Function	Yes	Don't Care	No	Don't Care	Don't Care
Global	No	Yes	Don't Care	Don't Care	Don't Care
Library	Yes	Don't Care	No	Don't Care	Don't Care
Constant	No	Don't Care	No	Yes	Don't Care
Typedef	Don't Care	Don't Care	Yes	Don't Care	Don't Care
Identifier	***** (your choice) *****				

Choosing the **Identifier** category from the C identifier options list allows the user to define these aspects of the search on the **Identifier Properties** dialog box (Figure 11).

The **Identifier Properties** dialog box allows you to:

- Combine multiple identifiers, such as globals and functions, in a single search
- Perform a search for any or all identifiers, that is wider in scope and not limited to the predefined categories on the **Browse C Code** dialog box

Figure 11: Identifier Properties Dialog Box



You can tell StP whether or not to search for each type of identifier listed on the dialog box, by choosing one of the following from its options list:

- **Yes**
- **No**
- **Don't Care**

For example, you could set up a search for local variable declarations by setting **Functions** and **Globals** to **No** and the other categories to **Don't Care**.

Specifying an Identifier's C Scope

You can qualify an identifier by entering its C scope in the **Func/Type Scope** field. If the construct is a data member, the entry in this field is assumed to be the name of a structure. Otherwise, it is assumed to be the name of a function. The field accepts a manually typed entry, or StP can supply the scope based on a result you select from a preliminary search.

For example, suppose you want to find all code references to a data member called *Name*. There may be many data members called *Name*, but the one you're interested in is a customer name. If you know the data structure in which the customer *Name* is defined, you could enter that structure's name (for example, *Customer_Info*) in the **Func/Type Scope** field to limit the search. If you don't know the structure name, you could search for all *Name* definitions and select the one you want from the results. StP automatically fills in the **Func/Type Scope** field for you from the selected result. Now you can perform the search with the References option selected to display all references to the data member *Name* that is defined in the *Customer_Info* structure.

To fill in the **Func/Type Scope** field automatically:

1. Make your selections and entries in the **Browse C Code** dialog box, leaving the **Func/Type Scope** field blank.
2. Click **Apply** to execute the query without closing the dialog box.
If more than one matching object is found, the results appear in the **Choose Proper Code Reference** selection dialog box.
3. Select one of the items in the selection dialog box for a subsequent search.
Its name appears in the **Browse C Code** dialog box with its C scope in the **Func/Type Scope** field.
4. Edit the search criteria as needed—for example, by selecting **References** instead of **Definitions**.
5. Click **OK** or **Apply** to execute the new query.

Using Pattern Matching in the Text Field

Table 8 describes the pattern matching characters you can use in the text field to specify a target for the search.

Table 8: Pattern Matching Characters

Character	Usage
* (asterisk)	Represents a character string of any length.
? (question mark)	Represents any single character.
[] (square brackets)	Enclose a range of characters, any one of which can occupy that text position. For example, [abcd]* finds all names that start with a, b, c or d.
- (hyphen)	Used within square brackets to separate the first and last characters of a range. For example, *[a-d] finds all names that end with a, b, c, or d.
! (exclamation point)	Negates a range of characters enclosed in square brackets. For example, *[!z] finds all names that do not end in z.

Selecting Files

The **Select Files to Search** dialog box appears when you select the **Use Focus** option on the **C Code Browser** dialog box and click the **Set/Change Focus** button.

Figure 12: Select Files to Search Dialog Box

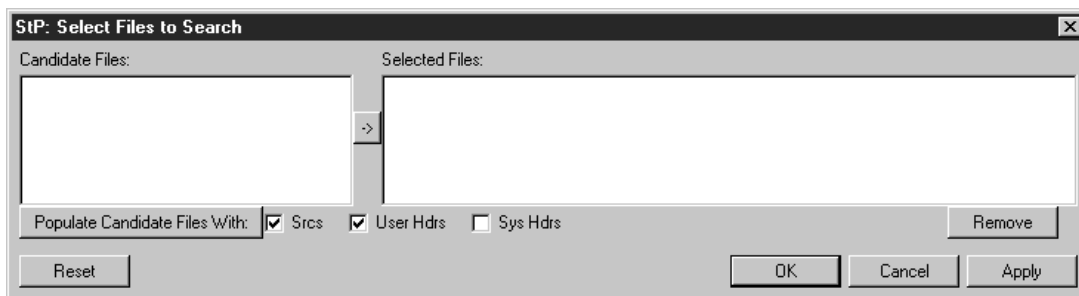


Table 9 lists options for this dialog box.

Table 9: Select Files to Search Dialog Box Summary

Element	Description
Candidate Files	A list of potential files from which you can select, which appears when you click the Populate Candidate Files With button.
Selected Files	The final list of selected files that you want to search.
Populate Candidate Files With button and options	A button and non-exclusive set of choices that determine which files appear in the Candidate Files list. You can select any combination of Srcs (source files), User Hdrs (user header files) and Sys Hdrs (system header files).
Arrow (->) button	A button to copy highlighted files from the candidate files list into the selected files list
Remove Button	A button to remove highlighted files from the selected files list.

The first time you use this dialog box during an StP session, the **Candidate Files** and **Selected Files** lists are empty. Once edited, the dialog box retains its contents for the remainder of the current StP session or until modified. If **Use Focus** is selected on the **Browse C Code** dialog box, make sure the **Selected Files** list has entries; otherwise, no files are searched.

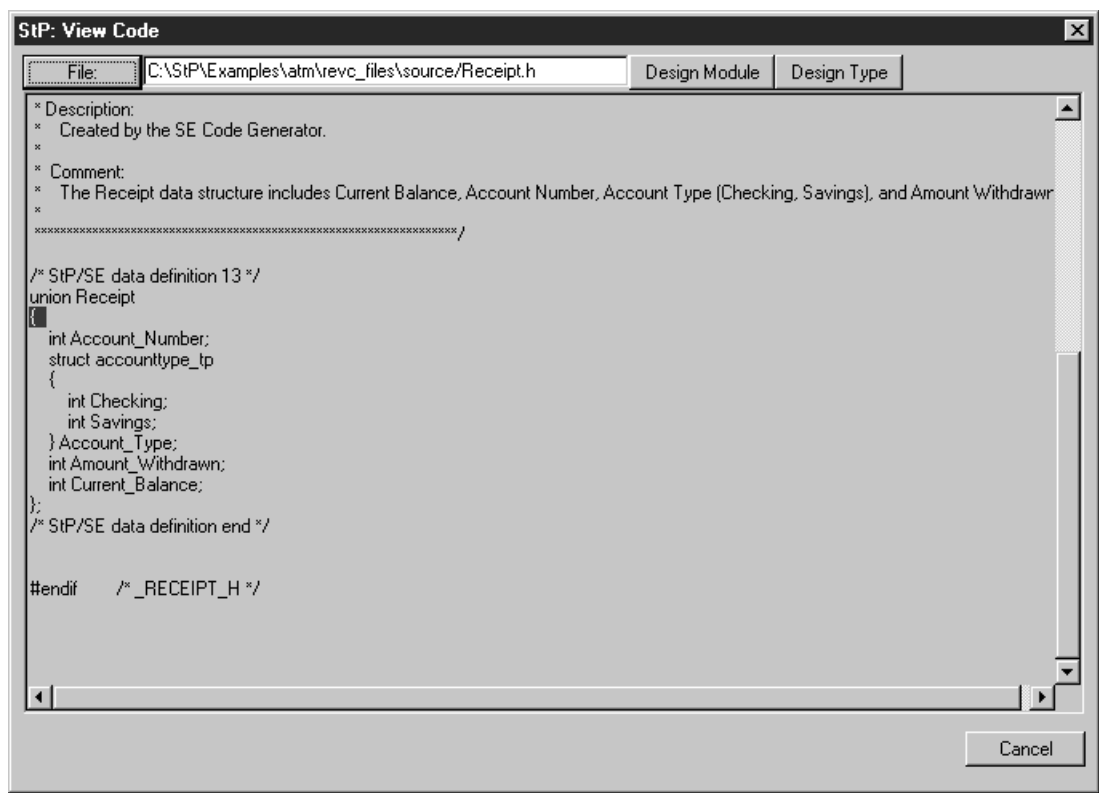
To use this dialog box:

1. Select any combination of the file type options (**Srcs**, **User Hdrs**, **Sys Hdrs**).
2. Click the **Populate Candidate Files With** button.
3. From the **Candidate Files** list, select the files you want to search and click the arrow (->) button on the dialog box to copy the selected files into the **Selected Files** list.
4. Click **OK** to retain your selections and exit this dialog box.
The **Browse C Code** dialog box reappears.

Viewing Source Code

The **View Code** dialog box, shown in Figure 13, is a read-only view window that displays C code associated with a code browser or source code navigation target.

Figure 13: View Code Dialog Box



The following table describes the **View Code** dialog box options.

Table 10: View Code Dialog Box Options

Option	Description
File button and field	Entering a directory and filename and pressing the File button displays the contents of the specified file in the view window.
Design Module button	If you select the name of a function in the displayed code, allows you to navigate to any structure chart in which the selected function is defined or referenced, by choosing from a displayed list.
Design Type button	If you select the name of a data structure element in the displayed code, allows you to navigate to any data structure diagram in which the selected element is defined or referenced, by choosing from a displayed list.

Examples

In this section a series of examples are shown. Each example is introduced with a sample piece of source code. You are shown how to set the various parameters on the **Browse C Code** dialog box to produce the desired effect.

Note: Unless otherwise specified, the **Use Focus** option should be unselected, and the **Selected Files** list in the **Select Files to Search** dialog should be empty.

Finding Writes on a Global Identifier

This example finds writes on a global identifier called “global_count.”

```
int global_count;
```

Figure 14: Finding Writes on a Global Identifier



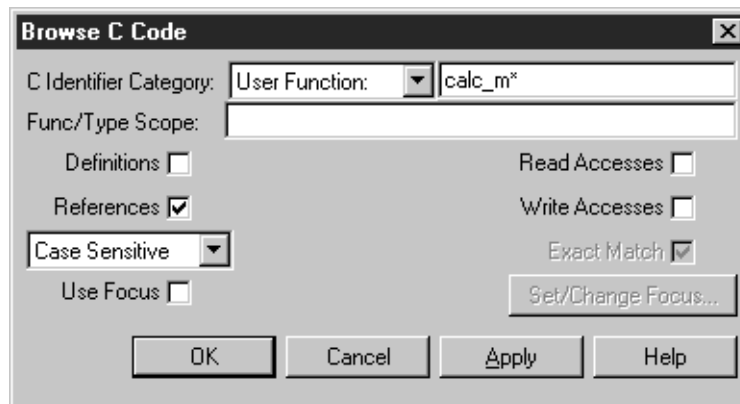
C Identifier Category:	Global
C identifier name:	global_c*
References option:	(selected)
Write Accesses option:	(selected)

Finding Calls Made to a Function

This example finds calls made to a function called “calc_mean.”

```
float calc_mean( int elements, float array[] )
```

Figure 15: Finding Calls Made to a Function

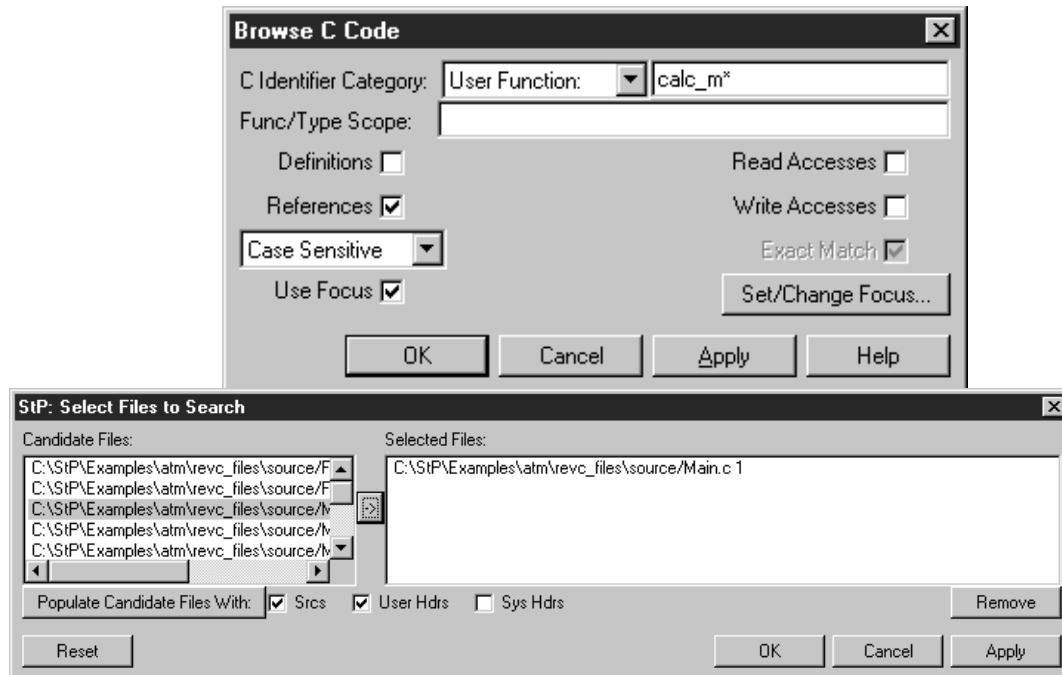


C Identifier Category:	User Function
C identifier name:	calc_m*
References option:	(selected)

Browsing Models and Code

This example finds calls made to a function called “calc_mean” from a source file called *Main.c*.

Figure 16: Finding Calls Made to a Function from a Source File



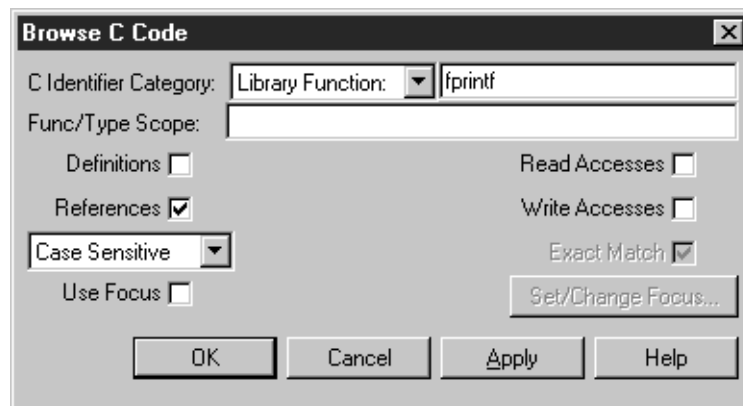
C Identifier Category:	User Function
C identifier name:	calc_m*
References options:	(selected)
Use Focus option:	Set focus to <i>Main.c</i> in the Select Files to Search dialog

Finding Calls Made to a Library

This example finds all calls to a library called “fprintf.”

```
fprintf( stderr, .. );
```

Figure 17: Finding Calls Made to a Library



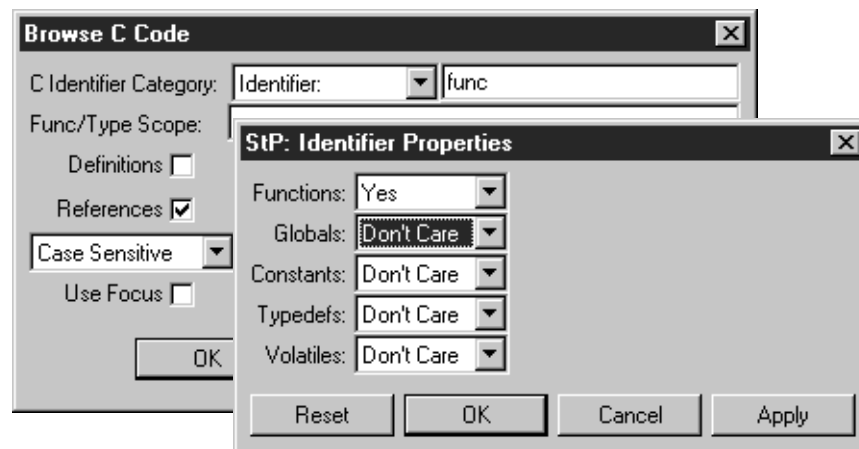
C Identifier Category:	Library Function
C identifier name:	fprintf
References option:	(selected)

Finding Where a Function Has its Address Taken

This example finds where a function called “func” has its address taken.

```
int (*any_function())[] = { func, fun2, fun3, fun4 };
```

Figure 18: Finding Where a Function Has its Address Taken



C Identifier Category:	Identifier
C identifier name:	func
Identifier Properties dialog:	Functions —Yes Others—Don't Care
References option:	(selected)

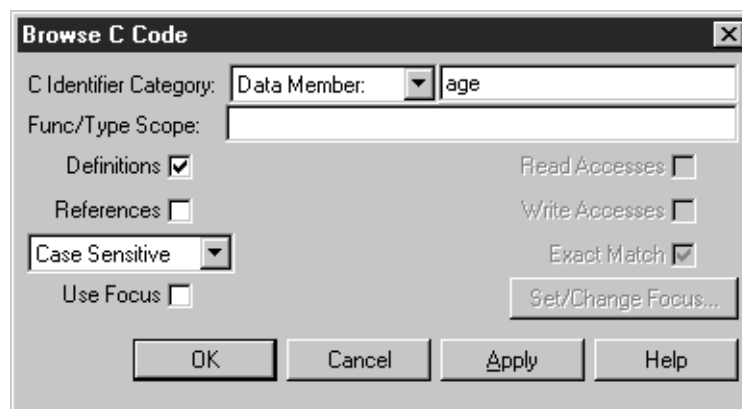
This search finds all normal calls to func. It also finds places where func’s address is read. For example, func’s address is read in the line of source code shown above. The category is set to **Identifier**. This is the correct setting when searching for accesses other than normal calls on either user-defined functions or library functions. However, when searching for conventional calls to user-defined functions, the **Function** category gives the fastest results.

Finding Where a Structure's Field is Defined

This examples finds where a data structure's field called "age" is defined for a data structure called "person."

```
struct person { char name[40]; int age; int code; };
```

Figure 19: Finding Where a Structure's Field is Defined

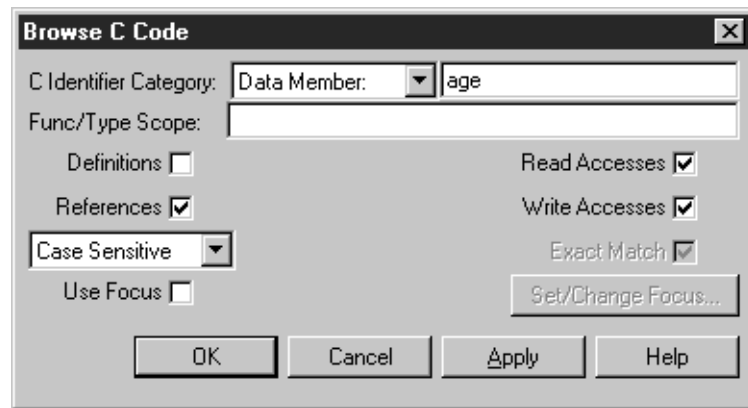


C Identifier Category:	Data Member
C identifier name:	age
Func/Type Scope:	person
Definitions option:	(selected)
References option:	(not selected)

Finding Where a Structure's Field is Used

This example finds where the data member is being written.

Figure 20: Finding Where a Structure's Field is Being Written



C Identifier Category:	Data Member
C identifier name:	age
Definitions option:	(not selected)
References option:	(selected)
Read Accesses option:	(selected)
Write Accesses option:	(selected)

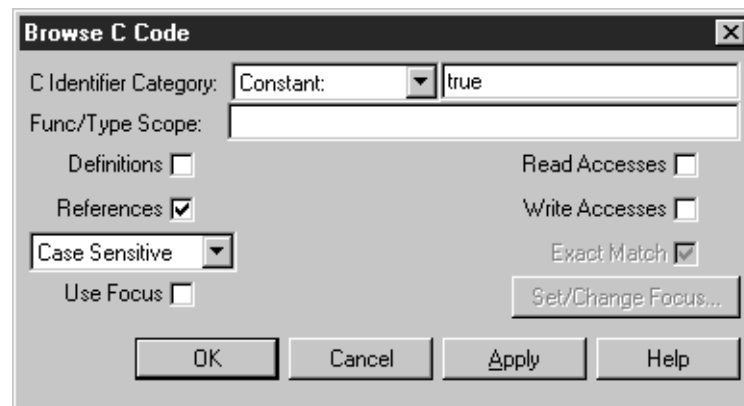
It is slightly faster to search for both read and write accesses than it is to search for accesses of just one type.

Finding Where an Enumeration's Field is Used

This example finds where an enumerations field is used.

```
typedef enum { true = -1, false } boolean;
```

Figure 21: Finding Where an Enumeration's Field is Used



C Identifier Category:	Constant
C identifier name:	true
References option:	(selected)

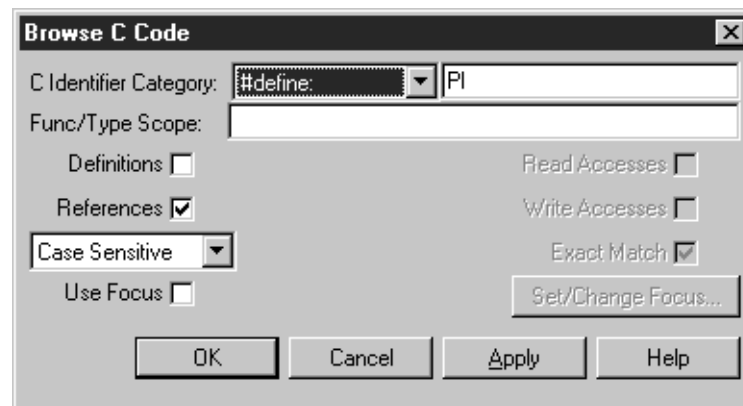
Note: StP Reverse Engineering treats enumerations as a collection of constants, rather than as data members.

Finding Where a #defined Value is Used

This example finds where a #defined value is used:

```
#define PI 3.141597
```

Figure 22: Finding Where a #defined Value is Used



C Identifier Category:	#define
C identifier name:	PI
References option:	(selected)

This finds every instance in which PI is used, except when it is used to declare the size of an array. It is not possible to locate where #define values are used within the bounds of an array definition; for example:

```
#define STRING_SIZE 200  
char name [ STRING_SIZE ];
```

This occurrence of STRING_SIZE is not located by the C Code Browser.

Listing All Statically Scoped Functions

This example lists all statically scoped functions.

Figure 23: Listing All Statically Scoped Functions

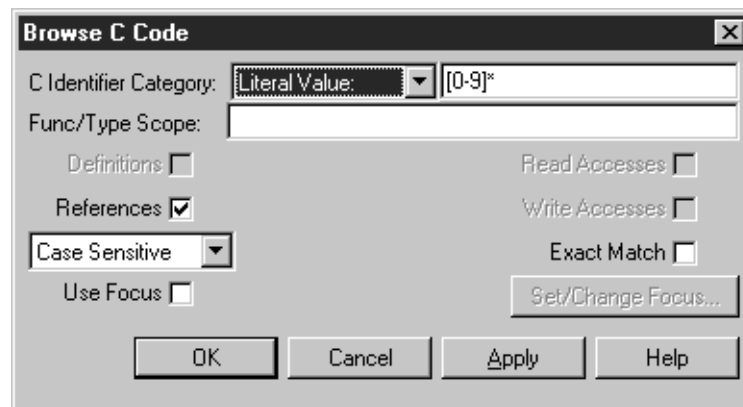


C Identifier Category:	Identifier
C identifier name	*
Identifier Properties dialog	Functions —Yes Globals —No Others—Don't Care
Definitions option:	(selected)
References option:	(not selected)

Finding All Numeric Literals

This example finds all numeric literals.

Figure 24: Finding All Numeric Literals



C Identifier Category: Literal Value

C identifier value: [0-9]*

References option: (selected)

Exact Match option: (not selected)

This finds every instance in which numeric literals are used. It also finds string literals that begin with a digit. The **C Code Browser** does not differentiate the type of the literal.

The **C Code Browser** does not search in array-bound declarations for the specified value.

Read-Only Locking on Semantic Model

While the **C Code Browser** is running, it holds read-only locks on the semantic model. Any number of processes can hold read-only locks on a single semantic model without any possibility of conflict; therefore, any number of users can run the **C Code Browser** at the same time on the same semantic model (provided there are enough licenses).

Reverse Engineering also uses locking on the semantic model to prevent users from simultaneously performing operations on the same data in the database. Neither the Reverse Engineering parser nor its diagram generator can run on the current system when the **C Code Browser** is in use, since these processes require both read and write locks in order to update the database. For more information, see “Semantic Model Locks” on page 11-26.

Interpreting Search Results

Some things to keep in mind when interpreting search results are:

- Searching for both declarations and references for data structures frequently produces the same line number twice, because an item is both declared and referenced at the same place.
- If a match occurs in a statement that has been split over several lines in the source code, the last line of the statement is highlighted, although the target object may not actually reside on that line.
- If you make changes to a file, the line number information held by the semantic model may no longer reflect what is in the file. This makes subsequent searches slightly inaccurate. Re-parsing the system causes the changed files to be reprocessed by Reverse Engineering, which updates the line number information.
- The **C Code Browser** does not search for calls made by a function because these are obvious from the source code and the structure charts.
- Array bounds are not searched. A declaration such as

```
int table[ 12 ], array[ ARRAY_SIZE ];
```

will not be found in a search for the literal ‘12’ or the #defined value of ‘ARRAY_SIZE’.

Troubleshooting

Use caution when running the **C Code Browser** on systems in which source files either failed to parse or were modified since the last parse. Also, use caution when running the **C Code Browser** on a system containing more than one executable. Check the parser output log for messages such as:

```
Parser Warning: duplicate function definitions for  
<funmane>--all future calls will be linked to this. Previous  
definition was in common.c new definition is in foo.c.
```

These represent external functions with multiple definitions, and sometimes Reverse Engineering links these incorrectly. This depends on the order in which the functions are defined and the order in which calls to the various functions are located in the source code. As calls are found, they are linked to and from the most recent definition of the named function.

Navigating from Model to Source Code

In addition to the **C Code Browser**, you can use navigation commands to navigate from a selected object in your model to associated source code. You can navigate from a structure chart, data structure diagram, or flow chart to associated source code displayed in:

- The StP code viewer (default editor, for viewing only)
- A user-specified editor (for viewing and editing)

Source code is associated with structure charts, data structure diagrams, and flow charts through StP's code generation or reverse engineering facilities. Some navigations to source code are available only if the system contains a semantic model created by StP's Reverse Engineering tool. If there is no semantic model, StP assumes the source code was created with StP's code generation facility. If the code was neither created by Reverse Engineering nor generated by StP, the navigation may not be accurate.

Setting a Default Editor

When you navigate from StP to source code files, using any of the **Source Code...** navigation commands, the code appears in the StP default source code viewer or editor.

To change the default code viewer/editor, set the *source_editor* ToolInfo variable to any supported viewer or editor.

For Windows NT these are:

- Windows Notepad (default when shipped)
- Microsoft Developer's Studio
- **View Code** dialog

The **View Code** dialog allows you to view, but not edit, the displayed code (for more information, see "Viewing Source Code" on page 12-24).

You cannot navigate back to StP from a source code editor.

To set up the default editor for the source code navigation, enter one of the lines shown in Table 11 in your ToolInfo file. For more information on setting ToolInfo variables, see *StP Administration*.

Table 11: Setting up the Source Code Navigation

To view the code from:	Edit or Add to your ToolInfo file:
Windows Notepad	<code>source_editor=notepad [FILE]</code>
Microsoft Developer's Studio Visual C++ 5.0 or greater	Remove the comment indicator for the entry: <code>source_editor=MSDev</code>
StP View Code dialog	Make sure the <i>source_editor</i> ToolInfo variable is unset.

Enter the lines exactly as shown; do not substitute anything for [FILE] or [LINE]. Use the [LINE] variable only if the editor can move to a particular line of a file upon startup.

Using the Navigation

When you navigate from StP to source code, you use one of the **GoTo** menu commands shown in Table 12.

Table 12: GoTo Commands for Source Code Navigation

Command	Navigate From	Navigate To
Source Code Definition	Data structure diagram (any data object)	Object's definition in the source code
	Structure chart (any program module or global)	
	Flow chart (any user call)	Source code for the selected user call
Source Code References	Structure chart (any module, library module, or global)	Object's references in the source code (if semantic model exists)
Flow Chart's Source Code Definition	Flow chart (entire diagram or any symbol)	Source code definition for the function modeled in this flow chart

Navigating to Source Code When a Semantic Model Exists

When you use one of the navigation commands listed in Table 12, StP searches for a semantic model in the current system. If StP finds a semantic model and the selected object in the semantic model database, the source code appears in the selected source viewer or editor.

To navigate from StP to source code when a semantic model exists:

1. Select an appropriate object in an StP/SE structure chart, data structure diagram, or flow chart.
2. From the **GoTo** menu, choose one of the source code navigation commands listed in Table 12.

The source code appears in the default source code viewer or editor.

Navigating to Source Code When a Semantic Model Does Not Exist

Without a semantic model, you can navigate only to an object's definition in the source code; navigating to object references requires a semantic model.

When you choose the **Source Code Definition** command, StP searches first for a semantic model. If it does not find one in the current system, it searches for the code in the default location for generated code. If it does not find the code in either location, it displays a window in which you can enter the directory, and optionally, the file containing the source code. By default, StP searches for code in a file bearing the name of the selected object. For example, if you selected a module named *Accounts*, StP looks for *Accounts.c* in the default or specified directory. StP saves the specified directory path and filename for future navigations to source code. If it cannot find the code in this location when you next navigate to source code, the pop-up window reappears, allowing you to specify a different directory and/or filename. If the code in the specified location is not associated with your model through reverse engineering or code generation, the navigation may not be accurate.

To navigate from StP to source code when a semantic model does not exist:

1. Select an appropriate object in an StP/SE structure chart, data structure diagram, or flow chart.
2. From the **GoTo** menu, choose one of the source code navigation commands listed in Table 12.

If StP finds the source code file, it navigates to the appropriate line in the code and displays the code in the selected source editor or viewer.

3. If StP cannot find the source code file, type the directory path and, optionally, the filename in the pop-up window that appears.
4. Click **OK**.

If StP finds the source code file, it navigates to the appropriate line in the code and displays the code in the selected source editor or viewer.

13 Renaming Symbols

This chapter describes how to rename a diagram symbol or table cell. Topics covered in this chapter are as follows:

- “What Are the Options for Renaming Symbols?” on page 13-1
- “Retyping a Label” on page 13-4
- “Renaming Objects Systemwide” on page 13-8

What Are the Options for Renaming Symbols?

StP provides two differing methods for renaming a diagram symbol or table cell. You can:

- Change the name of a symbol by retyping its label.
This change affects only the current diagram.
- Change the name of the underlying repository object using the **Rename Object Systemwide** command.
This ensures that the name is changed in all diagrams and tables in which it occurs.

Similarly, for SEFile and SEDirectory scope objects in StP/SE you can:

- Change the name (value) of the SEFile and/or SEDirectory object for certain data structure or structure chart objects, by editing the object's properties.
 - Globally rename the SEFile or SEDirectory object in the repository, using the **Rename Object System Wide** command on the StP Desktop **Tools** menu.
-

Choosing a Renaming Method

Use these guidelines to decide whether to rename a symbol or cell by retyping its label or by using the **Rename Object Systemwide** command.

Retype the label when:

- You have made a typo or otherwise mislabeled a symbol
- No other diagrams or tables in the system contain a symbol of the same name
- You want to create a new symbol with the same annotations and relationships as the current symbol

Use **Rename Object Systemwide** when you want to:

- Preserve information defined for this symbol in other diagrams and tables
- Propagate the name change throughout the system

For more information about the effects of renaming objects, see “How Renaming Affects Objects and Object References,” which follows.

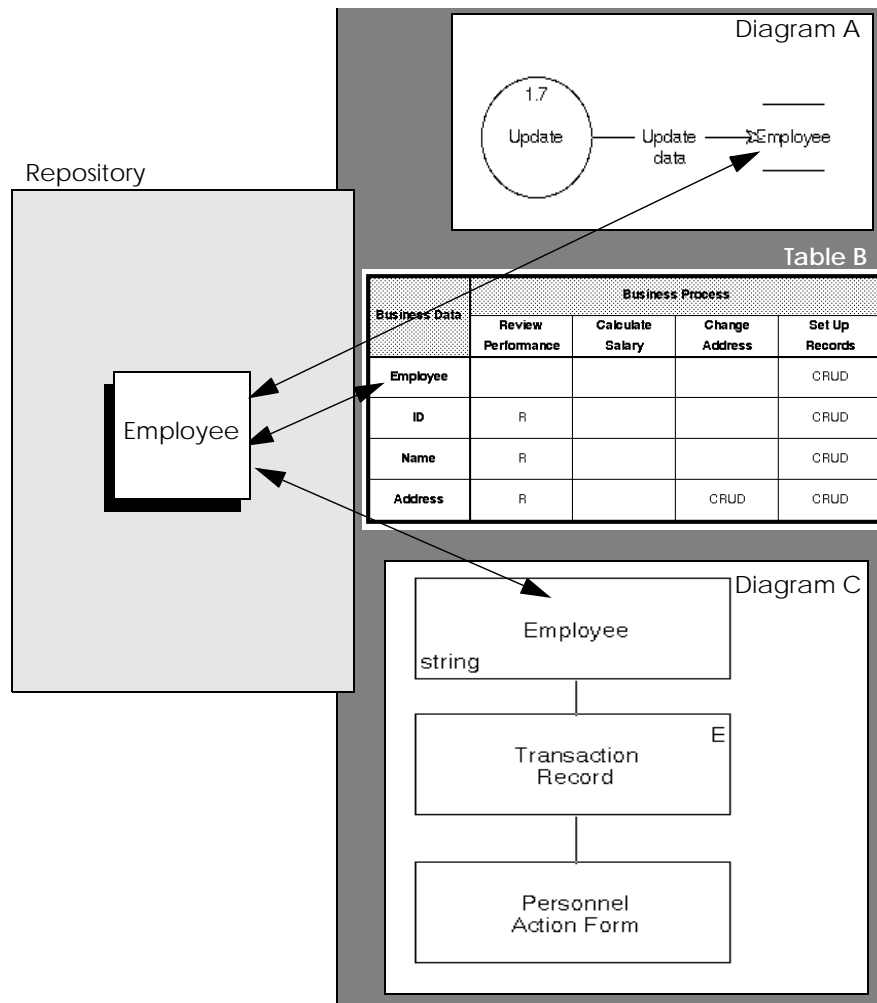
How Renaming Affects Objects and Object References

This section reviews the way that StP stores information in order to clarify the difference between relabeling a symbol and renaming it using the **Rename Object Systemwide** command. For detailed information about the Persistent Data Model, see *Object Management System*.

Each diagram symbol and table cell in the SE model is a symbolic reference to an object in the repository. A single repository object can be represented in any number of diagrams or tables. Each symbolic reference is a particular view of the object and contributes defining information to the object in the form of annotations and relationships to other objects. Any defining information that a particular diagram symbol or table cell contributes to its corresponding repository object automatically applies to all other symbols and table cells that represent the object. Figure 1 shows a repository object (*Employee*) that has references in two diagrams and a table.

What Are the Options for Renaming Symbols?

Figure 1: Repository Object and Symbolic References



The repository object and its symbolic references are associated by their common name. When you rename a reference on a diagram or table, you can choose either to break or to preserve the association to the underlying repository object. Renaming the reference by retyping its label breaks the association, since the name of the reference and the name of the object no longer correspond. Renaming the reference using the **Rename Object**

Systemwide command actually renames the repository object and all of its symbolic references, so the association is preserved.

Retyping a Label

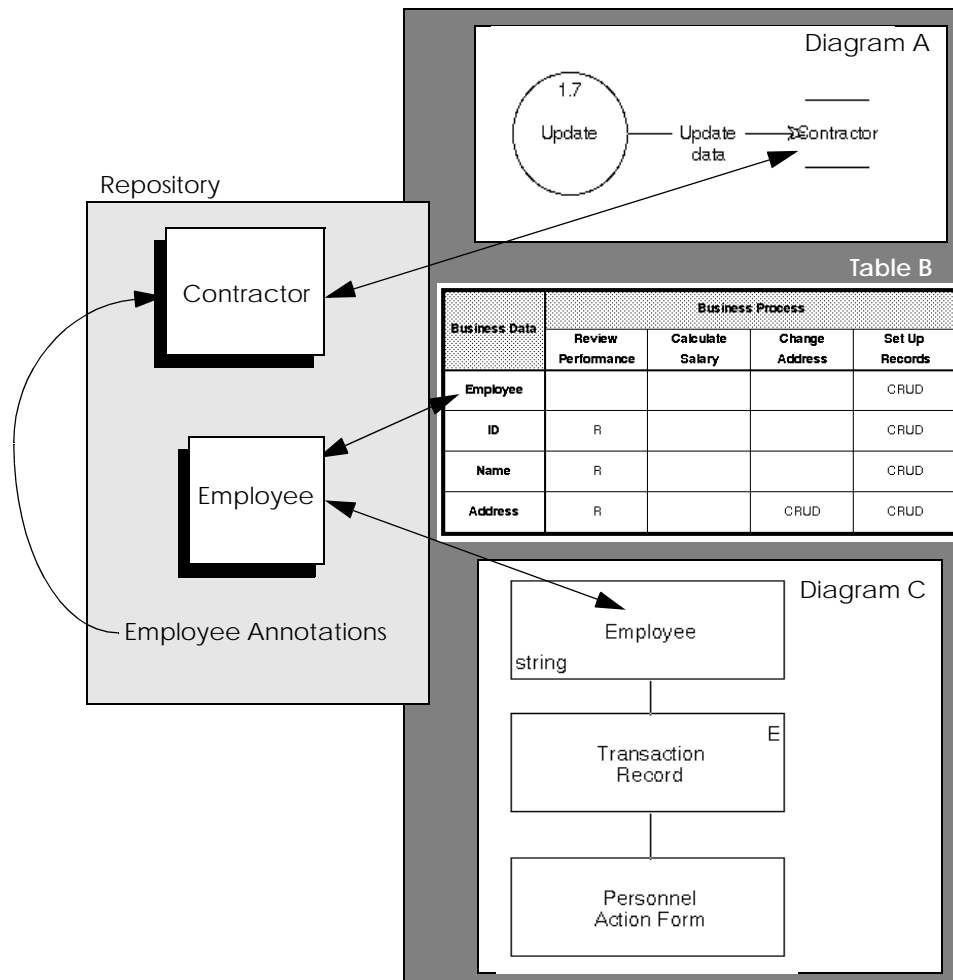
When you retype the label of a symbol, you map the symbol to a different repository object corresponding to the new name. Any associations defined in the current diagram or table are transferred to the new object. Because the scope of retyping a label is limited to the current diagram or table, relationships defined in other diagrams and tables are severed along with the mapping to the original object.

If the new label is not already in use within the system, a new repository object is created, and all of the annotations associated with the old object are copied to the new object. If the new label corresponds to an existing repository object, that object does not inherit annotations unless it has none of its own.

Remapping to an Object without Annotations

Figure 2 illustrates what happens when the new label maps to a newly-created repository object or an existing object that has no annotations. In this example, the *Employee* data store in Diagram A is relabeled *Contractor*.

Figure 2: Remapping to an Object without Annotations

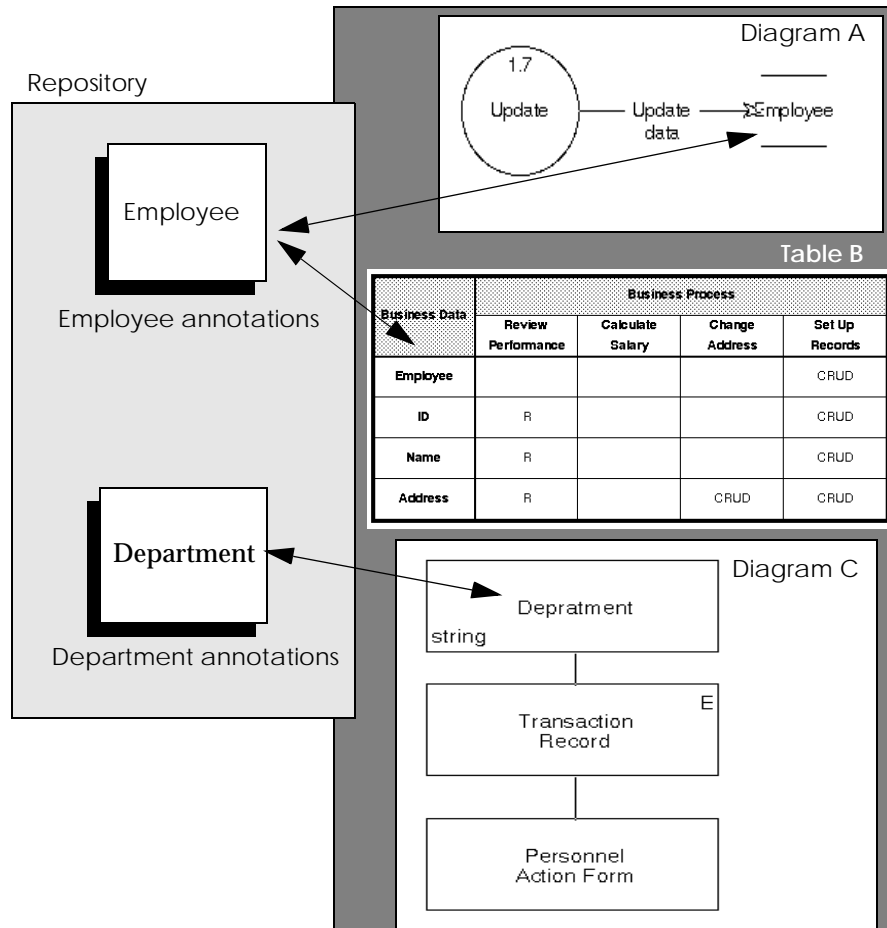


All of the annotations associated with the *Employee* object are copied over to the *Contractor* object. The attributes defined in Diagram A now belong exclusively to the *Contractor* object: the *Employee* object loses the flow linking it to the *Update* process. In turn, The *Contractor* object loses the association with the CRUD table B and the structure that includes the enumeration symbol *Transaction Record* and the sequence *Personnel Action Form* in Diagram C.

Remapping to an Object with Annotations

Figure 3 shows what happens when the new label maps to an existing object that already has annotations. In this example, the *Employee* sequence in Diagram C is relabeled *Department*. The structure defined in Diagram C is transferred to *Department*. *Employee* loses its structure, which includes *Transaction Record* and *Personnel Action Form*. Since *Department* already has annotations, it does not inherit any annotations from *Employee*.

Figure 3: Remapping to an Object with Annotations



Editing the Symbol or Cell Label

To relabel a symbol or table cell:

1. Double-click the symbol or table cell to select its label.
2. Type the new label.
3. Click the left mouse button outside of the symbol or table cell to terminate label editing.
4. Save the diagram or table.

Deleting Unreferenced Objects

When you relabel (and in consequence remap) the sole symbolic reference to a repository object, the object becomes “unreferenced.” It exists in the repository even though it is not depicted anywhere in the information model.

It is a good idea to delete unreferenced objects regularly. To do this, use the **Delete Unreferenced Objects in Current System Repository** command. This command is available from the StP Desktop **Repository > Maintain Systems** menu. For more information, see *StP Administration*.

Cloning Objects by Relabeling Symbols

You can take advantage of symbol relabeling to quickly “clone” similar objects. This avoids the repeated use of the Object Annotation Editor, table editors and dialog boxes to annotate symbols with identical data.

For example, you want to draw a structure chart in which *Manager*, *StaffMember* and *TempEmployee* are all sequences of type string. Rather than recreate this information three times, you can draw and annotate one sequence and use symbol relabeling to create the others.

To do this:

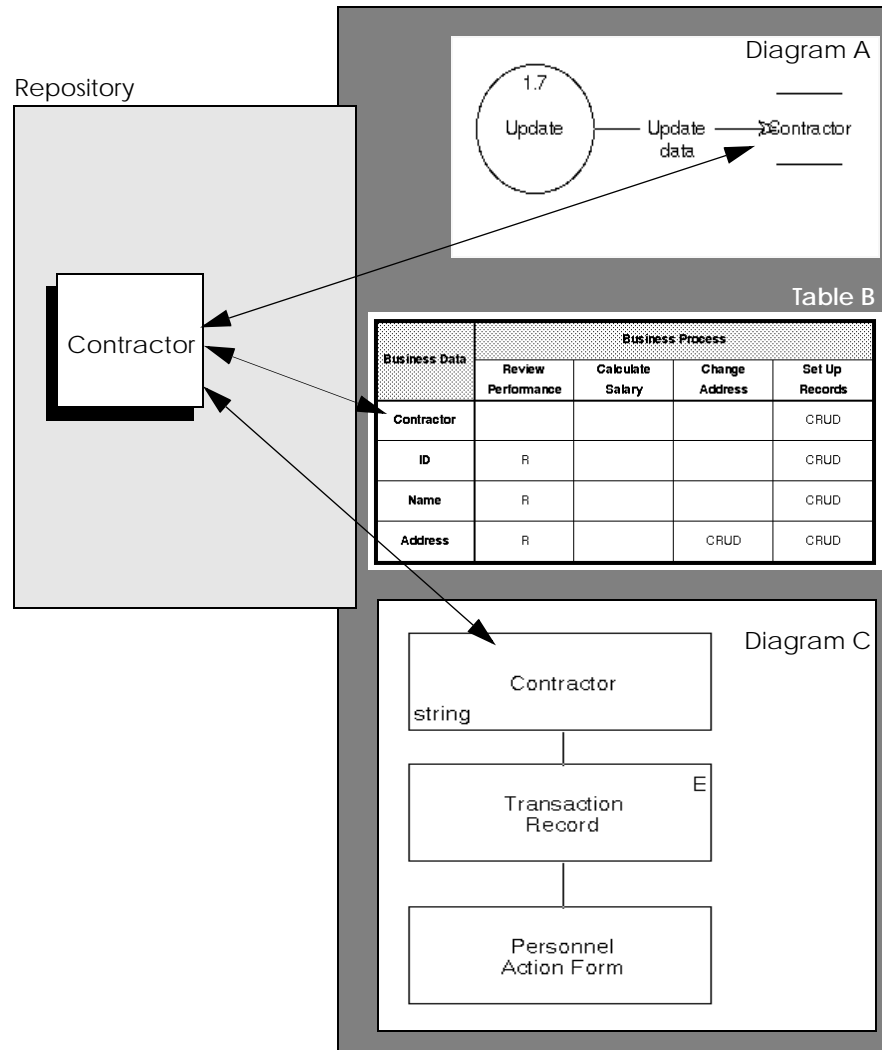
1. Draw, label, and annotate one sequence (for example, *Manager*). You designate a string by adding a type note with a value of string.
2. Save the diagram.
3. Copy the sequence symbol.
4. Relabel the duplicate sequence (for example, to *StaffMember*).
5. Save the diagram.
6. Repeat steps 3 to 5 to create the sequence *TempEmployee*.

Renaming Objects Systemwide

You can propagate a name change throughout the system using the **Rename Object Systemwide** command. When you use this command, the underlying repository object is updated, and the name change is reflected in every diagram and table in which the object is represented.

Figure 4 shows what happens when the *Employee* class in Diagram A is renamed to *Contractor* using the **Rename Object Systemwide** command.

Figure 4: Rename Object Systemwide



Objects That Can Be Renamed

Any element that can be labeled in a diagram or table can be renamed. You can also rename SEDirectory and SEFile scoping objects (see Table 1).

Table 1: StP/SE Rename Candidates

Tool	Object
Data Flow Editor	Data/Control flow Control flow into a Cspec bar Control flow out of a Cspec bar External Process Store
Data Structure Editor	Sequence Selection
Structure Chart Editor	Module
State Transition Editor	Event Action
Control Specification Editor	Control in cell Control out cell
File Objects	SEFile
Directory Objects	SEDirectory

Any element that derives its name from the names of related elements cannot be renamed directly. However, StP automatically adjusts these elements to reflect name changes in the objects on which they depend. For more information, see “Effects on Dependent Objects” on page 13-15.

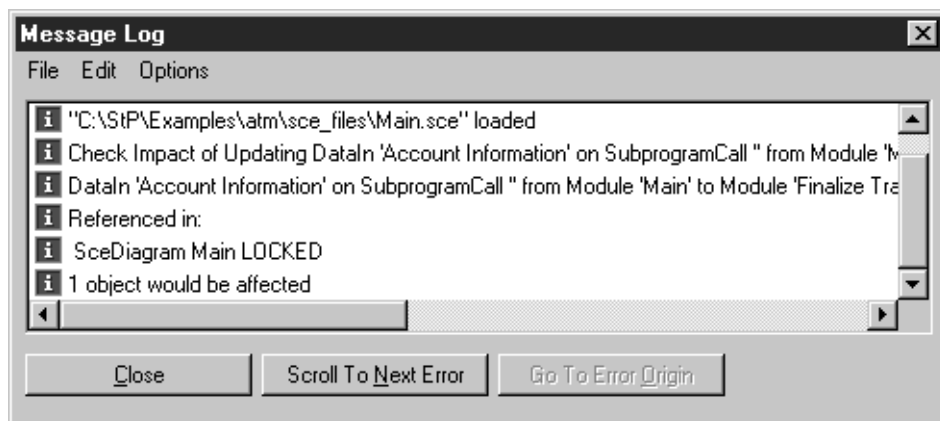
Renaming an Object from a Diagram or Table

To rename an object in the system repository from one of its references on a diagram or table:

1. Select the symbol or table cell.
2. From the **Edit** menu, choose **Rename Object Systemwide**.
3. In the **New Label** field of the **Rename** dialog box, edit the name.
4. Select those options you want to activate (see “Rename Options” on page 13-13).
5. Click **Check Impact**.

The StP Message Log lists the diagrams and tables that will be updated if you execute the rename operation.

Figure 5: Impact Report



If there is a conflict with an existing object or file name, a warning message is displayed.

6. Click **Rename**.

The object and all of its references are renamed. The names and/or signatures of dependent objects and files are adjusted to reflect the change.

The rename operation is confirmed in the Message Area. In addition, a broadcast message is sent out to any user who is editing a diagram or table containing references to the renamed object.

The rename operation is final, even if the diagram or table is not saved. To change back to the object's original name, you must repeat the global rename procedure.

Renaming SEFile and SEDirectory Objects

When you create a root node in a data structure diagram or a program module or global data in a structure chart, StP/SE scopes the newly created object to both an SEFile and an SEDirectory object. These objects determine the output file and directory for any C code for the data structure, program module, or global data object. You can change the file and directory scope values for the data structure or structure chart object by editing its **File** and/or **Directory** properties on the object's property dialog. This does not change the value of the SEFile or SEDirectory scope object itself. Rather, it assigns the data structure or structure chart object an entirely different SEFile or SEDirectory scoping object. This effectively creates a new data structure or structure chart object in the repository, whose scope distinguishes it from any like-named object of the same kind.

In some cases, you may want to change the value of an SEFile or SEDirectory scope object for all instances of its use in the current system. To globally change the value of SEFile or SEDirectory, use the StP Desktop **Rename Object System Wide** command.

Note: Renaming the scope object changes the directory or file specification for all data structure and/or structure chart objects scoped to that SEDirectory or SEFile object throughout the entire system.

To rename an SEDirectory or SEFile Object:

1. From the StP Desktop **Model Elements** category, select **File Objects** or **Directory Objects**.
2. Select an SEFile or SEDirectory object from the objects pane.
3. From the **Tools** menu, choose **Rename Object System Wide**.
4. In the **Rename** dialog box, type the new filename or directory specification.

5. Click **OK** or **Apply**.

The object and all of its references are renamed. The signatures of dependent objects are adjusted to reflect the change.

The rename operation is confirmed in the Message Area. In addition, a broadcast message is sent out to any user who is editing a diagram or table containing references to the renamed object.

The rename operation is final. To change the object's name back to the original scope specification, you must repeat the global rename procedure.

Rename Options

The **Rename Object** dialog box is the graphical interface for the **Rename Object Systemwide** command. The dialog box allows you to optionally rename certain related, but technically non-dependent, objects when you rename the currently selected object. (For automatic updates of dependent objects, see “Effects on Dependent Objects” on page 13-15.)

The basic **Rename** dialog box is shown in Figure 6.

Figure 6: Rename Object Dialog Box



Other versions of the **Rename** dialog box depend upon the type of symbol selected, each offering different options. Table 2 describes the options available from all the **Rename** dialog boxes.

Renaming Symbols

Table 2: Rename Options

Editor or Tool	Selection	Option	Renames
Data Flow Editor	Data Flow, Control Flow, Data Store,	Rename DSE Objects	Sequences and selections with the same name in data structure diagrams
		Rename DSE Diagram	Data structure diagram with the same name
	Control Flow	Rename ControlIn/ ControlOut in Cspec	Cells in Cspec tables that have the same name
	Process	Rename Offpage Processes	Offpage processes with the same name
	External	Rename Offpage Externals	Offpage externals with the same name
Data Structure Editor	Sequence, Selection	Rename Diagram	Data structure diagram with the same name
		Rename Data/Control Flows	Data and control flows with the same name in data flow diagrams
		Rename Data Stores	Stores with the same name in data flow diagrams
		Rename Type Annotations	These annotation items with the same names: Module Return types in structure charts, Parameter types in structure charts, Global Data Types in structure charts, Data Types in data structure diagrams
Structure Chart Editor	Module	Rename Diagram	Diagram with the same name
State Transition Editor	Event/ Action	New Event Label	Other events with the same name (Event/ Action cntx labels are re-evaluated)
		New Action Label	Other actions with the same name (Event/ Action cntx labels are re-evaluated)

Table 2: Rename Options (Continued)

Editor or Tool	Selection	Option	Renames
Control Specification Editor	ControlIn, ControlOut cell	Rename Control Flows	Control flows with the same name in data flow diagrams
		Rename DSE Data Objects	Sequences and selections with the same name in data structure diagrams
File Objects, Directory Objects	SEFile, SEDirectory	None	All instances of SEFile or SEDirectory throughout the entire system

Renaming Errors

Two instances of renaming errors occur when:

- The new name is identical to the name of an existing object that has annotations or references.
- Two users simultaneously rename either the same object or different objects with the same dependent object.

If the new name already exists in the repository, the Message Log displays an error message and the rename operation fails. If you cannot find the object in any diagram or table, it is probably an “unreferenced object” that you can delete. For more information, see “Deleting Unreferenced Objects” on page 13-7.

When two users attempt to rename the same object (or dependent object), the rename operation that is processed first succeeds. Subsequent rename operations can no longer find the target object and therefore fail, as indicated by an error message in the Message Log.

Effects on Dependent Objects

Some types of objects have names or signatures that are derived from the names of related objects. For example, the root node in a decomposition of a higher-level data structure diagram is dependent on the like-named node in the parent diagram.

Renaming Symbols

When you rename an object systemwide, StP:

- Automatically updates the names of any dependent objects
- Gives you the option of also renaming certain related objects that are not technically dependent

Automatic Updates

Table 3 lists dependent objects that derive their name or signature from a related object. StP automatically updates a dependent object when the object on which it depends is renamed.

Table 3: Automatic Updates on Dependent Objects

Editor	Source Symbol	Dependent Objects	What Happens to Dependent Objects upon Rename
Data Flow Editor	Data/Control Flow	Data/Control Flows (regardless of from/to nodes).	flow 1 = foo (from node a to c) flow 2 = foo (from node b to c) If flow 1 is renamed, flow 2 is also renamed. Flows with the same name and scope are renamed, without regard for the from/to nodes.
	Store	Stores	If a data store is renamed, only other data stores with the same name and scope are renamed.
Data Structure Editor	Sequence or Selection	Sequences and Selections	If a data structure object is renamed, only other data structure objects with the same name, type, and scope are renamed.
Structure Chart Editor	Module	Modules	If a structure chart program module is renamed, only other program modules with the same name and scope are renamed.

Table 3: Automatic Updates on Dependent Objects (Continued)

Editor	Source Symbol	Dependent Objects	What Happens to Dependent Objects upon Rename
State Transition Editor	Event	Events	If an event is renamed, other events with the same name are renamed and the Event/Action cntx labels are re-evaluated.
	Action	Actions	If an action is renamed, other actions with the same name are renamed and the Event/Action cntx labels are re-evaluated.

Optional Updates

When you rename certain objects, the **Rename** dialog box allows you to choose whether or not to also rename certain related objects that are not technically dependent; that is, their names are not derived. For example, when you rename a flow or store in a data flow diagram, you can choose to rename the associated data structure diagram, as well. For details, see “Rename Options” on page 13-13.

14 StP/SE Reports

StP/SE provides the capability of generating various SE-specific reports, including an Analysis Review, a Design Review, and a collection of C Code Metrics reports.

Topics covered in this chapter are as follows:

- “Analysis Review Report” on page 14-1
- “Design Review Report” on page 14-12
- “C Metrics Reports” on page 14-22
- “Generating Reports” on page 14-31

The Analysis Review and Design Review reports are generated using Query and Reporting System templates. You can customize these templates using the StP Query and Reporting Language. For details about this, see *Query and Reporting System*.

Analysis Review Report

The Analysis Review Report contains information about data flow and control flow diagrams. The document traverses the data flow diagrams in the analysis model hierarchically. The analysis model must follow the conventions of a *top* diagram, followed by proper decomposition of the model.

Analysis Review Report Example

This section shows an Analysis Review Report that was run on the ATM system and generated in Framemaker format. (For the purposes of this manual, all blank lines have been deleted to conserve space.)

SE Analysis Review Report

Project directory: d:\Proj_dir\

System: atm

Time: 11/09/99 17:12:40

User: young

DfeDiagram: top

04/16/99 13:33:05: FileCreate

04/16/99 13:34:49: FileRepSyncSucceed

Nodes for DfeDiagram: top

Process: Perform ATM Transaction

ProcessIndex:

RelativeIndex: 0

External: Bank

External: Customer Interface

External: Vendor

Links for top: top

DataFlow: Account Information

To Process: Perform ATM Transaction

From External: Bank

DataFlow: Cash

From Process: Perform ATM Transaction

To External: Customer Interface

DataFlow: Electronic Funds

From Process: Perform ATM Transaction

To External: Vendor

DataFlow: Receipt

From Process: Perform ATM Transaction

To External: Customer Interface

DataFlow: Transaction Data

To Process: Perform ATM Transaction

From External: Customer Interface

DataFlow: Transaction Data

From Process: Perform ATM Transaction

To External: Bank

ControlFlow: Access Account

From Process: Perform ATM Transaction

To External: Bank

ControlFlow: Account Accessed

To Process: Perform ATM Transaction

From External: Bank

ControlFlow: Acknowledgment

To Process: Perform ATM Transaction

From External: Vendor

ControlFlow: Eject Card

From Process: Perform ATM Transaction

To External: Customer Interface

ControlFlow: Payment Request

From Process: Perform ATM Transaction

To External: Vendor

ControlFlow: Transaction Request

To Process: Perform ATM Transaction

From External: Customer Interface

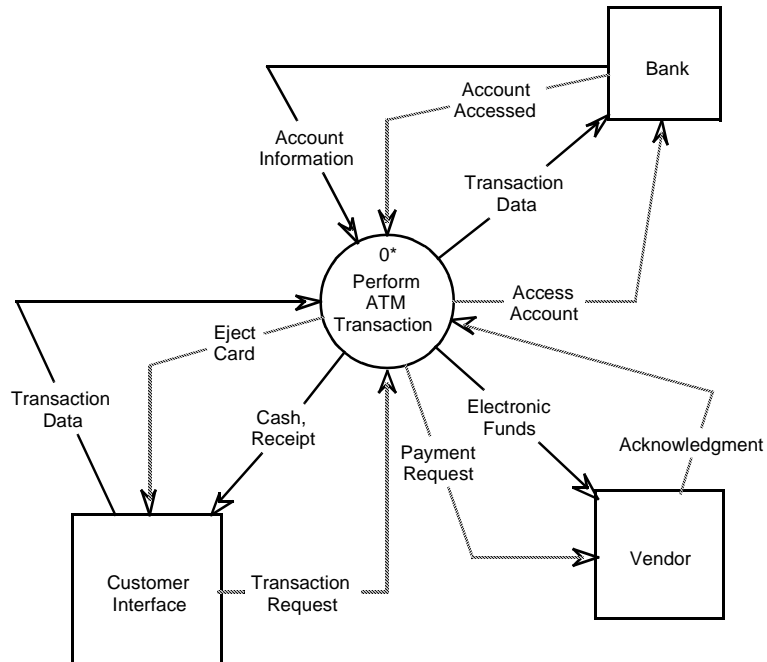


Figure 4: DfeDiagram: top (11/09/99 17:12:40)

DfeDiagram: 0

06/01/99 11:36:55: FileView

06/14/99 18:01:18: FileView

08/06/99 17:30:57: FileView

Nodes for DfeDiagram: 0

Process: Get Account Information

ProcessIndex:

RelativeIndex: 4

Process: Get Cash

ProcessIndex:

RelativeIndex: 1

Process: Make Deposit

ProcessIndex:

RelativeIndex: 3

Process: Pay Bill

ProcessIndex:

RelativeIndex: 2

Store: Account Information

Store: Transaction Data

SplitFlow: (0 400)

OffPageExternal: Bank

OffPageExternal: Customer Interface

OffPageExternal: Vendor

Links for 0: 0

DataFlow: Account Information

To Process: Pay Bill

From Store: Account Information

DataFlow: Account Information

From OffPageExternal: Bank

To Process: Get Account Information

DataFlow: Account Information

To Process: Get Cash

From Store: Account Information

DataFlow: Account Information

From Store: Account Information

To Process: Make Deposit

DataFlow: Account Information

To Store: Account Information

From Process: Get Account Information

DataFlow: Cash

From Process: Get Cash

To OffPageExternal: Customer Interface

DataFlow: Electronic Funds

To OffPageExternal: Vendor

From Process: Pay Bill

DataFlow: Receipt

From Process: Get Cash

To OffPageExternal: Customer Interface

DataFlow: Receipt

From Process: Pay Bill

To OffPageExternal: Customer Interface

DataFlow: Receipt

To OffPageExternal: Customer Interface

From Process: Make Deposit

StP/SE Reports

DataFlow: Transaction Data
 To OffPageExternal: Bank
 From Process: Get Account Information

DataFlow: Transaction Data
 From Process: Pay Bill
 To Store: Transaction Data

DataFlow: Transaction Data
 To Process: Get Cash
 From OffPageExternal: Customer Interface

DataFlow: Transaction Data
 From Process: Get Cash
 To Store: Transaction Data

DataFlow: Transaction Data
 To Process: Get Account Information
 From Store: Transaction Data

DataFlow: Transaction Data
 From Process: Make Deposit
 To Store: Transaction Data

DataFlow: Transaction Data
 To Process: Pay Bill
 From OffPageExternal: Customer Interface

DataFlow: Transaction Data
 From OffPageExternal: Customer Interface
 To Process: Make Deposit

ControlFlow: Access Account
 To OffPageExternal: Bank
 From Process: Get Account Information

ControlFlow: Account Accessed
 From OffPageExternal: Bank
 To Process: Get Account Information

ControlFlow: Acknowledgment
 From OffPageExternal: Vendor
 To Process: Pay Bill

ControlFlow: Eject Card
 From Process: Pay Bill
 To OffPageExternal: Customer Interface

ControlFlow: Eject Card
 To OffPageExternal: Customer Interface
 From Process: Make Deposit

ControlFlow: Eject Card
From Process: Get Cash
To OffPageExternal: Customer Interface

ControlFlow: Payment Request
To OffPageExternal: Vendor
From Process: Pay Bill

ControlFlow: Transaction Request
To Process: Make Deposit
From SplitFlow: (0 400)

ControlFlow: Transaction Request
To Process: Get Cash
From OffPageExternal: Customer Interface

ControlFlow: Transaction Request
From OffPageExternal: Customer Interface
To SplitFlow: (0 400)

ControlFlow: Transaction Request
To Process: Pay Bill
From SplitFlow: (0 400)

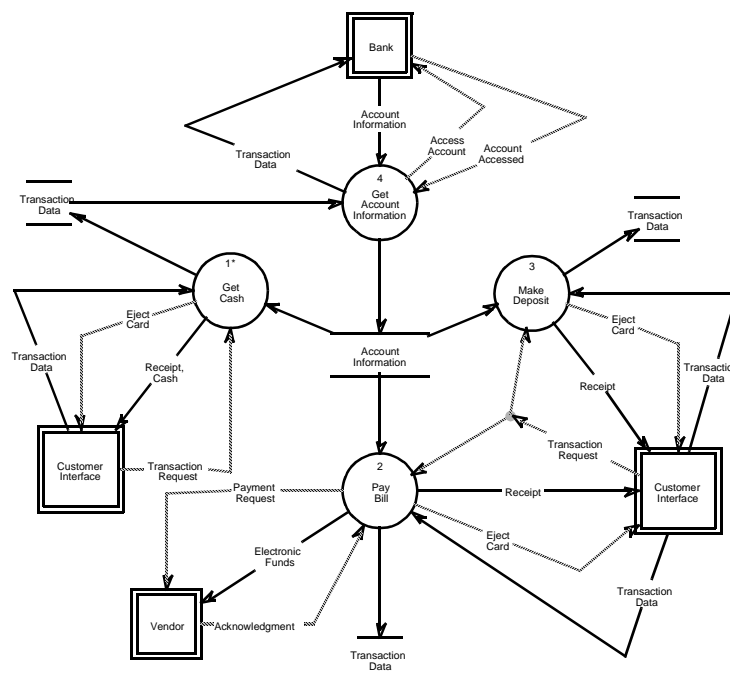


Figure 5: DfDiagram: 0 (11/09/99 17:12:40)

DfeDiagram: 1

06/01/99 11:37:31: FileView

06/14/99 17:48:28: FileView

06/23/99 11:48:33: FileView

Nodes for DfeDiagram: 1

Process: Complete Transaction

ProcessIndex:

RelativeIndex: 3

Process: Dispense Cash

ProcessIndex:

RelativeIndex: 1

Process: Initiate Transaction

ProcessIndex:

RelativeIndex: 4

Pspec:

The Initiate Transaction process validates the customer PIN.

ProcessSpec: True

Process: Specify Transaction Data

ProcessIndex:

RelativeIndex: 2

Cspec: 1

Store: Account Information

Store: Transaction Data

OffPageExternal: Customer Interface

Links for 1: 1

DataFlow: Account Information

To Process: Complete Transaction

From Store: Account Information

DataFlow: Account Information

To Process: Specify Transaction Data

From Store: Account Information

DataFlow: Amount

From Process: Specify Transaction Data

To Process: Dispense Cash

DataFlow: Cash

From Process: Dispense Cash

To OffPageExternal: Customer Interface

DataFlow: Receipt
 From Process: Complete Transaction
 To OffPageExternal: Customer Interface

DataFlow: Transaction Data
 From Process: Specify Transaction Data
 To Store: Transaction Data

DataFlow: Transaction Data
 To Process: Specify Transaction Data
 From OffPageExternal: Customer Interface

DataFlow: Transaction Data
 To Process: Complete Transaction
 From Process: Initiate Transaction

DataFlow: Transaction Data
 To Process: Initiate Transaction
 From OffPageExternal: Customer Interface

DataFlow: Valid PIN
 To Process: Initiate Transaction
 From Store: Account Information

ControlFlow: Cash Approved
 To Process: Dispense Cash
 From Cspec: 1

ControlFlow: Cash Dispensed
 From Process: Dispense Cash
 To Cspec: 1

ControlFlow: Eject Card
 To OffPageExternal: Customer Interface
 From Cspec: 1

ControlFlow: PIN Validated
 From Process: Initiate Transaction
 To Cspec: 1

ControlFlow: Print Receipt
 To Process: Complete Transaction
 From Cspec: 1

ControlFlow: Receipt Printed
 From Process: Complete Transaction
 To Cspec: 1

ControlFlow: Transaction Request
 From OffPageExternal: Customer Interface
 To Cspec: 1

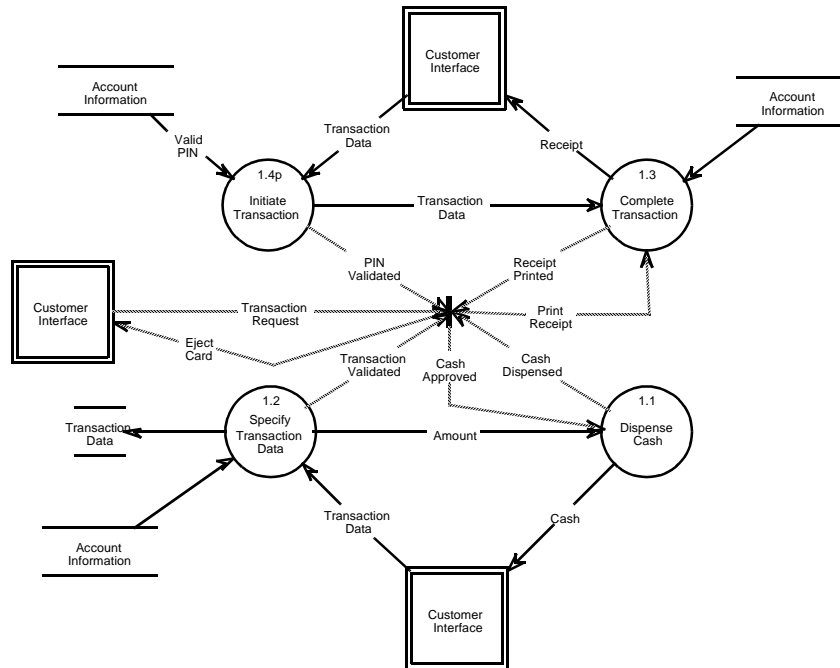


Figure 6: DfeDiagram: 1 (11/09/99 17:12:40)

Figure 7: ActionLogicTable: 1 (11/09/99 17:12:40)

	1	2	3	4	5	6	7	8
1	Action	Activate Processes				Control Out		
2		Complete Transaction	Dispense Cash	Initiate Transaction	Specify Transaction Data	Print Receipt	Cash Approved	Eject Card
3	Activate PIN Validation			1				
4	Activate Print Receipt	1				On		
5	Eject Card							On
6	Get Transaction Data				1		On	
7	Release Funds		1					
8	Request New Data				1		On	

Figure 8: EventLogicTable: 1 (11/09/99 17:12:40)

	1	2	3	4	5	6	7	8	9	10	11	12
1	Control In					Event						
2	PIN Vali date d	Tran sacti on Vali date d	Rece ipt Print ed	Cash Disp ense d	Tran sacti on Req uest	Card ,PIN enter ed	Cash Disp ense d	PIN Not OK	PIN OK	Rece ipt Print ed	Tran sacti on Not OK	Tran sacti on OK
3	1								On			
4	0							On				
5		1										On
6		0									On	
7			1							On		
8				1			On					
9					1	On						

Design Review Report

The Design Review Report is a design and code review document. It does not require the model to have been reverse engineered prior to use. The report provides information based on a hierarchical examination of the structure chart and/or data structure diagrams in a model.

Design Review Report Example

This section shows an excerpt from a Design Review Report that was run on the ATM system and generated in Framemaker format.

SE Design Review Report

Project directory: d:\Proj_dir\

System: atm

Time: 11/09/99 17:16:30

User: young

SceDiagram: Finalize_Transaction

04/16/99 13:33:05: FileCreate

04/16/99 13:35:33: FileRepSyncSucceed

11/09/99 17:11:53: FileView

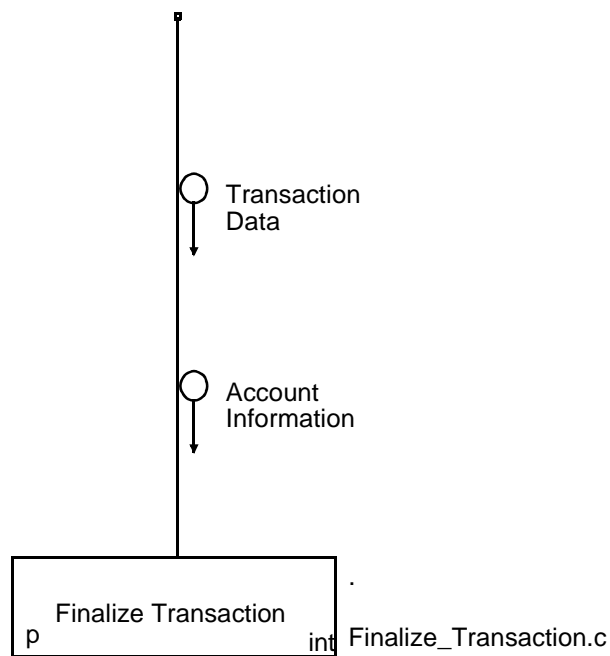


Figure 9: SceDiagram: Finalize_Transaction (11/09/99 17:16:30)

Nodes for SceDiagram: Finalize_Transaction

Module: Finalize Transaction

ModulePDL:

The Finalize Transaction module verifies the transaction and prints a receipt.

PDL: True

ModuleDefinition:

ModuleReturntype: int

StorageClass: static

SEFile: Finalize_Transaction.c

SEDirectory: .

SEFile: Finalize_Transaction.c

SEDirectory: .

Links for Finalize_Transaction: Finalize_Transaction

SubprogramCall:

To Module: Finalize Transaction

SEFile: Finalize_Transaction.c

SEDirectory: .

From FormalCaller:

DataIn: Account Information

DataIn: Transaction Data

SceDiagram: Main

04/16/99 13:33:06: FileCreate

04/16/99 13:35:35: FileRepSyncSucceed

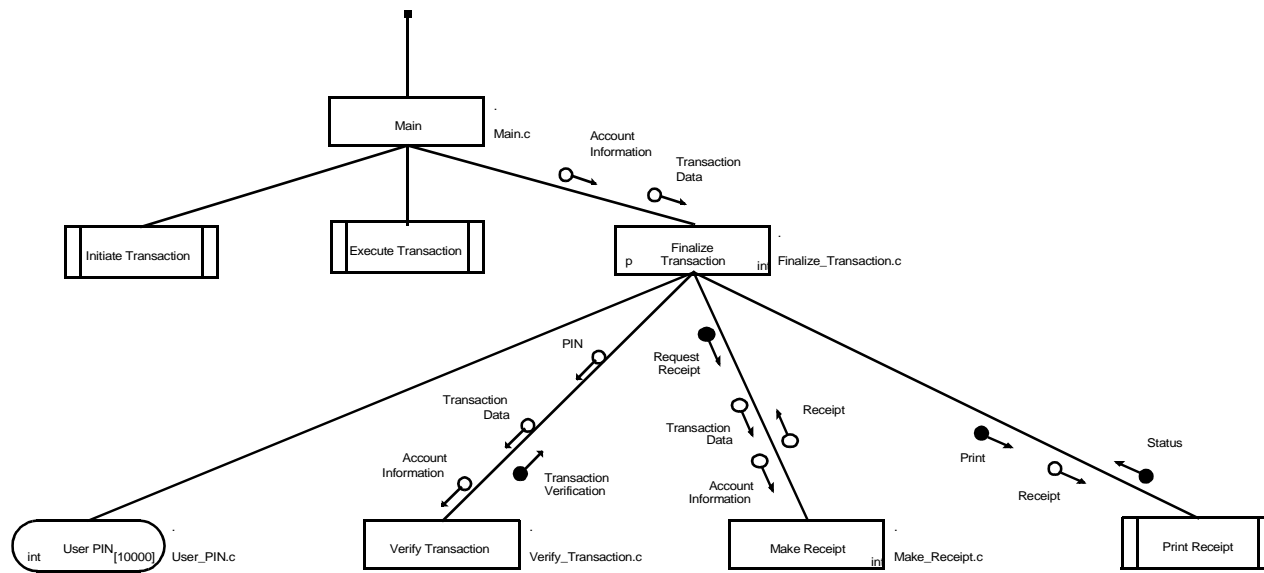


Figure 10: SceDiagram: Main (11/09/99 17:16:30)

Nodes for SceDiagram: Main

Module: Finalize Transaction

ModulePDL:

The Finalize Transaction module verifies the transaction and prints a receipt.

PDL: True

ModuleDefinition:

ModuleReturnType: int

StorageClass: static

SEFile: Finalize_Transaction.c

SEDirectory: .

Module: Main

ModuleDefinition:

SEFile: Main.c

SEDirectory: .

Module: Make Receipt

ModuleDefinition:

ModuleReturnType: int

SEFile: Make_Receipt.c

SEDirectory: .

Module: Verify Transaction

ModuleDefinition:

SEFile: Verify_Transaction.c

SEDirectory: .

DataModule: User PIN

GlobalDefinition:

DataType: int

StorageClass: const

ArraySize: 10000

SEFile: User_PIN.c

SEDirectory: .

LibraryModule: Execute Transaction

LibraryModule: Initiate Transaction

LibraryModule: Print Receipt

SEFile: Finalize_Transaction.c

SEFile: Main.c

SEFile: Make_Receipt.c

SEFile: User_PIN.c

SEFile: Verify_Transaction.c

SEDirectory: .

Links for Main: Main

SubprogramCall:

From Module: Finalize Transaction

SEFile: Finalize_Transaction.c

SEDirectory: .

To LibraryModule: Print Receipt

DataIn: Receipt

ControlIn: Print

ControlOut: Status

SubprogramCall:

To Module: Finalize Transaction

SEFile: Finalize_Transaction.c

SEDirectory: .

From Module: Main

SEFile: Main.c

SEDirectory: .

DataIn: Account Information

DataIn: Transaction Data

SubprogramCall:

From Module: Finalize Transaction

SEFile: Finalize_Transaction.c

SEDirectory: .

To Module: Make Receipt

SEFile: Make_Receipt.c

SEDirectory: .

DataIn: Account Information

DataIn: Transaction Data

DataOut: Receipt

ControlIn: Request Receipt

SubprogramCall:

From Module: Finalize Transaction

SEFile: Finalize_Transaction.c

SEDirectory: .

To Module: Verify Transaction

SEFile: Verify_Transaction.c

SEDirectory: .

DataIn: Account Information

DataIn: PIN

ParameterDefinition:

DataIn: Transaction Data
ControlOut: Transaction Verification
SubprogramCall:
 To DataModule: User PIN
 SEFile: User_PIN.c
 SEDirectory: .
 From Module: Finalize Transaction
 SEFile: Finalize_Transaction.c
 SEDirectory: .

SceDiagram: Make_Receipt

04/16/99 13:35:36: FileRepSyncSucceed
11/09/99 17:11:28: FileView
11/09/99 17:12:17: FileView

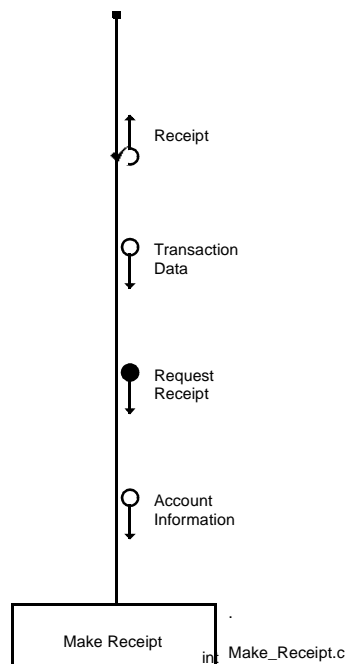


Figure 11: SceDiagram: Make_Receipt (11/09/99 17:16:30)

Nodes for SceDiagram: Make_Receipt

Module: Make Receipt

ModuleDefinition:

ModuleReturnType: int

SEFile: Make_Receipt.c

SEDirectory: .

SEFile: Make_Receipt.c

SEDirectory: .

Links for Make_Receipt: Make_Receipt

SubprogramCall:

To Module: Make Receipt

SEFile: Make_Receipt.c

SEDirectory: .

From FormalCaller:

DataIn: Account Information

DataIn: Transaction Data

DataOut: Receipt

ParameterDefinition:

DataType: Receipt

ParamStorageClass: register

ControlIn: Request Receipt

SceDiagram: Verify_Transaction

04/16/99 13:33:06: FileCreate

04/16/99 13:35:37: FileRepSyncSucceed

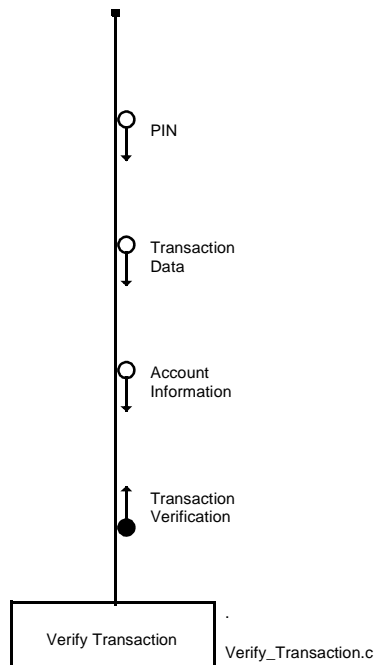


Figure 12: ScDiagram: Verify_Transaction (11/09/99 17:16:30)

Nodes for ScDiagram: Verify_Transaction

Module: Verify Transaction

ModuleDefinition:

SEFile: Verify_Transaction.c

SEDirectory: .

SEFile: Verify_Transaction.c

SEDirectory: .

Links for Verify_Transaction: Verify_Transaction

SubprogramCall:

To Module: Verify Transaction

SEFile: Verify_Transaction.c

SEDirectory: .

From FormalCaller:

DataIn: Account Information

DataIn: PIN

DataIn: Transaction Data

ControlOut: Transaction Verification

DseDiagram: Receipt

04/16/99 13:33:05: FileCreate

04/16/99 13:35:13: FileRepSyncSucceed

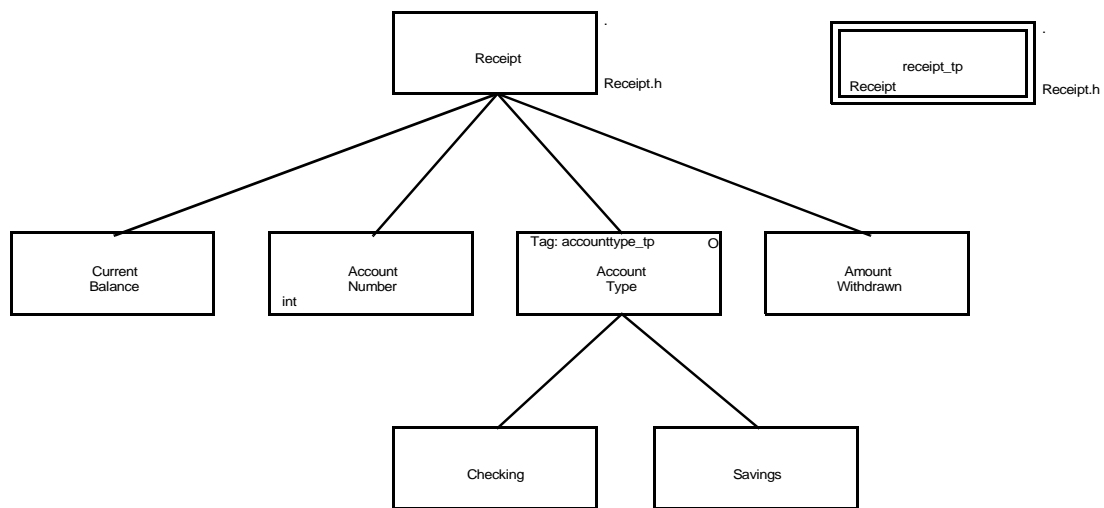


Figure 13: DseDiagram: Receipt (11/09/99 17:16:30)

Nodes for DseDiagram: Receipt

Selection: Account Type

DataDefinition:

Tag: accounttype_tp

SystemType: False

Sequence: Account Number

DataDefinition:

DataType: int

SystemType: False

Sequence: Amount Withdrawn

DataDefinition:

DataType:

SystemType: False

StP/SE Reports

Sequence: Checking

Sequence: Current Balance

Sequence: Receipt

DataDefinition:

SystemType: False

DataStructureComment:

The Receipt data structure includes Current Balance, Account Number, Account Type (Checking, Savings), and Amount Withdrawn.

SEFile: Receipt.h

SEDirectory: .

Sequence: Savings

SEFile: Receipt.h

SEDirectory: .

Typedef: receipt_tp

TypedefDefinition:

DataType: Receipt

SEFile: Receipt.h

SEDirectory: .

C Metrics Reports

StP/SE provides C metrics reports that are generated from an existing reverse engineered C semantic model:

- Top Function Report
- Complex Function Report
- Include Hierarchy Report
- Function Definition Report
- Function Localization Report
- Uncommented Objects Report

These reports enable you to help analyze your source code and the model-generation results.

Note: You must at least have generated structure charts with the Reverse Engineering tool before generating C Metrics Reports.

Top Function Report

This report lists all user-defined functions that are not called by any other functions. Each of these functions appears as the top function in a structure chart diagram. The report lists the top functions in order of decreasing importance to the functionality of the source code. An example of this report is shown in Figure 1.

The report shows the:

- Total number of lines of code for the function and its subordinates
- Total cyclometric complexity of the function and its subordinates
- Total number of function calls
- Depth of the call tree underneath the function

The **Access#** column provides a count of the number of times a function is accessed, other than being called. If a function has never been accessed (indicated by a zero in this column) then it is redundant within this system.

The report provides information about the most useful starting points for examining the hierarchy of functions described by the structure charts. After loading a top function's structure chart in the Structure Chart Editor, you can then navigate to all of that function's subordinates.

Figure 1: Top Functions Report

```

topfuns.rep - Notepad
File Edit Search Help

----- Top Function Report -----
=====

This report lists the potentially most important functions in
the source code. It can also be used to identify redundant
functions.
Each function appearing in this report is never explicitly
called in the source code, so the reason for it appearing here
could be :
a) It is a redundant function
b) It is called indirectly
c) It is a top-level function (e.g. main)
The number of accesses upon each function's address is listed in
the first column. If this is zero, then the function is never
called either indirectly or directly, so it is either a top-level
routine or a redundant routine. If the access number is non-zero,
the function could potentially be called indirectly, and so is
definitely not redundant.
The other columns are accumulated counts for this function and ALL
the functions it calls. Lines is the number of lines of code, Control
is the total McCabe Cyclomatic Complexity, Calls is the number of
function calls made, Depth is the depth of the call hierarchy beneath
the function.

Function Name      Access#  Lines   Control Calls   Depth   Source File
main               0       1684    206     381     6       ant_main.c
annotate_returns  0       174     22      29      3       db_ant.c
is_in_call_tree   0       12      1       1       1       ant_calls.c
handle_call_clash 1        9      0       1       1       ant_calls.c
handle_equal_types 2        6      0       1       1       db_ant.c
tidy_up           1        5      0       2       1       ant_main.c
cmp_calls         5        4      0       0       0       ant_calls.c
compare_file_ants 4        4      0       0       0       ant_ids.c
handle_equal_globals 0        4      0       1       1       db_ant.c
cmp_globals       2        4      0       1       1       db_ant.c

----- End of top function report -----

```

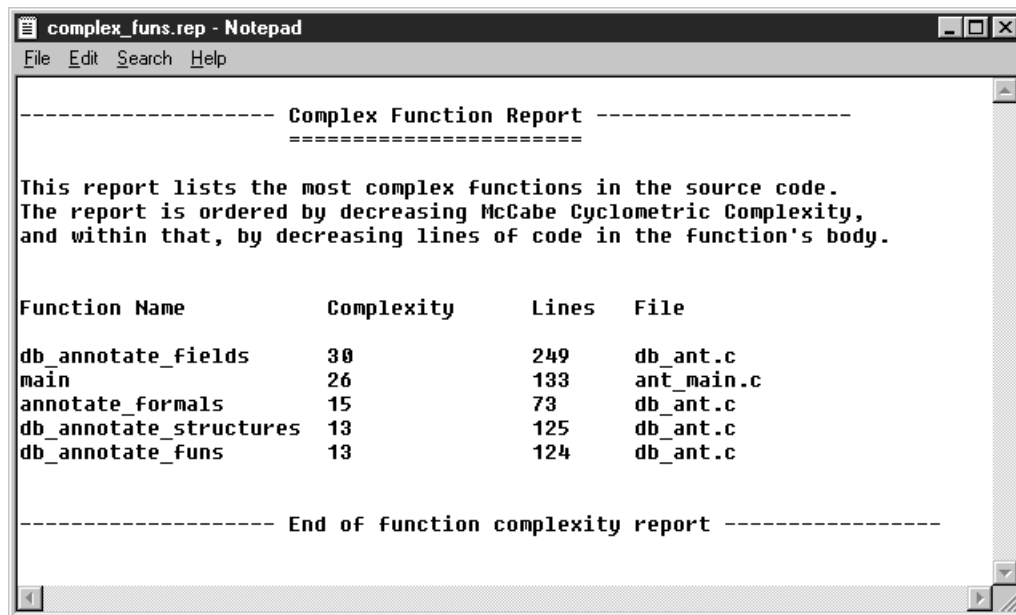

Complex Function Report

This report lists the largest and most complex user-defined functions on structure chart diagrams. These functions are the hardest to test, are the most likely to contain bugs, and require the most maintenance. (Ideally, functions should have a complexity of no more than 10.)

The report is useful for spotting similar and identical implementations of the same function. When source files representing many executables are parsed in a single system, this report can be used to determine which functions could be put into a shared library.

An example of this report is shown in Figure 2.

Figure 2: Complex Function Report



```
----- Complex Function Report -----  
=====
```

This report lists the most complex functions in the source code.
The report is ordered by decreasing McCabe Cyclometric Complexity,
and within that, by decreasing lines of code in the function's body.

Function Name	Complexity	Lines	File
db_annotate_fields	30	249	db_ant.c
main	26	133	ant_main.c
annotate_formals	15	73	db_ant.c
db_annotate_structures	13	125	db_ant.c
db_annotate_funs	13	124	db_ant.c

```
----- End of function complexity report -----
```

Include Hierarchy Report

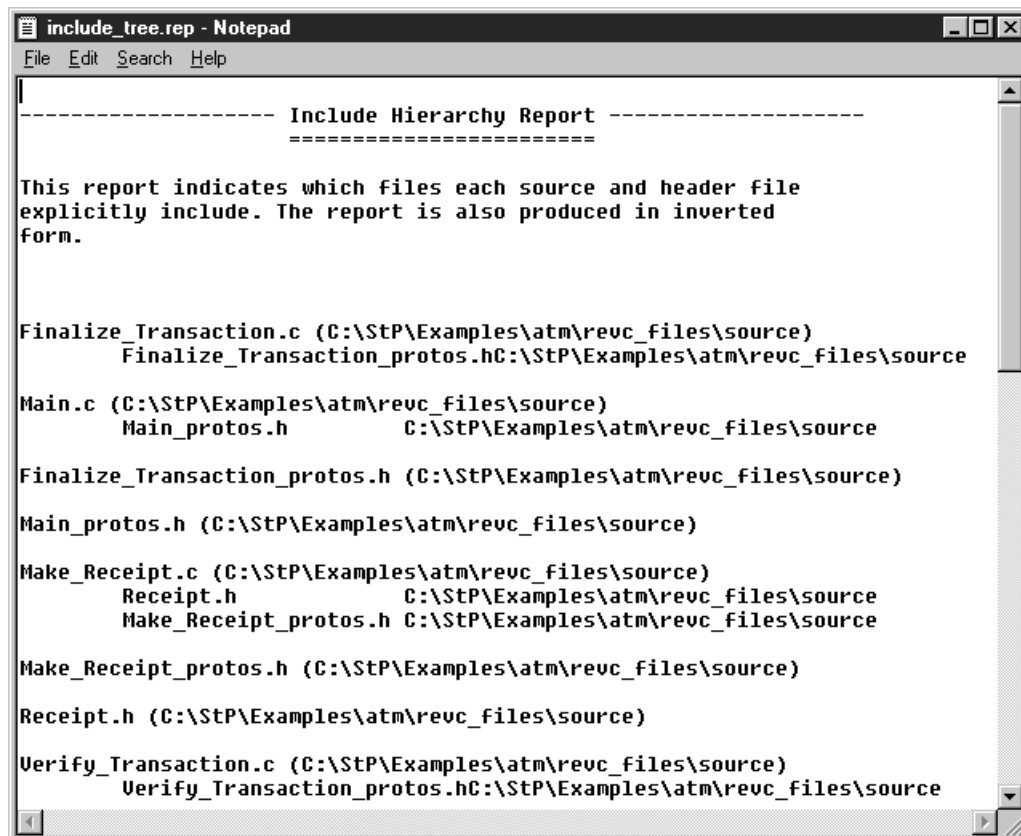
This report is useful for determining the effect of changes to header files.

The report has two sections:

- Include Hierarchy Report—Lists the files that a given file includes
- Included Hierarchy Report—Lists the files in which a given file is included

The following two figures show an example of the Include Hierarchy Report. Figure 3 shows the Include Hierarchy Report. Figure 4 shows the Included Hierarchy Report.

Figure 3: Include Hierarchy Report - Part 1



```
include_tree.rep - Notepad
File Edit Search Help

----- Include Hierarchy Report -----
=====

This report indicates which files each source and header file
explicitly include. The report is also produced in inverted
form.

Finalize_Transaction.c (C:\StP\Examples\atm\revc_files\source)
    Finalize_Transaction_protos.h C:\StP\Examples\atm\revc_files\source

Main.c (C:\StP\Examples\atm\revc_files\source)
    Main_protos.h C:\StP\Examples\atm\revc_files\source

Finalize_Transaction_protos.h (C:\StP\Examples\atm\revc_files\source)

Main_protos.h (C:\StP\Examples\atm\revc_files\source)

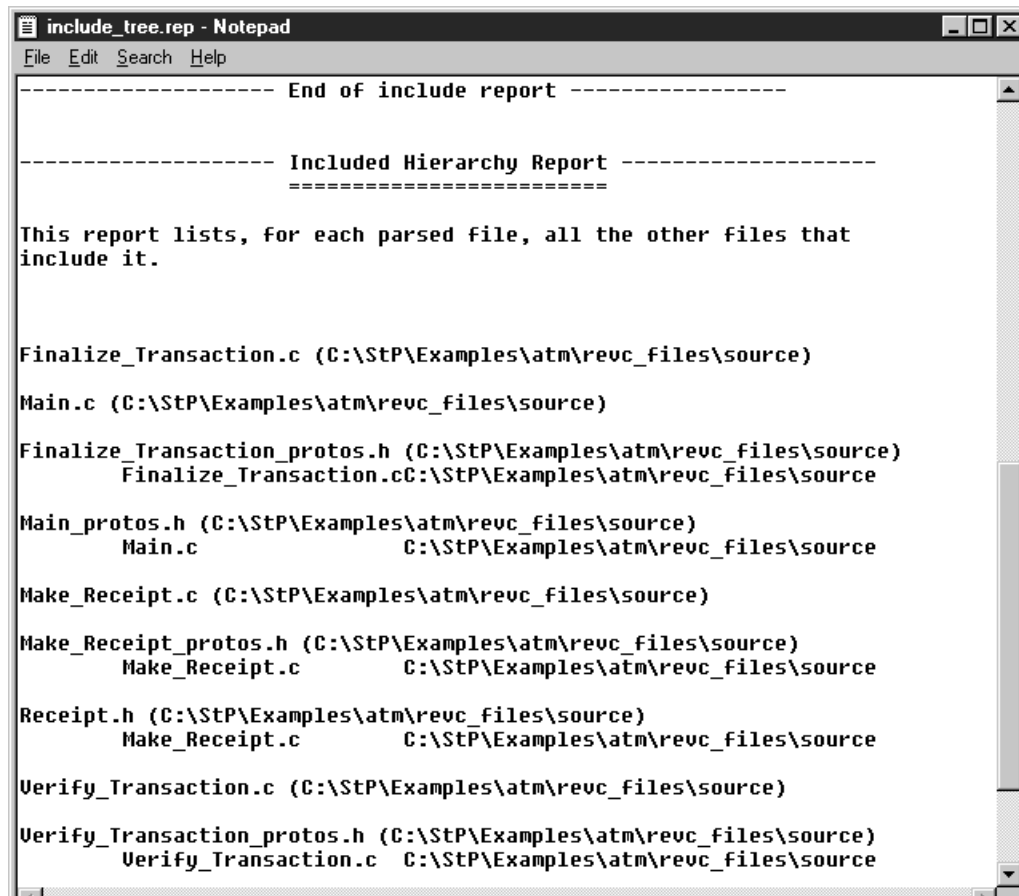
Make_Receipt.c (C:\StP\Examples\atm\revc_files\source)
    Receipt.h C:\StP\Examples\atm\revc_files\source
    Make_Receipt_protos.h C:\StP\Examples\atm\revc_files\source

Make_Receipt_protos.h (C:\StP\Examples\atm\revc_files\source)

Receipt.h (C:\StP\Examples\atm\revc_files\source)

Verify_Transaction.c (C:\StP\Examples\atm\revc_files\source)
    Verify_Transaction_protos.h C:\StP\Examples\atm\revc_files\source
```

Figure 4: Include Hierarchy Report - Part 2



```
include_tree.rep - Notepad
File Edit Search Help

----- End of include report -----

----- Included Hierarchy Report -----
=====

This report lists, for each parsed file, all the other files that
include it.

Finalize_Transaction.c (C:\StP\Examples\atm\revc_files\source)
Main.c (C:\StP\Examples\atm\revc_files\source)
Finalize_Transaction_protos.h (C:\StP\Examples\atm\revc_files\source)
    Finalize_Transaction.c C:\StP\Examples\atm\revc_files\source
Main_protos.h (C:\StP\Examples\atm\revc_files\source)
    Main.c C:\StP\Examples\atm\revc_files\source
Make_Receipt.c (C:\StP\Examples\atm\revc_files\source)
Make_Receipt_protos.h (C:\StP\Examples\atm\revc_files\source)
    Make_Receipt.c C:\StP\Examples\atm\revc_files\source
Receipt.h (C:\StP\Examples\atm\revc_files\source)
    Make_Receipt.c C:\StP\Examples\atm\revc_files\source
Verify_Transaction.c (C:\StP\Examples\atm\revc_files\source)
Verify_Transaction_protos.h (C:\StP\Examples\atm\revc_files\source)
    Verify_Transaction.c C:\StP\Examples\atm\revc_files\source
```

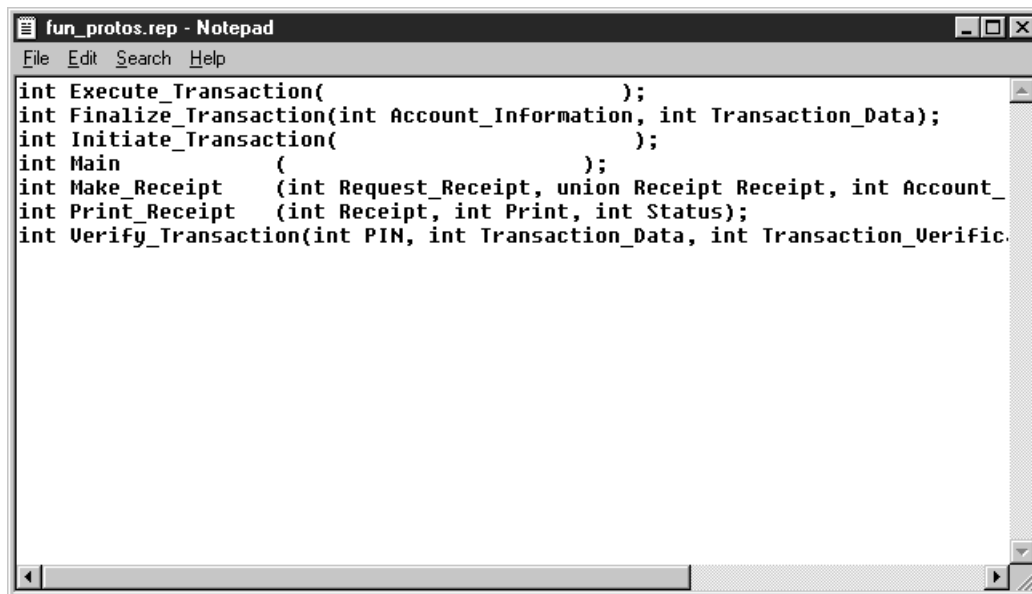
Function Definition Report

This report lists function prototypes for all of the user-defined (non-library) functions in the model's structure chart diagrams. The prototypes are composed of the function's name, return type, and the types and names of each of the function's parameters. They are listed in alphabetical order by function name.

You can generate this report as ANSI function prototypes, K&R prototypes, or as an alphabetical list of function names with the name of the source file in which they are defined.

An example of this report is shown in Figure 5.

Figure 5: Function Definition Report



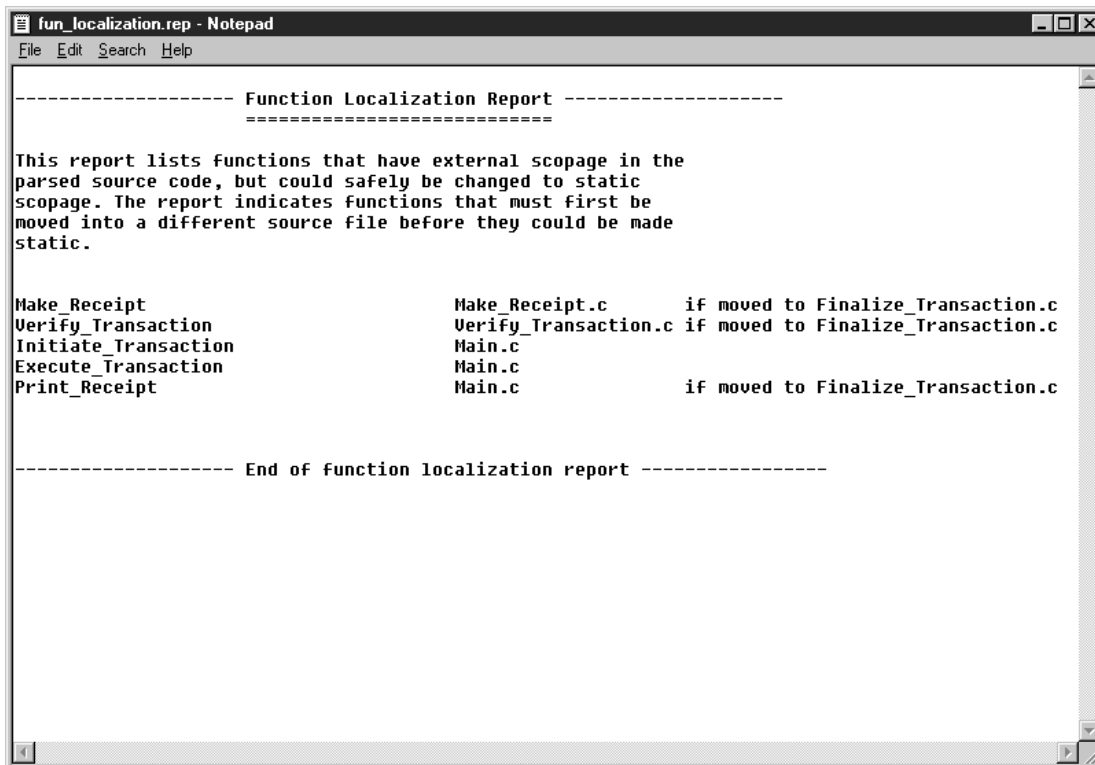
```
int Execute_Transaction( );
int Finalize_Transaction(int Account_Information, int Transaction_Data);
int Initiate_Transaction( );
int Main ( );
int Make_Receipt (int Request_Receipt, union Receipt Receipt, int Account_
int Print_Receipt (int Receipt, int Print, int Status);
int Verify_Transaction(int PIN, int Transaction_Data, int Transaction_Verific.
```

Function Localization Report

This report lists functions in structure chart diagrams that are currently externally scoped, but could be statically scoped without affecting the program. An example of this report is shown in Figure 6.

Brackets surrounding a file name indicate that the function could be made static if its definition is moved to the named source file.

Figure 6: Function Localization Report



```
fun_localization.rep - Notepad
File Edit Search Help

----- Function Localization Report -----
=====

This report lists functions that have external scopage in the
parsed source code, but could safely be changed to static
scopage. The report indicates functions that must first be
moved into a different source file before they could be made
static.

Make_Receipt                Make_Receipt.c      if moved to Finalize_Transaction.c
Verify_Transaction          Verify_Transaction.c if moved to Finalize_Transaction.c
Initiate_Transaction        Main.c
Execute_Transaction         Main.c
Print_Receipt               Main.c            if moved to Finalize_Transaction.c

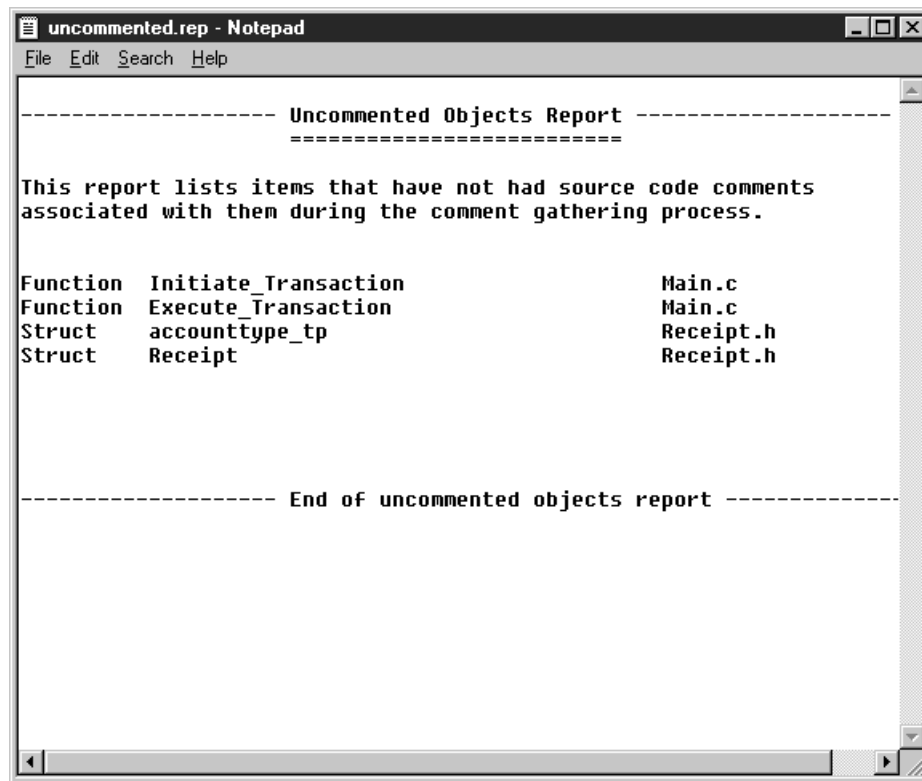
----- End of function localization report -----
```

Uncommented Objects Report

This report lists the functions, globals, and data structures for which no source code comments could be found. An example of this report is shown in Figure 7.

You may need to rerun the **Extract C Comments** command with different settings to extract more comments and add them to the semantic model, or check the source code for missing comments. For more information, see “Extracting Comments” on page 11-27.

Figure 7: Uncommented Objects Report



```
----- Uncommented Objects Report -----  
=====
```

This report lists items that have not had source code comments associated with them during the comment gathering process.

Function	Initiate_Transaction	Main.c
Function	Execute_Transaction	Main.c
Struct	accounttype_tp	Receipt.h
Struct	Receipt	Receipt.h

```
----- End of uncommented objects report -----
```

Generating Reports

Report generation commands are available from the StP Desktop. Table 1 lists the report generation commands available for StP/SE.

Table 1: Report Generation Commands

Desktop Menu	Command	For Details, See
Reports	Generate Analysis Review Report for Entire Model	“Summary of Analysis Review Report Options” on page 14-32
	Generate Design Review Report for Entire Model	“Summary of Design Review Report Options” on page 14-34
Code > Reverse Engineering	Generate C Code Metrics Reports	“Summary of C Code Metrics Report Options” on page 14-36

Using the Report Generation Dialog Boxes

Choosing any Report generation command described in Table 1 causes one of the Report Generation dialog boxes (Figure 8, Figure 9, and Figure 10) to appear. The Report Generation dialog box enables you to select various options for the selected report. Different options are presented for each of the reports.

To use a report generation dialog box:

1. Select a report generation command, as described in Table 1.
2. In the dialog box that appears, select options (as described in the following sections).
3. Click **OK** or **Apply**.

Summary of Analysis Review Report Options

The **Generate Analysis Review Report for Entire Model** command displays the **Generate Analysis Review Report** dialog box.

Figure 8: Generate Analysis Review Report Dialog Box

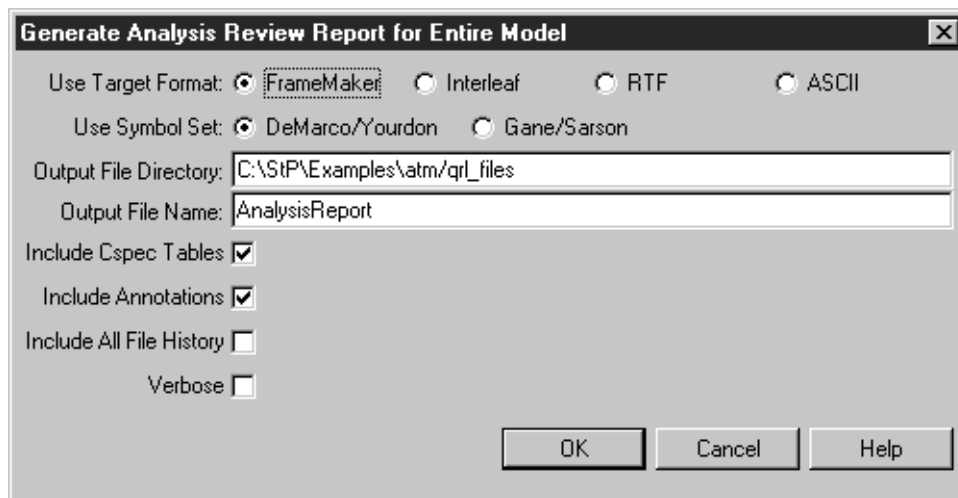


Table 2 is a summary of the options for the Analysis Review Report.

Table 2: Generate Analysis Review Report Options

Option	Description
Use Target Format	Select one of these formats for output: FrameMaker, Interleaf, RTF, ASCII
Use Symbol Set	Select symbol type: DeMarco/Yourdon or Gane/Sarson
Output File Directory	Shows default output directory. This field is editable.
Output File Name	Shows default output file. This field is editable.
Include Cspec Tables	If selected, includes information from Cspec tables in the report.
Include Annotations	If selected, includes information from annotations in the report.
Include All File History	If selected, includes information about the last three transactions performed against the file.
Verbose	If selected, displays report generation status messages in the StP Message Log as report is generated.

Summary of Design Review Report Options

The **Generate Design Review Report for Entire Model** command displays the **Generate Design Review Report** dialog box.

Figure 9: Generate Design Review Report Dialog Box

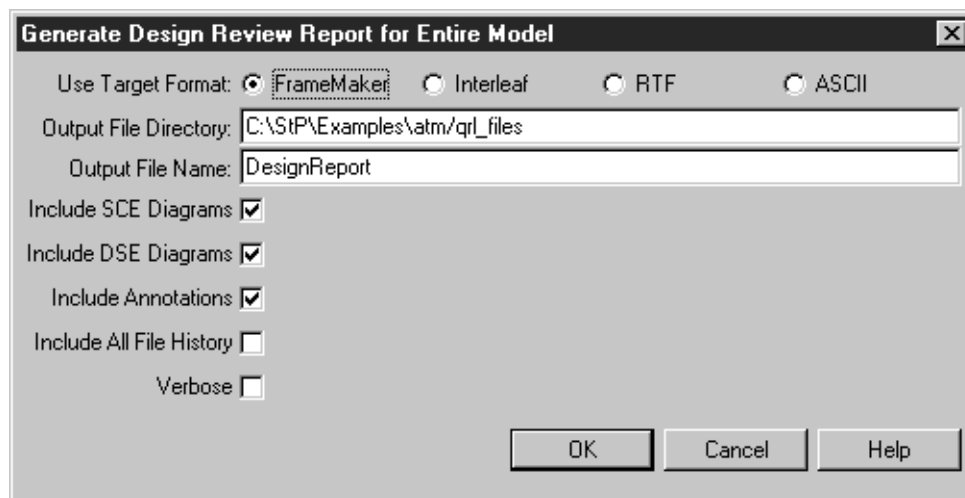


Table 3 summarizes the options for Design Review Report generation.

Table 3: Generate Design Review Report Options

Option	Description
Use Target Format	Select one of these formats for output: FrameMaker, Interleaf, RTF, ASCII
Output File Directory	Shows default output directory. This field is editable.
Output File Name	Shows default output file. This field is editable.
Include SCE Diagrams	If selected, includes information from structure chart diagrams in the report.
Include DSE Diagrams	If selected, includes information from data structure diagrams in the report.
Include Annotations	If selected, includes information from annotations in the report.
Include All File History	If selected, includes information about the last three transactions performed against the file.
Verbose	If selected, displays report generation status messages in the StP Message Log as report is generated.

Summary of C Code Metrics Report Options

The **Generate C Code Metrics Reports** command on the **Code > Reverse Engineering** menu displays the **Generate C Code Metrics Report** dialog box.

Figure 10: Generate C Code Metrics Report Dialog Box

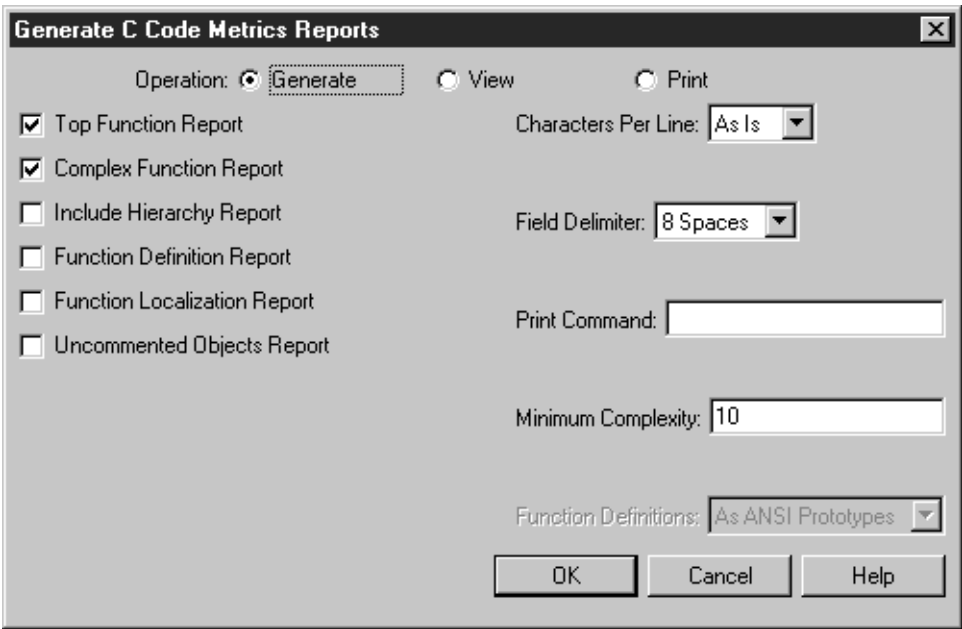


Table 4 summarizes the options for C Code Metrics Report generation.

Table 4: Generate C Code Metrics Report Options

Option	Description
Operation	Select one: Generate, View, Print . Generate —Generates a collection of reports into the current project and system directory. View —Generates all selected reports, and displays them one at a time in the StP default viewer or editor. Print —Not used in Windows NT. To print a report, choose the View operation and use the viewing application's Print command.
(report types)	Select one or more reports: Top Function Report Complex Function Report Include Hierarchy Report Function Definition Report Function Localization Report Uncommented Objects Report .
Characters Per Line	Select one: As Is, 65, 80, 100, 120 . The number of characters allowed on each report line.
Field Delimiter	Select one: Tab, 1 space, 4 spaces, 8 spaces . The amount of space that will appear between fields in the report.
Print Command	Not used in Windows NT. To print a report, choose the View operation and use the viewing application's Print command.
Minimum Complexity	For the Complex Functions Report only, specify the minimum level of complexity of functions to be included.
Function Definitions	For the Function Definition Report only, select one of the following: As ANSI Prototypes, As K&R Prototypes, As a Sorted List . The As a Sorted List option lists function names alphabetically and includes the name of the source file in which they are defined.

A **Application Types and PDM Types**

Each element in an SE model maps to an StP application type and a Persistent Data Model type. These types determine how project information is stored in the repository. For information about the Persistent Data Model types, see *Object Management System*.

Topics covered in this appendix are as follows:

- “Persistent Data Model Types” on page A-1
- “Application Types” on page A-2
- “Control Specification Editor (CSE) Types” on page A-3
- “Data Flow Editor (DFE) Types” on page A-5
- “Data Structure Editor (DSE) Types” on page A-6
- “Flow Chart Editor (FCE) Types” on page A-7
- “State Transition Editor (STE) Types” on page A-8
- “Structure Chart Editor (SCE) Types” on page A-9

Persistent Data Model Types

The Persistent Data Model (PDM) defines the attributes and relationships of objects in the repository in terms of abstract, persistent data types. An abstract data type consists of a data structure and its associated functions. All data created by StP applications is mapped to and inherits the attributes of one of these abstract data types.

This model provides a simple scheme for organizing and identifying objects in the repository while maintaining complete application independence.

Application Types

In diagram and table editors, each type of symbol or table cell represents an “application type” with a specific set of attributes. The link between these application types and the PDM is provided by the *app.types* file. This is an extensible file that maps types recognized by applications to types recognized by the StP repository.

Application types correspond to one of five PDM types:

- Node type
- Link type
- Cntx type
- Note type
- Item type

For example, on a data flow diagram, processes and data stores both map to the PDM node type, while data flows and control flows map to the PDM link type.

The following tables list all of the symbols used in each StP/SE editor along with their application types and the corresponding PDM types.

Note: Comments and comment links do not map to application or PDM types in the repository.

Control Specification Editor (CSE) Types

Table 1: CSE Application and PDM Types

SE Construct	Hsect	Vsect	Application Type	PDM Type	Which Cspec
ControlInCell	Controls	Control Ins	ControlIn	node	DET, PAT, ELT
ControlOutCell	Controls	Control Outs	ControlOut	node	DET, ALT
ProcessCell	Controls	Processes	Process	node	PAT, ALT
ControlInRow	Combinations	Control Ins	ControlCombination	node	DET, PAT, ELT
ControlOutRow	Combinations	Control Outs	ControlCombination	node	DET, ALT
ControlValueCell	Combinations	Control Ins	ControlValue	link	DET, PAT, ELT
		Control Outs	ControlValue	link	DET, ALT
ProcessTableProcess-ActivationCell	Combinations	Processes	ProcessActivation	link	PAT
ControlInCell	Control Ins/ Combinations	Control Ins	ControlIn	node	PAM
ProcessCell	Processes	Process Activations	Process	node	PAM

Application Types and PDM Types

Table 1: CSE Application and PDM Types (Continued)

SE Construct	Hsect	Vsect	Application Type	PDM Type	Which Cspec
ProcessRow	Control Ins/ Combinations	Process Activations	ProcessCombination	node	PAM
	Control Ins/ Combinations	Process Activations	Definition	link	PAM
ProcessActivationCell	Combinations	Processes	ProcessActivation	link	PAM, ALT
CurrentStateCell	Data	Data	State	node	STT
EventCell	Data	Data	Event	node	STT
ActionCell	Data	Data	Action	node	STT
NextStateCell	Data	Data	State	node	STT
			Transition	link	STT
			ActionInstance	cntx	STT
EventCell	Events	Events	Event	node	SEM
CurrentStateCell	States/ Combinations	States	State	node	SEM
ActionAndNext- StateCell	States/ Combinations	Events	Action	node	SEM
			State	node	SEM
			Transition	link	SEM
			ActionInstance	cntx	SEM
EventCell	Controls	Events	Event	node	ELT
EventDefinitionCell	Combinations	Events	Definition	link	ELT
ActionCell	Combinations	Actions	Action	node	ALT
ActionRow	Combinations	Actions	ControlCombination	node	ALT
			ActionControl- Definition	link	ALT

Data Flow Editor (DFE) Types

Table 2: DFE Application and PDM Types

SE Construct	Application Type	PDM Type
Anchor	FlowAnchor	node
Control flow arc or spline	ControlFlow	link (see note below)
Cspec bar	Cspec	node
Data flow arc or spline	DataFlow	link (see note below)
External	External	node
External store	External	node
Offpage external	OffPageExternal	node
Offpage process	OffPageProcess	node
Process	Process	node
	ProcessIndex	note
	RelativeIndex	item
Split flow	SplitFlow	node
Store	Store	node
Vertex	Vertex	N/A

Note: Each part of a data flow's or control flow's comma-separated name maps to one link object.

Data Structure Editor (DSE) Types

Table 3: DSE Application and PDM Types

SE Construct	Application Type	PDM Type
Component arc or spline	Component	link
Enumeration (see note below)	Enumeration	node
	SEFile	node
	SEDirectory	node
Selection (see note below)	Selection	node
	SEFile	node
	SEDirectory	node
Sequence (see note below)	Sequence	node
	SEFile	node
	SEDirectory	node
Typedef	Typedef	node
	SEFile	node
	SEDirectory	node

Note: The SEFile and SEDirectory application types represent scoping objects for typedefs and for sequence, selection, and enumeration objects that have no in links (root nodes).

Flow Chart Editor (FCE) Types

Table 4: FCE Application and PDM Types

SE Construct	Application Type	PDM Type
Code block	SECodeBlock	node
Comment	Comment	N/A
Do-While statement	SEUntil	node
Final state	SEFinalState	node
For statement	SEFor	node
If statement	SEIf	node
If-Else statement	SEIfElse	node
Initial state	SEStartState	node
Library call	SELibraryCall	node
Macro	SEMacro	node
Pointer call	SEPointerCall	node
SEFlowLink arc or spline	SEFlowLink	link
Source comment	SourceComment	N/A
Switch statement	SESwitch	node
Terminal	SETerminal	node
User call	SEUserCall	node
Void terminal	SEVoidTerminal	node
While statement	SEWhile	node

State Transition Editor (STE) Types

Table 5: STE Application and PDM Types

SE Construct	Application Type	PDM Type
Anchor	StartState	node
Event/Action	EventMap	node
	ActionMap	node
	ActionInstanceMap	cntx
State	State	node
Transition arc or spline	Transition	link
Vertex	Vertex	N/A

Note: The EventMap and ActionMap application types represent events and actions, respectively. The ActionInstanceMap application type represents the association between an event/action bar and a transition arc.

Structure Chart Editor (SCE) Types

Table 6: SCE Application and PDM Types

SE Construct	Application Type	PDM Type
Anchor	FormalCaller	node
Comment	Comment	N/A
Global data	GlobalData	node
	SEFile	node
	SEDirectory	node
Input data	DataIn	cntx
Input flag	ControlIn	cntx
Input/Output data	DataInOut	cntx
Library module	LibraryModule	node
Module	Module	node
	SEFile	node
	SEDirectory	node
ModuleCall arc or spline	SubprogramCall	link
Offpage connector	Offpage	node
Output data	DataOut	cntx
Output flag	ControlOut	cntx
Vertex	Vertex	N/A

Note: The SEFile and SEDirectory application types represent scoping objects for globals and modules.

B StP/SE Symbol Reference

This chapter describes the symbols and arcs used in the StP/SE diagram editors. Symbols are listed under the editor in which they appear. The chapter also includes tables listing the StP/SE display marks for each editor.

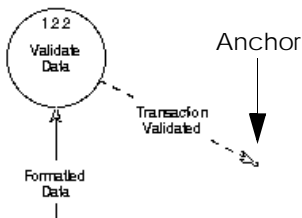
Topics covered in this appendix are as follows:

- “Data Flow Editor Symbols” on page B-2
 - “Data Structure Editor Symbols” on page B-7
 - “Flow Chart Editor Symbols” on page B-10
 - “Structure Chart Editor Symbols” on page B-16
 - “State Transition Editor Symbols” on page B-22
 - “Display Marks” on page B-25
-

Data Flow Editor Symbols

The alphabetically-listed symbols in this section are used to create data and control flow diagrams.

Anchor



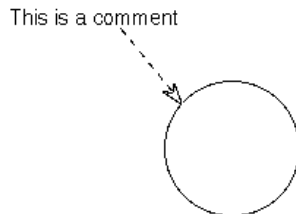
Definition: Small dot representing an offpage reference

Application Type: FlowAnchor

PDM Type: Node

Display Marks: None

Comment



Definition: A displayable note about a symbol

Application Type: None

PDM Type: None

Display Marks: None

Notes: You use the arc or spline symbol to draw links connecting a comment to the object to which it applies.

Control Flow (arc or spline)

Discrete flow:

----- Control Flow ----->>

Continuous flow:

----- Control Flow 2 ----->>

Definition: Arc or spline connecting processes with externals, other processes, and Cspec bars, that represents the movement of control information

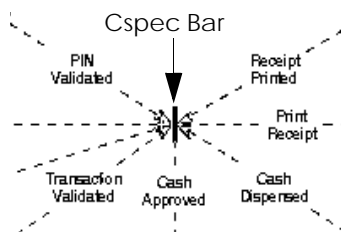
Arc Direction: Points to the destination symbol

Application Type: ControlFlow

PDM Type: Link

Display Marks: FlowType

Cspec Bar



Definition: Represents the real-time control aspects of a part of the system

Application Type: Cspec

PDM Type: Node

Display Marks: None

Data Flow (arc or spline)

Discrete flow:

----- Data Flow ----->>

Continuous flow:

----- Data Flow 2 ----->>

Definition: Arc connecting processes with externals, other processes, and data stores, that represents the movement of data

Arc Direction: Points to the destination symbol

Application Type: DataFlow

PDM Type: Link

Display Marks: FlowType

External

DeMarco/Yourdon:



Gane/Sarson:



Definition: A construct outside of the system being modeled that represents a physical device, a process, or a data source or sink

Application Type: External

PDM Type: Node

Display Marks: DuplicateExternal (Gane/Sarson only)

External Store



Definition: Data store that is outside of the system being modeled (appears only on the top diagram)

Application Type: External

PDM Type: Node

Display Marks: None

Offpage External



Definition: Indicates that an external exists on a higher level of the diagram hierarchy

Application Type: OffPageExternal

PDM Type: Node

Display Marks: None

Offpage Process



Definition: Indicates that a process exists on a higher level of the diagram hierarchy

Application Type: OffPageProcess

PDM Type: Node

Display Marks: None

Process

DeMarco/Yourdon:



Definition: Represents an operation that uses or generates data

Application Type: Process, ProcessIndex, RelativeIndex

PDM Type: Node, note, item

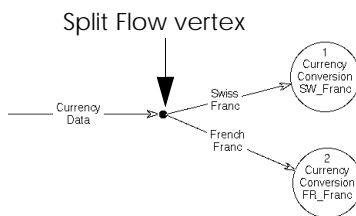
Display Marks: Pindex

Gane/Sarson:



Notes: Saving a process to the repository creates all three application types and PDM types. The ProcessIndex application type and its PDM Type (note) represent the part of the process index that indicates the parent diagram. The RelativeIndex application type and its PDM Type (item) represent the part of the index that is incremented for each process in the diagram.

Split Flow



Definition: Represents a common vertex for multiple flows

Application Type: SplitFlow

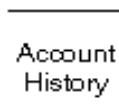
PDM Type: Node

Display Marks: None

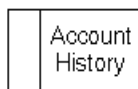
Notes: Does not exist as a symbol on the Symbol Palette. To create this symbol, choose the **Split Flow** command from the DFE menu.

Store

DeMarco/Yourdon:



Gane/Sarson:



Definition: A place where data is kept until it is used by a process

Application Type: Store

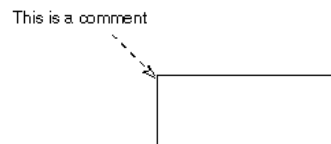
PDM Type: Node

Display Marks: DuplicateStore (Gane/Sarson only)

Data Structure Editor Symbols

The alphabetically-listed symbols in this section are used to create data structure diagrams.

Comment



Definition: A displayable note about a symbol

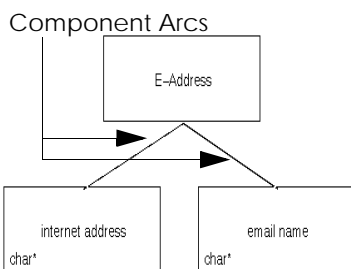
Application Type: None

PDM Type: None

Display Marks: None

Notes: You use the arc or spline symbol to draw links connecting a comment to the object to which it applies.

Component (arc or spline)



Definition: Arc or spline connecting component parts of a hierarchical data structure

Arc Direction: None

Application Type: Component

PDM Type: Link

Display Marks: None

Enumeration



Definition: Represents a limited number of specific values defined by its leaf nodes

Application Type: Enumeration, SEFile, SEDirectory

PDM Type: Node, node, node

Display Marks: SEDirectory, SEFile, Tag

Notes: All three application types and their associated PDM Types are created for root objects. The SEFile and SEDirectory types represent the object’s scoping information.

Selection



Definition: Represents an “or” choice relationship between two or more structures, as defined by its leaf nodes

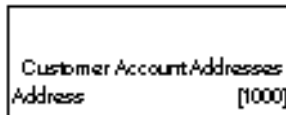
Application Type: Selection, SEFile, SEDirectory

PDM Type: Node, node, node

Display Marks: SEDirectory, SEFile, Tag, LeafType, ArraySize

Notes: All three application types and their associated PDM Types are created for root objects. The SEFile and SEDirectory types represent the object’s scoping information.

Sequence



Definition: Represents an “and” relationship composed of two or more structures, as defined by its leaf nodes

Application Type: Sequence, SEFile, SEDirectory

PDM Type: Node, node, node

Display Marks: SEDirectory, SEFile, Tag, LeafType, ArraySize

Notes: All three application types and their associated PDM Types are created for root objects. The SEFile and SEDirectory types represent the object’s scoping information.

Typedef



Definition: Assigns an additional name (alias) by which you can refer to a data object in a data structure diagram

Application Type: Typedef, SEFile, SEDirectory

PDM Type: Node, node, node

Display Marks: SEDirectory, SEFile, TypedefLeafType, ArraySize

Notes: When this object is saved to the repository, all three application types and their associated PDM Types are created. The SEFile and SEDirectory types represent the object’s scoping information.

Flow Chart Editor Symbols

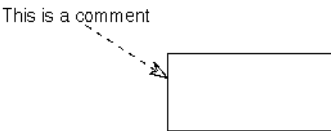
The alphabetically-listed symbols in this section are used to create flow charts.

Code Block



Definition: Contains an actual piece of code
Application Type: SECodeBlock
PDM Type: Node
Display Marks: None

Comment



Definition: A displayable note about a symbol
Application Type: None
PDM Type: None
Display Marks: None
Notes: You use the arc or spline symbol to draw links connecting a comment to the object to which it applies.

Do-While Statement



Definition: Represents a do-while statement with a true logic path (left arc) and a false logic path (right arc)
Application Type: SEUntil
PDM Type: Node
Display Marks: None

Final State



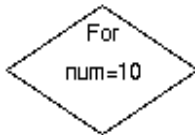
Definition: Logical end of a function

Application Type: SEFinalState

PDM Type: Node

Display Marks: None

For Statement



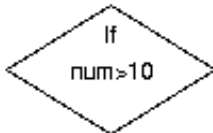
Definition: Represents a for statement with a true logic path (left arc) and a false logic path (right arc)

Application Type: SEFor

PDM Type: Node

Display Marks: None

If Statement



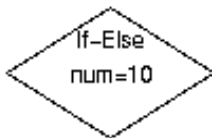
Definition: Represents an if statement with a true logic path (left arc) and a false logic path (right arc)

Application Type: SEIf

PDM Type: Node

Display Marks: None

If-Else Statement



Definition: Represents an if-else statement with a true logic path (left arc) and a false logic path (right arc)

Application Type: SEIfElse

PDM Type: Node

Display Marks: None

Initial State



Definition: Flow chart entry point indicating the beginning of the program flow for that function

Application Type: SEStartState

PDM Type: Node

Display Marks: None

Library Call



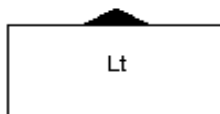
Definition: Indicates a call to a library function

Application Type: SELibraryCall

PDM Type: Node

Display Marks: None

Macro



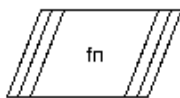
Definition: Represents a piece of code that is a macro

Application Type: SEMacro

PDM Type: Node

Display Marks: None

Pointer Call



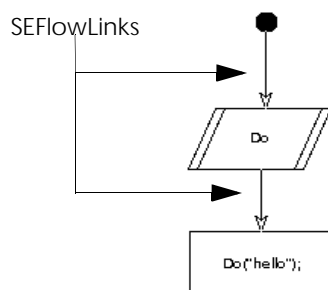
Definition: Indicates a call to a function pointer

Application Type: SEPointerCall

PDM Type: Node

Display Marks: None

SEFlowLink (arc or spline)



Definition: Arc or spline connecting the elements of a flow chart in the sequence of program execution

Arc Direction: In the direction of program flow

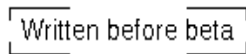
Application Type: SEFlowLink

PDM Type: Link

Display Marks: None

Notes: Arcs and splines can be labeled. It is helpful to assign labels such as 'True' or 'False' to arcs exiting conditional statements.

Source Comment



Definition: Represents a comment that appears in code

Application Type: None

PDM Type: None

Display Marks: None

Switch Statement



Definition: Represents switch statement with a true logic path (left arc) and a false logic path (right arc)

Application Type: SESwitch

PDM Type: Node

Display Marks: None

Terminal



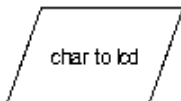
Definition: Represents the return of a value from within a particular branch of a function

Application Type: SETerminal

PDM Type: Node

Display Marks: None

User Call



Definition: Indicates a call to a user function
Application Type: SEUserCall
PDM Type: Node
Display Marks: None

Void Terminal



Definition: Indicates that no value is returned when the function terminates
Application Type: SEVoidTerminal
PDM Type: Node
Display Marks: None

While Statement

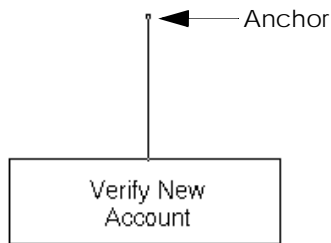


Definition: Represents while statement with a true logic path (left arc) and a false logic path (right arc)
Application Type: SEWhile
PDM Type: Node
Display Marks: None

Structure Chart Editor Symbols

The alphabetically-listed symbols in this section are used to create structure charts.

Anchor



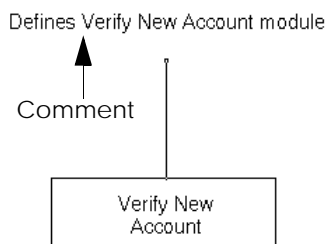
Definition: Indicates the defining point for a module

Application Type: FormalCaller

PDM Type: Node

Display Marks: None

Comment



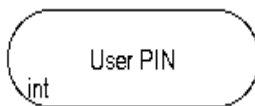
Definition: A displayable note about a symbol or diagram

Application Type: None

PDM Type: None

Display Marks: None

Global Data



Definition: Global data elements called by program modules

Application Type: GlobalData, SEFile, SEDirectory

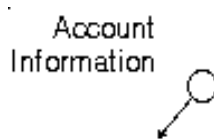
PDM Type: Node, node, node

Display Marks: AccessMode, ArraySize, GlobalReturnType, SEDirectory, SEFile

Notes: When this object is saved to the repository, all three application types and their associated PDM Types are created. The SEFile and SEDirectory types represent the object's scoping information.

Although access mode is a global data property, the AccessMode display mark itself is attached to the ModuleCall arc(s) connected to the global data symbol.

Input Data



Definition: Parameter symbol representing the transfer of data into a module

Application Type: DataIn

PDM Type: Cntx

Display Marks: ParamType

Input Flag



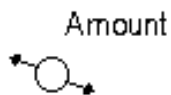
Definition: Parameter symbol representing the transfer of control information into a module

Application Type: ControlIn

PDM Type: Cntx

Display Marks: ParamType

Input/Output Data



Definition: Parameter symbol representing the updating of data by a module

Application Type: DataInOut

PDM Type: Cntx

Display Marks: ParamType

Iteration



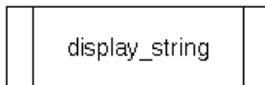
Definition: Procedural symbol representing a calling relationship within a loop construct

Application Type: None

PDM Type: None

Display Marks: None

Library Module



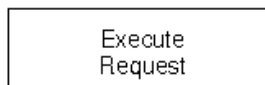
Definition: Represents a self-contained unit of code that exists outside of the modeled system

Application Type: LibraryModule

PDM Type: Node

Display Marks: None

Module



Definition: Represents the code in an identifiable program unit of the modeled system, such as a subprogram, subroutine, or function

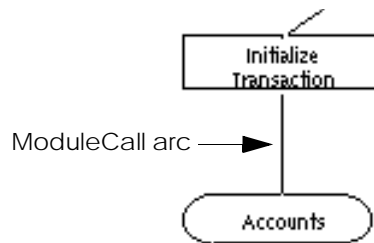
Application Type: Module, SEFile, SEDirectory

PDM Type: Node, node, node

Display Marks: LexicalModule, ModuleReturnType, Pdl, SEDirectory, SEFile

Notes: When this object is saved to the repository, all three application types and their associated PDM Types are created. The SEFile and SEDirectory types represent the object's scoping information.

ModuleCall (arc or spline)



Definition: Arc or spline representing the calling order of modules in a structure chart

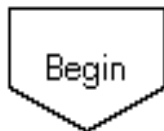
Arc Direction: None

Application Type: SubprogramCall

PDM Type: Link

Display Marks: None

Offpage Connector



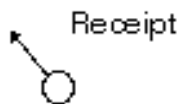
Definition: Connects to an offpage symbol on the same or another diagram that is the logical couple of the current diagram

Application Type: Offpage

PDM Type: Node

Display Marks: None

Output Data



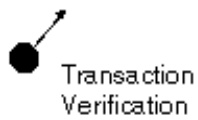
Definition: Parameter symbol representing the transfer of data out of a module

Application Type: DataOut

PDM Type: Cntx

Display Marks: ParamType

Output Flag



Definition: Parameter symbol representing the transfer of control information out of a module

Application Type: ControlOut

PDM Type: Cntx

Display Marks: ParamType

Selection



Definition: Procedural symbol representing an “if” or “case” statement in the calling module

Application Type: None

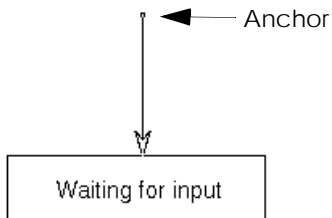
PDM Type: None

Display Marks: None

State Transition Editor Symbols

The alphabetically-listed symbols in this section are used to create state transition diagrams.

Anchor



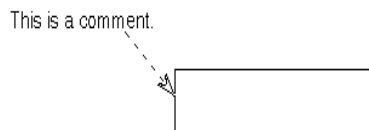
Definition: Indicates which state symbol represents the initial (start) state

Application Type: StartState

PDM Type: Node

Display Marks: None

Comment



Definition: A displayable note about a symbol

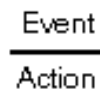
Application Type: None

PDM Type: None

Display Marks: None

Notes: You use the arc or spline symbol to draw links connecting a comment to the object to which it applies.

Event/Action



Event

Action

Definition: An event and an action that cause a transition between states

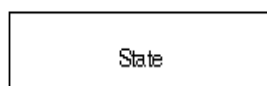
Application Type: EventMap, ActionMap, ActionInstanceMap

PDM Type: Node, node, cntx

Display Marks: None

Notes: All three application types and their corresponding PDM Types are created for each event/action bar. The EventMap and ActionMap types represent events and actions, respectively. ActionInstanceMap represents the association between an event/action bar and a transition arc.

State



State

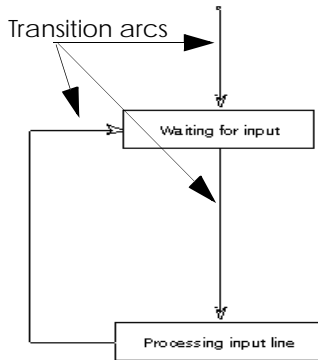
Definition: Shows the status of the system before, after, and between real-time events

Application Type: State

PDM Type: Node

Display Marks: None

Transition (arc or spline)



Definition: Arc or spline representing a transition between states

Arc Direction: In the direction of the transition

Application Type: Transition

PDM Type: Link

Display Marks: None

Display Marks

The following tables describe the display marks used in each StP/SE diagram editor, and the objects to which they can be applied.

Data Flow Editor

Table 1: Data Flow Diagram Display Marks






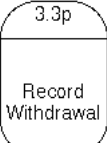

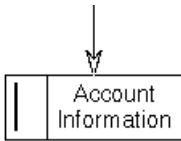
Name	Example		Description	Object
Pindex	De Marco-Yourdon 	Gane-Sarson 	A process index in the format x.y, where x is the parent process index and also the current diagram level (diagram's name) and y is an integer that is incremented for each process added to the diagram.	Process
(with decomposition indicator)	De Marco-Yourdon 	Gane-Sarson 	An asterisk following the process index indicates the existence of a decomposition for that process.	
(with process specification indicator)	De Marco-Yourdon 	Gane-Sarson 	The lowercase p following the process index indicates the existence of a Pspec annotation that describes this process.	

Table 1: Data Flow Diagram Display Marks (Continued)

Name	Example	Description	Object
FlowType	<p>Discrete flow:</p> <p>———— Data Flow —————></p> <p>----- Control Flow -----></p> <p>Continuous flow:</p> <p>———— Data Flow 2 —————>></p> <p>----- Control Flow 2 ----->></p>	Indicates whether the flow is discrete (single-headed arrow) or continuous (double-headed arrow).	Data flow or control flow
DuplicateExternal (Gane/Sarson only)		Diagonal line across lower right corner indicating this external has been used more than once.	External
DuplicateStore (Gane/Sarson only)		Bold vertical bar indicating this data store has been used more than once.	Store

Data Structure Editor

Table 2: Data Structure Diagram Display Marks

Name	Example	Description	Object
SEDirectory		Text outside upper right corner identifies the directory object to which this data object is scoped.	Root nodes
SEFile		Text outside lower right corner identifies the file object to which this data object is scoped.	
Tag		Text preceded by the word Tag : in the upper left corner of intermediate nodes, referring to the C structure tag for the struct, union, or enum. The tag is stored as the Tag item on a Data Definition note.	Intermediate nodes
LeafType		Text in lower left corner indicating the data type of the object. Data type is stored as the Data Type item on a Data Definition note.	Sequence, Selection
TypedefLeafType		Text in lower-left corner indicates the data type of the typedef. The data type is stored as the Data Type item on a Typedef Definition note.	Typedef
ArraySize		Text in lower right corner identifying the array size. Array size is stored as the Array Size item on a Data Definition note.	Sequence, Selection, Typedef

Structure Chart Editor

Table 3: Structure Chart Display Marks

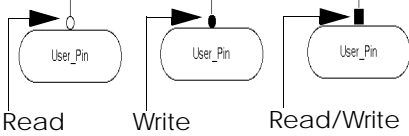

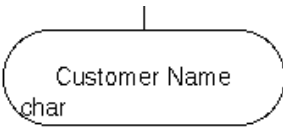
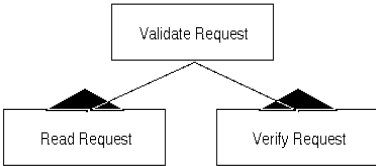
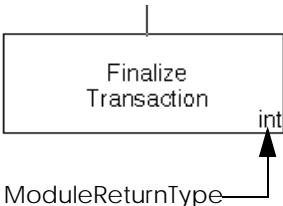
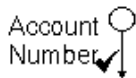
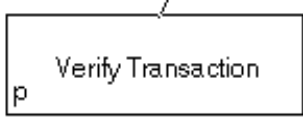
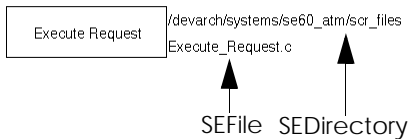
Name	Example	Description	Object
AccessMode		A hollow or solid circle or a solid square appearing at the end of a link between a module and a global data object, indicating the mode used to access the global data.	Global Data
ArraySize		Bracketed text in lower right corner identifying the array size of a global data object.	
GlobalReturnType		Text in lower left corner indicating the data type for a global data module. Default is int.	
LexicalModule		Solid triangle at the top of a module indicating that it is lexically included within its parent module.	Program Module
ModuleReturnType		Text in lower right corner indicating the return data type for the module. Default is int.	

Table 3: Structure Chart Display Marks (Continued)

Name	Example	Description	Object
ParamType		Check mark indicating that a data type has been specified for the parameter.	Parameter
Pdl		Letter P in lower left corner that indicates a PDL file exists for this module.	Program module
SEDirectory		Identifies a directory to which a program module or global data module is scoped.	Program module, Global data
SEFile		Identifies a file to which a program module or global data module is scoped.	

Index

- (hyphen) pattern matching
 - character 12-22
- ! (exclamation point) pattern matching
 - character 12-22
- #include files
 - parsing source code (RE) 11-17, 11-18, 11-22
- * (asterisk) pattern matching
 - character 12-22
- ? (question mark) pattern matching
 - character 12-22
- @ (At sign)
 - function pointer data types 10-10
- [] (square brackets) pattern matching
 - characters 12-22

A

- aborts in FCE diagrams
 - reverse engineered 11-79
- abstract data types, C
 - comment extraction for (RE) 11-29
 - generating C code for 10-16
- Access Mode annotation item (SCE) 7-34
- access modes to global data (SCE)
 - display mark 7-10, 7-34, 7-36, B-28
 - specifying 7-34
- Action in Action Logic Table command (STE) 5-6
- Action Logic Table command (DFE) 3-8
- action logic tables 5-14, 6-15, 6-23

- actions in Cspec tables
 - balancing with STE diagrams 6-30
 - described 6-6
 - editing labels 6-19
 - navigation from 6-13
 - representing 6-4
 - validating 9-8
- actions in STE diagrams
 - described 5-10
 - renaming, effects on dependent objects 13-17
 - representing 5-10 to 5-12
 - symbol for 5-10
 - transitions associated with 5-10
 - validating 9-6
- actual parameters, *See* parameters in SCE diagrams
- All Control Spec Tables tool 6-8
- All References command (DSE) 4-5
- All References command (SCE) 7-7, 7-8
- Allocate Requirements command
 - Data Flow Editor 3-7, 3-8
 - Data Structure Editor 4-5
 - described 1-11
 - State Transition Editor 5-6
 - Structure Chart Editor 7-7, 7-8
- Allocate Requirements to Object command
 - State Transition Editor 5-6
 - Structure Chart Editor 7-7, 7-8
- Allowed Value annotation item (DSE) 4-16, 6-19, 6-20

-
- Analysis Review Report
 - described 14-1
 - dialog box 14-32
 - generating 2-8
 - anchors
 - external stores in DFE diagrams 3-18
 - initial states in STE diagrams 5-7
 - offpage externals in DFE diagrams 3-16, 3-20
 - offpage processes in DFE diagrams 3-17, 3-20
 - symbol for (DFE) 3-20, B-2
 - symbol for (SCE) 7-37, B-16
 - symbol for (STE) 5-7, B-22
 - annotations
 - See also* Object Annotation Editor, object properties
 - described 1-9
 - for C code generation 10-3, 10-5
 - for data flow diagrams 3-28
 - for data structure diagrams 4-7, 4-17, 10-5
 - for requirements 1-12
 - for SEFile and SEDirectory objects 2-9
 - for state transition diagrams 5-13
 - for structure chart diagrams 7-22, 10-3
 - generated by Reverse Engineering 11-2, 11-81 to 11-83
 - online descriptions of 1-10
 - ANSI C 10-1, 10-6, 10-26
 - Aonix
 - documentation comments xix
 - Technical Support xviii
 - websites xix
 - application types
 - defined A-2
 - for control specification table cells A-3
 - for data flow diagram symbols A-5
 - for data structure diagram symbols A-6
 - for flow chart diagram symbols A-7
 - for state transition diagram symbols A-8
 - for structure chart diagram symbols A-9
 - persistent data and 1-13
 - Apply Filter command
 - in SCE diagrams 7-47
 - in STE diagrams 5-13
 - Apply to Multiple Selection option
 - Properties dialog (DSE) 4-21
 - Properties dialog (SCE) 7-27
 - apptypes, *See* application types
 - arcs
 - Component arc symbol (DSE) B-7
 - Default Arc Type command (DFE) 3-19
 - in DFE diagrams 3-19
 - ModuleCall symbol (SCE) B-20
 - SEFlowLink symbol (FCE) B-13
 - Transition symbol (STE) 5-9, B-24
 - arrays
 - Array Size annotation item (DSE) 10-19, 11-83, B-27
 - Array Size annotation item (SCE) 7-36, 10-13, 11-82, B-28
 - Array Size property (DSE) 4-20, 4-28, 10-5, 10-19
 - Array Size property (SCE) 7-27, 7-36, 10-4, 10-13
 - ArraySize display mark 4-7, 7-10, 7-36, B-27, B-28
 - in DSE diagrams 4-7, 4-28, 10-19
 - in SCE diagrams 7-36, 10-13
 - Arrow button (->)
 - Select Files to Search dialog box 11-50, 12-23
 - arrow icons (-> <-)
 - Synchronization Differences dialog box (RE) 11-52
 - ASCII format 9-12
 - Attach to Arc command (SCE) 7-17
- ## B
- Backus-Naur Format files
 - creating BNF script 4-40
 - Backus-Naur format files 2-7, 4-38
 - BNF files, *See* Backus-Naur format files
 - Browse C Code dialog box 12-13

Browse Code button
 Synchronization Differences dialog box (RE) 11-53, 11-54
Browse commands
 Browse C Code 12-14
 Browse Structured Models 2-9, 12-5
Browse Design button
 Synchronization Differences dialog box (RE) 11-53, 11-54
Browse menu (SE Browser) 12-6
browser *See* SE Browser, C Code Browser
browsing
 See also C Code Browser, SE Browser, navigation
 repository for SE objects 12-2 to 12-12
 repository with OMS
 queries 12-6 to 12-10
 semantic model 12-13 to 12-38

C

C code
 browsing, *See* C Code Browser
 directories and files for code 4-22, 7-28, 10-2, 10-22, 10-23
 displayed in text editor/viewer 10-25, 11-54, 12-24, 12-38 to 12-41
 editing generated code 10-28
 embedded code, avoiding parsing (RE) 11-19 to 11-22
 generating from graphical model, *See* C code generation
 generating semantic model from, *See* parsing source code
 illegal characters in identifiers 10-21
 illegal variable names 10-21
 modifying generated code 10-27
 PC features, parsing (RE) 11-8, 11-11
 removing previously generated 10-25, 10-28
 synchronizing RE model with 1-1, 11-2, 11-5, 11-45 to 11-56
 tagged fields in generated files 10-28
C Code Body annotation note 10-4, 10-13, 10-26, 11-37, 11-82

C Code Browser
 C programming constructs 12-14
 described 1-8, 12-1, 12-13
 exact match text searches 12-18
 examples 12-25 to 12-36
 identifier categories 12-17
 Identifier Properties dialog box 12-20
 identifier scope qualification 12-21
 identifier selection, customized 12-19
 objects not searched for 12-37
 pattern matching 12-21
 results, interpreting 12-37
 Reverse Engineering and 11-44, 12-37
 search options 12-16
 semantic model locks 11-26, 12-37
 specifying files to search 12-22
 starting 12-14
 troubleshooting 12-38
 using 12-14 to 12-25
 viewing code 12-24
C code generation
 See also C code
 commands for 10-22
 described 1-1, 1-8, 2-7, 10-1
 directories and files for output 10-2, 10-22, 10-23, 10-25
 DSE annotations for 10-5
 DSE support for 10-14 to 10-20
 editing code 10-28
 file extensions for 10-23, 10-25
 from SEFile/SEDirectory objects 2-7
 from selected diagrams 2-7
 Generate C Code dialog box 10-23
 incremental 10-27
 modifying generated code 10-27
 options for 10-25
 overriding default output location 10-23
 Query and Reporting Language scripts 10-2
 removing previously generated code 10-25, 10-28
 SCE annotations for 10-3
 SCE support for 10-6 to 10-14

-
- tagged fields in generated files 10-28 to 10-34
 - validation checks for generating 9-14
 - viewing generated code 10-25
 - C Code Metrics reports
 - described 14-22 to 14-30
 - dialog box 14-36
 - C dialects
 - parsing source code (RE) 11-17, 11-19
 - C identifiers
 - illegal characters in 10-21
 - C Implementation option (Generate C Code dialog) 10-26
 - C storage classes
 - generating C code for 10-4, 10-12, 10-14
 - representing in SCE 7-29
 - Storage Class property (SCE) 7-26, 7-30
 - values for 7-29
 - C structure tags in DSE diagrams
 - adding 4-29
 - display mark for 4-7, B-27
 - generating C code for 10-5, 10-17
 - Call's Callee command (FCE) 8-6
 - Call's Caller command (FCE) 8-6
 - Call's Flow Chart command (FCE) 8-6
 - Caller command (SCE) 7-7
 - calls
 - See also objects called*
 - browsing semantic model for 12-27 to 12-28
 - in FCE diagrams 8-8, B-12, B-15
 - in SCE diagrams 7-11
 - reverse engineered in FCE diagrams 11-72
 - Candidate Files scrolling list
 - Select Files to Search dialog 11-49, 12-23
 - Change Diagram Generation Options button
 - Synchronize Model dialog box (RE) 11-47, 11-48
 - Change Process Index command (DFE) 3-9, 3-40
 - Change Process Index dialog box 3-41
 - Check RE Semantic Model Consistency command 9-14, 11-6, 11-27
 - Check RE Semantic Model Locks command 11-6, 11-26
 - Check Semantics commands
 - Check Cspec Semantics 9-10
 - Check Semantics 9-10
 - Check Semantics for Selected Objects 2-9, 9-7
 - Check Semantics for Whole Model 2-9, 9-10
 - Check Semantics Selectively 2-9, 9-10, 9-12
 - Check Table Semantics 9-10
 - Check Semantics dialog 9-12
 - Check Semantics Selectively dialog box 9-13
 - Check Syntax command 8-15, 9-2
 - Choose Files for Partial Model button
 - Synchronize Model dialog box (RE) 11-47, 11-48
 - Choose Proper Code Reference dialog box 12-15
 - Clear Browser command (SE Browser) 12-12
 - cloning objects 13-7
 - code blocks in FCE diagrams
 - reverse engineered 11-71
 - symbol for 8-9, B-10
 - Collapse command (DFE) 3-9, 3-43
 - commands
 - Control Specification Editor GoTo menu 6-10 to 6-13
 - Data Flow Editor DFE menu 3-9
 - Data Flow Editor GoTo menu 3-7
 - Data Structure Editor DSE menu 4-6
 - Data Structure Editor GoTo menu 4-5
 - Desktop, StP/SE specific 2-7
 - Desktop, using 2-5
 - Flow Chart Editor GoTo menu 8-6
 - for generating C code 10-22
 - for reorganizing data flow diagrams 3-38
 - frequently used in editors 1-9
-

-
- report generation 14-31
 - Reverse Engineering 11-4
 - SE Browser's SE menu 12-12
 - semantic checking 9-10
 - source code navigation 12-40
 - State Transition Editor GoTo menu 5-6
 - Structure Chart Editor GoTo menu 7-7
 - Structure Chart Editor SCE menu 7-8
 - comment extraction (RE)
 - comment delimiters 11-30
 - described 11-2, 11-5, 11-27
 - evaluating and improving 11-33
 - Extract Source Code Comments dialog box 11-28
 - formatting comment text 11-32
 - gap setting 11-29, 11-31
 - line length specification 11-30
 - options 11-29
 - position setting 11-31
 - saving parameters 11-30
 - swear word detection 11-33
 - uses of 11-33
 - using 11-30
 - comments
 - C data structure 10-5, 10-14
 - C function header 10-4
 - DFE symbol for 3-21, B-2
 - DSE symbol for B-7
 - FCE source code comments 8-15, B-14
 - FCE symbol for B-10
 - generating C code for 10-26
 - SCE symbol for B-16
 - STE symbol for B-22
 - Uncommented Objects Report 14-30
 - compiler directives
 - default 11-17
 - defining for parsing (RE) 11-17, 11-18
 - disabling during parsing (RE) 11-16, 11-17
 - compiler errors prevention 10-19
 - Complex Function Report
 - described 14-25
 - dialog box 14-37
 - conditional tests in FCE diagrams
 - representing 8-10
 - reverse engineered 11-73, 11-74
 - symbol for 8-10
 - context diagrams in DFE
 - creating 3-22 to 3-23
 - example of 3-23
 - control flow symbol B-3
 - control flows, *See* flows in DFE diagrams
 - control in/out cells in Cspec tables
 - balancing with DFE control flows 6-30
 - control combinations 6-6, 9-7, 9-8
 - described 6-6
 - editing labels 6-19
 - navigation from 6-10, 6-11, 6-13
 - representing control flows 6-4
 - validating 9-7, 9-8
 - control in/out parameters in SCE diagrams 7-15
 - control signals, *See* flows in DFE diagrams, parameters in SCE diagrams
 - Control Specification Editor
 - Data Flow Editor and 1-4
 - described 1-7, 6-1, 6-8
 - GoTo menu 6-9
 - Hatley/Pirbhai methodology 1-5
 - SE-specific Edit menu commands 6-14
 - SE-specific Tools menu commands 6-14
 - starting 6-9
 - State Transition Editor and 5-1
 - StP Desktop commands 2-5
 - supported methodology 1-3
 - control specification tables
 - combinational logic 6-3
 - components of 6-4 to 6-7, 6-22 to 6-29
 - control signal flow 6-2
 - copying, renaming, and deleting 6-8
 - creating 6-15 to 6-17
 - creating from state transition diagrams 5-14
 - described 1-5, 6-2 to 6-7, 6-22 to 6-29
 - editing 6-18 to 6-21
 - labels for table elements 6-4, 6-5, 6-18, 6-19
 - navigations 6-9 to 6-13, 6-16
-

-
- object mapping to application/PDM
 - types A-3
 - populating 5-5, 6-15, 6-18, 6-20
 - sequential logic 6-3
 - switching between 6-21
 - validating 6-30, 9-7 to 9-8
 - control values in Cspec tables
 - described 6-6
 - populating cells with 6-14, 6-18, 6-19, 6-20
 - representing 6-4
 - validating 9-7
 - CSE *See* Control Specification Editor
 - Cspec Bar command (STE) 5-6
 - Cspec Bar on Data Flow Diagram
 - command (CSE) 6-10, 6-11, 6-12, 6-13
 - Cspec bars in DFE diagrams
 - navigation from 3-7, 3-8
 - representing control
 - specifications 3-33 to 3-34
 - symbol 3-33, B-3
 - validating 9-5
 - Cspecs, *See* control specification tables
 - D**
 - Data Definition annotation note
 - (DSE) 4-7, 4-27, 4-28, 6-20, 10-17, 10-19, 11-83, B-27
 - Data Definition annotation note
 - (SCE) 7-36
 - Data Definition command (CSE) 6-10, 6-11, 6-13
 - data definitions, C 10-15
 - Data Flow Diagram command (STE) 5-6
 - data flow diagrams
 - See also* Data Flow Editor
 - Analysis Review Report and 2-8
 - comments in 3-21
 - context diagram 3-22 to 3-23
 - creating 3-12 to 3-37
 - decompositions 3-24
 - defining data and control
 - information 3-35
 - described 1-4, 3-1, 3-12
 - display marks 3-10, B-25
 - filters to hide data or control flows 3-37
 - naming 3-12
 - navigations 3-6 to 3-8
 - object mapping to application/PDM
 - types A-5
 - process index, changing 3-40
 - renaming hierarchies 2-10, 3-38 to 3-40
 - reorganizing 3-38 to 3-55
 - reports 14-1
 - representing objects 3-15
 - validating 3-56, 9-3
 - Data Flow Editor
 - See also* data flow diagrams
 - Control Specification Editor and 1-4, 3-32
 - Data Structure Editor and 1-5
 - DeMarco/Yourdon methodology 1-5, 3-3
 - described 1-7, 3-1, 3-2
 - DFE menu 3-9
 - Gane/Sarson methodology 1-5, 3-3
 - GoTo menu 3-6
 - Hatley/Pirbhai methodology 1-5
 - starting 3-4
 - State Transition Editor and 5-1
 - StP Desktop commands 2-4, 2-8, 2-10
 - supported methodology 1-3
 - symbols 3-5, B-2 to B-6
 - data flow symbol B-3
 - data flows, *See* flows in DFE diagrams
 - data parameters, *See* parameters in SCE diagrams
 - data stores in DFE diagrams
 - defining in DSE 3-35
 - display mark for duplicates 3-11, B-26
 - external 3-18, B-4
 - qualifications, removing 3-50
 - renaming, effects on dependent
 - objects 13-16
 - symbol for 3-18, B-6
 - validating 9-3, 9-4
 - Data Structure Comment annotation note
 - (DSE) 10-5, 10-14, 10-26, 11-83
-

data structure diagrams
 See also Data Structure Editor
 annotations for C code generation 10-5
 Backus-Naur format files for 4-38
 creating 4-8 to 4-16
 data elements 4-9 to 4-10
 decomposing objects 4-15, 4-33
 defining DFE control signals 4-15
 defining SCE data types 7-40
 described 1-5, 4-1, 4-8
 display marks B-27
 drawing children on 4-24, 4-36
 generating C code from 10-14 to 10-20
 intermediate nodes 4-10, 4-14
 leaf nodes 4-9, 4-14
 navigations 4-4 to 4-5, 4-36
 object mapping to application/PDM types A-6
 object properties 4-17 to 4-30
 reports 14-12, 14-30
 reverse engineered 11-2, 11-66 to 11-70
 root nodes, *See* root nodes in DSE diagrams
 scoping 4-21 to 4-26
 uncommented data structures 14-30
 validating 4-40, 9-6, 9-14
 Data Structure Editor
 See also data structure diagrams
 Data Flow Editor and 1-5, 3-35
 described 1-7, 4-1, 4-2
 DSE menu 4-6
 GoTo menu 4-4
 Jackson methodology 1-6
 Properties dialog 4-17 to 4-21
 starting 4-3
 StP Desktop commands 2-4, 2-7
 Structure Chart Editor and 1-5
 supported methodology 1-3
 symbols 4-3, B-7 to B-9
 Data Type annotation item
 in DSE diagrams 10-18, 11-83, B-27
 in SCE diagrams 10-11, 10-14, 11-82
 data types
 abstract 4-15, 4-30
 assigning to DSE objects 4-7, 4-27 to 4-28
 assigning to SCE objects 7-31 to 7-32
 browsing repository for abstract types 12-4, 12-11
 defining in DSE 4-15, 7-40
 display marks B-27, B-28
 in DSE diagrams 10-15, 10-18
 in SCE diagrams 10-4, 10-5
 predefined C types 7-31, 7-40
 system type 4-30, 10-5, 10-19
 validating in DSE diagrams 9-6
 validating in SCE diagrams 9-9
 Decision Table command (DFE) 3-7
 decision tables 6-24
 Decomposition command (DFE) 3-7
 Definition command (SCE) 7-7, 7-38
 Delete Selected Differences button
 Synchronization Differences dialog box (RE) 11-53
 Delete Unreferenced Objects in Current System Repository
 command 13-7
 deleting unreferenced repository objects 13-7
 DeMarco/Yourdon Data Flow Diagrams
 See Data Flow Editor
 DeMarco/Yourdon methodology 1-3, 1-5, 2-4, 3-3
 design documentation, creating 1-1
 Design Module button
 View Code dialog box 11-56, 12-25
 Design Review Report
 described 14-12
 dialog box 14-34
 generating 2-8
 Design Type button
 View Code dialog box 11-56, 12-25
 desktop, *See* StP Desktop
 DFE menu 3-9
 DFE, *See* Data Flow Editor
 diagram generation (Reverse Engineering)
 choosing diagram types 11-35

-
- control level settings for structure
 - charts 11-38, 11-43, 11-58 to 11-63
 - described 11-34
 - diagram backups 11-35
 - flow chart elements, including/
 - suppressing 11-39
 - function code, including 11-37
 - Generate New Model dialog box 11-34
 - generating new model 11-34 to 11-44
 - global variable access, including 11-38
 - library calls, including/
 - suppressing 11-38, 11-39 to 11-41, 11-42
 - limiting size of SCE diagrams 11-38
 - options for generating diagrams 11-35, 11-37
 - positioning calls in SCE
 - diagrams 11-38
 - replacing existing model 11-45
 - Set Diagram Generation Options dialog box 11-36
 - space between symbols,
 - specifying 11-38, 11-39
 - system type data structures,
 - including 11-39
 - unused data structures,
 - including 11-39, 11-43
 - width compression of DSE
 - diagrams 11-39
 - diagram types, described 1-4
 - dialog boxes
 - Analysis Review Report 14-32
 - Browse C Code 12-13
 - Change Process Index 3-41
 - Check Semantics 9-12
 - Check Semantics Selectively 9-13
 - Choose Proper Code Reference 12-15
 - Design Review Report 14-34
 - Execute Query 12-9
 - Extract Source Code Comments 11-28
 - Generate BNF 4-39
 - Generate C Code 10-23
 - Generate C Code Metrics Report 14-36
 - Generate New Model 11-34
 - Generate Pspec 3-30
 - Identifier Properties 12-19, 12-20
 - Makefile Options 11-12
 - Options 3-10, 3-19, 4-6, 7-9
 - Parse Source Code 11-6, 11-9
 - Parser Preprocessor Options 11-15
 - Properties (DSE) 4-17 to 4-21
 - Properties (SCE) 7-23 to 7-27
 - Rename Hierarchy 3-39
 - Rename Object (diagram/table objects) 13-13
 - Select Files to Search 11-48, 12-22
 - Set Diagram Generation Options 11-36
 - Synchronization Differences 11-52
 - Synchronize Model 11-46
 - View Code 11-54, 12-24
 - directories
 - See also* SEDirectory objects
 - Directory property (DSE) 4-20, 4-26
 - Directory property (SCE) 7-26, 7-28
 - for generated code 4-22
 - for Reverse Engineering 11-2, 11-7, 11-23, 11-83 to 11-85
 - Directory field (Generate C Code dialog) 10-23, 10-25
 - display marks
 - Data Flow Editor 3-10, B-25
 - Data Structure Editor 4-6 to 4-7, B-27
 - descriptions of, by editor B-25 to B-29
 - Display Marks tab of Options dialog box 3-10, 4-6, 7-9
 - Structure Chart Editor 7-9 to 7-10, B-28
 - Do All Design Updates button
 - Synchronization Differences dialog box (RE) 11-53
 - Do Design Additions button
 - Synchronization Differences dialog box (RE) 11-53
 - Do Design Deletions button
 - Synchronization Differences dialog box (RE) 11-53
 - do-while statements in FCE diagrams
 - reverse engineered 11-74
 - symbol for B-10
 - DSE Data Definition command (DFE) 3-7, 3-8, 3-36
-

DSE menu 4-6
DSE, *See* Data Structure Editor
DuplicateExternal display mark
(DFE) 3-11, B-26
DuplicateStore display mark (DFE) 3-11,
B-26

E

Edit PSpec Note command (DFE) 3-9,
3-28
editing
 Edit Annotation command 2-9
 Edit Diagram Annotation
 command 1-9
 Edit PDL Note command (SCE) 7-9,
7-43
 Edit QBE as OMS Query command (SE
 Browser) 12-8
 Edit Table Annotation command 1-9
 generated C code 10-28
 SEFile and SEDirectory object
 annotations 2-9
Editor field (Generate C Code
 dialog) 10-25
entry points in FCE diagrams 8-7
enumerations in DSE diagrams
 browsing source code for 12-33
 described 4-13
 reverse engineered 11-69
 symbol for 4-13, B-8
errors
 navigating to source of in model 9-14
 renaming objects systemwide 13-15
 Reverse Engineering parsing 11-22
event definitions in Cspec tables
 described 6-6
 editing cells 6-19
 representing 6-4
Event in Event Logic Table command
(STE) 5-6
Event Logic Table command (DFE) 3-8
event logic tables 5-14, 6-15, 6-25
event/action bars in STE diagrams
 associating with transitions 5-10

 example of 5-11
 filters to hide or show 5-13
 moving 5-12
 navigating from 5-6
 symbol for 5-10, 5-12, B-23
 validating 9-6
events in Cspec tables
 balancing with STE diagrams 6-30
 described 6-6
 editing labels 6-19
 navigation from 6-13
 representing 6-4
 validating 9-8
events in STE diagrams
 defining in Cspec tables 6-1
 described 5-10
 renaming, effects on dependent
 objects 13-17
 representing 5-10 to 5-12
 symbol for 5-10
 transitions associated with 5-10
 validating 9-6
Execute OMS Query command 12-8
Execute QBE command 12-7
Exit command, File menu 1-9
exit points in FCE diagrams
 final state symbol 8-13, B-11
 representing 8-12
 reverse engineered 11-76
 terminal symbol 8-13, B-14
 void terminal symbol 8-13, B-15
Explode command (DFE) 3-9, 3-46
external stores, *See under* data stores
external variables, reverse
 engineering 11-66
externals in DFE diagrams
 display mark for duplicates 3-11, B-26
 symbol for 3-15, B-4
Extract Source Code Comments command
(RE) 11-4, 11-30

F

FCE, *See* Flow Chart Editor

-
- File button
 - View Code dialog box 11-55, 12-25
 - File Name field (Generate C Code dialog) 10-23, 10-25
 - File property
 - data structure diagrams 4-20, 4-26
 - structure charts 7-26, 7-28
 - files
 - See also* SEFile objects
 - Backus-Naur format files (DSE) 4-38
 - for generated code 4-22, 7-27
 - for Reverse Engineering 11-23, 11-85 to 11-86
 - makefiles for Reverse Engineering 11-18
 - process specifications (DFE) 3-28
 - Program Design Language files (SCE) 7-44
 - saving semantic errors to 9-12
 - selecting for browsing 12-18, 12-19, 12-22
 - selecting for parsing (RE) 11-8
 - specifiers file for parsing code (RE) 11-17, 11-19
 - storing model information in 1-13
 - Fill Browser command (SE Browser) 12-12
 - Fill button
 - Properties dialog (DSE) 4-19, 4-20
 - Properties dialog (SCE) 7-24, 7-26
 - Filter button and text field
 - Parse Source Code dialog box (RE) 11-7
 - filters
 - Hide/Show IncludedModules (SCE) 7-47
 - Hide/Show Libraries (SCE) 7-47
 - Hide/Show ParameterLabels (SCE) 7-47
 - HideAllControlFlowsShowAllDataFlows (DFE) 3-37
 - HideAllDataFlowsShowAllControlFlows (DFE) 3-37
 - HideEventAction (STE) 5-13
 - ShowEventAction filter (STE) 5-13
 - final states in FCE diagrams, symbol for 8-13, B-11
 - Flow Chart command (SCE) 7-7
 - flow chart diagrams
 - See also* Flow Chart Editor
 - associating with SCE-defined functions 8-7
 - calls, representing 8-8
 - conditional tests, representing 8-10
 - creating 8-7 to 8-15
 - described 1-6, 8-1, 8-7
 - entry points 8-7
 - exit points, representing 8-12
 - incorporating code segments 8-9
 - navigations 8-5 to 8-6
 - object mapping to application/PDM types A-7
 - reverse engineered 11-2, 11-71 to 11-81
 - source comments 8-15
 - validating 8-15
 - Flow Chart Editor
 - See also* flow chart diagrams
 - described 1-7, 8-1, 8-2
 - GoTo menu 8-5
 - starting 8-3
 - StP Desktop commands 2-4
 - supported methodology 1-3
 - symbols 8-3, B-10 to B-15
 - Flow Chart's Callee command (FCE) 8-6
 - Flow Chart's Caller command (FCE) 8-6
 - Flow Chart's SCE Module command (FCE) 8-6
 - Flow Chart's Source Code Definition command (FCE) 8-6, 12-40
 - Flow Type annotation note and item (DFE) 3-20
 - Flow Type display mark (DFE) 3-11, 3-19
 - flows in DFE diagrams
 - balancing 3-57
 - browsing repository for 12-3, 12-4, 12-11
 - continuous 3-19
 - control flows 1-4, 3-19, 3-32, 3-33, 6-1, 9-3, B-3
 - data flows 3-19
-

-
- defining control signals in DSE 4-15
 - defining in DSE 3-35
 - described 3-19
 - discrete 3-19
 - drawing 3-32, 3-33
 - flow type 3-11, 3-19
 - flow type display mark B-26
 - hiding data or control information 3-37
 - loopback 3-26
 - merging 3-53
 - navigation from 3-8
 - qualifications, removing 3-50
 - renaming, effects on dependent objects 13-16
 - reversing 3-55
 - splitting 3-52, 3-53
 - symbol for 3-19, 3-32, B-3
 - symbol for split flows B-6
 - validating 9-3, 9-4, 9-5
 - FlowType display mark (DFE) B-26
 - for statements in FCE diagrams
 - reverse engineered 11-74
 - symbol for B-11
 - formal parameters, *See* parameters in SCE diagrams
 - forward declarations 10-7
 - FrameMaker MIF format 9-12
 - Function Definition Report
 - described 14-28
 - dialog box 14-37
 - Function Localization Report
 - described 14-29
 - dialog box 14-37
 - functions
 - See also* library modules, program modules
 - C storage class identifiers 7-29
 - calls 10-12
 - code bodies 10-4, 10-12, 10-13, 10-26
 - comment extraction for (RE) 11-29
 - complex 14-25
 - declarations 10-9
 - definitions 10-7
 - externally versus statically scoped 14-29
 - header comments 10-4, 10-8
 - header parameters 7-41, 10-9
 - included 7-32
 - pointers 10-10
 - prototypes 10-7, 14-28
 - representing in SCE diagrams 7-11, 8-7
 - return types 10-4, 10-11
 - reverse engineered 11-37, 11-65
 - top 14-23
 - uncommented 14-30
- G**
- Gane/Sarson Data Flow Diagrams tool, *See* Data Flow Editor
 - Gane/Sarson methodology 1-3, 1-5, 2-4, 3-3
 - Generate BNF commands
 - Generate BNF (Desktop command) 2-7, 4-39
 - Generate BNF for All Diagrams (Desktop command) 2-7, 4-39
 - Generate BNF for Diagram 4-6, 4-39
 - Generate BNF dialog box 4-39
 - Generate C Code Body option (Generate C Code dialog) 10-13, 10-26
 - Generate C commands
 - Generate C (Desktop command) 2-7, 10-22
 - Generate C Code (SCE command) 7-9, 10-22
 - Generate C for Entire Model 2-7, 10-22
 - Generate Comments option (Generate C Code dialog) 10-8, 10-14, 10-26
 - Generate New Model command (RE) 11-4, 11-34
 - Generate PDL commands (SCE)
 - Generate PDL 2-8, 7-9, 7-44
 - Generate PDL for All Diagrams 2-8, 7-45
 - Generate Pspec commands (DFE)
 - Generate Pspec 2-8, 3-9, 3-29
 - Generate Pspec for All Diagrams 2-8, 3-30
 - Generate Pspec dialog box 3-30

Generate Report commands
 Generate Analysis Review Report for
 Entire Model 2-8, 14-31
 Generate C Code Metrics
 Reports 14-31
 Generate Design Review Report for
 Entire Model 2-8, 14-31
generating
 C code from graphical model 1-1,
 10-1 to 10-34
 graphical model from source code 1-1,
 11-34 to 11-43
Global Access property (SCE) 7-26, 7-34
Global Comment annotation note
 (SCE) 11-82
global data in SCE diagrams
 access modes 7-10, 7-34
 array size 7-10, 7-36, 10-13
 C storage classes 7-29
 data types 7-10, 7-31, 7-40, 10-4, 10-14
 described 7-12
 display marks 7-10, 7-32, B-28, B-29
 drawing 7-14
 generating C code for 10-13
 navigation from 7-8
 reverse engineered 11-38, 11-66
 scoping to directories and files 7-10,
 7-27 to 7-29
 symbol for 7-13, B-17
 uncommented 14-30
Global Definition annotation note
 (SCE) 7-30, 10-12, 10-13, 10-14,
 11-82
Global Type command (SCE) 7-8, 7-40
Global Usage annotation note (SCE) 7-34
global variables
 comment extraction for (RE) 11-29
GlobalReturnType display mark
 (SCE) 7-10, 7-31, B-28
Go to Error Origin button (message
 log) 9-14
GoTo menu
 Control Specification Editor 6-9
 Data Flow Editor 3-6

Data Structure Editor 4-4
Flow Chart Editor 8-5
State Transition Editor 5-4
Structure Chart Editor 7-6

H

Hatley/Pirbhai methodology 1-3, 1-5
header files
 parsing errors (RE) 11-23
 parsing source code (RE) 11-10, 11-17,
 11-18
Help menu, Object Annotation
 Editor 1-10
Hide/Show Groups command (SE
 Browser) 12-5
HTML format 9-12

I

Identifier Properties dialog box 12-19
if statements in FCE diagrams
 reverse engineered 11-73
 symbol for B-11
if-else statements in FCE diagrams
 reverse engineered 11-73
 symbol for B-12
ifndef statements 10-19
Implementation File Extension field
 (Generate C Code dialog) 10-23,
 10-25
Include Hierarchy Report
 described 14-26
 dialog box 14-37
Included File annotation item 10-20
Included File annotation item (DSE) 10-5
Included File annotation item (SCE) 10-4
included files
 generating C code 10-4, 10-5, 10-20
 ifndef statements 10-19
 parsing source code (RE) 11-18
 reports 14-26
initial states in FCE diagrams
 reverse engineered 11-71
 symbol for 8-7, B-12

- initial states in STE diagrams
 - drawing 5-7
 - symbol for 5-7
 - validating 9-6
- input data symbol B-17
- input flag symbol B-18
- input/output data symbol B-18
- input/output parameters, *See* parameters in SCE diagrams
- Interface (Header) File Extension field (Generate C Code dialog) 10-23, 10-25
- iterations in SCE diagrams
 - representing 7-19
 - reverse engineered 11-58
 - symbol for 7-18, 7-19, B-18

J

Jackson methodology 1-3, 1-6

K

Kernighan & Ritchie C 10-1, 10-6, 10-26

L

- LeafType display mark (DSE) 4-7, B-27
- Lexical Include annotation note (SCE) 7-33
- LexicalModule display mark (SCE) 7-10, 7-33, B-28
- library functions in FCE diagrams
 - representing calls to 8-8
 - symbol for calls to B-12
- library modules in SCE diagrams
 - described 7-13
 - drawing 7-14
 - hiding/showing 7-47
 - navigation from 7-8
 - reverse engineered 11-39 to 11-42, 11-65
 - symbol for 7-13, B-19
- library templates for Reverse Engineering 11-40 to 11-42

- locks
 - on semantic model 11-26, 12-37
 - status of on semantic model 11-6, 11-26

M

- macros in FCE diagrams, symbol for 8-10, B-13
- Makefile Options dialog 11-12
- Makefile reader (RE)
 - macros 11-15
 - makefiles 11-18
 - Parse Source Code dialog option 11-8
 - recursive makes 11-14
 - subordinate targets 11-15
 - using 11-9, 11-12 to 11-15
- Merge Flow command (DFE) 3-10, 3-53
- Message Log
 - printing error messages 9-12
- Microsoft Word RTF format 9-12
- MIF (FrameMaker) format 9-12
- models
 - described 1-2
 - storing information about 1-13
- Module Comment annotation note (SCE) 10-4, 10-8, 10-26, 11-82
- Module Definition annotation note (SCE) 7-30, 10-4, 10-9, 10-11, 10-12, 11-82
- Module Is field, Properties dialog (SCE) 7-26, 7-33
- Module PDL annotation note (SCE) 7-42
- Module Return Type annotation item (SCE) 10-11, 11-82
- Module Storage Class annotation item (SCE) 10-12
- ModuleReturnTypes display mark (SCE) 7-10, 7-31, B-28
- modules, *See* program modules, library modules, global data

N

- Name field
 - Properties dialog (DSE) 4-20

Properties dialog (SCE) 7-26
navigation
Control Specification Editor
 commands 6-9 to 6-13
Data Flow Editor commands 3-6 to 3-8
Data Structure Editor
 commands 4-4 to 4-5
Flow Chart Editor
 commands 8-5 to 8-6
SE Browser options 12-6, 12-10
State Transition Editor
 commands 5-4 to 5-6
Structure Chart Editor
 commands 7-6 to 7-8
to code in text editor/viewer 10-25,
 11-3, 11-54, 12-24, 12-38 to 12-41
to Control Specification Editor 3-6, 5-5,
 5-6, 6-9, 6-15
to Data Flow Editor 3-4, 5-5, 5-6, 6-10,
 12-10
to Data Structure Editor 3-6, 4-3, 6-10,
 7-6, 12-10
to DSE parent structures 4-36
to Flow Chart Editor 7-6, 8-3
to Requirements Table Editor 3-6, 4-4,
 5-5, 7-6
to State Transition Editor 3-6, 5-3, 6-10
to Structure Chart Editor 7-4, 8-5, 12-10
nested structures, reverse
 engineering 11-68
notation
DeMarco/Yourdon 1-5
Gane/Sarson 1-5
Hatley/Pirbhai 1-5
Jackson 1-6
method references 1-3
methods in Data Flow Editor 3-3
Yourdon/Constantine 1-6

O

OAE, *See* Object Annotation Editor
Object Annotation Editor
 See also annotations, object properties
 data structure diagrams 4-17
 described 1-9

requirements annotations 1-11
state transition diagrams 5-13
structure chart diagrams 7-22
Object Management System query
 language 12-2, 12-6
object properties
 annotating objects using property
 sheets 1-9
 C code generation and 10-3
 DSE annotations for C code
 generation 10-5
 SCE annotations for C code
 generation 10-3
 setting in DSE 4-17 to 4-30
 setting in SCE 7-22 to 7-36
object references, described 1-14, 13-2
objects, *See* repository
offpage connectors in SCE diagrams
 navigation from 7-8
 symbol for 7-20, B-20
 using 7-20 to 7-22
 validating 9-9
offpage externals in DFE diagrams
 anchors representing 3-20
 navigation from 3-8
 symbol for 3-16, B-4
 validating 9-4
Offpage on Other Diagram command
 (SCE) 7-8, 7-22
Offpage on This Diagram command
 (SCE) 7-8, 7-22
offpage processes in DFE diagrams
 anchors representing 3-20
 navigation from 3-8
 symbol for 3-17, B-5
 validating 9-4
OMS Repository Browser
 described 12-1, 12-2
On Selection command 1-10
Open command, File menu 1-9
Options dialog box
 Default Arc tab 3-19
 Display Marks tab 3-10, 4-6, 7-9
output data symbol B-20

output flag symbol B-21

P

Parameter Definition annotation note
(SCE) 7-30, 10-11, 10-12, 11-82

parameters in SCE diagrams

actual versus formal 7-15, 7-41

browsing repository for 12-4, 12-11

C storage classes 7-29

data types 7-10, 7-31, 7-40, 10-4

display marks 7-10

formal 10-9

hiding/showing labels 7-47

input data B-17

input flags B-18

input/output data B-18

moving 7-17

navigation from 7-7

output data B-20

output flags B-21

ParamType display mark 7-32, B-29

representing 7-15 to 7-17

reverse engineered 11-63

symbols for 7-15, B-17, B-18, B-20, B-21

validating 9-9

ParamType display mark (SCE) 7-10,
7-31, B-29

Parent command (DFE) 3-7, 3-8

Parent command (DSE) 4-5

Parser Preprocessor Options dialog
box 11-15

parsing source code (RE)

aborting parser 11-10

C dialects in specifiers file 11-6, 11-19

compiler directives 11-16, 11-17, 11-18

described 11-2, 11-5, 11-6

embedded languages 11-19 to 11-22

errors file 11-23

file problems 11-10

file specification 11-10 to 11-11

include file specification 11-18

incremental mode 11-8, 11-9, 11-11

makefile options 11-13

Makefile Reader 11-8, 11-12

Parse Source Code command 11-4, 11-9

Parse Source Code dialog box 11-6,
11-9

parser options 11-7

PC features 11-8, 11-11

performance 11-8

preprocessing source files 11-23

preprocessor options 11-8, 11-15

starting parser 11-9

troubleshooting 11-22 to 11-24

version control systems 11-11

pattern matching in C Code

Browser 12-21

Pdl display mark (SCE) 7-10, 7-43, B-29

PDL files, *See* Program Design Language
files

PDM, *See* Persistent Data Model

Perform Check Only option (Generate C
Code dialog) 10-26

Persistent Data Model

application types and 1-13

querying repository for PDM

types 12-2, 12-6, 12-8

storing information as persistent
data 1-13

Pindex display mark (DFE) 3-11, 3-29,
B-25

pointer calls in FCE diagrams

representing 8-8

symbol for B-13

Populate Candidate Files With button

Select Files to Search dialog box 11-49,
12-23

printing

C Code Metrics reports 14-37

error messages 9-12

Print command, File menu 1-9

suggested diagram width for 11-38

procedural symbols in SCE diagrams

reverse engineered 11-58

using 7-18 to 7-19

process activations in Cspec tables

described 6-7

editing cells 6-19

-
- navigation from 6-13
 - Process Activation Matrix command (DFE) 3-7
 - process activation matrixes 6-26
 - Process Activation Table command (DFE) 3-7
 - process activation tables 6-27
 - representing 6-4
 - validating 9-8
 - process index, changing 3-40
 - Process on Data Flow Diagram command (CSE) 6-11
 - process specifications
 - annotations for 3-9, 3-28
 - creating 3-27 to 3-31
 - DFE decompositions versus 3-24, 3-27
 - example 3-27
 - generating 2-8, 3-9, 3-29
 - modifying QRL script for 3-30
 - viewing 3-31
 - processes in Cspec tables
 - described 6-6
 - editing labels 6-19
 - navigation from 6-11
 - process combinations 6-7, 9-8
 - Process on Data Flow Diagram command (CSE) 6-11, 6-13
 - representing 6-4
 - validating 9-7
 - processes in DFE diagrams
 - browsing repository for 12-4, 12-11
 - collapsing 3-43
 - decomposing 3-13, 3-24
 - defining 3-24
 - display marks 3-11, 3-29, B-25
 - exploding 3-46, 3-46 to 3-50
 - indexes 3-40 to 3-43
 - initial process 3-22
 - naming 3-12
 - primitive 3-24, 3-27
 - process specifications 3-9, 3-27 to 3-31
 - symbol for 3-16, B-5
 - validating 9-4, 9-5
 - Program Design Language files
 - creating PDL annotations 7-42
 - creating PDL script 7-45
 - described 7-41
 - generating 2-8, 7-44
 - parameters in 7-15
 - viewing 7-46
 - program modules in SCE diagrams
 - browsing repository for 12-4, 12-11
 - C storage classes 7-29
 - calls 7-11, 7-18
 - defining 7-37 to 7-40
 - described 7-11
 - display marks 7-10, 7-32, B-28, B-29
 - drawing 7-14
 - hiding/showing lexical includes 7-47
 - lexically included 7-10, 7-12, 7-32
 - navigation from 7-7
 - PDL specifications 7-10, 7-12, 7-41 to ??, 7-46, ?? to 7-46
 - renaming, effects on dependent objects 13-16
 - return types 7-10, 7-31, 7-40, 10-4, 10-11
 - reverse engineered 11-57
 - scoping to directories and files 7-10, 7-27 to 7-29
 - symbol for 7-11, B-19
 - validating 9-9
 - Properties command
 - Data Structure Editor 4-14, 4-18, 4-26
 - Structure Chart Editor 7-24
 - Pspec annotation note (DFE) 3-28
 - Pspecs, *See* process specifications 3-27
- ## Q
- qualifications (DFE), removing from model 3-50
 - queries, *See* browsing
 - Query and Reporting Language scripts
 - C code generation 10-2
 - generating BNF files (DSE) 4-40
 - generating PDL files (SCE) 7-45
 - generating process specifications (DFE) 3-30
 - Query menu (SE Browser) 12-6
-

R

re_stdfuncs file 11-40

read/write

- access modes to global data 7-34
- parameters in SCE diagrams 7-15

real-time extensions to structured analysis 1-3, 1-5, 5-1

Recompile Relationships command (SE Browser) 12-12

Reduce to Selected Rows command (SE Browser) 12-12

Remove All Qualifications in Diagram command (DFE) 3-10, 3-51

Remove button

- Select Files to Search dialog box 11-50, 12-23

Remove Previously Generated Code option (Generate C Code dialog) 10-25, 10-28

Remove Selected Files from RE Semantic Model command 11-25

Rename Hierarchy command (DFE) 2-10, 3-38 to 3-40

Rename Hierarchy dialog (DFE) 3-39

Rename Object Systemwide command
See also renaming

Check Impact button on Rename dialog 13-11

described 1-15, 13-1, 13-3, 13-8

effects on dependent objects 13-15

errors 13-15

objects that can be renamed 13-10

Rename dialog and options (diagram/table objects) 13-13 to 13-15

renaming objects from diagrams or tables 13-11

renaming related objects 13-13, 13-16

renaming SEDirectory and SEFile objects 2-9, 13-12 to 13-13

when to use 13-2

renaming

- See also* Rename Object Systemwide command

by retyping symbol/cell labels 13-1, 13-2, 13-4 to 13-8

data flow decomposition

hierarchies 2-10, 3-38 to 3-40

objects in repository 2-9

Rename Object Systemwide command
versus retyping labels 13-1, 13-2

to clone objects 13-7

Replace command, Edit menu 3-19

reports

Analysis Review 2-8

C Code Metrics 1-8, 11-3, 11-5

described 14-1 to 14-30

Design Review 2-8

dialog boxes 14-31 to 14-37

generating 14-31 to 14-37

repository

browsing 1-15, 12-2 to 12-12

described 1-12, 1-13

object mapping 1-14, A-1 to A-9

object references, described 1-14, 13-2

querying 12-6 to 12-10

renaming stored objects 1-15

unreferenced objects, deleting 13-7

requirements

allocating with OAE 1-11

allocating with RTE 1-10

Requirements Table Editor

described 1-7, 1-10

StP Desktop commands 2-5

Reset button

Parse Source Code dialog box (RE) 11-8

Return Type command (SCE) 7-7, 7-40

Reverse Engineering

browsing to code or model 11-54

C Code Browser and 11-26, 11-44

commands 11-4

comment extraction 11-2, 11-5, 11-27 to 11-33

described 1-8, 11-1 to 11-3

diagram generation 11-2, 11-4, 11-34 to 11-43

directory structure 11-83 to 11-86

-
- interpreting generated
 - diagrams 11-56 to 11-81
 - maintaining/updating design
 - model 11-44 to 11-56
 - modifying code or diagrams 11-2
 - overview 11-2 to 11-6
 - parsing source code 11-2, 11-5,
11-6 to 11-24
 - procedure overview 11-4
 - removing contents from semantic
model 11-25
 - reports 11-3, 11-5, 14-22
 - semantic model file listing 11-24
 - semantic model locks 11-26
 - starting 11-4
 - StP Desktop commands 2-8, 11-4
 - synchronizing model with source
code 11-2, 11-5, 11-45 to 11-56
 - validating semantic model 9-14
 - Reverse Flow command (DFE) 3-10, 3-55
 - root nodes in DSE diagrams
 - creating 4-14
 - display marks for B-27
 - identically named 4-22
 - partial structures scoped to 4-24
 - scoping 4-22 to 4-26
 - RTF (Microsoft Word) format 9-12
- S**
- Save As command 1-9, 4-15
 - Save command 1-9
 - Save Differences button
 - Synchronization Differences dialog box
(RE) 11-53
 - SCE menu 7-8
 - SCE, *See* Structure Chart Editor
 - Scoped Decomposition command
(DSE) 4-5, 4-15
 - scoping
 - child to parent in DSE diagrams 4-21
 - modules and global data in SCE
diagrams 7-10, 7-27 to 7-29
 - root nodes in DSE
diagrams 4-22 to 4-26
 - supported programming language
capabilities 1-3
- SE Browser**
- Browse menu 12-6
 - browsing repository 12-10 to 12-12
 - creating OMS queries from QBE
entries 12-8
 - cut and paste query construction 12-7
 - described 1-7, 12-1 to 12-2
 - describing objects to search for 12-4
 - editing hierarchies, effect of 12-12
 - Execute Query dialog box 12-9
 - executing OMS queries 12-6 to 12-10
 - GoTo menu 12-6, 12-10
 - hiding/showing table sections 12-5
 - manipulating contents of 12-12
 - menus 12-5
 - object ids 12-4
 - OMS query language queries 12-8
 - pattern matching 12-7, 12-8
 - Query menu 12-6
 - Query-By-Example rows 12-4, 12-7
 - rows, displaying/clearing 12-12
 - SE menu 12-6, 12-12
 - Sort menu 12-6
 - starting 2-9, 12-5
 - table example 12-2
 - table sections 12-3
 - targets 12-11 to 12-12
 - using 12-2 to 12-5
- SE File Definition annotation note 10-20
 - SE File Definition annotation note
(DSE) 10-5
 - SE File Definition annotation note
(SCE) 10-4
 - SE menu (SE Browser) 12-6, 12-12
 - searching, *See* browsing
 - SEDirectory objects
 - annotating 2-9
 - C code output directory 10-22
 - display mark (DSE) 4-7, 4-22, B-27
 - display mark (SCE) 7-10, B-29
 - renaming 13-1, 13-12
 - scoping in DSE diagrams 4-22, 4-24,
4-26
-

-
- scoping in SCE diagrams 7-27
 - StP Desktop commands 2-5
 - SEFile objects
 - annotating 2-9
 - C code output files 10-22
 - display mark (DSE) 4-7, 4-22, B-27
 - display mark (SCE) 7-10, B-29
 - renaming 13-1, 13-12
 - scoping in DSE diagrams 4-22, 4-24, 4-26
 - scoping in SCE diagrams 7-27
 - StP Desktop commands 2-5
 - Select Files to Search dialog box
 - C Code Browser 12-22
 - Reverse Engineering 11-48 to 11-50
 - Selected Files scrolling list (Select Files to Search dialog) 11-49, 12-23
 - selections in DSE diagrams
 - described 4-12
 - renaming, effects on dependent objects 13-16
 - reverse engineered 11-66
 - symbol for 4-12, B-8
 - validating 9-6
 - selections in SCE diagrams
 - representing 7-18
 - reverse engineered 11-58, 11-59
 - symbol for 7-18, B-21
 - semantic checks
 - applying from an editor 9-11
 - applying from StP Desktop 9-11
 - applying selectively 2-9, 9-12 to 9-13
 - checking entire Cspec 9-7
 - command descriptions 9-10
 - Control Specification Editor 6-30, 9-7 to 9-8
 - Data Flow Editor 3-56, 3-57, 9-3
 - Data Structure Editor 4-41, 9-6
 - described 9-2
 - Flow Chart Editor 8-15
 - navigating from error to source 9-14
 - State Transition Editor 5-15, 9-6
 - Structure Chart Editor 7-48, 9-9
 - types of 9-2 to 9-9
 - semantic model
 - browsing for C constructs 11-5, 12-1, 12-13 to 12-38
 - C Code Browser locks 12-37
 - checking for corruption of 11-27
 - checking lock status of 11-6, 11-26
 - creating, *See* parsing source code (RE)
 - listing files in 11-24
 - removing contents from 11-25
 - Reverse Engineering locks 11-26
 - validating 9-14
 - sequences in DSE diagrams
 - described 4-11
 - renaming, effects on dependent objects 13-16
 - reverse engineered 11-66
 - symbol for B-9
 - validating 9-6
 - Set Column Values from Data Definition command (CSE) 6-14, 6-18, 6-21
 - Set Label command (CSE) 6-18, 6-19, 6-21
 - Set/Change Focus button (C Code Browser) 12-15, 12-19, 12-22
 - Show All Children command (DSE) 4-6, 4-36
 - Sort menu (SE Browser) 12-6
 - source code, *See* C code, semantic model
 - Source Code Definition command
 - Data Structure Editor 4-5
 - described 12-40
 - Flow Chart Editor 8-6
 - Structure Chart Editor 7-7, 7-8
 - source code navigation
 - described 12-38
 - setting default editor 12-39
 - with a semantic model 12-40
 - without a semantic model 12-41
 - Source Code References command (SCE) 7-7, 7-8, 12-40
 - source_editor ToolInfo variable 12-39
 - spline symbol 3-5, 4-4, 5-4, 7-5, 8-4
 - See also* arcs
 - Split Flow command (DFE) 3-10, 3-52
 - split flow symbol B-6
-

-
- starting
 - Control Specification Editor 6-9
 - Data Flow Editor 3-4
 - Data Structure Editor 4-3
 - Flow Chart Editor 8-3
 - SE Browser 2-9
 - State Transition Editor 5-3
 - StP Desktop 2-3
 - Structure Chart Editor 7-4
 - State Event Matrix command 3-8, 5-6
 - state event matrixes 5-14, 6-15, 6-28
 - State on State Transition Diagram
 - command (CSE) 6-12
 - State Transition Diagram command 3-7, 6-12, 6-13
 - state transition diagrams
 - See also* State Transition Editor
 - anchor symbol 5-7
 - arc symbol 5-9
 - control specifications and 6-2
 - creating 5-7 to 5-12
 - creating associated control specification tables 5-14
 - described 1-5, 5-1, 5-7
 - drawing initial states 5-7
 - event/action bar symbol 5-10, 5-12
 - example of 5-12
 - filters to hide event/action bars 5-13
 - moving event/action bars 5-12
 - navigations 5-4 to 5-6
 - object mapping to application/PDM types A-8
 - representing events and actions 5-10 to 5-12
 - representing states and transitions 5-8
 - state symbol 5-7, 5-8
 - validating 5-15, 9-6
 - State Transition Editor
 - See also* state transition diagrams
 - Control Specification Editor and 5-1
 - Data Flow Editor and 1-4, 5-1
 - described 1-7, 5-1, 5-2
 - filters 5-13
 - GoTo menu 5-4
 - Hatley/Pirbhai methodology 1-5
 - starting 5-3
 - StP Desktop commands 2-4
 - supported methodology 1-3
 - symbols 5-4, B-22 to B-24
 - validating models 5-15
 - State Transition Table command 3-7, 5-6
 - state transition tables 5-14, 6-15, 6-29
 - states in Cspec tables
 - balancing with STE diagrams 6-30
 - described 6-7
 - editing labels 6-19
 - navigation from 6-12
 - representing 6-4
 - validating 9-8
 - states in STE diagrams
 - defining in Cspec tables 6-1
 - described 5-8
 - representing 5-8
 - symbol for B-23
 - validating 9-6
 - static variables, reverse engineering 11-66
 - STE, *See* State Transition Editor
 - Storage Class annotation item (SCE) 7-30, 10-12, 10-14, 11-82
 - Storage Class property (SCE) 10-4, 10-12, 10-14
 - storage classes, *See* C storage classes
 - stores, *See* data stores
 - StP Desktop
 - commands summary 2-7 to 2-10
 - described 1-7, 2-1
 - model categories 2-4
 - starting 2-3
 - StP Repository, *See* repository
 - StP/Structured Environment
 - features 1-6, 1-16
 - introduction to 1-1 to 1-16
 - supported methodologies 1-2, 1-5, 1-6, 3-3
 - using the editors 1-8
 - structure chart diagrams
 - See also* Structure Chart Editor
 - annotations for C code generation 10-3
-

-
- browsing 12-3
 - calling relationships, representing 7-18
 - creating 7-11 to 7-22
 - data and control information,
 - representing 7-15
 - described 1-6, 7-1, 7-11
 - display marks 7-9 to 7-10, B-28
 - drawing 7-13
 - filters to hide/show objects 7-47
 - generating C code from 10-6 to 10-14
 - hierarchical relationships 11-56
 - loop constructs 11-58
 - navigations 7-6 to 7-8
 - object mapping to application/PDM
 - types A-9
 - object properties 7-25 to 7-30
 - offpage connectors for
 - simplifying 7-20 to 7-22
 - ordering objects in 7-13
 - procedural symbols 7-18 to 7-20
 - Program Design Language (PDL) files
 - for 7-41 to 7-46
 - reports 14-12, 14-23, 14-25, 14-28, 14-29
 - reverse engineered 11-2, 11-56 to 11-66
 - root modules 7-11
 - validating 7-48, 9-9, 9-14
 - Structure Chart Editor
 - See also* structure chart diagrams
 - Check Semantics Selectively
 - dialog 9-12
 - Data Structure Editor and 1-5
 - described 1-7, 7-1, 7-2
 - GoTo menu 7-6
 - Properties dialog 7-23 to 7-27
 - SCE menu 7-8
 - starting 7-4
 - StP Desktop commands 2-4, 2-7, 2-8
 - supported methodology 1-3
 - symbols 7-4, B-16 to B-21
 - Yourdon/Constantine
 - methodology 1-6
 - Structure Chart Library Module command (FCE) 8-6
 - Structure Chart Module command (FCE) 8-6
 - structure declarations, reverse
 - engineering 11-66
 - Structure Tag property (DSE) 4-20, 10-5, 10-17
 - structure tags, *See* C structure tags
 - structured analysis
 - Data Flow Editor and 3-1
 - Data Structure Editor and 4-1
 - real-time extensions 5-1, 6-1
 - supported methodology 1-3, 1-5, 1-6
 - structured design
 - Data Structure Editor and 4-1
 - Flow Chart Editor and 8-1
 - Structure Chart Editor and 7-1
 - supported methodology 1-3, 1-6
 - structured engineering 1-1, 1-2
 - Structured Model (Desktop category) 2-5
 - subroutines, representing in SCE
 - diagrams 7-11, 7-32
 - switch statements in FCE diagrams
 - reverse engineered 11-73
 - symbol for B-14
 - symbols
 - Data Flow Editor 3-5, B-2 to B-6
 - Data Structure Editor 4-3, B-7 to B-9
 - Flow Chart Editor 8-3, B-10 to B-15
 - State Transition Editor 5-4, B-22 to B-24
 - Structure Chart Editor 7-4, B-16 to B-21
 - symbol reference B-1 to B-29
 - symbol-to-object mapping A-2 to A-9
 - synchronizing model with source code (RE)
 - adding new elements to model 11-53
 - applying differences 11-52, 11-54
 - browsing graphical model 11-53, 11-54
 - browsing source code 11-53, 11-54
 - changing options 11-47
 - described 1-1, 1-8, 11-2, 11-5, 11-45
 - detected differences, described 11-50
 - displaying saved differences 11-47
 - file specification for partial
 - synchronization 11-47, 11-48
 - finding existing differences 11-47
 - options summary 11-46
-

- redrawing model with differences 11-53
- removing differences from model 11-53
- reporting differences 11-50, 11-52
- saving differences 11-53
- Synchronization Differences dialog box 11-52
- Synchronize Model command 11-44, 11-46
- Synchronize Model dialog box 11-46
- syntax checks
 - Control Specification Editor 6-30
 - Data Flow Editor 3-56
 - Data Structure Editor 4-40
 - described 9-1
 - Flow Chart Editor 8-15
 - navigating from error to source 9-14
 - State Transition Editor 5-15
 - Structure Chart Editor 7-48
- system include search directories
 - parsing source code (RE) 11-17
- System Type annotation item (DSE) 11-83
- System Type property (DSE) 4-21, 10-5

T

- Table Types command (CSE) 6-21
- tables, *See* control specification tables
- Tag annotation item (DSE) 4-7, 10-17, B-27
- Tag display mark (DSE) 4-7, B-27
- Technical Support xviii
- terminal symbol in FCE diagrams 8-13, B-14
- text editors for viewing code 12-39
- ToolInfo variables
 - re_token_file 11-20, 11-22
 - relative index 3-14
 - source_editor 12-39
 - use_offpages 3-21
- Top Function Report
 - described 14-23
 - dialog box 14-37

- transitions in Cspec tables, validating 9-8
- transitions in STE diagrams
 - described 5-9
 - events and actions associated with 5-10
 - representing 5-9
 - symbol for 5-9, B-24
 - validating 9-6
- troubleshooting
 - source code parser (RE) 11-22 to 11-24
- Type annotation item
 - in DSE diagrams 4-27
- Type command (DSE) 4-5
- Type Definition command (SCE) 7-7, 7-40
- Type property (DSE) 4-20, 10-5, 10-18
- Type property (SCE) 7-26, 7-31, 10-4, 10-11, 10-14
- Type's Definition command (DSE) 4-5, 4-10, 4-15, 4-30, 4-31
- Typedef Definition annotation note (DSE) 4-27, 4-28, 10-18, 11-83, B-27
- typedefs in DSE diagrams
 - data type 10-18
 - described 4-13
 - parsing errors (RE) 11-23, 11-24
 - reverse engineered 11-70
 - symbol for 4-13, B-9
 - TypedefLeafType display mark 4-7, B-27
 - validating 9-6

U

- Uncommented Objects Report
 - described 14-30
 - dialog box 14-37
- union declarations, reverse engineering 11-66
- unreferenced objects, deleting 13-7
- update parameters in SCE diagrams 7-15
- updating diagrams with code changes, *See* synchronizing model with source code

- Use Alternate File Name option (Generate C Code dialog) 10-23, 10-25
- Use Focus option (C Code Browser) 12-15, 12-22
- user functions in FCE diagrams
 - representing calls to 8-8
 - symbol for calls to B-15
- user include search directories
 - parsing source code (RE) 11-17
- user_re_funs file 11-40

V

- validating
 - commands for 2-9
 - control specification tables 6-30, 9-7 to 9-8
 - data flow diagrams 3-56, 9-3
 - data structure diagrams 4-40, 9-6
 - flow chart diagrams 8-15
 - models for C code generation 9-14, 10-26
 - semantic model for consistency 9-14
 - state transition diagrams 5-15, 9-6
 - structure chart diagrams 7-48, 9-9
 - system models 9-1 to 9-14
- variable argument functions 10-4, 10-9
- Variable Arguments annotation item (SCE) 10-4, 10-9, 11-82
- Verbose option (Generate C Code dialog) 10-26
- View Code dialog box
 - C Code Browser 12-24
 - Reverse Engineering 11-54
- View File option (Generate C Code dialog) 10-25
- View Generated PDL command (SCE) 7-9, 7-46
- View Generated PSpec command (DFE) 3-9, 3-31
- void terminal symbol in FCE diagrams 8-13, B-15

W

- while statements in FCE diagrams
 - reverse engineered 11-74
 - symbol for B-15

Y

- Yourdon/Constantine methodology 1-3, 1-6

