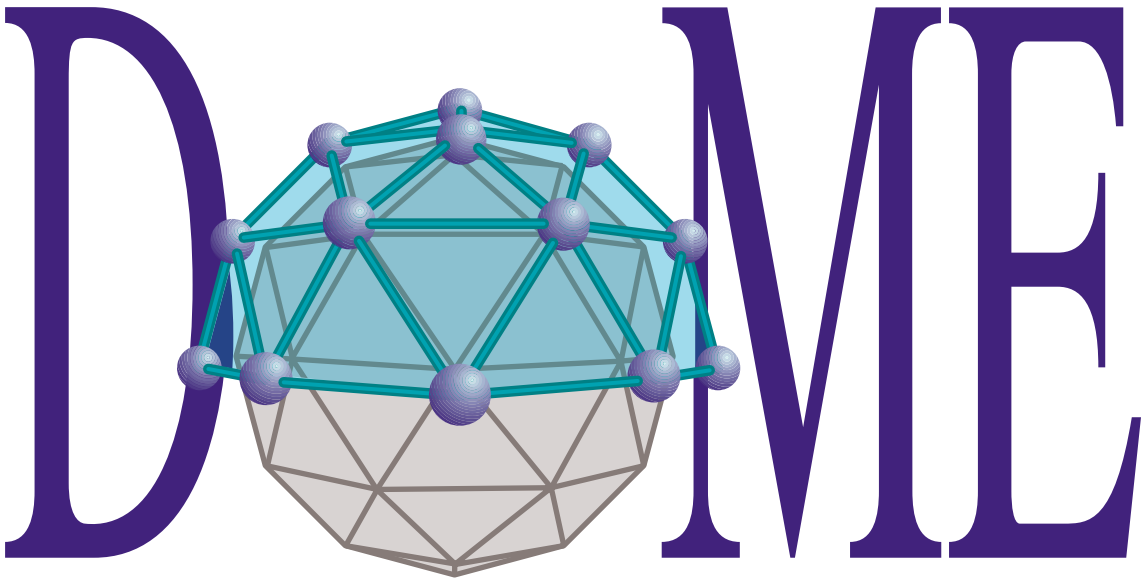


Extensions Manual

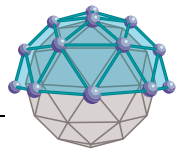
Legal Notices

This is version 5.2 of the DoME Extensions Manual.

Email: dome-info@htc.honeywell.com
Web: www.htc.honeywell.com / dome

Trademarks

Contents



...Preface

About This Guide vi
Revision History vii
Related Documents. vii
Conventions Used in This Guide viii
Appearance of Windows & Screen Elements viii
Typographic Conventions viii
The Mouse Button Dilemma ix
Mouse Button Operations x
How to Reach Us... xi

1...Introduction

. . In This Chapter 1
Extension Language 2
DoME Extension Languages 2
Projector 2
Alter 3
Inter-Operability 4
Programming Environment 4
DoME Extension Facilities. 4

2...Projector

. . In This Chapter 5
Description. 6
Diagrams 6
Nodes 7
Procedure 7
Port 7
Statement Block 7
Alter Code 8
Constant 8
Variable 8
Merge 8
Fork 8
Conditional 9
Connectors 9
Data Flow 9
Control Flow 9

3...Alter

. . In This Chapter 11
Description. 12
Extensions to R4 Scheme 12
General Purpose Extensions 12
User Interfacing 12
GrapE Interface 12
Object Oriented Programming 15
Operating System Interface 17
Dictionaries 17

4...Programming Tools

. . In This Chapter 19
Overview 20
Projector Environment Viewer 20
Ready Nodes 21
Data Flows 21
Control Flows 21
Alter / Projector Browser 21
Structure 21
Types 22
Methods 22
Description 22
Alter Evaluator 23
Structure 23
Definitions 24
Environments 24
Programs 25
Expressions 26
External Representations 26
Alter Environment Viewer 26
Structure 26
Activation Stack 27
Bindings 27
Alter Object Inspector 28
Structure 29
Alter List Inspector 29
Structure 29

5...Programming in Projector

. . In This Chapter 31
Code Generator Example 32
Initial Setup 33
Coding Scenario 33

6...Programming in Alter

. . In This Chapter 47
Opening an Evaluator 48
Evaluating Expressions 48
Entering a Program 49
Evaluating Definitions 49
Saving your Alter Program 50
Closing an Evaluator 50
Opening an Alter Program File 50
Opening a Program File From an Evaluator 50
Printing 51

7...Plug-In Functions

. . In This Chapter 53
Function Calling Mechanism 54
Function Entry Point 55
Registering Functions 55
Examples 57

Count all the nodes 57
Count the semantic nodes 58
Summarize nodes and arcs 60
Summary Report 61

8...Print Drivers

. . In This Chapter 65
Description 66
Driver Functions 67
GraphicsContext Operations 71
Color Operations 71
The Procedure Map 71
Registering a Driver 72
Example Driver 73

9...Document Generators

. . In This Chapter 79
Document Markup 80
Document Generators 82
Stylesheets 83
Text Formatters 83
The Generation Process 83

10...SGML Generators

. . In This Chapter 85
Query Operations 86
Registering a Generator 87
Example Generator 88
Choosing a DTD 88
Creating node types 89
Writing the generator 90

11...Text Formatters

. . In This Chapter 93
Overview 94
Formatting Operations 95
Font Operations 96
Text-Style Operations 96
Quantity Operations 97
Registering a Formatter 98

12...Stylesheets

. . In This Chapter 101
Overview 102
Registering a Stylesheet 102
Example Stylesheet 103

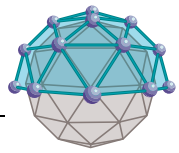
A...Scheme

. . In This Appendix 105
Selected References 106
Unimplemented R4 Scheme Elements 106

Index

. 107

Preface



This preface includes the following topics...

- About this guide
- Revision history
- Related documents
- Conventions used in this guide
- How to reach us

About This Guide

This manual describes the extension system available within the DoME toolset that allows the user to extend the capabilities of the toolset without requiring source code to it. The manual describes the extension languages that are available and the facilities that make use of the extension languages.

This guide includes:

Chapter 1 Introduction — *A description of an extension language, the extension languages available in DoME, and the facilities built around the extension languages*

Chapter 2 Projector — *How to use DoME's graphic extension language*

Chapter 3 Alter — *Describes DoME's textual extension language*

Chapter 4 Programming Tools — *Describes the DoME extension system programming tools such as the browsers and debuggers*

Chapter 5 Programming in Projector — *An example of using Projector to create a plug-in function*

Chapter 6 Programming in Alter — *An example of using the DoME Evaluator to execute simple Alter routines*

Chapter 7 Plug-In Functions — Describes how to integrate plug-in functions to extend the capabilities of tools

Chapter 8 Print Drivers — *Describes how to integrate print drivers to allow printing of graphs in different formats*

Chapter 9 Document Generators — *Describes the document generation approach taken by the DoME toolset*

Chapter 10 SGML Generators — *Describes the SGML generation portion of the document generator*

Chapter 11 Text Formatters — *Describes the Text Formatter portion of the document generator*

Chapter 12 Stylesheets — *Describes the Stylesheet portion of the document generator*

Appendix A Scheme — *Contains references to Scheme related documents*

Revision History

Table 1 describes the evolution of this document. When you communicate with us, please identify the documentation and software versions you are using.

Table 1 DoME Extension Manual Revision History

Publication Number	Rev.	Date	Description
TRG-M99-002	A	1/99	Updated to DoME Version 5.2
TRG-M98-002	A	8/98	Updated to DoME Version 5.1
TRG-M97-002	A	3/97	Original manual updated and reorganized to support DoME version 5.0

Related Documents

This guide is your primary reference to the DoME extension system. Other documents describing DoME capabilities and disciplines include...

DoME Guide Manual

The primary “how to” and reference for the DoME core tool-set.

Alter Programmer’s Reference Manual

Technical description of *Alter*, DoME’s resident variant of the *Scheme* extension language. A general-purpose programming language, Alter can be used to write DoME code generators, document generators, and a host of other specialized tools.

Conventions Used in This Guide

Throughout this guide and related documents, various conventions are used to identify technical terms, computer-language constructs, mouse and keyboard operations, and window / screen element appearance.

Appearance of Windows & Screen Elements

The windows and screen elements shown in this guide depict what you would typically see in the Microsoft Windows 95 environment. If you are running DoME on the Macintosh, UNIX, or Windows NT platform, your actual DoME windows and screen elements will look different with respect to the title bar, buttons, menus, and other desktop widgetry.

☞

The important point we'd like to make here is that DoME is platform-independent, and performs identically on UNIX, Macintosh, and all supported flavors of Windows...regardless of the desktop decor and widgetry used.

Typographic Conventions

In this guide and related documentation, you will encounter various items distinguished by specific fonts or symbols:

Table 2 Formatting Conventions

Example	Description
dome_dir	Variable—Indicates an element for which you must supply a value
`~/model.dome`	Literal text—Often used for file path-names and operating system commands
Transcript show: 'Hello'.	Code fragments
<RETURN>, <ESCAPE>, <CTRL-C>, <SELECT>, <OPERATE>	Key names/mouse button names—Brackets and names are not to be entered literally
APPLY, REVERT, FILE:NEW, LAYOUT:ARC:ROUTEARC	Widgetry selections—Button or menu selections are indicated by name...sub-menus are delimited by colons
☞	Notes, cautions, warnings—Indicated by this symbol pointing to the text

The Mouse Button Dilemma

Since DoME runs under multiple platforms, we're obliged to deal with three distinct breeds of mice in a democratic manner: the one-, two-, and three-button varieties.

In the case of one-button mice, for example, it would be confusing (if not insulting) to refer to the <LEFT>, <MIDDLE> and <RIGHT> buttons. (After all, quantity seldom means quality.)

To bypass the potentially serious problem of button envy between our three breeds of mice, we've decided to follow the convention that *ParcPlace-Digitalk* uses in its documentation—to use mouse button names that are generically descriptive: <SELECT>, <OPERATE>, and <WINDOW>. Use the descriptions below to identify your specific mouse button(s) name(s).

Table 3 Mouse Button Names

<SELECT>	*Select* a window, object, or menu item; position the pointer or highlight text
<OPERATE>	Bring up a menu of *operations* that are appropriate for the current view or *selected object; i*n certain modes, this button may have *special* functions
<WINDOW>	Bring up the menu of actions that can be performed in *any* DoME window (except dialogs); also used to *cancel* certain operations like arc creation and node placement

1 *One-button mouse*—The lone mouse button is <SELECT>. To access the <OPERATE> menu, press the <OPTION> key and click the mouse button. To access the <WINDOW> menu, press the <COMMAND> key and click the mouse button.

2 *Right-handed two-button mouse*—The left and right buttons are <SELECT> and <OPERATE>, respectively. To access the <WINDOW> menu, press the <CTRL> key and click the <OPERATE> button simultaneously. (If you operate your mouse left-handed, these buttons may be reversed.)

3 *Right-handed three-button mouse*—The correspondence is from left to right: Left = <SELECT>; Middle = <OPERATE>; Right = <WINDOW>. (If you operate your mouse left-handed, these buttons may be reversed.)

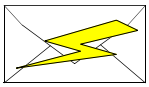
Mouse Button Operations

The following table describes the actions you can perform with your mouse buttons in the DoME environment.

Table 4 Mouse Button Operations

When you see..	Do this...
<CLICK>	Press and release the <SELECT> mouse button.
<DOUBLE-CLICK>	Press and release the <SELECT> mouse button twice in rapid succession without moving the mouse pointer.
<SHIFT>-<CLICK> <CTRL>-<CLICK> <META>-<CLICK>	While holding down the <SHIFT>, <CTRL>, or <META> key, press and release the <SELECT> mouse button.

How to Reach Us...

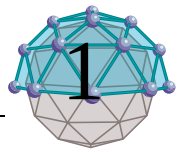


We'd love to get your feedback on the DoME software and documentation. Feel free to drop us a note...

Email: dome-info@htc.honeywell.com
Web: www.htc.honeywell.com/dome

With all communications please include the version number of the software and/or documentation, as well as the type of computer and operating system you are using.

Introduction



. . In This Chapter

This chapter describes...

- What is an extension language (page 2)
- What extension languages are available from within DoME (page 2)
- Facilities within DoME that make use of the extension languages (page 4)

Extension Language

An extension language is the facility by which a Component / Infrastructure Developer, as described in the DoME Guide introduction, implements model transformation and analysis tools. In addition to providing traditional programming capabilities, an extension language usually provides extended capabilities required by the tool making use of the extension language. DoME makes use of two extension languages that are fully inter-operable.

DoME Extension Languages

DoME's two extension languages are named Project and Alter and are the basis of the DoME extension system. This manual describes Projector / Alter with examples covering Plug-In Functions, Print Drivers, and Document Generators.

With model-based development tools such as DoME, you first build a formal, graphical description of the system your customer wants, then you generate artifacts -- including documentation and source code -- automatically from your model.

The Projector / Alter extension system sprouted out of our desire to give our users a way to write generators themselves. We began with the premise that the user should be able to specify a translator graphically. This makes the feature a very natural extension to DoME, and reduces the learning curve for most users. Since some (but not all) translators were likely to have a prominent algorithmic component, we quickly added the concept of a textual face to the language. Consequently, there are two mutually compatible ways of specifying artifact generators in DoME; Projector is the graphical language and Alter is the textual language.

Projector

Projector is a visual programming language (VPL) that resembles a data flow or control block-diagraming language with operators connected by wires[1]. Projector operators are implemented as subdiagrams where each subdiagram contains a refinement of the operator being defined. Projector directly supports graph execution. Users can graphically specify an operation and then immediately execute it. Execution of a Projector diagram proceeds in a data flow-like fashion in which operator calls can be made as soon as they have data available on all inputs. The Projector debugger allows the user to see the operation executed in an animated fashion making it easier to check the correctness of the operation.

1 In fact, the first prototype of Projector was based on an existing data flow language called ControlH used to specify control laws for embedded avionics controls applications.

Projector is a DoME tool, therefore Projector programs are created using the standard DoME editing functions. If you have used other DoME tools, Projector will probably feel very familiar to you and provide you with an intuitive and natural programming environment. If you haven't used a DoME tool before see the DoME Guide for information on the basic steps involved in creating and editing a DoME model.

Alter

Alter is a nearly complete implementation of the R^4 Scheme language[2]. Alter contains various extensions to standard Scheme in order to enhance its functionality as a DoME extension language. These extensions include user interfacing primitives, operating system primitives, remote procedure call capabilities, model querying capabilities, and additional support for object-oriented programming.

As a variant of the Scheme language, Alter provides programmers with a simple, yet powerful, tool for writing extensions. Some of the advantages of a Scheme-based language are simple syntax, straightforward semantics, extensibility, publicly available formal specification, and adaptability to many programming paradigms.

Alter provides a simple and elegant tool for expressing algorithms. If you are unfamiliar with the Scheme language please refer to the selected references in Appendix A: Scheme on page 105. Scheme is a member of the family of Lisp-like languages, a set of programming languages known for their simplicity and ease of use.

> *This is one of the great advantages of Lisp-like languages: They have very few ways of forming compound expressions, and almost no syntactic structure. All of the formal properties can be covered in an hour, like the rules of chess. After a short time we forget about syntactic details of the language (because there are none) and get on with the real issues -- figuring out what we want to compute, how we will decompose problems into manageable parts, and how we will work on the parts.*[3]

You can learn Alter in a short amount of time, leaving you more time to concentrate on the important task of writing extensions that will make your modeling environment more productive and useful.

2 Clinger, W. and Rees, J. (eds), "Revised^4 Report on the Algorithmic Language Scheme"

3 Abelson, Harold, and Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, New York: The MIT Press, 1985, p. xvi.

Inter-Operability

The Projector and Alter languages overlap significantly in their data and control architectures. In fact, each can transfer data and control into the other. Alter operators (procedures in Scheme parlance) can call operators defined in Projector and vice versa. If a Projector operator calls an Alter operator, execution on that operator does not start until all of the input wires (parameters) have been satisfied with available data. If an Alter operator calls a Projector operator, that operator can proceed immediately because Alter code must supply values for all inputs simultaneously.

The two languages have much of the same functionality but, they each have a distinctive programming style. Projector's strength is describing flows of data and transfer of control, but it is very weak at describing the intricate algorithmic details that often lurk in the bowels of any computer program. Alter's strength is simply and concisely describing algorithms, but due to the functional nature of scheme programs they are often fragmented, and getting a feel for the flow of information and control within an Alter program can be a grueling task. The integration of Projector and Alter allows the user to take advantage of the strengths and avoid the weaknesses of each language.

Programming Environment

Whether a translator is implemented in Projector or Alter there are numerous capabilities available to the user that directly support the development of a translator. These capabilities include a system browser, a suite of predefined operators, a debugging / tracing facility, and an error reporting mechanism. These capabilities provide for a very functional iterative development environment.

DoME Extension Facilities

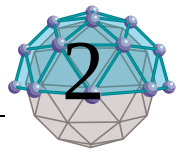
DoME provides several facilities that make use of the extension languages including Plug-In Functions, Print Drivers, and Document Generators. Each of these facilities are described in great detail in later chapters of this document.

The Plug-In Function facility allows the user to execute any Projector / Alter program simply by selecting it from the **TOOLS:PLUG-INS** menu of an editor.

The Print Driver facility allows the user to print the contents of a diagram to a file in a specialized format.

The Document Generator facility allows the user to easily create new documents.

Projector



. . In This Chapter

This chapter describes...

- The Projector Editor (page 6)
- The nodes that may be used in a Projector Diagram (page 7)
- The connectors that may be used in a Projector Diagram (page 9)

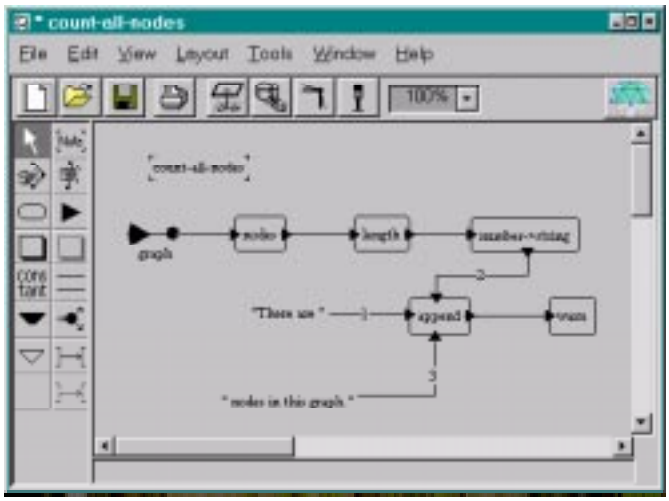
Description

This chapter assumes you are familiar with the basic features of DoME. In particular, this chapter presents Projector, a visual programming language that is the graphical part of DoME's extension languages. If you are unfamiliar with the basic operations of DoME please refer to the DoME Guide manual.

One way to think of a Projector program is to envision a flow of data through the diagram. Input data enters the diagram through input ports. Data flows transport data between ports. Procedure nodes perform operations upon data. Control flows pass control from one node in the diagram to another and can be used to synchronize operations.

Before getting started, here's a simple example of what a Projector diagram looks like. When executed it presents a dialog to the user specifying the number of nodes in a diagram. As you can see, there are several tools for creating nodes and connectors within the diagram. Each tool is described in detail in the following sections.

Figure 1 Example Projector Diagram



Diagrams

All diagrams in a Projector model represent implementations of procedures except the top most diagram which represents the starting point for the model.

In addition to the standard properties, a Projector implementation has the following additional properties:

- Break - If break is set to true then the Projector debugger is opened at the point when the implementation is to begin execution.
- Class - The name of the class to associate this implementation with. Since Projector is object oriented it is possible for a procedure to have multiple implementations

each of which handles a different class of object.

- Entry - When a Projector model is used as a Plug-In Function this value represents the name of the procedure that should be called upon start-up. This property is only applicable to the top diagram of the model.
- Load File - When a Projector model is used as a Plug-In Function the file specified by this value is loaded before execution of the entry procedure begins. The load file is a file containing Alter code. This property is only applicable to the top diagram of the model.

Nodes

There are nine different types of nodes that can be placed inside a Projector diagram. These include Procedures, Ports, Statement Blocks, Alter Codes, Constants, Variables, Merges, Forks, and Conditionals.

Procedure

A Procedure represents a call to an Alter operation, an Alter procedure, or a Projector procedure implementation. If the procedure has no implementation then the procedure must be a call to either an Alter Operation or an Alter procedure which has the same name as the procedure. If the procedure has an implementation then the call is resolved during runtime to determine if the procedure implementation is executed or an Alter operation is executed.

Procedure nodes are hierarchical and can have multiple subdiagrams. The procedures in a projector diagram are reusable so in addition to the standard editor there is a Shelf of reusable procedures that can be edited via the Shelf Browser.

In addition to the standard properties, a procedure has the following additional property:

- Category - The category is used by the Alter / Projector Browser as an organizational aide.

Port

A port is a point where either data enters or exits a node. Ports can be placed on procedures, statement blocks, and Alter code nodes.

Statement Block

A Statement Block allows for block structuring within a diagram. It prevents execution of the nodes within it until all of its input ports have data and all incoming control flows are ready.

In addition to the standard properties, a statement block has the following additional property:

- Break - If break is set to true then the Projector debugger is opened at the point when the statement block is to begin

execution.

Alter Code

An Alter Code node provides a mechanism to execute alter code from within a Projector diagram. An Alter Code node can have multiple inputs and multiple outputs. When an Alter Code node is executed, an environment is created in which the input ports' names are bound to the arguments that were passed in based strictly on the ordering of the input ports. When the code is done executing the result is passed directly to the output port if there is only one output port or the result, which is a list of associations where the key is the output port name and the value is the result for the port, is broken apart and passed to the individual output ports.

In addition to the standard properties, an alter code node has the following additional property:

- Code - The alter code to be executed.

Constant

A Constant passes the result of evaluating its value to each of its outgoing data flows.

In addition to the standard properties, a constant has the following additional properties:

- Show - If the constant has no name then the value property of the constant is normally displayed in its entirety. The show property specifies how many characters of the value property should be shown in the diagram. If zero is specified and no name is present then the entire value property is displayed.
- Value - The alter code to be executed that computes the constant. The code can be as complex as necessary or as simple as the number 1.

Variable

A Variable is essentially a global variable. Variables with the same name used in multiple diagrams represents the same value.

In addition to the standard properties, a variable has the following additional property:

- Collect - Should the data being assigned to the variable replace the data or append it to the end of the current value where the current value must be a list.

Merge

A Merge node is used to pass data on from one of many data flows to a single data flow. A merge node can execute as soon as there is data on one incoming data flow.

Fork

A Fork is used to split a data flow into two or more data flows.

Conditional

A Conditional node provides a mechanism to pass control to another node. It accepts a single input. If the input is false then those nodes connected to the false control port are sent a control token. If the input is not false then those nodes connected to the true control port are sent a control token.

Connectors

There are two different types of connectors that can be placed inside a Projector diagram. These are Data Flows and Control Flows.

Data Flow

A Data Flow is used to pass data along to connected nodes. The data can be transferred in different ways. If the data is transferred in simple mode then each piece of data is passed directly down the data flow. If the data is transferred in maintain mode then the data is kept on the data flow even after the destination has received the value. If the data is transferred in build mode then all data on the data flow is grouped into a collection before it is passed to the destination node. If the data is transferred in reduce mode then the data, which must be a collection, has each element placed individually on the data flow and passed to the destination node.

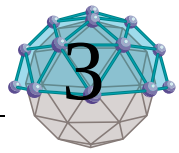
In addition to the standard properties, a data flow has the following additional properties:

- Trace - If trace is set to true then as data is added to and removed from the data flow a message is shown in the DoME transcript window stating that fact.
- Transfer - Specifies how the data is transferred to the destination node. See Above.

Control Flow

A Control Flow is used to pass control tokens to destination nodes. Nodes that have control flows entering them cannot be executed until all incoming control flows have tokens on them. After a node is executed all outgoing control flows have tokens placed on them.

Alter



. . In This Chapter

This chapter describes...

- The Alter extension language (page 12)
- The extensions add to Alter (page 12)

Description

Alter is based on the Scheme programming language. If you are unfamiliar with the basic syntax and semantics of the Scheme programming language please see the Scheme Appendix starting on page 105 for selected references. This chapter provides an overview of the Alter programming language.

Alter is a nearly complete implementation of R^4 Scheme. Most, but not all, of the features of R^4 Scheme have been implemented. The Scheme Appendix starting on page 105 contains a description of those elements not currently implemented in Alter.

Extensions to R^4 Scheme

Alter includes some extensions to R^4 Scheme. These extensions include the following:

- General purpose additions and extensions to primitives
- Extensions for user interfacing
- Extensions for manipulating GrapE structures
- Support for Object-oriented programming (OOP)
- Operating System Interface
- Dictionaries

The following sections briefly describe each of these extensions. Please see the Alter Programmers Reference Manual for more information.

General Purpose Extensions

Some primitives have been added to Alter for performance and/or convenience purposes. The semantics of some Scheme primitives have been expanded to function on objects of types other than those defined in the standard.

User Interfacing

Alter operates in a windowed, graphics-capable environment. In order to take advantage of this environment some extra primitives are included in Alter. These extra primitives include procedures to open dialogs that request information from the user.

Please see “User Requests” in the Alter Programmer’s Refernce Manual for more information on these user interface extensions.

GrapE Interface

Alter was developed in order to provide a way for users to write their own back-end generators that would produce code and documentation, or perform analysis of models. In order to provide this capability, Alter contains many primitives that allow the user to traverse, modify, and query the GrapE objects that comprise DoME models.

A complete list of GrapE Interface primitives can be found in the Alter Programmer's Reference Manual. The following section highlights the capabilities provided by these extensions.

navigation

Several primitives provide the capability to traverse a graph. These primitives allow users to move from node to node along the connectors that connect nodes and to move up and down within a hierarchical model.

Most algorithms for solving problems on a graph examine or process each node or connector. Such algorithms are often referred to as *graph traversals*. The Projector / Alter extension system provides a user with the ability to write graph traversal algorithms.

A node in a graph can be connected to other nodes via connectors. To get a list of the connectors entering the node use the `incoming-arcs` method. To get a list of the connectors leaving the node use the `outgoing-arcs` method.

Some nodes in graphs are hierarchical. To get the parent of a node use the `parent` method. To get a list of the subdiagrams of a hierarchical node use the `subdiagrams` method.

A connector in a graph usually has a direction. The node at the source of the connector is called the origin and the node at the end of the connector is called the destination. You can use the `source` method to get the source node for a connector and the `destination` method to get the destination node for a connector.

The following example demonstrates how to use the model querying functions to perform a depth-first search of a graph.

Example 1: Depth-First Search

```
(find-operation mark!)
(add-method (mark! (node) self)
   (letrec
      (  (d (get-property"userSettings"
                          (graph self)))
         (m (dictionary-ref d 'marked-nodes
'())) )
      (set! m (cons self m))
      (dictionary-set! d 'marked-nodes m)
      (set-property! "userSettings"
                     (graph self) d) ) )
(find-operation unmark!)
(add-method (unmark! (graphmodel) self)
   (letrec((d(get-property "userSettings"
self)))
```

```
        (dictionary-set! d 'marked-nodes '())
        (set-property! "userSettings" self d)))

   (find-operation unmarked?)
   (add-method (unmarked? (node) self)
      (letrec
         ((d(get-property"userSettings"
                          (graph self)))
          (m(dictionary-ref d 'marked-nodes '())))
         (if(member self m) #f #t)))

   (find-operation dfs)
   (add-method (dfs (node) self visit)
      (let
         ((o (map destination
                   (outgoing-arcs self))))
         (visit self)
         (mark! self)
         (for-each (lambda (n) (dfs n visit))
                   (select o unmarked?))))
```

modifying

The set-property! primitive provides a general facility for modifying the values of an object's properties. Primitives also exist for adding / removing nodes and connectors to / from a model and changing the display properties (such as position) of objects.

querying

A DoME model is an information model. An information model is similar to a database. The nodes in the model are similar to records in a database. The connectors are similar to relationships. Like records in a database, nodes have fields called *properties*. These properties contain information of various types. Unlike most relationships in databases, connectors can also have properties. The Projector / Alter extension system provides the user with the ability to write programs that query information models in much the same way a database would be queried for information.

To get the value of a property use the `get-property` method. To set the value of a property use the `set-property!` method. To determine whether an object has a particular property set use the `has-property-set?` method. To remove any value that is bound to a property use the `unset-property!` method.

Besides these generic querying methods, many standard properties can be queried by other predefined methods. For example the name property of any GrapEThing can be obtained with the `name` method and set with the `name-set!`

method. Please see the GrapE Interface section in the Alter Programmer's Reference Manual for a description of other standard model query methods.

Object Oriented Programming

Alter provides support for object-oriented programming (OOP). These extensions allow users to define their own types and define operations for those types. Alter's implementation of types and operations is derived from the Oaklisp language[1]. The following section highlights some of the capabilities provided by Alter's partial implementation of the Oaklisp language.[2]

Objects and Types

Everything in Alter is an object. Objects are instances of types. A type defines a set of operations that can be performed on instances of itself. The `make` procedure allows users to define new types as well as create instances of types. Also provided are primitives for querying objects for their type, subtypes, supertypes and instance variables.

In order to allow for abstraction and modular design of objects with complex behavior, a type can have multiple supertypes.

To create an object in Projector / Alter use the `make` operation. By default, the Projector / Alter system provides two `make` methods.

One method is defined on for objects of type `type`.

```
(make type) => an object
```

This method is used to make instances of the argument *type*.

The second method is defined for objects of type `meta-type`.

```
(make type ivars supertypes) => a type
```

This method is used to create new types. The new type inherits from the types in the *supertypes* list and has instance variables defined in the *ivars* list.

Operations and Methods

Methods for performing operations on an object are provided by the object's type. Methods are inherited from supertypes. Therefore, a subtype only needs to provide those methods which distinguish it from its supertypes.

1 Pearlmutter, Barak, and Kevin Lang, "The Oaklisp Language Manual"

2 This section is adapted from "The Oaklisp Language Manual".

Operations are objects in their own right. When an operation is performed on an object, the method defined for the object's type is used. In order for methods to be defined for a particular operation, the operation must exist. Operations are created with the `find-operation` operation.

```
(find-operation op-name) => an operation
```

Alter first checks the current lexical environment for a binding. If one exists and the value is an operation, Alter returns that value. If one exists and it is not an operation, an error occurs. If a user-defined binding does not exist, but a predefined binding does, Alter binds the symbol in the current lexical environment to a surrogate operation that allows the user to add methods without disrupting the space of predefined symbols; a surrogate handles calls just like a normal operation, except that it can forward calls to the predefined operation if it is given an object that falls outside of its interface range.

If neither a user-defined or predefined binding exists, Alter creates a new operation and binds it to the given symbol. The return value of find-operation is the new or existing operation (or surrogate).

Once an operation has been created the `add-method` operation can be used to define methods for the operation on types.

```
(add-method (op (type) arg-list) body)
=>procedure
```

Methods are procedures. The procedure defined by the `add-method` operation is invoked when the operation *op* is performed on an object of type *type*. The *arg-list* provides the formal arguments for the procedure and the *body* provides the expressions that make up the body of the procedure. The procedure created by `add-method` could also be created by the following `lambda` expression.

```
(lambda (arg-list) body) => procedure
```

Sometimes a method doesn't want to override the inherited method completely, but only wishes to extend its behavior. The `^super` operation allows dispatching to supertypes.

```
(^super type op self args)
```

This is just like (*op self args*) except that the method search begins at *type* rather than at the type of *self*. The argument *type* specifies which supertype the method search should begin at. This is necessary due to the existence of multiple inheritance.

Operating System Interface

R^4 Scheme provides minimal specification of interfaces to the operating system. Alter functions in an environment that provides highly portable access to the underlying operating system. By taking advantage of this operating system interface, Alter can provide some extra primitives to programmers to allow them to access the host file system in a generic way.

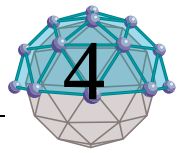
Please see the OS Interface section in the Alter Programmer's Reference Maanual for more information on these OS interface extensions.

Dictionaries

In order to provide efficient keyed access to information Alter provides a *dictionary-type*. A dictionary is like an alist except that it has an implementation other than a list. This implementation provides more efficient access to information.

Please see the Dictionaries section in the Alter Programmer's Reference Manual for more information on the *dictionary-type* and its operations.

Programming Tools



. . In This Chapter

This chapter describes...

- The Projector Environment Viewer (page 20)
- The Alter / Projector Browser (page 21)
- The Alter Evaluator (page 23)
- The Alter Environment Viewer (page 26)
- The Alter Object Inspector (page 28)
- The Alter List Inspector (page 29)

Overview

DoME provides a set of tools for developing Projector / Alter programs. The tools include the following:

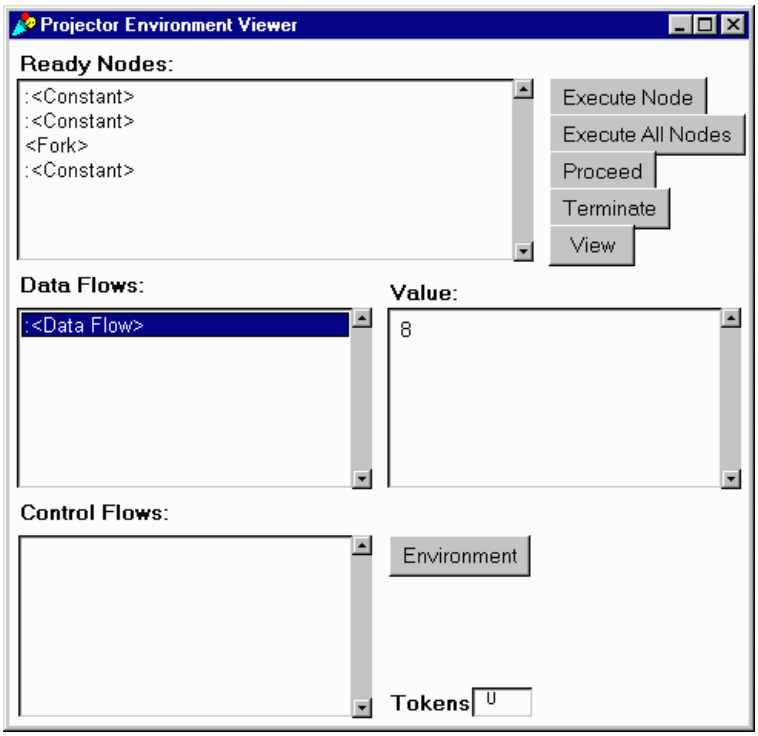
- Projector Editor (described in the Chapter “Projector” on page 5)
- Projector Environment Viewer
- Alter / Projector Browser
- Alter Evaluator
- Alter Environment Viewer
- Alter Object Inspector
- Alter List Inspector

To familiarize you with the basic features of the Projector / Alter Programming Tools, this chapter will lead you through a brief description of each of the tools.

Projector Environment Viewer

The Projector Environment Viewer provides a mechanism for examining the Projector execution environment, stepping through a diagram, and debugging a Projector program. It is opened when a break is encountered on an procedure implementation or a statement block.

Figure 2 Projector Environment Viewer



The Projector Environment Viewer is made up of three areas: ready nodes, data flows, and control flows areas.

Ready Nodes

The ready nodes area displays the nodes in the diagram that are ready to be executed. The user can select any one of the ready nodes and press the **EXECUTE NODE** button to have that node execute. Pressing the **EXECUTE ALL NODES** button causes all ready nodes to execute. Pressing the **PROCEED** button causes the view to go away and allows execution of the Projector diagram to continue. Pressing the **TERMINATE** button causes the viewer to close and terminate execution of the diagram. Pressing the **VIEW** button causes the selected ready node to be selected in the diagram in which it is contained and if that diagram is not open it is opened for the user.

Data Flows

The data flows area shows all of the data flows that are in the diagram. Selecting a data flow will cause its value to be displayed in the Value field.

Control Flows

The control flows area shows all of the control flows that are in the diagram. Selecting a control flow will cause the tokens field to display the number of tokens on that control flow. There is also an Environment button in the control flows area that opens an Alter Environment Viewer for the user.

Alter/ Projector Browser

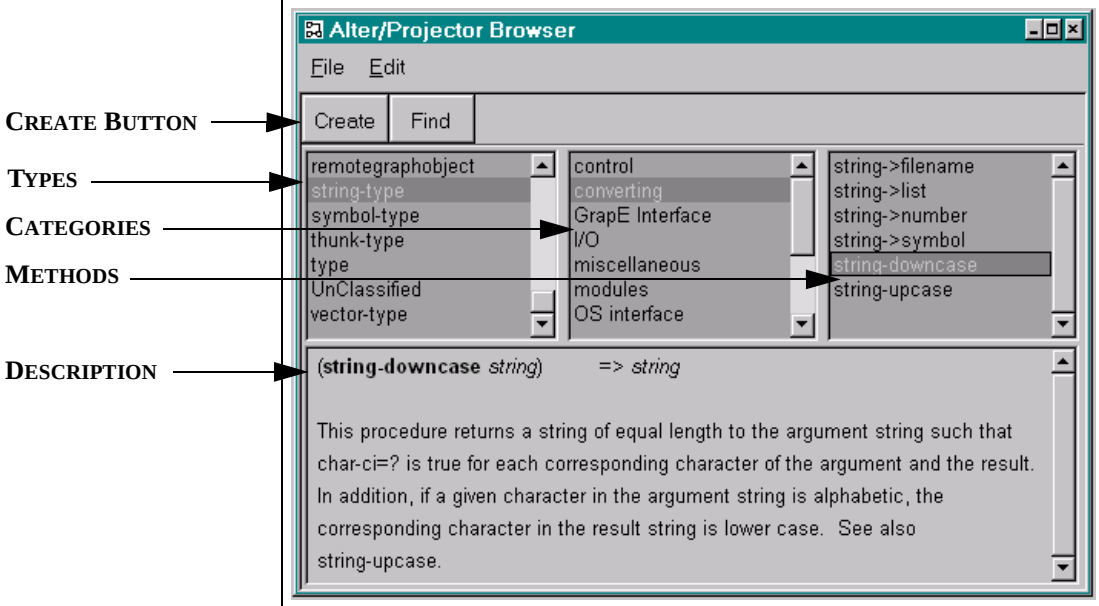
The Alter / Projector Browser offers a mechanism for users to browse types and methods that have been defined in an environment. Its capabilities include browsing classes, categories, and methods, and creating instances of methods for use in Projector diagrams.

To open a Projector Browser (more than one can be open at a time), select the Tools:Browser menu from a projector editor or click on the **BROWSER** button in an Alter Evaluator.

Structure

The Alter / Projector Browser has a title bar, a tool bar, three upper views, and one lower view as shown in Figure 3. Each view provides a lower level of detail in the environment library, ending with the description subview, which describes a single method. The **CREATE** button is used by Projector programmers to create operation calls objects in a Projector diagram. To create an operation call, press the **CREATE** button, move to a Projector diagram, and press the <Select> button at the location where the operation call should be located.

Figure 3 The Alter/Projector Browser



Types

The type of an object determines its behavior when operations are performed on it. A type specifies the behavior of an object by providing methods that are used to perform operations on that object. A single type, such as **string-type**, can respond to any number of operations. For the sake of convenience and conceptual clarity, they are placed in functional groups called categories. In the second view, all of the categories for the currently selected type are displayed. For string-type, there are twelve categories. The “converting” category is selected in Figure 3.

Methods

The third view displays all of the methods in the currently selected category. These methods are objects in their own right. They can be created dynamically, stored in data structures, returned as results of other operations, and so on.

Description

The bottom view is not a list manager like the other three. It is a read only view that provides a description of the method that is selected in the third view. The description is different depending on which programming environment was used to create the method.

Alter methods

The description for a method that was implemented using Alter is given as text. The textual description is formatted in the following way. The header provides a template for a call to the method and a description of the type of object the method

evaluates to. Method names in templates are in **boldface** type, while arguments are *italicized*. The symbol "=>' should be read "evaluates to". Thus the header line

(**stringcase-down** *string*) => *string*

indicates that the method **stringcase-down** takes one argument, a *string*, and evaluates to *string*.

Projector methods

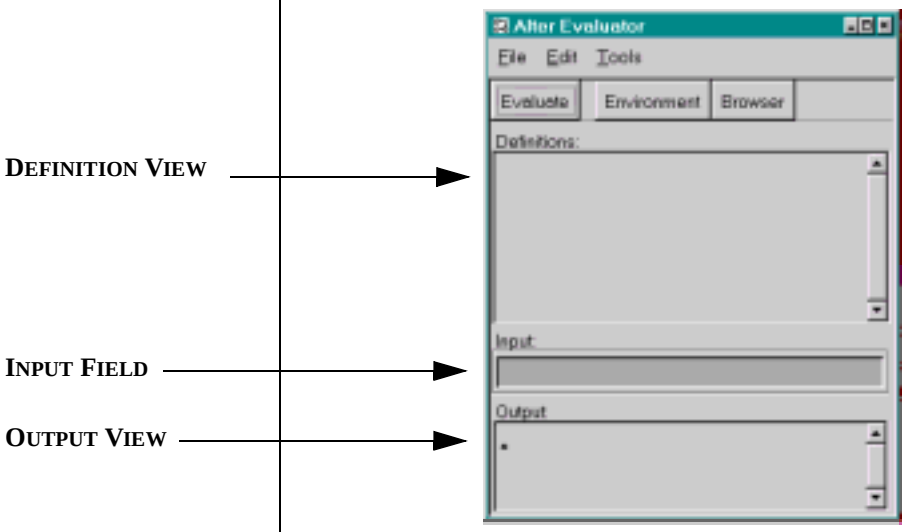
The description for a method that was implemented using Projector is displayed graphically as a Projector Procedure node. The node displayed is the same as the node displayed on the Projector shelf.

Alter Evaluator

The Alter Evaluator is a tool for writing, testing and debugging Alter programs. The evaluator also provides access to the other Alter Programming Tools.

To open a new evaluator (more than one can be open at a time), select **TOOLS:ALTER EVALUATOR** menu option from the DoME Launcher main menu or select the **TOOLS:ALTER EVALUATOR** menu option from any DoME diagram editor.

Figure 4 The Alter Evaluator



Structure

The evaluator has a title bar, a menu bar, a tool bar, two views, and a field as shown in Figure 4. The **TITLEBAR** indicates the filename; the **MENUBAR** offers access to additional functionality; the **ENVIRONMENT** button offers access to the Alter Environment Viewer; the **BROWSER** button provides access to the Projector Browser; the **DEFINITIONS** view provides an area for editing expressions and definitions that constitute

an Alter program; the **INPUT** field allows entry of expressions to be evaluated interactively; and finally, the **OUTPUT** view displays the results of evaluated expressions and definitions.

Definitions

The Alter programming language provides a means for using names to refer to computational objects.[1] Such a name identifies a *variable* whose *value* is the object.

In Alter the operator for naming things is called `define`.

```
(define pi 3.14)
```

Evaluating this expression causes the Alter interpreter to associate the value `3.14` with the name `pi`.

Once a name has been defined to be a value, the value can be referred to by its name. For example, if we have evaluated the `define` expression from above, then

```
pi => 3.14
```

The expression `pi` evaluates to its value.

`Define` provides a simple means for abstraction. It allows us to use simple names to refer to the results of compound operations. The Alter interpreter allows these name-object associations, called *bindings*, to be created incrementally. This feature promotes incremental development and testing of programs. Due to this feature, Alter programs often consists of a large number of simple procedures.

Environments

The possibility of associating values with symbols and later retrieving them, requires that the interpreter maintain some memory that keeps track of the bindings. This memory is called the *environment*. The memory that contains definitions is the *global*, or *top-level*, environment.

Often we wish to keep track of the history of a system as a program runs. This state is often kept track of by a variable. In order to keep track of the state of a system we must have a way to change the value associated with a name.

Alter has an assignment operator, called `set!`, that allows programs to change the value associated with a name. The name `set!` reflects the naming convention in Scheme that operators that change the value of variables end with an exclamation point.

```
(set! pi (+ 3 2))
```

1 This section adapted from Abelson and Sussman, *The Structure and Interpretation of Computer Programs* New York: The MIT Press, 1985.

`Set!` changes `pi` so that its value is the result of evaluating `(+ 3 2)`. The operator set! presents a problem. The variable pi is a name defined in the global environment, and is freely accessible to any procedure. With the existence of `set!`, it becomes necessary to be able to hide some variables so that their values cannot be indiscriminately changed.

We can use the `let` procedure to create local environments. Bindings in local environments are only accessible from within that environment and environments that are local to it. For example,

```
(define circumference
   (let ( (pi 3.14) )
      (lambda (radius)
         (* pi (* radius radius))) ) )
```

In this example, a local environment is created. In this environment the symbol `pi` is bound to the value `3.14`. A procedure is defined within the local environment. Inside this procedure the symbol `pi` is referenced. Because the procedure is within the *scope* of the local environment, it refers to the value `3.14`. The value 3.14 is bound to the symbol `pi` only in this local environment. Any bindings in the global environment still exist. For example, assuming our previous examples were evaluated in the same environment in which circumference was defined

```
(circumference pi) => 78.5
```

Since `pi` is bound to 5, the result of `(+ 3 2)`, in the global environment, a call to circumference with pi as it argument returns the circumference of a circle with radius 5.

Programs

A program consists of a sequence of expressions and definitions. Programs are typically stored in files or entered interactively into the **DEFINITIONS VIEW** of the Evaluator.

Definitions occurring at the top level of a program can be interpreted declaratively. They cause bindings to be created in the top level environment. Expressions occurring at the top level of a program are interpreted imperatively; they are executed in order when the program is invoked or loaded, and typically perform some kind of initialization.

Program code that is entered interactively into the **DEFINITIONS VIEW** is evaluated upon selection of the **EDIT:EVALUATE** item from the menu bar. Program code that is loaded from a file is evaluated immediately. The result of evaluating each expression or definition in the program is displayed in the **OUTPUT** view of the Evaluator.

Expressions

An Alter expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

The **INPUT FIELD** of the Evaluator allows the user to enter expressions interactively. The expressions are evaluated in the top-level environment upon pressing the <RETURN> key and the results are presented in the Evaluator's **OUTPUT VIEW**.

External Representations

An important concept in Alter is that of the *external representation* of an object as a sequence of characters. For example, an external representation of the integer 28 is the sequence of characters "28", and an external representation of the integers 8 and 13 is the sequence of characters "(8 13)".

The **OUTPUT VIEW** of the Evaluator presents the external representation of objects that result from the evaluation of expressions and definitions, if one exists. If no external representation exists, a pound sign (#) followed by a name enclosed in angle brackets (<>) is displayed. For example, a procedure might be displayed as "#<procedure1234>".

Alter Environment Viewer

When a program error occurs, an Environment Viewer appears, displaying the activation stack, which lists procedures that were waiting for a return when the breakdown occurred. The Environment Viewer enables you to trace the program flow leading to the error, examine the values of arguments at each stage of execution, and view the bindings that are defined in the execution environment.

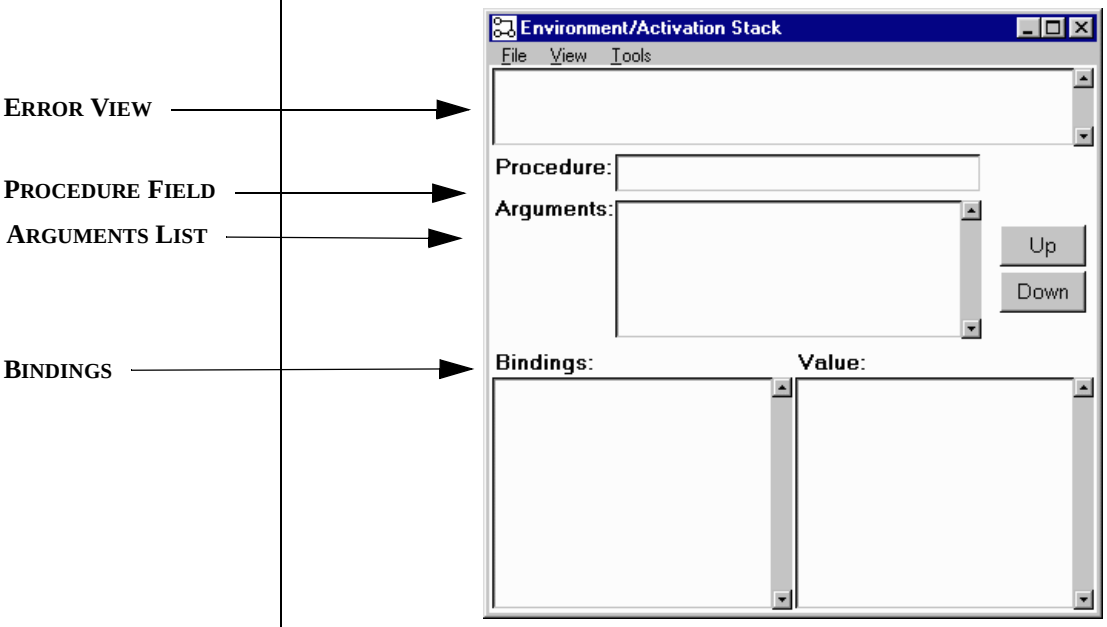
You can also invoke the Environment Viewer by pressing the **ENVIRONMENT BUTTON** on any Evaluator Window (as with the Evaluator Window, more that one Environment Viewer can be open at a time). When the Environment Viewer is invoked this way, it does not display an error message (no error occurred) nor does it display anything in the activation stack (nothing is being executed). In this mode of operation, the Environment Viewer provides a view of the bindings that are currently defined in the environment.

Structure

The Environment Viewer has a title bar, menubar, two buttons, a field, and four views as shown in Figure 5. The **UP** button causes the previous record in the activation stack to be displayed; the **DOWN** button causes the next record in the activation stack to be displayed; the **ERROR** view displays the error message; the **PROCEDURE** field displays the name of the procedure that is on the top of the activation stack; the **ARGUMENTS** list displays the values of the arguments to the procedure named in the **PROCEDURE** field; the **BINDINGS** list

displays the symbols that are bound in the environment; and finally, the **VALUE** view displays the object that the currently selected variable in the **BINDINGS** list is bound to.

Figure 5 The Alter Environment Viewer



Activation Stack

The activation stack is a stack of activation records. An activation record can be thought of as a pair whose car is an executable (procedure or operation) and whose cdr is the list of arguments passed to the executable. Each time a procedure is called it is placed on the activation stack. When a procedure is through being executed it is removed from the stack.

The activation stack area of the Environment Viewer consists of the **PROCEDURE** field, **ARGUMENTS** list, **UP** button, and **DOWN** button. This area enables the user to select a particular activation record for viewing (by moving up and down in the stack) and displays the contents of the current record. The name of the executable of the selected activation record is displayed in the **PROCEDURE** field and the list of arguments is displayed in the **ARGUMENTS** list. The user can inspect an argument by selecting it in the **ARGUMENTS** list and choosing **INSPECT** from the <OPERATE> menu.

Bindings

Any identifier that is not a syntactic keyword may be used as a variable. A variable may name a location where a value can be stored. A variable that does so is said to be bound to the location. The set of all visible bindings in effect at some point in a program is known as the *environment* in effect at that point. The value stored in the location to which a variable is bound is called the variable's value. By abuse of terminology,

the variable is sometimes said to name the value or to be bound to the value. This is not quite accurate, but confusion rarely results from this practice.

Alter is a statically scoped language with block structure. To each place where a variable is bound in a program there corresponds a *region* of the program text within which the binding is effective. The region is determined by the particular binding construct that establishes the binding; if the binding is established by a lambda expression, for example, then its region is the entire lambda expression. Every reference to or assignment of a variable refers to the binding of the variable that established the innermost of the regions containing the use.

If there is no binding of the variable whose region contains the use, then the use refers to the binding for the variable in the top level environment, if any; if there is no binding for the identifier, it is said to be *unbound*.

The initial (or "top level") Alter environment starts out with a number of variables bound to locations containing useful values, most of which are primitive procedures that manipulate data. These "predefined bindings" are not displayed in the bindings area of the Environment Viewer. Only bindings that are created or modified by the user are displayed.

The bindings area of the Environment Viewer consists of the **VARIABLES** list and **VALUE** view. This area enables the user to browse the environment that is in effect at the point in the execution when the error occurred. If the viewer was invoked manually by the user (not in response to an error), then the bindings area displays the top level environment.

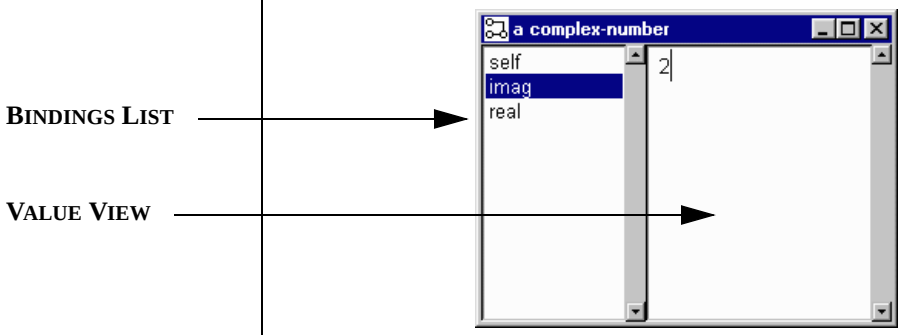
The **VARIABLES** list displays all bound variables in the environment. The **VALUE** view displays the value that the currently selected variable is bound to. The user can inspect a variable by selecting it in the **VARIABLES** list and choosing **INSPECT** from the <OPERATE> menu.

Alter Object Inspector

An object inspector is a window that is used to examine objects other than lists (lists are inspected by list inspectors, see Alter List Inspector below). An object inspector also allows the user to evaluate code within the environment of the inspector.

To open an object inspector, use one of the methods available to inspect bindings on one of the programming tools.

Figure 6 Alter Object Inspector



Structure

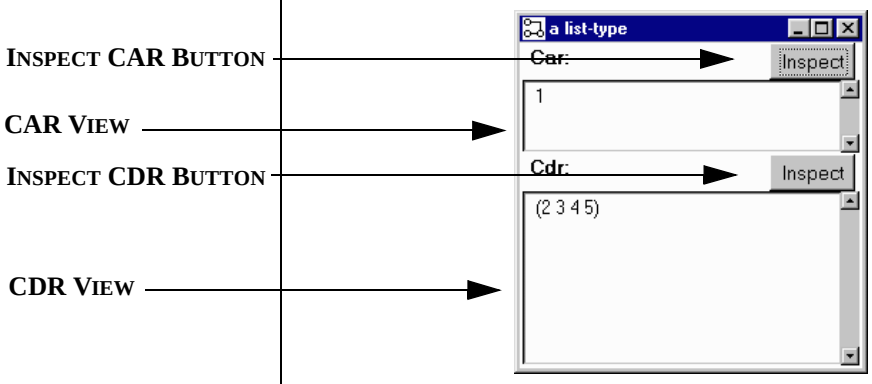
An object inspector has a **TITLE BAR**, a **BINDINGS** list and a **VALUE** view. The **TITLE BAR** indicates the type of object being inspected. The **BINDINGS** list provides a list of the bound variables in the inspectors environment. The **VALUE** view displays the value bound to the variable selected in the **BINDINGS** list. The **VALUE** view also allows the user to enter and evaluate Alter expressions. The expression can be simply evaluated for effect or the result of the expressions can be printed or inspected.

Alter List Inspector

A list inspector is a window that is used to examine a list object. The list inspector displays the contents of the CAR field and the contents of the CDR field. It also allow either of the fields to be inspected.

To open a list inspector, use one of the methods available to inspect bindings on one of the programming tools to inspect a variable that is bound to a list.

Figure 7 Alter List Inspector



Structure

A list inspector has a **TITLE BAR**, two buttons, and two views. The **TITLE BAR** indicates the type of object being inspected. The **INSPECT** CAR button allows the user to inspect the contents of the list's CAR field. The CAR **VIEW** displays the

contents of the list's CAR field. The **INSPECT** CDR button allows the user to inspect the contents of the list's CDR field. And finally, the CDR **VIEW** displays the contents of the list's CDR field.

Programming in Projector 5

. . In This Chapter

This chapter describes...

- How to use Projector to create a Plug-In Function (page 32)

This tutorial provides a tour of the key features of the Projector language and editing tool[1]. This is accomplished by the construction of a sample translator. A code generator to generate Alter code from a Coad-Yourdon Object-Orient Analysis (CYOOA) model is designed and implemented.

It is assumed that the reader has a basic knowledge of computer programming languages. The reader should certainly be familiar with the Scheme language, and understand the concepts of Scheme procedures. In addition, it is assumed that the reader has read the DoME Guide introduction and understands concepts such as the DoME inspector and using DoME-wide menus.

If you are unfamiliar with the basic operations of DoME please see the DoME Guide manual.

It is also assumed that the reader is familiar with object-oriented analysis, design and programming and is familiar with the CYOOA notation and the DoME CYOOA tool.

If you are unfamiliar with the DoME Coad-Yourdon Object-Oriented Analysis tool please see the Coad-Yourdon Object-Oriented Analysis appendix in the DoME Guide manual.

Code Generator Example

This tutorial will lead you through the steps necessary to complete the development of an Alter code generator for the DoME Coad-Yourdon O-O Analysis tool.

The code generator will produce Alter type definitions for each of the classes defined in a CYOOA model. The generator can easily be enhanced to generate method definitions for accessor functions for instance variables and to generate method stubs for any services defined on a class.

The specified Projector model will have four Projector procedures:

1 **generate-alter** — the entry point to the program.

2 **get-output-port** — prompts the user for a filename and returns an output port on that file.

3 **write-definition** — generates the type definition code for a class.

4 **write-class-definition** — generates the Alter code necessary to define the class and bind it to a variable.

The model will also contain four Projector Alter code blocks:

1 This example may be found in the .../tools/cyooa/lib/cy2alter.pro file that is delivered with DoME.

1 **write-class-comment** — generates a header comment for the class.

2 **superclasses** — returns the superclasses of a class.

3 **dc-parent** — used by superclasses procedure.

4 **gs-parents** — used by superclasses procedure.

Some support procedures will be defined as well.

Initial Setup

Initially, the user starts DoME. The DoME launcher will appear after a few moments. The user opens a new model by selecting **FILE:NEW**, and then selecting *Projector Diagram* from the list.

Coding Scenario

As a result of opening a new Projector Diagram, the top level diagram is displayed in an editor. Many predefined procedures are defined and available through the Alter / Projector Browser (see page 21). The browser provides the user with a view of the operators / procedures and classes available for use in developing a Projector program, including procedures defined by the user.

This tutorial uses a mixture of the top-down and the bottom-up design approaches. Pure top-down or pure bottom-up design is possible but for ease of understanding the tutorial a mixture of both is used. The tutorial will first guide the user through the creation of the main procedure named `generate-alter`. Next, the `get-output-port`, `write-definition`, and `write-methods` procedures are created. Finally, the implementation for the `generate-alter` procedure is specified and invoking the code generator is described.

To open the shelf, select **VIEW:SHELF**. The shelf is a library of reusable components of a program that are defined by the user. It also serves as an area for the user to define new procedures.

To create an procedure on the shelf, select the procedure tool which is the rounded rectangle on the toolbar, and drop the procedure in the interface pane by pressing the <SELECT> mouse button. A newly created procedure named TBD will be displayed in the pane. Whenever a procedure is created, Projector will give it a default name of TBD (meaning “to be determined”).

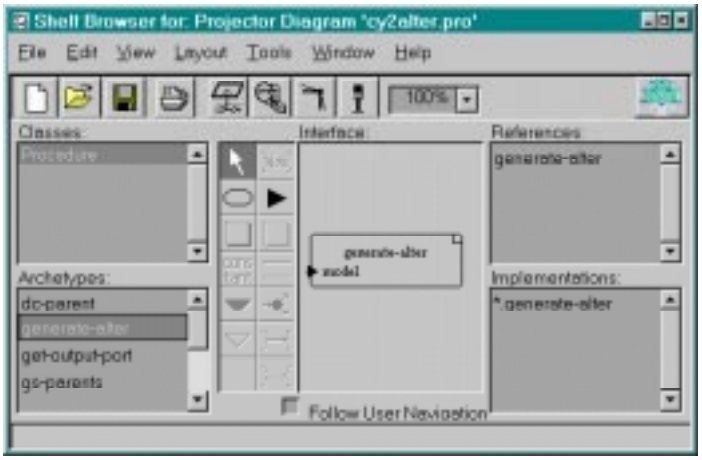
After the procedure is created, select it and press the return key. A dialog is opened to prompt the user for the name of the procedure. Type `generate-alter` and click OK.

Next, select the port tool which is the filled triangle on the toolbar, and place the port on the left edge of the `generate-alter` operator. Select the port, and then select the properties inspector. Many users find it useful to open a properties inspector and move it to a corner of their screen and just leave it open rather than opening a new inspector every time one is required.

Notice the input port already has a name, TBD. As with procedures, whenever a port is created, Projector will give it a default name. As with procedures this will be TBD. Change the name of the port to *model* by typing in the name field at the top of the inspector. Press the **APPLY** button to make these changes take effect.

Also displayed in the object inspector window are the values of two other port properties: `direction` and `show name`. `Direction` indicates whether the port is an input port or output port. Select *In* from the direction pop-up. `Show Name` indicates whether the name of the port should be displayed on the procedure or not. Select true from the show name pop-up. Press **APPLY** to make these changes take effect.

Figure 8 Projector Shelf Browser

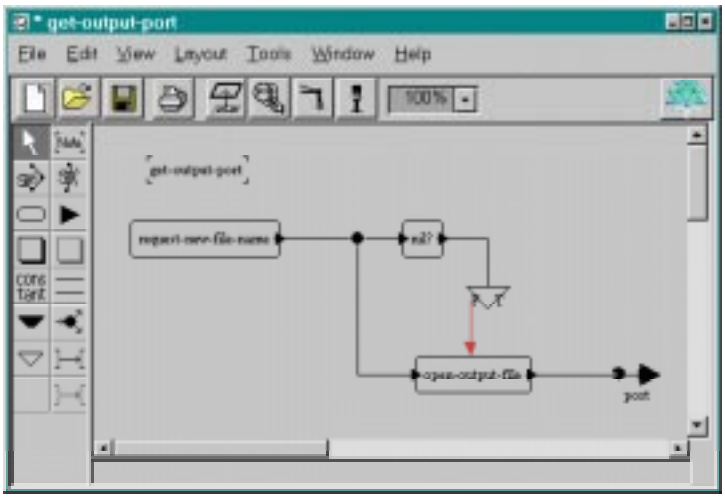


The port should now “point” in towards the center of the rectangle and the name of the port should be displayed as well. Chances are the default size of the procedure is not big enough to display both the procedure name and the port name without overwriting one with the other. To make things more legible, resize the procedure so that there is enough room to display both names. To resize the procedure, place the mouse pointer over one of the corners of the procedure and press and hold the <SELECT> mouse button. Drag the corner of the procedure until it is of the desired size and then release the mouse button. Your shelf should now look something like Figure 8.

Before the implementation for the `generate-alter` procedure can be created, the `get-output-port` and `write-definition` procedures must be built. In a similar manner as to how the `generate-alter` procedure was made, create the `get-output-port` procedure. It will have one output port named `port`.

Next, create an implementation for `get-output-port`. Click with the <OPERATE> mouse button in the implementation pane, and select create. A dialog will appear prompting you to select a model type. You can choose between an `Alter Procedure` and a `Projector Diagram`. Selecting an `Alter Procedure` allows you to enter an implementation for the procedure using the textual language Alter. Selecting `Projector Diagram` will open up a new Projector editor that you can use to specify the implementation of the procedure using the graphical language Projector. Select `Projector Diagram` for the implementation of `get-output-port`. A new editor will appear. Notice that the output port that you defined on the `get-output-port` procedure shows up in this editor.

Figure 9 Projector implementation of `get-output-port`

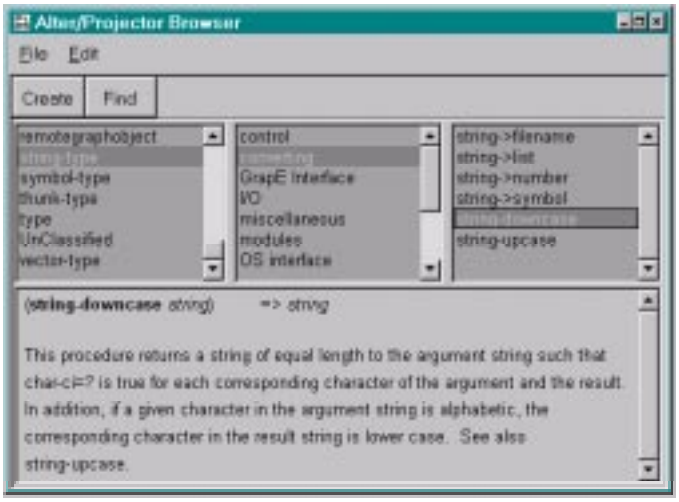


The complete implementation, which is shown above, requires:

- one reference to the predefined operator `request-new-file-name`,
- one reference to the predefined operator `open-output-file`,
- one reference to the predefined operator `nil?`,
- one `splitter`, and
- one `conditional` to test the result of the `nil?` operator.

In order to access the predefined operators you must open the Alter / Projector Browser. Select **TOOLS:BROWSER** from the main menu of either the Projector Diagram editor or the Projector Shelf editor. DoME will load the definitions of the predefined operators if they have not already been loaded and then open a window like the one shown below.

Figure 10 Projector Browser



First, we will add the `request-new-file-name` operator to the implementation diagram. The **ALTER/PROJECTOR BROWSER** provides search capability to make it easier to find particular operators. Click with the <OPERATE> mouse button in the class pane of the browser and select search. A dialog will appear prompting you for a name to search for. The name you type can include the wildcard character `*`. Type in `request*` and press <RETURN>. A pop-up list of operator names that matched your search expression will appear. Select `request-new-file-name` from the pop-up list. If the description text for the operator has not been loaded, DoME will load the description text and then select the operator in the method pane of the browser. Press the **CREATE** button. The cursor will change to the name of the operator. Place the cursor over the implementation diagram and press the <SELECT> mouse button. A reference to the predefined operator will be created in the implementation diagram.

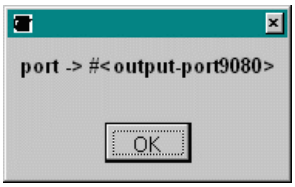
Next, perform similar actions to add references to the predefined operators `nil?` and `open-output-file`.

Now, select the conditional tool from the tool bar. The conditional tool is an unfilled triangle. Place the conditional in the implementation diagram.

Next, select the data flow tool from the tool bar. The data flow tool is represented by a solid arrow. Click on the output port of the `request-new-file-name` operator. A solid line will appear attached to the cursor. Move the cursor to the output port node of the diagram named `port` and click on it. Now there is a data flow between the output port of the request-file-name operator and the output port of the implementation diagram. Notice that DoME has changed the selected tool back to the pointer tool.

You can now execute this diagram and inspect the result that is placed on the output port by the `request-new-file-name` operator. Select **TOOLS:EXECUTE** from the main menu. A dialog will appear prompting you to choose a file. The `request-new-file-name` operator will return the filename you select or `nil` if cancel is selected. Select any filename and press **OK**. A dialog appears like the one shown below.

Figure 11 Diagram Execution Result



Press **OK** to make the dialog go away.

Next, select the data flow that you placed between the request-new-file-name operator and the output port and press the delete key. The data flow will be removed from the diagram.

Now, connect together the operators in the diagram. Since there are a lot of wires to place, permanently select the data wire tool. Do this by holding down the shift key when selecting the tool. Now that tool will stay active until another tool is selected.

If at any point the user needs to cancel a wire draw command, it can be done by clicking the <WINDOW> mouse button. This can be useful if one has accidentally started a connection which one didn't intend.

Place a data flow between the output port of the `request-new-file-name` operator and the input port of the `nil?` operator. Notice that DoME does not reselect the pointer tool like it did before. Place a data flow between the output port of the `nil?` operator and the top of the conditional node. Be careful not to connect the data flow to the sides of the conditional with the `T` or the `F`. Only control flows can be connected to these sides. Finally, place a data flow between the output port of the `open-output-file` operator and the output port node of the implementation diagram.

The result of the `request-new-file-name` operator needs to be input to the `open-output-file` operator as well as the `nil?` operator. This can be accomplished using a `splitter`. Select the `splitter` tool from the tool bar. The `splitter` tool is represented by a solid filled circle with three arrows, one in-coming and two out-going. After selecting the `splitter` tool, click on the middle of the data flow that connects the `request-new-file-name` operator to the `nil?` operator. A `splitter` will appear in the data flow.

Now, use the data flow tool to connect the splitter to the input port of the `open-output-file` operator.

Finally, we need to connect a control flow from the false port of the conditional to the `open-output-file` operator. This will cause the `open-output-file` operator to execute only if the user actually selects a file. Select the control flow tool from the tool bar. The control flow tool is represented by a dotted arrow (as opposed to the solid arrow of the data flow). Click on the `F` port of the conditional and then on the `open-output-file` operator. Click on the operator, not one of the ports, control flows cannot be connected to input or output ports.

The implementation diagram for `get-output-port` is now finished. However, it probably isn't very neat looking. A quick way to spruce up the diagram is by using the straighten arcs button. First, let's select the entire diagram. Do this by holding down the <SELECT> mouse button and drawing a box around the entire diagram. When the button is released, all of the objects should be selected. Hit the straighten arcs button. Projector will straighten the various wires as best it can without moving any other objects

Now you can test the implementation by selecting **TOOLS:EXECUTE** from the main menu as you did before. When you are done testing select **FILE:CLOSE** from the main menu. This will only close the window and not the entire model, unless this is the last window that is open for this model.

The next step is to create the `write-definition` operator. Go to the shelf browser and create the `write-definition` operator, it will have two input ports, named `class` and `port`, and no output ports.

Before the implementation of the `write-definition` operator can be created the `write-class-comment` and `write-class-definition` operators must be defined. These two operators will have Alter implementations so this will give us a chance to demonstrate implementing an operator in Alter and show how Alter and Projector inter-operate.

Create the `write-class-comment` operator on the shelf, it will have two input ports, named `class` and `port`, and no output ports. Click with the <OPERATE> mouse button in the implementation pane of the shelf and select **CREATE**. Again a dialog is display prompting you for the type of implementation you wish to create. This time select `Alter Procedure`.

An **ALTER PROCEDURE CODE** inspector is displayed. Type the Alter implementation given below into the inspector window and press **APPLY** to make the changes take effect.

Example 2: Alter implementation of write-class-comment operator

```
(display ";;; ---" port)
(newline port)
(display ";;; --- TYPE: " port)
(display (name class))
(newline port)
(display ";;; --- DESCRIPTION: " port)
(display (description class) port)
(newline port)
(display ";;; ---" port)
(newline port)
```

Before the `write-class-definition` operator can be created, the `superclasses` operator needs to be defined. The `superclasses` operator makes use of two support operators, `dc-parent` and `gs-parents`, that we will define first.

Create the `dc-parent` operator on the shelf, it will have an input port named `genspec` and an output port named `parent`. Create an Alter implementation for the operator using the following Alter code.

Example 3: Alter implementation of dc-parent operator

```
(let* (  (arcs (select (outgoing-arcs genspec))
                       (lambda (a) (is-a? a domaingslink)))) )
     (if (null? arcs) nil (destination (car arcs)))
```

Next create the `gs-parents` operator. It will have an input port named `class` and an output port named `parents`. Create an Alter implementation for the operator using the following code:

```
(map destination
    (select (outgoing-arcs class)
       (lambda (a) (is-a? a domaingslink))))
```

Next lets define the `superclasses` operator. This will demonstrate making calls to Projector operators in Alter code.

Create the `superclasses` operator on the shelf, it will have one input port named class and one output port named list. Create an `Alter` implementation for the operator.

In the following code we reference the two Projector operators that were just created. The operators have Alter implementations but they are still Projector operators. We could re-implement either of the operators with a `Projector Diagram` implementation and the following code would still work:

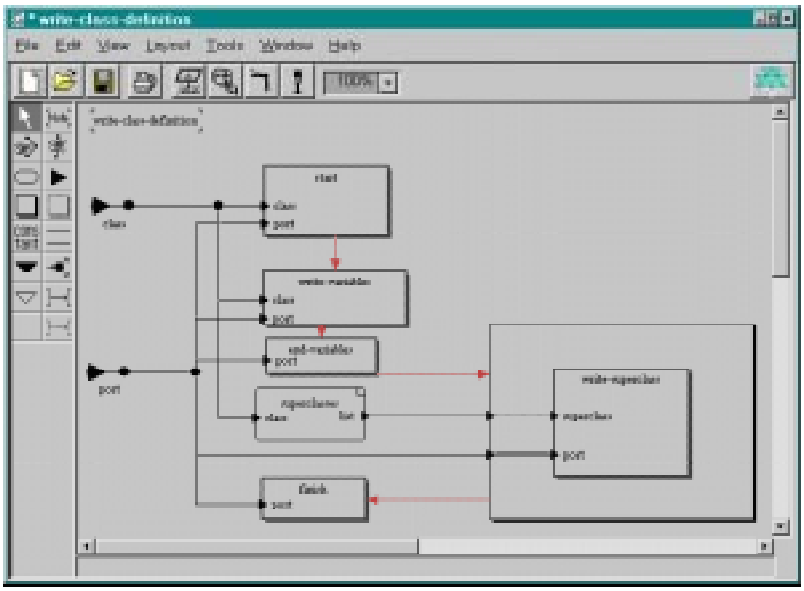
Example 4: Alter implementation of superclasses operator

```
(map dc-parent (gs-parents class))
```

Now we can implement the `write-class-definition` operator. Create the `write-class-definition` operator on the shelf. It will have two input ports, named `class` and `port`, and no output ports. In this implementation we will demonstrate the use of `Alter Code` blocks. These code blocks can be thought of an Alter code that is "inlined" in the graphical Projector Diagram. The complete implementation, which is shown in the figure below, of the `write-class-definition` operator requires:

- four `Alter Code Blocks`,
- one reference to the `superclasses` operator, and
- one `Statement Block`.

Figure 12 Projector implementation of write-class-definition



Select the `Alter Code` tool from the tool bar. The `Alter Code` tool is the shadowed gray box. Place three code blocks on the implementation diagram. Name them `start`, `write-variables`, and `finish`. Place two input ports on the `start` code block and name them `class` and `port`. Place two input ports on the `write-variables` code block and also name them `class` and `port`. Place one input port on the `finish` code block and name it `port`.

Connect the `class` input node to the `class` input ports on the `start` and `write-variables` code blocks and the `port` input node to the `port` input ports on each of the code blocks. Use the `data flow` tool in conjunction with the `splitter` tool to accomplish this.

Select the `start` code block and bring up a property inspector on it. Use the following Alter code to define its `code` property.

```
(display "(define " port)
(display (string-downcase (substitute (name class) " " "-")) port)
(display " (make type (list " port)
```

Next, use the following code to define the `write-variables` code block.

```
(for-each
    (lambda (a)
        (begin
            (display " '" port)
            (display (string-downcase (substitute (name a) " " "-
")) port)))
        (get-property "attributes" class))
```

Finally, use the following code to define the `finish` code block.

```
(display ")))" port)
(newline port)
```

Next, place a reference to the superclasses operator on the implementation diagram. Connect this reference's `class` input port to the `class` input node and it's `port` input port to the `port` input node. Again use the `data flow` tool to accomplish this.

Select the `Statement Block` tool from the tool bar. The `Statement Block` tool is a shadowed box drawn with solid black lines. Place a Statement Block in the implementation diagram and name it write-superclasses. Place two input ports on the block, and name them `class` and `port`. Connect the `class` port to the `list` output port of the superclasses operator reference. Connect the `port` input port to the `port` input node.

Next, select the `Alter Code` tool and place a code block inside the `write-superclasses` statement block and name it `write-superclass`. You can place an `Alter Code Block` inside a `Statement Block` by dropping the `Alter Code Block` on top of the `Statement Block`. The `Statement Block` will expand to absorb the `Alter Statement Block`, but you can resize the `Statement Block` as well.

Add two input ports to the `write-superclass` code block and name them `superclass` and `port`. Connect the `superclass` input port to the `class` input port on the `write-superclasses` statement block. make sure that you start the connection at the statement block's input port so that the data flows from the statement block to the `write-superclass` Alter code block. In the same way connect the port input port on the statement block to the port input port on the Alter code block. Use the following code to define the Alter code block:

```
(display " " port)
(display (string-downcase (substitute (name class) " " "-")) port)
```

The return value of the `superclasses` operator is a list of classes. The code inside the statement block expects to receive one class at a time. We can cause the block to be iterated once for each element in the list by changing the type of the data flow. Select the data flow that connects the `superclasses` operator's `list` output port to the statement block's `class` input port and inspect its properties.

The data flow's `transfer` property specifies how the data flow transfers its data from begin to end.

- **simple** — transfers whatever is placed on its origin to the destination unmodified.
- **maintain** — transfers whatever is placed on its origin to the destination unmodified and continues to return that value indefinitely.
- **build** — collects the elements that are placed on its origin into a list until no more objects are available and then transfers the built list to its destination.
- **reduce** — takes any list that is placed on its origin and transfers the elements of the list one at a time to its destination.

Set the data flow's `transfer` property to reduce and press the **APPLY** button to make the changes take effect. Now the superclasses that are placed on the data flow will be transmitted one at a time to the statement block and the

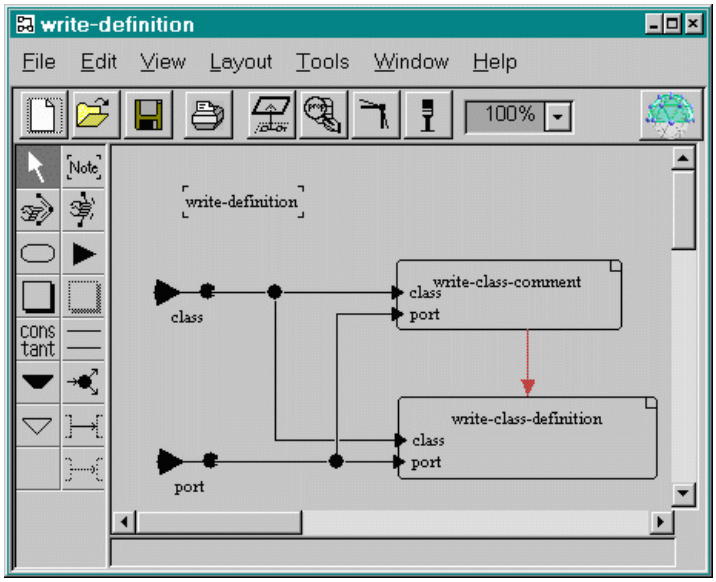
statement block will be executed once for each element in the list. We have in effect *graphically* constructed a `for-each` statement.

As the last part of the implementation of the `write-class-definition` operator place a control flow from the start block to the write-variables block, from the write-variables block to the write-superclasses statement block and from the write-superclasses statement block to the finish block. This will ensure that the graph is executed in the proper order so that the code gets witten to file properly.

Now we can continue with the implementation of the `write-definition` operator. The complete implementation, which is shown below, of the `write-definition` operator requires:

- one reference to the `write-class-comment` operator and,
- one reference to the `write-class-definition` operator.

Figure 13 Projector implementation of write-definition



Select the write-definition operator in the Archetypes pane on the shelf browser and create a `Projector Diagram` implementation for the operator. As before, click the <OPERATE> mouse button in the implementation pane of the shelf browser, select **CREATE** and choose `Projector Diagram` as the model type for the implementation. Now, in the implementation diagram which has just opened, create references to `write-class-comment` and `write-class-definition`. This is done by selecting the archetype, pressing the <OPERATE> mouse button in the **REFERENCES**

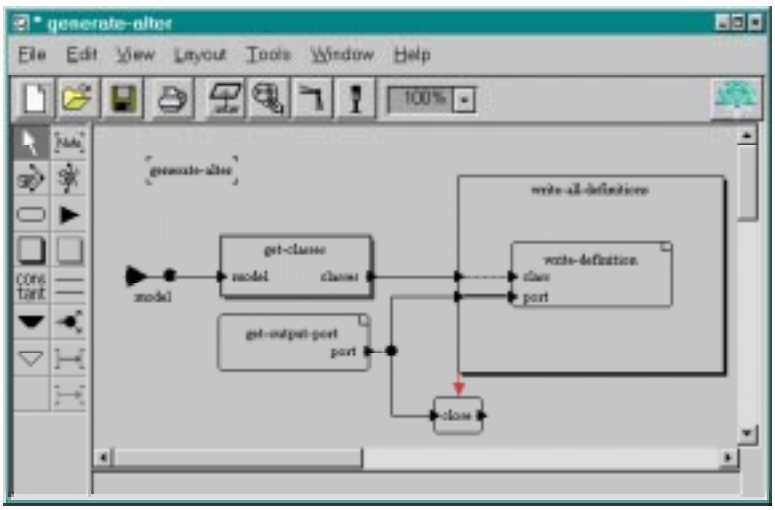
pane, selecting `create` from the pop-up menu and then clicking the <SELECT> mouse button in the diagram in which the reference is to be placed. In this case, that is the `write-definition` implementation diagram.

As can be seen in the figure above, the input `class` is connected to the `class` input ports on both operators and the input `port` is connected to the `port` input port on both operators. This is accomplished by using `splitters` to split the data flow. A control flow is placed from the `write-class-comment` operator to the `write-class-definition` operator to ensure that the comment is written before the definition.

Finally, we can return to the `generate-alter` operator and construct its implementation. Select the `generate-alter` operator in the shelf browser and, as you have done before for other operators, create a `Projector Diagram` implementation for it. The complete implementation, which is shown below, of the `generate-alter` operator requires:

- one `Alter Code Block`,
- one `Statement Block`,
- one reference to the predefined close operator,
- one reference to the `get-output-port` operator, and
- one reference to the write-definition operator.

Figure 14 Projector implementation of generate-alter



As you have done before, create an Alter Code Block and place it in the implementation diagram for the `generate-alter` operator. Name the block `get-classes` and give it one input port named `model` and an output port named `classes`. Use the following code to implement the `get-classes` block:

```
(select (nodes model)
    (lambda (n) (is-a? domainclass)))
```

Next drop in a statement block and name it `write-all-definitions`. Give it two input ports and name them `classes` and `port`. Place a reference to the `write-definition` operator in the statement block. Connect the `classes` port of the statement block to the `class` port of the operator with the data flow tool. Set the `transfer` property of the data flow to `reduce` so that the `write-definition` operator gets executed once for each class. Connect the `port` on the statement block to the `port` on the operator and set its `transfer` property to `maintain`.

Next, drop a reference to the `get-output-port` operator. Connect the `port` output port of the `get-output-port` operator to the `port` input port of the statement block with the data flow tool. Connect the `model` input node to the `model` input port of the `get-classes` code block. Connect the `classes` output port of the code block to the `class` input port of the `statement block`.

Select **TOOLS:BROWSER** from the main menu in order to open the Projector Browser. Search for the close operator and drop a reference to it into the implementation diagram. Place a splitter into the data flow that connects the `get-output-port` operator to the statement block. Connect the `splitter` to the input port of the `close` operator. Leave the output port of the `close` operator unconnected.

The port cannot be closed until the statement block is done executing. Therefore, put a control flow from the statement block to the `close` operator.

As the final step, we need to specify the entry point for the program. To do this select **VIEW:TOP OF MODEL DIAGRAM** from the main menu of any of the editors that are part of the Projector model. The top level diagram will be displayed in an editor. Drop a reference to the `generate-alter` operator into the diagram. Select the graph label node (upper left hand corner) and inspect its properties. Enter `generate-alter` for its Entry property and press the **APPLY** button to make the change take effect. This completes the coding scenario.

Programming in Alter 6

. . In This Chapter

This chapter describes...

- How a user interacts with the Alter Evaluator

To familiarize yourself with the basic features of the Alter language and the Alter programming tools, this chapter will lead you through a set of task oriented steps one would use to write an Alter program.

The following topics will be covered in this chapter.

- Opening an Alter Evaluator
- Evaluating expressions
- Entering program text - (scrolling, editing commands)
- Evaluating a program (definitions -> environment bindings)
- Saving program text (definitions)
- Exiting the Alter Evaluator
- Opening existing programs
- Printing program text

Opening an Evaluator

The Alter Evaluator is a convenient tool for writing, testing and debugging Alter programs. The evaluator also provides access to the other tools in the Alter Programming Environment.

Once the DoME launcher is up you can open a new Alter Evaluator using the DoME Launcher.

1 Select the TOOLS:ALTER EVALUATOR menu option from the DoME launcher.

An Alter Evaluator appears.

Alternatively, you can open an Alter Evaluator from any DoME graph editor.

1 Create a new model.

A new DoME graph editor appears for the type of model you selected.

2 Select the TOOLS:ALTER EVALUATOR menu option from the graph editor's main menu.

An Alter Evaluator appears.

Evaluating Expressions

An Alter expression is a construct that returns a value, such as a variable reference, literal, procedure call, or conditional.

Alter expressions can be evaluated interactively using the **INPUT FIELD**.

1 Enter (quote a) **in the INPUT FIELD and press the** <RETURN> **key.**

The symbol a is displayed in the **OUTPUT PANEL**.

2 **Enter** (- (+ 1 2) 3) **in the INPUT FIELD and press the** <RETURN> **key.**

The number 0 is displayed in the **OUTPUT PANEL**.

3 **Enter** "Hello DoME!" **in the INPUT FIELD and press the** <RETURN> **key.**

The string "Hello DoME!" is displayed in the **OUTPUT PANEL**.

4 **Enter** (make grapething) **in the INPUT FIELD and press the** <RETURN> **key.**

#<value: a GrapEThing> is displayed in the **OUTPUT PANEL**.

Entering a Program

When you first open a new Alter Evaluator, you will have access to a new Alter environment. Besides the predefined bindings (those bindings of symbols to procedures, operations and types provided by default) the environment will contain no bindings.

Definitions can be entered interactively in the **DEFINITIONS PANE**. When the definitions are evaluated they cause bindings to be created in the top level environments.

1 **Enter definition code in the DEFINITIONS PANE.**

For this example enter the following code:

```
(define add4 (lambda (n) (+ n 4) ) )
(define sub3 (lambda (n) (- n 3) ) )
(define add1 (lambda (n) (add4 (sub3 n)) ) )
```

The **DEFINITIONS PANE** and the **INPUT FIELD** of the Evaluator window provide the basic editing operations that are available in many text editors.

Evaluating Definitions

1 **Select the EDIT:EVALUATE menu option.**

The Alter code in the **DEFINITIONS PANE** is parsed and evaluated. The result of the define procedure is displayed in the **OUTPUT/RESULTS PANEL**.

2 **Type** (add4 16) **in the Input Field and press the** <RETURN> **key.**

20 is displayed in the **OUTPUT/RESULTS PANEL**.

3 **Type** (sub3 16) **in the Input Field and press the** <RETURN> **key.**

13 is displayed in the **OUTPUT/RESULTS PANEL**.

4 **Type** (add1 16) **in the Input Field and press the** <RETURN> **key.**

17 is displayed in the **OUTPUT/RESULTS PANEL**.

Saving your Alter Program

You can save your definitions to a file and load them back in as a program later. In order to allow DoME to recognize that the file contains Alter code it is necessary to make the first line in the file a comment.

1. **Add a comment line as the first line in the DEFINITIONS PANE.**

   Insert the cursor at the far left of the first line in the definitions pane. Type a semi-colon and press the <RETURN> key.

2. **Select FILE:SAVE or FILE:SAVE AS.**

   SAVE will save the current definitions to the previously saved name. If you have not yet saved the file, or you select **SAVE AS**, a dialog window will open allowing you to specify where to save the definitions.

3. **Select the directory in which you wish to place the saved model.**

4. **Type in the file name. Press OK or <RETURN>.**

   The title bar will now contain the name of the file to which you saved the definitions.

Closing an Evaluator

1. **Select FILE:QUIT.**

   The Evaluator window will close.

Opening an Alter Program File

Once a program has been saved in a file, you may re-open it and modify it at any time.

1. **Select the OPEN button on the DoME Launcher window, or select FILE:OPEN from an open DoME graph editor window.**

2. **Find the file you want and select it.**

3. **Click on the OK button or press the <RETURN> key.**

   The contents of the program file is displayed in the **DEFINITIONS PANE** of a new Alter Evaluator Window. The definitions are parsed and evaluated.

Opening a Program File From an Evaluator

You can also load a program file into a Evaluator window that is already open. The bindings created by the definitions in the file augment any bindings already defined through the Evaluator window.

1. **Select FILE:OPEN from an open Evaluator window.**

   An “Open File Dialog” will open.

2. **Find the file you want and select it.**

3 Click on the OK button or press the <RETURN> key.

Any old text is removed from the **DEFINITIONS PANE** of the Evaluator window. The contents of the program file is displayed in the **DEFINITIONS PANE** of the Alter Evaluator Window. The definitions are parsed and evaluated. The bindings created by the evaluation are added to, or modify, any bindings that were previously defined through the Evaluator window.

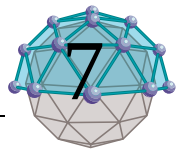
Printing

The contents of the definitions pane of a Evaluator window can be sent to a printer.

1 Select FILE:PRINT in the Evaluator window containing the definitions you want to print.

The text in the **DEFINITIONS PANE** is sent to the default printer.

Plug-In Functions



. . In This Chapter

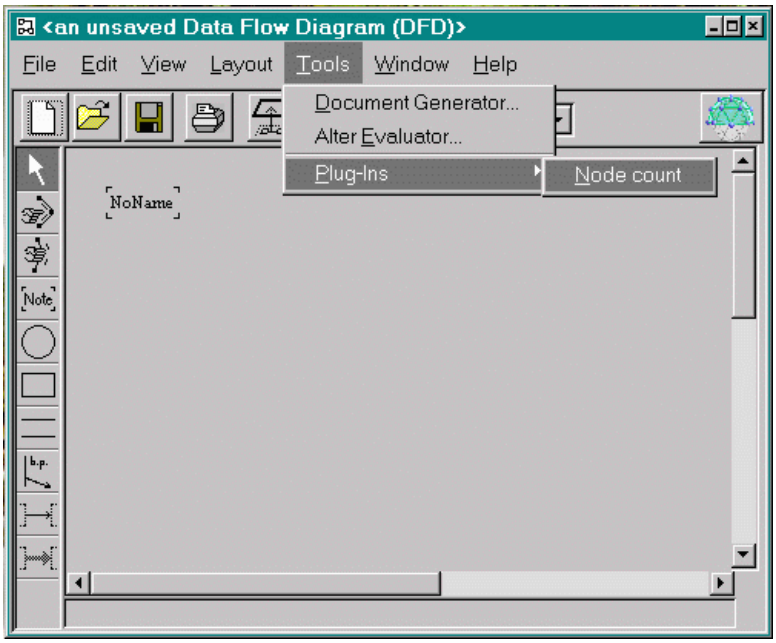
This chapter describes...

- How to associate a plug-in function with DoME (page 54)
- Examples of plug-in functions (page 57)

You can write new functions for DoME to execute on a model. These functions can be used to perform many different operations such as model analysis, conversion to another notation, document and code generation, or graph execution.

Users can register their new functions with DoME. Once a function is registered it can be invoked from the **TOOLS:PLUG-INS** menu (see Figure 15 DoME Tools Menu below).

Figure 15 DoME Tools Menu



Plug-In functions are written using the Projector / Alter extension system. This chapter provides the requirements that users must conform to when writing plug-in functions. It also provides a few examples of plug-in functions.

Function Calling Mechanism

When you register a plug-in function with DoME you specify a *function name*, a *source file*, and a *graph type*. DoME uses the function name as the menu item label in the **TOOLS:PLUG-INS** menu for your function. DoME uses the graph type to determine when to make a function available in the **TOOLS:PLUG-INS** menu. A function appears in the **TOOLS:PLUG-INS** menu only if its graph type matches the type of graph being edited. DoME uses the source file as the source code of the function. When a function is invoked for the first time during a session, DoME loads and evaluates the code in the source file.

When DoME loads the source code for a plug-in function, it stores the entry point to the function along with the current time stamp of the source file. When a user invokes a function, by selecting it from the **TOOLS:PLUG-INS** menu, DoME checks to see whether the source file has been loaded.

If it has not been loaded, then DoME proceeds to load and evaluate the source code.

If the source file has already been loaded then DoME checks the stored time stamp against the current time stamp of the source file. If the time stamps are equal then DoME uses the stored entry point. If the time stamps differ DoME asks the user whether the source file should be reloaded. If the user responds "yes" then DoME reloads the file and replaces the stored entry point and time stamp with the new ones. If the user responds "no" then DoME uses the stored entry point.

Once DoME has performed the necessary checks and performed any necessary loads it invokes the plug-in function. It does this by evaluating the entry point, which is assumed to be an operator that takes one argument, with the model currently being edited as the lone argument.

Once DoME invokes the function, it waits for the function to complete execution before taking control again. Therefore, during function execution it is up to the function to handle all interaction with the user, the graph, and the operating system using the appropriate Alter primitives. A function can be interrupted by pressing <CONTROL> - c.

Function Entry Point

In order to invoke a plug-in function, DoME must know what the entry point for the function is. The entry point is an operator that takes one argument.

If the source file for the plug-in function contains Alter code then the entry point is the result of evaluation of the last expression in the file.

If the source file for the plug-in function contains a Projector diagram then the entry point is specified as a property of the top level graph.

Registering Functions

In order for DoME to recognize the existence of a plug-in function, you need to register it. Plug-in functions are registered in one or more function description files. All function description files are named "function.dom". A function description file is a text file that describes one or more functions and has the following form:

```
[DoMEUserFunctionList driverspec . . .]
```

Where driverspec looks like:

```
[DoMEUserFunctionSpec
   functionName: 'menu-string'!
   sourceFile: 'pathname'!
   graphType: #symbol!
   keySequence: 'char'!
]
```

where

functionName	is a string that will be used to form the entry in the **TOOLS:PLUG-INS** menu. An ampersand (&) can be placed to the left of the character that should be the accelerator key.
sourceFile	is a string giving the filename of the function's definition file.
graphType	is a string representing one of the DoME graph model types. GraphModel is the top of the hierarchy. This function will appear in the **TOOLS:PLUG-INS** menu of editors that are editing graphs of this type.
keySequence	is a single character that can be used to invoke the function from within DoME via a shortcut key.

You can also include the contents of other function specification files with an entry of the following form:

```
[DoMEFileInclude sourceFile: 'pathname'!]
```

where

pathname	is a string giving the full pathname of the function specification file to include.

DoME looks for function description files in particular places depending on your configuration. If you have the environment variable DoMEUserFunctions set, DoME will first look there for a description file then DoME will try the filenames represented by the expression:

```
(construct
   (construct
      (construct
         (construct (dome-home) "tools")
```

```
        "*")
      "etc")
    "function.dom")
```

Examples

This section includes some example plug-in functions.

Count all the nodes

In this example you will create a function that counts all the nodes in a graph and returns the result in a dialog box.

In order to count all the nodes in a graph it is first necessary to query the graph for its nodes. Projector / Alter provides a simple mechanism for obtaining the nodes of a graph with the `nodes` procedure. For example,

```
(nodes G)
```

will evaluate to a list of the nodes in graph G.

The `length` procedure is a standard Scheme procedure that returns the number of elements in a list. We will use this to count the number of nodes.

Once we have the result we need to display it to the user. Projector / Alter provides some capability to interact with the user. The `warn` procedure can be used to display a message to the user. For example,

```
(warn "Greetings universe!")
```

will cause a dialog box to be displayed with the message `"Greetings universe!"` and a button labeled `"ok"`.

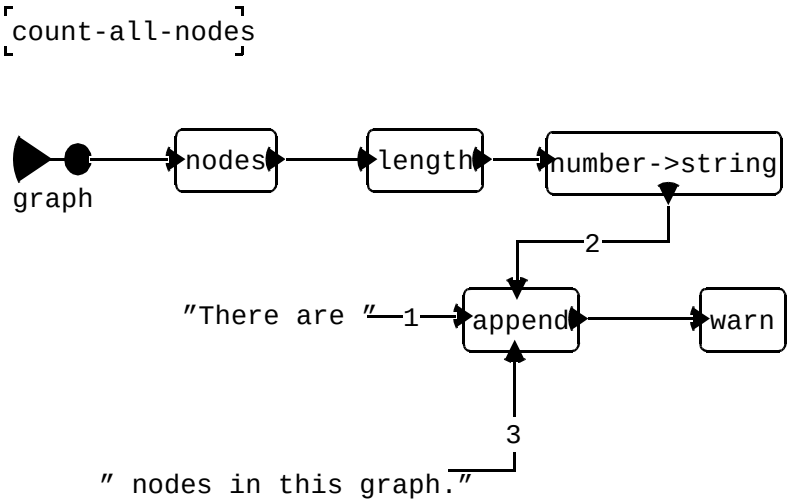
The following Alter code defines a procedure that implements this simple node counter.

Example 5: Alter implementation of count-all-nodes

```
(define (count-all-nodes graph)
   (let ( (n (length (nodes graph))) )
        (warn (append "There are "
            (number->string n)
            " nodes in this graph.")) ) )
```

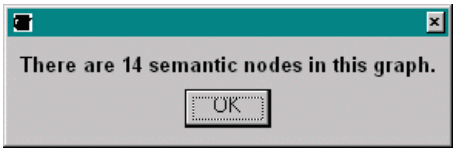
The following is a Projector diagram that also implements the node counter.

Figure 16 Projector implementation of count-all-nodes



Invoking this function on a graph produces a dialog similar to the following:

Figure 17 Example dialog from count-all-nodes execution



Count the semantic nodes

You may have noticed in the previous example that the number of nodes returned was greater than you might have expected. This is because in DoME many more things are nodes than you might expect. For example the label that is usually located in the upper left corner of a DoME graph is a node (called a `Graph Label`). In fact the graph label is a special type of a more general type of DoME node called a `Note`.

Notes are part of all DoME graphs. They are similar to a comment in a program and are therefore semantically unimportant. To obtain the count that you probably expected to get, we need to eliminate the semantically unimportant nodes and only count the semantic nodes.

To obtain a list of only the semantic nodes we need to eliminate all nodes that are of type `netnote`. The standard Scheme procedure select provides us with a mechanism for selecting particular items from a list. For example,

```
(select '(1 2 3 4) (lambda (e) (> e 2)))
```

will return a list containing all the elements of the list `(1 2 3 4)` that are greater than 2. So, the list `(3 4)` is returned.

To select nodes of a particular type we will need to determine whether a node is of a particular type. Projector / Alter provides the `is-a?` predicate that will do just this. For example,

```
(is-a? A B)
```

will return true if the type of A is equal to B or the type of A inherits from B, otherwise it will return false.

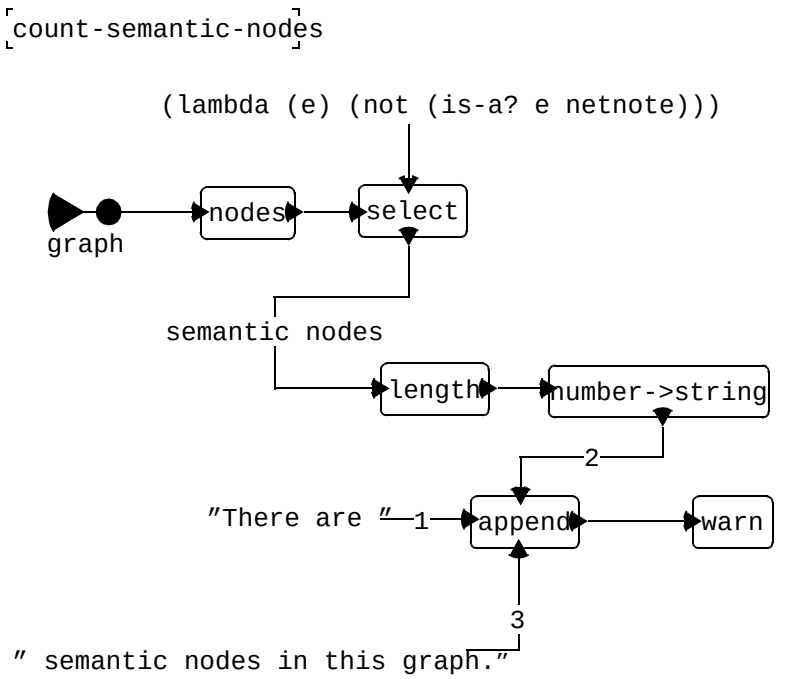
The following Alter code defines a procedure that implements this semantic node counter.

Example 6: Alter implementation of count-semantic-nodes

```
(define (count-semantic-nodes graph)
   (let ( (n (length (select (nodes graph)
                (lambda (e) (not (is-a? e
netnote))) ) ) ) )
        (warn (append "There are "
            (number->string n)
           " semantic nodes in this graph.")) )
)
```

The following is a Projector diagram that also implements the semantic node counter.

Figure 18 Projector implementation of count-semantic-nodes



Summarize nodes and arcs

Besides nodes, most DoME graphs include arcs as well. Often, DoME graphs includes many different kinds of nodes and arcs. Therefore, we could provide the user with more information about their model by providing a summary of the number and types of objects in their graph.

To do this we will need to get a string that describes the type of an object. Projector / Alter provides the `what-are-you` procedure that returns a string describing the object. For example,

```
(what-are-you (make graphmodel))
```

returns the string “Graph Model”.

We will also need to group objects together by type so that we can get a count of the different types of objects in the graph. Projector / Alter provides the procedure get-type that returns the type of an object. For example,

```
(get-type A)
```

will return the type of A.

We will make use of a dictionary to store the results until we are ready to display them. The Projector / Alter procedure `make-dictionary` creates a dictionary. For example,

```
(make-dictionary)
```

will return a new instance of `dictionary`. The Projector / Alter procedures `dictionary-set!` and `dictionary-ref` provide access to the entries in a dictionary. For example,

```
(dictionary-set! D key value)
```

will insert `value` into dictionary `D` and associate it with `key` and

```
(dictionary-ref D key default)
```

will return the value in `D` that is associated with `key` if it exists and return `default` otherwise.

The following Alter code implements a function that displays a summary of the different types of objects contained in a graph.

Example 7: Alter implementation of summarize-graph

```
(define (summarize-graph graph)
   (letrec
      (  (c (components graph))
         (d (make-dictionary))
         (s (make-string 0))
         (nl (list->string (list #\newline)))
         (tb (list->string (list #\tab)))
```

```
          (add-to-d (lambda (e)
             (let
                ( (cnt
                   (dictionary-ref d (what-are
you e) 0)) )
                (dictionary-set! d
                                    (what-are-you
e)
                                    (+ cnt 1)) )
) )
          (add-to-s (lambda (e)
             (letrec
                ( (cnt (number->string
                         (dictionary-ref d e -
1))) )
                (set! s (append s nl e ": " tb
cnt))))) )
    (for-each add-to-d c)
    (for-each add-to-s (dictionary-keys d))
    (warn (append "Summary:" nl s nl)) ) )
```

Summary Report

Users may wish to produce a report that summarizes the contents of the model. Simple plain text documents can be generated using plug-in functions. The following example creates a simple model summary report, a listing of all of the components in a model with their descriptions and rationale. The function traverses the nodes and each of its subdiagrams.

The document is laid out as follows:

Title

1.0 <top level mission name>

An example entry in the inventory list looks like this:

TopModel (State-Transition Diagram)

DESCRIPTION: An example state-transition diagram.
RATIONALE: This model is part of an example.

There is one such entry for each component of the graph.

Model Queries

To get the components of the graph use the **`components`** procedure.

```
(components graph) => list of components
```

To write an entry, get the name, description and rationale properties from the object using the **`name`**, **`description`** and **`rationale`** procedures.

```
(name component) => string
(description component) => string
(rationale component) => string
```

Each entry also requires a description of its type. To get a string that describes the object's type use the **what-are-you** procedure.

```
(what-are-you component) => string
```

Word Wrap

To make the text more readable use the **word-wrap** procedure to create lines no longer that the width of the page.

```
(word-wrap string page-width verbatum?)
```

See the Alter Programmer's Reference Manual for more information on the `word-wrap` procedure.

Output Files

To direct the output to a source other than the system transcript use the procedure **with-output-to-file**.

```
(with-output-to-file filename proc)
```

See the Alter Programmer's Reference Manual for more information on the `with-output-to-file` procedure.

Generator

Put all these together into an Inventory Report generator.

```
;;; ---------------------------------------
;;; --- This Alter module declares the methods
;;; --- used to write an inventory report for
;;; --- a GrapEThing.
;;; ---------------------------------------
(find-operation display-inventory-property)
(add-method
   (display-inventory-property (grapething)
self prop)
      (for-each
         lambda (s) (display-indented-line s
3))
      (word-wrap (append (string-upcase prop)
                  ": "
                  (get-property prop self))
                  72
                  #t) ) )

(find-operation display-indented-line)
(add-method (display-indented-line (string-
type) s i-sz)
   (display (append (make-string i-sz #\space)
s))
   (newline) )

(find-operation without-crs)
(add-method (without-crs (string-type) self)
   (list->string
      (map
         (lambda (c) (if (eq? c #\newline)
```

```
#\space c))
    (string->list self))) )

(find-operation write-inventory-entry)
(add-method (write-inventory-entry
(grapething) self)
   (newline)
   (display (without-crs (get-property “name”
self)))
   (display (append “ (“ (what-are-you self)
“)”))
   (newline)
   (display-inventory-property self
“description”)
   (display-inventory-property self
“rationale”) )

(find-operation write-inventory)
(add-method (write-inventory (graphmodel)
graph)
   (let ( (proc (lambda () (display “Model
Inventory Report”)
            (newline)
            (display “----------------------”)
            (newline)
            (write-inventory-entry graph)
            (for-each write-inventory-entry
                     (components graph)))
       ) )
   (with-output-to-file “” proc) ) )

write-inventory ; entry-point
```

The operation **display-inventory-property** gets the property named in the **prop** argument, appends the property name and a colon to the beginning of the property value, uses word-wrap to create lines of text no longer than the width of the page, and then uses display-indented-line to display the lines of text indented 3 spaces.

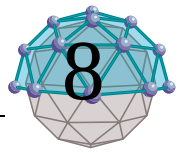
The operation display-indented-line adds the number of spaces indicated by the i-sz argument to the string passed in the s argument, displays the result on the default output port followed by a newline.

The without-crs operation removes any carriage returns from the argument and returns the result.

The write-inventory-entry operation displays the text for the inventory entry for one GrapEThing. The text is displayed on the default output port.

The write-inventory operation opens a window and writes an inventory report for the argument graph.

Print Drivers



. . In This Chapter

This chapter describes...

- The DoME print driver facility (page 66)
- An example print driver (page 73)

Description

A DoME print driver allows the user to print a graph in a specialized format to a file. DoME has several predefined print drivers including postscript, gif, and (frame)maker interchange format.

You can write new printer drivers for DoME using the Alter language. Such drivers are seamlessly integrated into DoME and made available for use through the normal print dialog.

Each DoME printer driver must supply a list of procedures for performing the various graphic operations used for displaying graphs. You can name the procedures anything you want; DoME hooks up to them through a simple protocol (described below).

The printer driver procedures are as follows:

Procedure Title	**Description**
initialize	This is the first call made to a printer driver. It is typically used for setting up resources and global data.
preamble	This is the second call made to a printer driver. It is typically used for setting up scaling information and initializing the print device.
line	Draws a simple line from one point to another point. If this procedure is not supplied by the driver but polyline is, DoME uses the polyline procedure.
rectangle	Draws a rectangle (filled or unfilled). The upper left and lower right corners of the rectangle are given. If this procedure is not supplied by the driver but polyline is, DoME uses the polyline procedure. If neither rectangle nor polyline are provided, DoME uses the line procedure.
polyline	Draws an open polyline (filled or unfilled). The polyline is given as a list of points (x . y). If the polyline procedure is not supplied by the driver but line is, DoME uses the line procedure.
arc	Draws an arc (filled or unfilled). The arc is given as a bounding box, and a start angle and sweep angle. If the arc procedure is not supplied by the driver, but polyline is, DoME uses the polyline procedure.

Procedure Title	Description
string	Displays a single line of text. If a string in the DoME model is more than one line long, DoME will make multiple calls to this procedure, once for each line of text.
postamble	This is the second to last call made to a printer driver. It is typically used for sending wrap-up data to the print device.
finalize	This is the last call made to a printer driver. It is typically used for releasing supplementary resources used during the printing.

DoME decomposes the display of the graph into these low-level calls. For example, to display an arc with an ordinary arrow head, DoME will make the following calls:

1 polyline procedure, with the pen color the same as the background, to blank out an area behind the trunk of the arc

2 polyline procedure, with a visible pen color, to draw the trunk or the arc

3 arc procedure, specifying filled, with the same color as in step 2, to draw the arrow head.

When you want to print a model (using the File / Print menu command), DoME looks for a file that describes the user-defined drivers that are available. It then uses the information in this file to augment the print dialog to give you access to those drivers.

Upon loading the source code for a printer driver, DoME expects the last expression in the file to be a list that tells which Alter procedures implement which driver functions.

Driver Functions

This section describes the interface that DoME expects for each driver function listed in the previous table. You may name the procedures in your driver anything you want; the names given in the list below are the symbols DoME looks for in the associative list at the end of the source file.

The first two arguments to each driver function are the graphics context (gc) and the output port (port). The graphics context is an object that holds information about pen color, line style, etc., and is described in more detail later. The output port is where the driver should send its characters for driving the print device.

(initialize *gc port*)

This is the first call made to a printer driver. It is typically used for setting up resources and global data.

(preamble *gc port print-size graph-bounds*) ⇒ *scale-info*

This is the second call made to a printer driver, and is also called at the beginning of subsequent graphs if more than one is being printed (i.e. the Print Child Graphs box was checked in the print dialog). It is typically used for setting up scaling information and initializing the print device.

print-size	The desired size, in inches, of the resulting printed form of the graph.
graph-bounds	The graph's current bounds, as a rectangle with units of pixels. The rectangle is represented as a list of the form $((ul_x . ul_y) . (lr_x . lr_y))$, where "ul" means upper-left and "lr" means lower-right.

A return value is expected from preamble, which is a list of the form $(s_x\ s_y\ t_x\ t_y)$. On subsequent calls to driver functions that use points, DoME will first transform screen coordinates into driver coordinates according to the following formula:

$$x = s_x(x_s + t_x)$$

$$y = s_y(y_s + t_y)$$

where $(x_s . y_s)$ is a point in screen coordinates.

(line *gc port from to*)

Draws a simple line from one point to another point. If this procedure is not supplied by the driver but polyline is, DoME uses the polyline procedure.

from	The origin end of the line. The is given as a pair $(x . y)$.
to	The destination end of the line. The point is given as a pair $(x . y)$.

(rectangle *gc port rectangle fill*)

Draws a rectangle (filled or unfilled). The upper

left and lower right corners of the rectangle are given. If this procedure is not supplied by the driver but polyline is, DoME uses the polyline procedure. If neither rectangle nor polyline are provided, DoME uses the line procedure.

rectangle	The upper left and lower right corners of the rectangle, given as a list of the form $((ul_x . ul_y) . (lr_x . lr_y))$.
fill	A boolean. If true, the rectangle should be filled (with the current pen color).

(polyline *gc port point-list fill*)

Draws an open polyline (filled or unfilled). The polyline is given as a list of points (x . y). If the polyline procedure is not supplied by the driver but line is, DoME uses the line procedure.

point-list	A list of the vertices, as points (pairs of the form $(x . y)$). The polyline is considered to be a closed shape if the last point is equal to the first point.
fill	A boolean. If true, the polyline should be filled (with the current pen color).

(arc *gc port bounding-box start-angle sweep-angle fill*)

Draws an arc (filled or unfilled). The arc is given as a bounding box, and a start angle and sweep angle. Angles are given in degrees, with positive angles indicating clockwise displacements, negative angles counterclockwise.

bounding-box	A rectangle of the form $((ul_x . ul_y) . (lr_x . lr_y))$ that circumscribes the ellipse (only the portion of the ellipse as specified by start-angle and sweep-angle should be drawn).
start-angle	The starting angular position for the arc, measured from three o'clock (in the screen coordinate system).
sweep-angle	The angular path of the arc rela-

tive to the starting angle. These angles are specified in the (possibly skewed) coordinate system of the ellipse. For example, the angle between three o'clock and a line from the center of the ellipse to the top right corner of the bounding rectangle is always 45 degrees, even if the bounding rectangle is not square.

fill — A boolean. If true, the arc should be filled (with the current pen color), so that it looks like a pie wedge whose tip is at the center of the *bounding-box*.

(string *gc port string alignment location extent*)

Displays a single line of text. If a string in the DoME model is more than one line long, DoME will make multiple calls to this procedure, once for each line of text.

string — A string containing a single line of text, with no line breaks.

alignment — A string indicating the alignment of the supplied text. The string may be one of {"Left", "Right", "Center", or "Justified"}. The interpretation of *location* depends on *alignment*.

location — A point (*x* . *y*) specifying the location for the text. If *alignment* is "Left", then *location* specifies the extreme upper-left corner of the text. If *alignment* is "Right", then *location* specifies the extreme upper-right corner of the text. If *alignment* is "Center" or "Justified", then *location* specifies the exact center of the text. The *extent* parameter may be used to help position the text on the print device.

extent — The width and height (*w* . *h*) of the string as it appears on the screen (scaled according to the

formula described under preamble).

(postamble *gc port*)

This is the second to last call made to a printer driver if only one graph is being printed. If more than one graph is being printed (because the Print Child Graphs box was checked in the print dialog), the postamble procedure is called at the end of every graph. It is typically used for sending wrap-up data to the print device.

(finalize *gc port*)

This is the last call made to a printer driver. It is typically used for sending finish-up commands to the print device, and for releasing supplementary resources used during the printing.

GraphicsContext Operations

You can apply the following operations to a graphics context, the first argument given to all of the printer driver functions:

(**face** *altergraphicscontext*) ⇒ *string*

(**landscape** *altergraphicscontext*) ⇒ *boolean*

(**line-width** *altergraphicscontext*) ⇒ *integer*

(**paint** *altergraphicscontext*) ⇒ *array*

(**paint-color** *altergraphicscontext*) ⇒ *colorvalue*

(**paint-style** *altergraphicscontext*) ⇒ *symbol*

(**line-style** *altergraphicscontext*) ⇒ *symbol*

These are described in more detail in the DoME Alter Programmer's Manual.

Color Operations

The paint-color operation returns a colorvalue that holds the individual primary color components of the current pen color. You can use the following operations on colorvalue instances to obtain the individual color components: **red, blue, green, cyan, magenta, yellow, hue, saturation, brightness**. All of these operations return a real number between 0 and 1.

The Procedure Map

The last expression in a driver must be an associative list that has the form:

```
((function-symbol . procedure) . . .)
```

where *function-symbol* must be one of the following symbols: initialize, preamble, line, rectangle, polyline, arc, string, postamble, or finalize. *Procedure* must be a procedure defined earlier in the process of loading the driver source file.

A driver is not required to supply all of the procedures. DoME will automatically convert a graphics operation to use a different driver routine if the preferred one is missing. For example, if the *polyline* procedure is missing, DoME will convert polylines into individual line segments and make multiple calls to the *line* procedure. If DoME can't find a substitute procedure, it raises an error. Note that there is no substitute for the *string* procedure.

The example driver included in this chapter has a sample procedure map at the very end.

Registering a Driver

In order for DoME to recognize the existence of a user-defined driver, you must create a driver description file and place it in one of the locations DoME searches. The file must be named "pformats.dom". A driver description file describes one or more drivers and has the following form:

```
[DoMEPrintFormatList driverspec . .
.]
```

Where driverspec looks like:

```
[DoMEPrintFormatSpec
    formatName: 'name-of-format'!
    sourceFile: 'pathname'!
    fileOnly: boolean!
    fileSuffix: 'suffix'!
    handlesChildren: boolean!
    handlesLandscape: boolean!
]
```

where

formatName	is a short string that will be used to augment the print dialog's format menu.
sourceFile	is a string giving the filename of the driver source code file.
fileOnly	is either `true` or `false`. If `true`, the format can only be printed to a file; DoME will not allow it to be sent directly to the printer.
fileSuffix	is a string to be used for suggesting a filename suffix if the graph is to be printed to a file.
handlesChildren	is either `true` or `false`. If `true`, the format can handle printing subdiagrams.

handlesLandscape is either `true` or `false`. If `true`, the format can handle printing the diagram in landscape mode.

If you have the environment variable DoMEPrintFormats set, DoME will look there first for a driver description file and then DoME will look in the files represented by the following Alter expression:

```
(construct
   (construct
      (construct
         (construct (dome-home) 'tools')
         "*")
      'etc')
   'pformats.dom')
```

Example Driver

What follows is a trimmed-down, example printer driver that prints the graph in DXF format. The driver does not support all of the features of DXF, so the rendering is only approximate.

```
;; PLEASE NOTE: Whenever numbers are supplied by DoME
as
;; arguments to this print engine, those numbers may
be
;; integers, fractions or floats.
;; It is up to the print engine to convert them into
whatever
;; form is needed for the particular print format.

(define *scale* '(1.0 . 1.0))       ;; a global
definition


;; ------------------------------------------------
----------
;; dxf-preamble

(define (dxf-preamble context port print-size graph-
bounds)
  (let ((width (- (cadr graph-bounds) (caar graph-
bounds)))
   (height (- (cddr graph-bounds) (cdar graph-
bounds))))
    (let ((scale (exact->inexact (/ print-size (max
width height)))))
      (show-pair 0 "SECTION" port)
      (show-pair 2 "HEADER" port)
      (show-pair 9 "$LIMMIN" port)
      (show-point 0 0 port)
      (show-pair 9 "$LIMMAX" port)
```

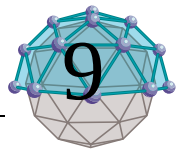
```
      (show-point 8 10 port)
      (show-pair 0 “ENDSEC” port)
      .
      .
      .
      (show-pair 2 “ENTITIES” port)
      (set! *scale* (cons scale (- scale)))
      (list scale (- scale)
       (- (caar graph-bounds)) (- (cddr graph-
bounds))))))

;; ------------------------------------------------
----------
;; dxf-rectangle

(define (dxf-rectangle context port rect fill)
  (if (or (< (brightness (paint-color context)) 1.0)
      (> (saturation (paint-color context)) 0.0))
      (if fill
      (begin
       (show-pair 0 “SOLID” port)
       (show-pair 8 1 port)
       (show-point (caar rect) (cdar rect) port)
       (show-point (cadr rect) (cdar rect) port 11)
       (show-point (caar rect) (cddr rect) port 12)
       (show-point (cadr rect) (cddr rect) port 13))
   (begin
    (dxf-polyline context port
              (list (car rect)
               (cons (cadr rect) (cdar rect))
               (cdr rect)
               (cons (caar rect) (cddr rect))
               (car rect))
              #f)))))

;; ------------------------------------------------
----------
;; dxf-polyline

(define (dxf-polyline context port vertices fill)
  (if (or (< (brightness (paint-color context)) 1.0)
      (> (saturation (paint-color context)) 0.0))
      (begin
       (show-pair 0 “POLYLINE” port)
       (show-pair 8 1 port)
       (show-pair 66 1 port)
       (show-pair 70 0 port)
       (let ((thickness (/ (lineWidth context)
72.0)))
    (show-pair 40 thickness port)
```

```
    (show-pair 41 thickness port))
      (for-each (lambda (v)
        (show-pair 0 "VERTEX" port)
        (show-pair 8 1 port)
        (show-point (car v) (cdr v) port))
       vertices)
      (show-pair 0 "SEQEND" port)
      (show-pair 8 1 port))))

;; ------------------------------------------------
----------
;; dxf-arc
;; This driver does not support arcs. We will supply
a routine
;; so that DoME does not raise an exception.

(define (dxf-arc context stream rectangle start-angle
sweep-angle fill)
  '())

;; ------------------------------------------------
----------
;; dxf-string

(define (dxf-string context port string alignment
position extent)
  (show-pair 0 "TEXT" port)
  (show-pair 8 "TEXT" port)
  (let ((relScale (get-property "relativeScale"
context)))
    (let ((offset (cond ((equal? alignment "Left")
           (cons 0 0))
          ((equal? alignment "Center")
           (cons (/ (car extent) 2) (* 0 relScale
(cdr *scale*))))
          ((equal? alignment "Right")
           (cons (car extent) 0)))))
      (let ((x (- (car position) (car offset)))
       (y (+ (cdr position) (cdr offset))))
   (show-point x y port)
   (show-pair 40 (* 12 (car *scale*)) port)
   (show-pair 1 string port)
   (show-pair 50 0.0 port)))))

;; ------------------------------------------------
----------
;; dxf-finalize

(define (dxf-postamble context port)
  (show-pair 0 "ENDSEC" port)
  (show-pair 0 "EOF" port))
```

```
;;
**************************************************
**********
;; The following procedures are not called directly
by DoME, but ;; rather by this print engine.
**************************************************
**********
;; show-pair
;;
;; Write out a pair of values that constitues a DXF
data group

(define (show-pair first second port)
  (display first port)
  (newline port)
  (display second port)
  (newline port))

;;
**************************************************
**********
;; show-point
;;
;; Write out an x-y coordinate with the specified
group ID offset.

(define (show-point x y port . offset)
  (let ((group (if (null? offset) 10 (car offset))))
    (display group port)
    (newline port)
    (display (exact->inexact x) port)
    (newline port)
    (display (+ group 10) port)
    (newline port)
    (display (exact->inexact y) port)
    (newline port)))

;;
**************************************************
**********
;; End of local procedures
;;
**************************************************
**********

;; Procedure map

(list
 (cons ‘preamble dxf-preamble)
 (cons ‘postamble dxf-postamble)
 (cons ‘rectangle dxf-rectangle)
```

```
(cons ‘polyline dxf-polyline)
(cons ‘string dxf-string))
```

Document Generators



. . In This Chapter

This chapter describes...

- The DoME documentation generation facility

You can write plug-in functions in DoME that generate plain text documentation and code from a model (See "Summary Report" on page 61). You can also generate documentation that is compatible with your preferred text processing environment. These functions perform all the activities necessary to produce the final output document. They traverse and query the model, create the document's content, and structure and format that content into a final document.

Producing documentation as described in the previous examples has its drawbacks:

- Changing the format of document requires modification to the generator program.
- Producing output that is compatible with a different text processing environment requires modification to the generator program.
- Producing the same document with multiple styles or for multiple text processors requires multiple generator programs.

DoME provides a better way to produce documentation that removes the drawbacks listed above.

Document Markup

A document has content, structure and style.[1] Content is the series of words, spaces, and punctuation contained in the document. Structure refers to the way the document is divided into paragraphs, chapters and sections. Style dictates how the different structure elements are displayed.

Traditionally, the publishing process proceeded as follows. The author would write the document using some media (hand-written, type written, etc.). When the author turned the manuscript over to the publisher for publishing the copy-editor would annotate it with instructions to the type-setter concerning layout, fonts, spacing, indentation, etc. These annotations were known as *markup*. Then the type-setter would use these instructions to layout the type. Finally the document would be printed using the type.

As computers became more widely used, publishers began using electronic publishing programs. These programs process data files that contain the text of the document interspersed with processing instructions explaining the actions to be taken at that point. The various instructions are the equivalent of

1 This section is a watered down version of a couple chapters in "Practical SGML" written by Eric van Herwijnen. Please see this book for a more thorough and complete discussion of SGML.

traditional document mark up. The mark up for the document could be expressed in one of many formatting languages depending on the text formatter being used.

The mark up in these documents is known as *specific* markup. It is specific in that it indicates the specific instructions to be carried out at a certain point during document processing.

Many formatting languages allow commands to be grouped together into macros. Often these macros can be stored in a separate file and shared among many documents. Placing the macros in a separate file makes maintenance of a uniform style easier. Changes made in one place propagate to all documents. All documents processed using the macro set have the same style.

Macros lead to the concept of *generic* markup. With generic markup the exact processing instructions to be taken by the formatter are not specified in the document. Instead, markup is used to specify the structure of the document. Macros are assumed to correspond one-to-one with the structural *elements* of the document.

Using generic markup, the purpose of the various parts of the document are specified without considering the appearance. The exact processing instructions are contained in the macro definitions that correspond to the elements of the document. Therefore, a generically marked up document specifies the content and structure of the document but not the style. The style is specified in a stylesheet.

The set of macros used to format a generically marked up document is a *stylesheet*. More than one stylesheet can be applied to a document. Using one stylesheet, chapter headings may appear in bold-face type, using another they might appear in upper case large font. Stylesheets even extend to the display of a document on media other that the printed page. For example, chapters could be used to indicate that a new window should be opened whose title bar contains the chapter heading text and whose window contains the contents of the chapter. Stylesheets allow a document's appearance to change easily, allowing publishers to use the same document source for journal articles and later for an anthology.

Standard Generalized Markup Language (SGML) provides a standard language for specifying generic markup. Using SGML, document types can be defined. A document type definition (DTD) specifies the following:

- *Names* and *content* of all structural *elements* that are allowed in a document of that type.
- The number of times an element may appear.
- The order in which the elements must appear.

- The contents of all elements.
- Attributes of tags and their default values.

Many standard DTD's have been designed and published by various organizations. Probably the most well known DTD is the Hyper Text Markup Language (HTML) DTD used by the World Wide Web (WWW).

The language defined by a DTD can be used to describe the logical structure of the document, but no semantics are applied to these logical elements by the language. These semantics (i.e., skip a line and indent three spaces, etc.) are provided by a style sheet. *Document Style Semantics and Specification Language* (DSSSL) will, when approved as an international standard, provide a standard language for specifying processing instructions to be applied to the various elements of a document. An SGML document instance combined with a DSSSL stylesheet instance can be used by a SGML / DSSSL aware text formatter to produce a final output document in some page description language (PDL) such as PostScript. The PDL version of the document can be displayed directly on its intended output device.

DoME's document generation and processing capabilities have been infused with the spirit of these two international standards. Though DoME does not completely implement either of these standards currently, many of their benefits are passed on to DoME users.

Document Generators

SGML is used to describe documents with a tree-like structure. Since the document is a tree it can be represented internally as a tree. DoME represents SGML documents as trees. Thus, the process of generating an SGML document is the process of building a tree representing the document.

Included with DoME is a set of standard Alter libraries for document generation. These libraries include the definition of a node type, sgmlnode, useful for producing SGML documents. The type defined in the standard library includes the definition of standard operations used by DoME style sheets and text formatters to process the tree.

Using the standard library, the process of document generation amounts to producing a tree constructed out of instances of sgmlnode and its subtypes. A DoME SGML document generator is a procedure or operation that returns a tree.

See "SGML Generators" on page 85 for more information about document generators.

Stylesheets

Alter is used to specify document stylesheets. The form of an Alter stylesheet specification is strongly influenced by DSSSL, but is in no way an implementation of the proposed DSSSL standard.

DoME stylesheets provide a generic way of specifying the formatting to be applied to the logical elements (nodes) of an SGML document (tree).

A DoME stylesheet consists of the definition of one operation. The operation includes method definitions for each of the different node types contained in a document that will be processed using the stylesheet. In the process of formatting a document, a text formatter will invoke the operation on each node in the tree and carry out the formatting operations specified by the stylesheet method.

DoME stylesheets make use of standard formatting operations supported by DoME document formatters.

See “Stylesheets” on page 101 for more information about stylesheets.

Text Formatters

Text formatters are objects that respond to the processing instructions contained in a stylesheet. The output from a text formatter can be anything from a page description language (PDL) like PostScript to a plain text approximation of the formatted document to raw SGML code.

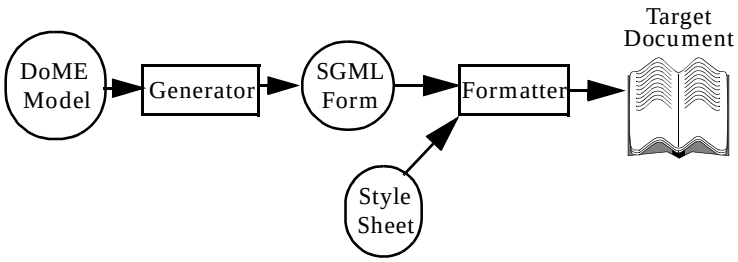
A text formatter is an Alter type. Certain standard methods are defined on the formatter. These methods are used in stylesheet methods to specify formatting instructions. The response to the invocation of one of these standard methods by a stylesheet method varies depending on the formatter. Each formatter produces output that is compatible with a particular text processing environment. Users can write custom text formatters for their preferred document presentation environment. This environment might be a laser printer, a word processor or an HTML browser. Each of these environments requires a source document that is written in a different language.

See “Text Formatters” on page 93 for more information about text formatters.

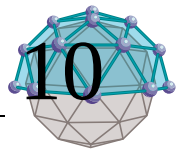
The Generation Process

The DoME document generation process is depicted in Figure 19.

Figure 19 DoME Document Generation Process



SGML Generators



. . In This Chapter

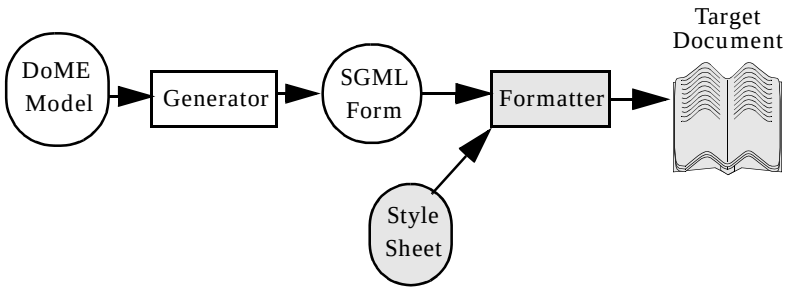
This chapter describes...

- The SGML document generator

You can write user-defined functions in DoME that produce a SGML document. These generators are the first stage in the DoME document generation process. The result of the SGML document generator is passed to a text formatter that, with direction from a stylesheet, produces the final formatted document in some page description language (PDL).

The DoME document generation process is depicted in Figure 20. The non-grayed parts of the figure represent the SGML portions of the generation process.

Figure 20 DoME Document Generation Process



Query Operations

A DoME SGML document generator builds a tree structure in main memory that represents an instance of an SGML document type. The nodes that the tree are built from must be instances of a subtype of sgmlnode which provides certain standard query operations. These standard query operations are used by stylesheets to query the tree for information. The stylesheet takes this information and invokes operations on a text formatter.

The standard SGML tree node query operations are as follows:

(children nd) => list

Returns a list containing a node for each child of nd.

(attributes nd) => list

Returns a list containing one node for each attribute of nd.

(parent nd) => list

Returns a list containing the parent of the node, if the node has a parent, and otherwise returns the empty list.

(owner nd) => list

Returns a list containing the owner of nd, if nd has an owner, and otherwise returns the empty list.

(contents nd) => string

Returns a string representing the contents of nd.

(map-children nd proc) => list

This is the map procedure invoked with a list whose elements are the children of nd.

(for-each-child nd proc) => nil

This is the for-each procedure invoked with a list whose elements are the children of nd.

(map-self+children nd proc) =>list

This is the map procedure invoved with a list whose elements are nd and each of the children of nd.

(node=? nd_1 nd_2) => boolean

Returns #t if nd_1 and nd_2 represent the same node in the same tree, and otherwise returns #f.

Registering a Generator

In order for DoME to recognize the existence of a user-defined SGML document generator, you must create a generator description file and place it in one of the locations DoME searches. All SGML document generator files are named "document.dom". An SGML generator description file describes one or more generators and has the following form:

```
[DoMESGMLDocList generatorspec . . .]
```

Where generatorspec looks like:

```
[DoMESGMLGeneratorSpec
   functionName: 'menu-string'!
   sourceFile: 'pathname'!
   graphType: #symbol!
   documentType: #symbol!
   outputTypes: [OrderedCollection '#symbol'!*]
]
```

where

functionName	is a string that will be used to form the entry in the **DOCUMENT** menu of the document generator dialog.
sourceFile	is a string giving the filename of the generator's definition file.
graphType	is a string representing one of the DoME graph model types. GraphModel is the top of the hierarchy. This generator will appear in the **DOCUMENT** menu of the document generator dialog of the editors that are editing graphs of this type.

documentType	is a string describing which style types are appropriate for this generator.
outputTypes	is a collection of strings describing where a document can put output to. Valid values are #file, #window, #printer, and #directory. If unspecified then #file, #window, and #printer are used.

If you have the environment variable DoMESGMLDocuments set, DoME will look there first for a SGML document generator file and then DoME will look in the files represented by the following Alter expression:

```
(construct
   (construct
      (construct
         (construct (dome-home) 'tools')
         "*")
      'etc')
   'document.dom')
```

Example Generator

The following section will lead you through the construction of a simple SGML document generator. The example builds on the example found in "Summary Report" on page 61.

Choosing a DTD

The first step in constructing an SGML document generator is to choose a target document type. This involves choosing a DTD from among those that are publicly available or designing your own. In this section we will continue the example began in "Summary Report" on page 61. We will enhance that example by making the Model Inventory Report a formatted document. The following document type definition will be used for this document:

```
<!-- Inventory DTD for Manual Examples -->
<!ELEMENT prop - - (propname, propval) >
<!ELEMENT propname - - (#PCDATA) >
<!ELEMENT propval - - (#PCDATA) >
<!ELEMENT name - - (#PCDATA) >
<!ELEMENT type - - (#PCDATA) >
<!ELEMENT title - - (#PCDATA) >
<!ELEMENT entry - - (name, type, desc, rat)>
<!ELEMENT invent - - (title,entry*) >
```

Creating node types

Before writing the generator you will need to create node types that match each of the elements in the DTD. The node types will be used to represent each of the element types in the document. The stylesheet will use these types to apply the proper formatting to each element in the document.

The following type definitions will suffice for this DTD.

```
;;; ----------------------------------------
---
;;; <!ELEMENT prop - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-prop
   (make type '() (list sgml-node)))
(name-set! invent-prop "prop")
;;; ----------------------------------------
---
;;; <!ELEMENT propname - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-propname
   (make type '() (list sgml-node)))
(name-set! invent-propname "propname")
;;; ----------------------------------------
---
;;; <!ELEMENT propval - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-propval
   (make type '() (list sgml-node)))
(name-set! invent-propval "propval")
;;; ----------------------------------------
---
;;; <!ELEMENT name - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-name
   (make type '() (list sgml-node)))
(name-set! invent-desc "name")
;;; ----------------------------------------
---
;;; <!ELEMENT type - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-type
   (make type '() (list sgml-node)))
(name-set! invent-desc "type")
```

```
;;; ----------------------------------------
---
;;; <!ELEMENT title - - (#PCDATA) >
;;; ----------------------------------------
---
(define invent-title  (make type '() (list sgml-
node)))
(name-set! invent-desc "title")
;;; ----------------------------------------
---
;;; <!ELEMENT entry - - (name, type, desc, rat)
>
;;; ----------------------------------------
---
(define invent-entry
   (make type '() (list sgml-node)))
(name-set! invent-entry "entry")
;;; ----------------------------------------
---
;;; <!ELEMENT invent - - (title, entry*) >
;;; ----------------------------------------
---
(define invent-invent
   (make type '() (list sgml-node)))
(name-set! invent-desc "invent")
;;; ----------------------------------------
---
```

Each of these types are subtypes from the type sgml-node which is provided with DoME in the standard library sgmlnode.lib.

Writing the generator

The generator is a procedure or operation that takes one argument and returns an SGML tree. The generator constructs the tree based on information in the model that is passed in as its sole argument. The SGML document generator is very similar to generating a plain text document. The main difference is that instead of displaying text, the SGML generator creates nodes in the tree.

The following code will create an SGML tree representing an instance of the invent DTD.[1]

```
(find-operation inventory-property)
(add-method
   (inventory-property (grapething) self prop)
      (let( (nd1 (make invent-prop))
            (nd2 (make invent-propname))
```

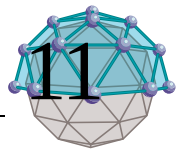
1 This example may be found in the .../tools/alter/examples/inv-sgml.alt file that is delivered with DoME.

```
            (nd3 (make invent-propval)) )
         (set-contents! nd2 prop)
         (set-contents! nd3
                        (without-crs
                           (get-property prop
self)))
         (add-child-first nd1 nd3)
         (add-child-first nd1 nd2)
      nd1) )

(find-operation without-crs)
(add-method (without-crs (string-type) self)
   (list->string
      (map
         (lambda (c) (if (eq? c #\newline)
#\space c))
    (string->list self))) )

(find-operation inventory-entry)
(add-method (inventory-entry (grapething) self)
   (let( (nd1 (make invent-entry))
         (nd2 (make invent-name))
         (nd3 (make invent-type)) )
      (set-contents! nd2
                     (without-crs
                        (get-property “name”
self)))
      (set-contents! nd3 (what-are-you self))
      (add-child-first nd1
                (inventory-property self
“rationale”))
      (add-child-first nd1
                (inventory-property self
“description”))
      (add-child-first nd1 nd3)
      (add-child-first nd1 nd2)
      nd1) )

(find-operation inventory)
(add-method (inventory (graphmodel) graph)
   (let( (nd1 (make invent-invent))
         (nd2 (make invent-title)) )
      (set-contents! nd2 “Model Inventory
Report”)
      (add-child-first nd1 (inventory-entry
graph))
      (add-child-first nd1 nd2)
```

```
        (add-children-last nd1
                  (map inventory-entry
    (components graph)))
        nd1) )

    inventory ; entry-point
```

The operation inventory is the entry point for this document generator. The last line of the file that contains the definitions for this generator should evaluate to the operation inventory.

Text Formatters



. . In This Chapter

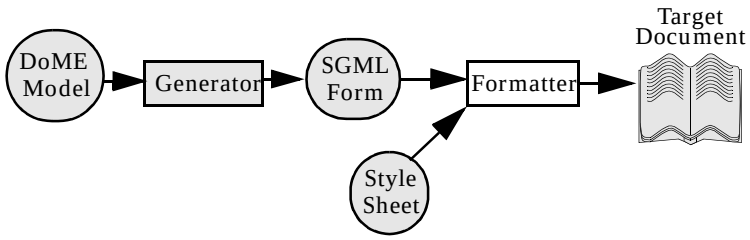
This chapter describes...

- The DoME text formatter facility

You can write new text formatters for DoME. Such formatters are seamlessly integrated into DoME and made available for use through the Generate dialog. These formatters are the second stage in the DoME document generation process. The result of the SGML document generator is passed to a text formatter that, with direction from a stylesheet, produces the final formatted document in some page description language (PDL).

The DoME document generation process is depicted in Figure 21. The non-grayed parts of the figure represent the text formatter portions of the generation process.

Figure 21 DoME Document Generation Process



Overview

A text formatter is an object that knows how to write documents in a form compatible with particular document processing environment. This processing environment may be a word processor, printer, etc. The formatter takes the appropriate actions to create a document that is formatted in the style specified by a stylesheet.

The context-type formatter is provided with all DoME installations. This type provides basic functionality to create plain text (without word-wrap) documents from an SGML tree and a stylesheet.

Context-type implements all the standard functions for formatters, therefore any subtypes of context-type meet the requirements for a formatter. The user can then specialize the standard operations for their new formatter and make use of the pre-existing operations for those that do not need specialization.

Also included with DoME are formatters to create documents in Maker Interchange Format (MIF), Interleaf ASCII Format (IAF), Rich Text Format (RTF), and plain text with word-wrap (TXT). There is also a formatter that simply writes out the raw SGML without any formatting applied (it essentially ignores the stylesheet). These formatters are all subtypes of the context-type formatter.

Formatting Operations

A DoME text formatter generates a final document from a SGML document represented as a tree under the guidance of a stylesheet. The stylesheet specifies which operations are to be invoked whenever a node of a particular type is encountered in the tree. The stylesheet also dictates how the tree is traversed. The stylesheet expects that certain standard formatting operations will be supported by the formatter.

The standard text formatting operations are as follows:

(process-node f style-op nd) => unspecified

`Uses the stylesheet operation style-op to procecss the node nd.`

(process-children f style-op nd) => unspecified

`Uses the stylesheet operation style-op to process the children of nd.`

(format f style-op output root-node) => unspecified

`Uses the stylesheet operation style-op to process the SGML tree rooted at root-node.`

(initialize f) => f

(write-preamble f) => unspecified

(write-postamble f) => unspecified

(open f [file]) => unspecified

(close f) => unspecified

(cr f) => unspecified

(start-para f) => unspecified

(anchor f text) => unspecified

(link f text) => unspecified

(comment f text) => unspecified

(put-string f text)=> unspecified

(default-font f) => font

(set-default-font! f font) => unspecified

(current-font f) => font

(set-current-font! f font)=> unspecified

(default-style f) => style

(set-default-style! f style) => unspecified

(current-style f) => style

(set-current-style! f style)=> unspecified

Formatters that are subtyped from context-type contained in the ALter standard library `context.lib` will automatically meet these requirements. To create a new formatter the user can simply specialize those operations that need specialization in order to produce the new formatter.

Font Operations

(family font) => string

(set-family! font string) => unspecified

(bold font) => boolean

(set-bold! font boolean) => unspecified

(italic font) => boolean

(set-italic! font boolean) => unspecified

(size font) => quantity

(set-size! font quantity) => unspecified

(strikeout font) => boolean

(set-strikeout! font boolean) => unspecified

(underline font) => boolean

(set-underline! font boolean) => unspecified

Text-Style Operations

(sindent style) => quantity

(set-sindent! style quantity) => unspecified

(eindent style) => quantity

(set-eindent style quantity) => unspecified

(findent style) => quantity

(set-findent! style quantity) => unspecified

(sp-before style) => quantity

(set-sp-before! style quantity) => specified

(sp-after style) => quantity

(set-sp-after! style quantity) => specified

(alignment style) => symbol

(set-alignment! style symbol) => unspecified

(withprev style) => boolean

(set-withprev! style boolean) => unspecified

(withnext style) => boolean

(set-withnext! style boolean) => unspecified

Quantity Operations

(quantity? obj) => boolean

(quantity-equal? $quantity_1$ $quantity_2$) => boolean

(magnitude quantity) => number

(in number) => quantity

(in quantity) => number

(pt number) => quantity

(pt quantity) => number

(mm number) => quantity

(mm quantity) => number

(cm number) => quantity

(cm quantity) => number

(twips number) => quantity

(twips quantity) => number

(add q_1 ...) => quantity

(sub q_1 ...) => quantity

(mul q_1 ...) => quantity

(div q_1 ...) => quantity

Registering a Formatter

In order for DoME to recognize the existence of a user-defined text formatter, you must create a formatter description file and place it in one of the locations DoME searches. All text formatter files are named "dformats.dom". A text formatter description file describes one or more formatters and has the following form:

```
[[DoMEDocumentFormatList formatspec . . .]
```

Where formatspec looks like:

```
[DoMEDocumentFormatSpec
   formatName: 'menu-string'!
   sourceFile: 'pathname'!
   fileSuffix: 'string'!
]
```

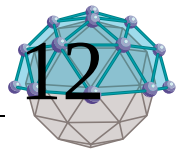
where

formatName	is a string that will be used to form the entry in the **FORMAT** menu of the document generator dialog.
sourceFile	is a string giving the filename of the text formatter's definition file.
fileSuffix	is a string appended to the filename specified from the document generator dialog.

If you have the environment variable DoMEDocumentFormats set, DoME will look there first for a text formatter file and then DoME will look in the files represented by the following Alter expression:

```
(construct
   (construct
      (construct
         (construct (dome-home) 'tools')
         "*")
      'etc')
   'dformats.dom')
```

Stylesheets



. . In This Chapter

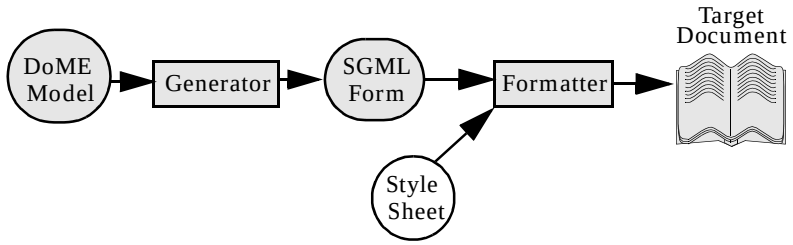
This chapter describes...

- The DoME stylesheet facility

You can write stylesheets for DoME text formatters using Alter. Such stylesheets direct the process of creating a final formatted document. Stylesheets are used in the second stage of the DoME document generation process.

The DoME document generation process is depicted in Figure 22. The non-grayed parts of the figure represent the stylesheet portions of the generation process.

Figure 22 DoME Document Generation Process



Overview

The stylesheet is a procedure or operation that takes two arguments. The return value of this procedure or operation is unspecified. The first argument is a node in the tree and the second argument is a formatter.

The stylesheet defines the operations the formatter should perform when it encounters a node of a certain type. These operations include such things as changing the font size, starting a new paragraph or writing out a string.

A stylesheet is an Alter operation with the following characteristics:

- It takes two arguments. The first is a node in an SGML tree. The second is an instance of a DoME text formatter.
- It only makes use of the standard tree querying operations on the SGML node to traverse and query the tree and its nodes.
- It invokes only standard text formatting operations on the DoME text formatter.
- Makes use of only standard operations defined on font-description, text-style and quantity-type.

Registering a Stylesheet

In order for DoME to recognize the existence of a user-defined stylesheet, you must create a stylesheet description file and place it in one of the locations DoME searches. All stylesheet files are named “docstyle.dom”. A stylesheet description file describes one or more stylesheets and has the following form:

```
[[DoMEStyleSheetList stylesheetspec . . .]
```

Where stylesheetspec looks like:

```
[DoMEStyleSheetSpec
   styleName: 'menu-string'!
   sourceFile: 'pathname'!
   documentTypes: [OrderedCollection
#'symbol'*!]
]
```

where

styleName	is a string that will be used to form the entry in the **STYLE SHEET** menu of the document generator dialog.
sourceFile	is a string giving the filename of the stylesheet's definition file.
documentTypes	is a collection of symbols describing which documents the stylesheet can be used in conjunction with.

If you have the environment variable DoMEDocumentFormats set, DoME will look there first for a text formatter file and then DoME will look in the files represented by the following Alter expression:

```
(construct
   (construct
      (construct
         (construct (dome-home) 'tools')
         "*")
      'etc')
   'docstyle.dom')
```

Example Stylesheet

The following code is an example of a stylesheet that specifies formatting for a document that is an instance of the invent DTD (see "Choosing a DTD" on page 88).[1] This style sheet approximates the formatting given to the Model Inventory Report in the example in "Summary Report" on page 61.

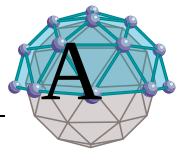
```
(find-operation style-op)
(add-method (style-op (invent-invent) self f)
   (let( (cf (current-font f))
         (cs (current-style f)) )
      (set-family! cf "Courier")
      (set-sp-before! cs (pt 12))
      (process-children f style-op self) ) )
```

1 This example may be found in the .../tools/alter/examples/invent.sty file that is delivered with DoME.

```
(add-method (style-op (invent-title) self f)
   (set-size! (current-font f) 14)
   (set-bold! (current-font f) #t)
   (set-sp-after! (current-style f) (pt 12))
   (set-underline! (current-font f) #t)
   (start-para f)
   (put-string f (contents self)) )

(add-method (style-op (invent-entry) self f)
   (start-para f)
   (process-children f style-op self) )

(add-method (style-op (invent-name) self f)
   (put-string f (contents self)) )

(add-method (style-op (invent-type) self f)
   (letrec( (cf (current-font f))
            (cfi (italic cf)) )
         (put-string f " (")
         (if (not cfi) (set-italic! cf #t))
         (put-string f (contents self))
         (if (not cfi) (set-italic! cf #f))
         (put-string f ")") ) )

(add-method (style-op (invent-prop) self f)
   (let( (cs (current-style f)) )
         (set-sp-before! cs (pt 2))
      (set-sp-after! cs (pt 0))
      (set-sindent! cs (in (/ 1 4)))
      (start-para f)
      (process-children f style-op self) ) )

(add-method (style-op (invent-propname) self f)
   (set-bold! (current-font f) #t)
   (put-string f (contents self))
   (put-string f ": ") )

(add-method (style-op (invent-propval) self f)
   (put-string f (contents self)) )

style-op ; stylesheet operation
```

The operation style-op is the stylesheet operation in this case. The last line of the file that contains the definitions for this stylesheet should evaluate to the operation style-op.

Scheme



. . In This Appendix

This appendix describes...

- References to Scheme related information (page 106)
- Scheme Elements not currently implemented in Alter (page 106)

Selected References

William Clinger and Jonathan Rees, editors. "Revised^4 Report on the Algorithmic Language Scheme". University of Oregon Technical Report CIS-TR-90-02.

http://www.cs.indiana.edu/scheme-repository/home.html

http://www-swiss.ai.mit.edu/scheme-home.html

Unimplemented R^4 Scheme Elements

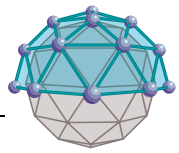
Alter is a nearly complete implementation of R^4 Scheme. The following elements of R^4 Scheme have *not* been implemented in Alter yet.

- Alter is not properly tail-recursive.
- Scheme continuations are not fully implemented. Currently, the extent of the escape procedure is restricted to the activation of the call-with-current-continuation that defined it. Therefore, its use is pretty much restricted to structured, non-local exits from loops or procedure bodies.
- Complex numbers.
- Promises.
- Macros.
- The following *essential* procedures are not implemented:

  case
  quasiquote
  call-with-current-continuation (call/cc) - partially implemented
  call-with-input-file
  call-with-output-file
  input-port?
  output-port?
  read
- The following *non-essential* procedures are not implemented:

  delay
  rationalize
  make-rectangular
  make-polar
  real-part
  imag-part
  magnitude
  angle
  force
  transcript-on
  transcript-off

Index



Symbols

"Operate" mouse button ix
"Select" mouse button ix
"Window" mouse button ix
^super 16

A

about this guide vi
activation stack. See Alter
add-method 16
Alter 3, 20, 66, 102
    activation stack
    environment viewer
    evaluator window 23
    expression
    external representation
    Programmer's Reference Manual vii
alter code block 8
assignment 24

B

bindings 16, 24, 25, 27
browser. See Projector
buttons, mouse ix

C

circumfrence 25
code generators vii
color value 71
components 61
conditional 9
constant 8
control flow 9
conventions viii

D

data flow 9
define 24
definitions 25
dictionary 17
document generators vii
Document Style Semantics and Specification Language 82
Document Type Definition 81
DoME
    Programmer's Manual vii
DXF 73

E

environment 24, 25, 27
    global 24, 25
    local 25
    top-level 24, 25
evaluator. See Alter
expression. See Alter
external representation. See Alter

F

face 71
file
    opening 50
find-operation 16
fork 8

G

get-property 61
graph
    printing 51
graphics context 71
guide
    contents vi
    conventions used viii
    description vi
    publication number vii
    related documents vii
    revision history vii
    version vii
    window / screen appearance viii

H

Hyper Text Markup Language 82

I

Interleaf ASCII Format 94
inventory 92

L

lambda 16
landscape 71
let 25
line-width 71

M

macros 81
make 15
Maker Interchange Format 94
markup 80
merge 8
method 22
mouse buttons ix

O

Oaklisp 15
object 15
object-oriented programming 15
operation 15

P

page description language 82, 86
paint 71
paint-color 71
paint-style 71
port 7
printing 51
procedure 7
program 25
Programmer's Manual, DoME vii
Programmer's Reference Manual, Alter vii
Projector 2, 6
  browser
  environment viewer
publication number vii

R

related documents vii
Rich Text Format 94

S

Scheme 3, 12
Scheme extension language vii
scope 25
set! 24
Standard Generalized Markup Language 81
statement block 7
style-op 104
stylesheet 102
subtype 15
supertype 15

T

type 15, 22

U

UNIX viii
User-defined functions
  registering 55

V

value 24
variable 8, 24
viewer. See Alter
viewer. See Projector

W

what-are-ypu 62
Windows viii
with-output-to-file 62
word-wrap 62
World Wide Web 82
write-inventory 63