

# ASCEND IV

## Advanced System for Computations in ENgineering Design

A portable mathematical modeling environment.

Release 0.8

Sept 26, 1997

Partial revision for release 0.9

June 20, 1998

### Authors:

Arthur Westerberg and research group

Department of Chemical Engineering, Carnegie Mellon University

Benjamin Allan, Vicente Rico-Ramirez, Mark Thomas, Kenneth Tyner

Other Helpful People Too Numerous To Mention Here

### Sponsors 1993-1997:

Computer Aided Process Design Consortium

Institute for Complex Engineered Systems

National Science Foundation grant for the Engineering Design Research Center

US Department of Energy

We are deeply indebted to the authors and contributors at large who created Tcl/Tk. Many thanks  
Dr. Osterhout!

### Keywords:

ASCEND, CONOPT, EDRC, FORTRAN, GAMS, GNU license, GUI, ICES, LSODE, Levenberg-Marquardt, MINLP, NLP, Newton, ODE, Tcl, Tk, UNIFAC, boundary value, chemical engineering, collocation, complex engineered system,, conditional modeling, copyleft, degrees of freedom, design, design research center, distillation, dynamic, engineering design, free software, freeware, initial value, initialization, interactive, large-scale, linear algebra, linear equations, mathematical modeling, mixed integer, modeling system, nonideal thermodynamics, nonlinear program, object oriented, optimization, ordinary differential equation, Pitzer vapor, reactive distillation, scalable, scaling, simulation, solving, structural analysis, Wilson liquid.

## Documentation Bird's Eye View

ASCEND IV	1
Advanced System for Computations in ENgineering Design	1
Documentation Bird's Eye View	2
Documentation Detail Map	4
A typical scenario for running the ASCEND system	18
Getting Started with ASCEND	22
Script	26
Library	38
Browser	50
Solver	60
The Data Probe Window	72
ASC PLOT	78
Display slave	86
ASCEND Units	90
The ASCEND Toolbox	96
The System Utilities Window	100
Font Selection Dialog	108
The Print Dialog	112
Solved simple modeling problems with ASCEND	116
A Conditional Modeling Example: Representing a Superstructure	122
A Simple Chemical Engineering Flowsheeting Example	144
The ASCEND predefined collection of models	162
The ASCEND IV language syntax and semantics	164
Units library	218
Brief History of ASCEND	232
INDEX	236



## Documentation Detail Map

<b>A typical scenario for running the ASCEND system</b>	<b>18</b>
<b>Getting Started with ASCEND</b>	<b>22</b>
:: Philosophy	22
- Getting the ASCEND system and installing it	22
- Starting ASCEND	23
o ASCENDDIST	23
o ASCENDHELP	23
o ASCENDLIBRARY	23
<b>Script</b>	<b>26</b>
Figure ASCEND's Script Window	26
:: The Script Menu Bar	27
- Script File Menu	27
o New File	27
o Read File	27
o Import File	27
o Exit ASCEND	27
o Save	27
o Save As	27
o Buffer List	27
- Script Edit Menu	27
o Record actions	27
o Select all	28
o Delete statements	28
o Cut	28
o Copy	28
o Paste	28
- Script Execute Menu	28
o Run statements selected	28
o Step through statements selected	28
- Script Options window	28
o Save all options and appearances for all windows	28
- Script View window	29
o Font	29
o Save Script appearance	29
o Save all appearances	29
- Script Tools window	29
- Script Help menu	29
o On SCRIPT	29
o On getting started with ASCEND	29
o About ASCEND IV	30
:: The Script Language	30
- Summary	30

	<i>o</i>	<i>&lt;arg&gt;</i>	30
	<i>o</i>	<i>&lt;a1,a2&gt;</i>	30
	<i>o</i>	<i>&lt;a1 a2&gt;</i>	30
	<i>o</i>	<i>[a1]</i>	30
	<i>o</i>	<i>[a,b]</i>	30
	<i>o</i>	<i>qlfdid</i>	30
	<i>o</i>	<i>qlfpid</i>	30
	<i>o</i>	<i>{}</i>	30
-		Quick reference:	31
	<i>o</i>	<i>ASSIGN</i>	31
	<i>o</i>	<i>BROWSE</i>	31
	<i>o</i>	<i>CLEAR_VARS</i>	31
	<i>o</i>	<i>COMPILE</i>	31
	<i>o</i>	<i>DELETE</i>	31
	<i>o</i>	<i>DISPLAY*</i>	31
	<i>o</i>	<i>INTEGRATE</i>	31
	<i>o</i>	<i>MERGE</i>	31
	<i>o</i>	<i>PLOT</i>	31
	<i>o</i>	<i>PRINT</i>	31
	<i>o</i>	<i>PROBE</i>	31
	<i>o</i>	<i>READ</i>	31
	<i>o</i>	<i>REFINE</i>	31
	<i>o</i>	<i>RESTORE*</i>	31
	<i>o</i>	<i>RESUME</i>	31
	<i>o</i>	<i>RUN</i>	31
	<i>o</i>	<i>SAVE*</i>	31
	<i>o</i>	<i>SHOW</i>	31
	<i>o</i>	<i>SOLVE</i>	31
	<i>o</i>	<i>WRITE</i>	31
-		Commands	32
	<i>o</i>	<i>ASSIGN</i>	32
	<i>o</i>	<i>BROWSE</i>	32
	<i>o</i>	<i>CLEAR_VARS</i>	32
	<i>o</i>	<i>COMPILE</i>	32
	<i>o</i>	<i>DELETE</i>	32
	<i>o</i>	<i>DISPLAY</i>	32
	<i>o</i>	<i>INTEGRATE</i>	32
	<i>o</i>	<i>MERGE</i>	33
	<i>o</i>	<i>OBJECTIVE</i>	33
	<i>o</i>	<i>PLOT</i>	33
	<i>o</i>	<i>PRINT</i>	33
	<i>o</i>	<i>PROBE</i>	33
	<i>o</i>	<i>READ</i>	33
	<i>o</i>	<i>REFINE</i>	34
	<i>o</i>	<i>RESTORE</i>	34
	<i>o</i>	<i>RESUME</i>	34

	<i>o</i>	<i>RUN</i>	34
	<i>o</i>	<i>SAVE</i>	34
	<i>o</i>	<i>SHOW</i>	34
	<i>o</i>	<i>SOLVE</i>	34
	<i>o</i>	<i>WRITE</i>	35
	<i>o</i>		35
::		Script Window Bindings	35
	<i>o</i>	<i>M1</i>	35
	<i>o</i>	<i>M1-Drag</i>	35
	<i>o</i>	<i>Shift-M1[-Drag]</i>	35
	<i>o</i>	<i>Double-M1</i>	35
	<i>o</i>	<i>Double-M1-Drag</i>	35
	<i>o</i>	<i>Triple-M1</i>	35
	<i>o</i>	<i>Triple-M1-Drag</i>	35
	<i>o</i>	<i>M2</i>	35
	<i>o</i>	<i>M2-Held-Down</i>	35
	<i>o</i>	<i>M3</i>	35
	<i>o</i>	<i>Control-M1</i>	35
	<i>o</i>	<i>Control-k</i>	36
	<i>o</i>	<i>Control-w</i>	36
	<i>o</i>	<i>Meta-w</i>	36
	<i>o</i>	<i>Control-y</i>	36
	<i>o</i>	<i>Meta-y</i>	36
<b>Library</b>			<b>38</b>
		Figure ASCEND Library Window.	38
		Figure Data structure used to store type defini-	
tions.		40	
::		Menu Bar	40
-		The file Menu	40
	<i>o</i>	<i>Read types from file</i>	40
	<i>o</i>	<i>Close window</i>	41
	<i>o</i>	<i>Exit ASCEND</i>	41
-		The Edit Menu	41
	<i>o</i>	<i>Create simulation</i>	41
	<i>o</i>	<i>Suggest methods</i>	41
	<i>o</i>	<i>Delete Simulation</i>	41
	<i>o</i>	<i>Delete all types</i>	42
		Figure The Create Simulation Dialog	42
-		The Display Menu	42
	<i>o</i>	<i>Code</i>	42
	<i>o</i>	<i>Ancestry</i>	42
	<i>o</i>	<i>Refinement hierarchy</i>	42
	<i>o</i>	<i>External functions</i>	42
	<i>o</i>	<i>Hide type</i>	42
	<i>o</i>	<i>UnHide type</i>	42
	<i>o</i>	<i>Hide/Show Fundamentals</i>	43

	Figure Select the fundamental type to Hide or	
Unhide.	43	
-	The Find Menu	43
	o <i>ATOM by units</i>	43
	o <i>Type by name</i>	43
	Figure The Library's Find Type dialog.	44
	o <i>Type by fuzzy name</i>	44
	o <i>Pending statements</i>	44
	o <i>To Display</i>	44
	o <i>To Console</i>	44
	o <i>To File</i>	44
-	The Options Menu	44
	o <i>Generate C binary</i>	45
	o <i>Simplify compiled equations</i>	45
	o <i>Save options</i>	45
-	The View Menu	45
	o <i>Font</i>	45
	o <i>Open automatically</i>	45
	o <i>Save appearance</i>	45
-	The export Menu	45
	o <i>Simulation to Browser</i>	46
	o <i>Simulation to Solver</i>	46
	o <i>Simulation to Probe</i>	46
-	The help Menu	46
	o <i>On LIBRARY</i>	46
::	Type Refinement Hierarchy Window	46
	Figure The Type Refinement Window.	47
	Figure The Parts window displays the parts.	47
	Figure The Hierarchy Roots Window.	49
<b>Browser</b>		<b>50</b>
	Figure ASCEND's Browser window.	50
::	The Menu Bar	51
-	BROWSER File menu	51
	o <i>Read values</i>	51
	o <i>Write values</i>	51
	o <i>Close window</i>	51
	o <i>Exit ASCEND</i>	51
-	BROWSER Edit Menu	51
	o <i>Run method</i>	51
	o <i>Clear Vars</i>	52
	o <i>Set value</i>	52
	o <i>Refine</i>	52
	o <i>Merge</i>	52
	o <i>Compile</i>	53
	o <i>Resume Compilation</i>	53
	o <i>Create Part</i>	53

-	BROWSER Display menu	53
	<i>o</i> Attributes	53
	<i>o</i> Relations	53
	<i>o</i> Conditional Relations	53
	<i>o</i> Logical Relations	53
	<i>o</i> Conditional Logical Relations	54
	<i>o</i> Whens	54
	<i>o</i> Plot	54
	<i>o</i> Statistics	54
-	BROWSER Find menu	54
	<i>o</i> By name	54
	<i>o</i> By type	54
	<i>o</i> Aliases	56
	<i>o</i> Where created	56
	<i>o</i> Clique	57
	<i>o</i> Eligible variables	57
	<i>o</i> Active Relations	57
	<i>o</i> Operands	57
	<i>o</i> Pendings	57
-	BROWSER Options menu	57
	<i>o</i> Hide Passed Parts	57
	<i>o</i> Suppress Atoms	57
	<i>o</i> Display Atom Values	57
	<i>o</i> Check Dimensionality	58
	<i>o</i> Save Options	58
	<i>o</i> Hide Names	58
	<i>o</i> UnHide Names	58
-	BROWSER view menu	58
	<i>o</i> Font	58
	<i>o</i> Open automatically	58
	<i>o</i> Save window appearance	58
-	BROWSER Export menu	58
	<i>o</i> to Solver	58
	<i>o</i> Many to Probe	58
	Figure Filtering instances sent to the Probe	59
	<i>o</i> Item to Probe	59
-	BROWSER Help menu	59
	<i>o</i> On BROWSER	59
	<b>Solver</b>	<b>60</b>
	Figure Solver Window	60
::	The Solver Menu Bar	61
-	Solver File Menu	61
	<i>o</i> Close Window	61
	<i>o</i> Exit ASCEND	61
-	Solver Edit Menu	61
	<i>o</i> Remove instance	61



	o	Select objective	61
-		Solver Display Menu	61
	o	Status	61
	o	Unattached variables	61
	o	Unincluded relations	61
	o	Incidence matrix	61
		Figure The Incidence Matrix	62
-		Solver Execute Menu	62
	o	Solve	62
	o	Single step	62
	o	Integrate	63
-		Solver Analyze menu	63
	o	Reanalyze	63
	o	Debugger	63
	o	Overspecified	63
	o	Find dependent eqns.	63
	o	Find unassigned eqns.	63
	o	Evaluate unincluded eqns.	63
	o	Find vars near bounds	63
	o	Find vars far from nominal	64
-		Solver View Menu	64
	o	Font	64
	o	Open automatically	64
	o	Save Solver appearance	64
-		Solver Export Menu	64
	o	to Browser	64
	o	to Probe	64
::		Solver Button Bar	64
	o	Solver Select Button	64
	o	Solver Options Button	65
	o	Halt Button	65
-		General parameters page	65
		Figure General Parameter Page	65
::		Available Solvers	67
-		QRSlv	67
::		Debugger	69
		Figure The Debugger Window	70
		<b>The Data Probe Window</b>	<b>72</b>
::		Overview	72
		Figure Probe window	73
::		The File menu	73
	o	New buffer	73
	o	Read file	74
	o	Save	74
	o	Save as	74
	o	Print	74

	<i>o</i>	<i>Close window</i>	74
	<i>o</i>	<i>Exit ASCEND</i>	74
	<i>o</i>	<i>Buffer list</i>	74
::		The Edit Menu	74
	<i>o</i>	<i>Highlight all</i>	74
	<i>o</i>	<i>Remove selected names</i>	74
	<i>o</i>	<i>Remove all names</i>	74
	<i>o</i>	<i>Remove UNCERTAIN names</i>	74
	<i>o</i>	<i>Copy</i>	74
::		The View Menu	75
	<i>o</i>	<i>Font</i>	75
	<i>o</i>	<i>Open automatically</i>	75
	<i>o</i>	<i>Save window appearance</i>	75
::		The Export Menu	75
	<i>o</i>	<i>to Browser</i>	75
	<i>o</i>	<i>to Display</i>	75
::		The Probe Filter	75
-		The Help Menu	75
		Figure Probe import filter	76
		<b>ASC PLOT</b>	<b>78</b>
::		Plot maker	78
		Figure The Ascend Plot Window	78
-		The Edit Menu	79
-		The Execute Menu	79
		Figure The Create Data Window	80
-		The Display Menu	81
		Figure The Graph Generics Window	82
		Figure Complete Plot	83
::		Navigation	84
		Figure Phase Diagram	85
		<b>Display slave</b>	<b>86</b>
::		Overview	86
		Figure Display slave window	86
::		Display File Menu	87
	<i>o</i>	<i>Print</i>	87
	<i>o</i>	<i>Close window</i>	87
	<i>o</i>	<i>Exit ASCEND</i>	87
::		Display Edit Menu	87
	<i>o</i>	<i>Cut</i>	87
	<i>o</i>	<i>Copy</i>	87
	<i>o</i>	<i>Paste</i>	87
::		Display View Menu	87
	<i>o</i>	<i>Show comments in code</i>	87
	<i>o</i>	<i>Save Display options</i>	87
	<i>o</i>	<i>Font</i>	87
	<i>o</i>	<i>Open automatically</i>	88

o	Save window appearance	88
-	Font	88
-	Open automatically	88
-	Display Help Menu	88
::	Title line	88
	<b>ASCEND Units</b>	<b>90</b>
::	The Menu Bar	90
o	Units vs dimensions	90
o	Typical use	90
	Figure The Units of measure window	91
-	Units File Menu	91
o	Read file	91
o	Save file	91
o	Close window	91
o	Exit ASCEND	92
-	Units Edit Menu	92
o	Set precision	92
o	Set basic units	92
-	Units Display Menu	92
o	Show all units	92
-	Units View Menu	92
o	SI(MKS) set	92
o	US Engineering set	92
o	CGS set	92
o	Font	92
o	Open automatically	92
o	Save window appearance	92
-	Units Help Menu	93
::	An essay on units vs dimensions	93
-	On UNITS	94
	<b>The ASCEND Toolbox</b>	<b>96</b>
	Figure The ASCEND Toolbox window.	96
::	Exit	97
::	Ascplot	97
::	Help	97
::	Utilities	97
::	Internals	97
::	Bug Report	98
	<b>The System Utilities Window</b>	<b>100</b>
::	Overview	100
	Figure The System Utilities window manages	
	ASCEND's interaction with the operating system and with other pro-	
	grams.	100
::	Variables	101
-	WWW Root URL	101
-	WWW Restart Command	102

-	WWW Startup Command	102
-	ASCENDLIBRARY Path*	102
-	Scratch Directory	103
-	Working Directory	103
-	Plot Program Type	103
-	Plot Program Name	103
-	Text Edit Command	103
-	Postscript Viewer	104
-	Spreadsheet Command	104
-	Text Print Command	104
-	PRINTER Variable*	104
-	ASCENDDIST Directory*	104
-	TCL_LIBRARY Environment Variable*	105
-	TK_LIBRARY Environment Variable*	105
::	Buttons	105
-	OK	106
-	Save	106
-	Read	106
-	More	106
-	Help	106
	<b>Font Selection Dialog</b>	<b>108</b>
::	Overview	108
	Figure The font selection dialog.	108
::	Font Menu	109
::	Style Menu	109
::	Cancel Button	109
::	OK Button	109
::	Current Font Sample	110
::	Font Sampler Area	110
::	Point Size Slider	110
::	Current Font Selection	110
::	Setting the Default Font	110
	<b>The Print Dialog</b>	<b>112</b>
::	Overview	112
	Figure The print dialog.	112
::	Settings	112
-	Destination	112
-	Printer	114
-	Name of file	114
-	Enscript flags	114
-	User print command	114
::	Buttons	115
-	OK	115
-	Help	115
-	Cancel	115
	<b>Solved simple modeling problems with ASCEND</b>	<b>116</b>

::	Roots of a polynomial	116
-	Problem statement	117
-	Answer	117
::	Numerical integration of tabular data	118
-	Problem statement	118
-	Answer	119
<b>A Conditional Modeling Example: Representing a Superstructure</b>		<b>122</b>
Figure Superstructure used in the example of the application of the when statement122		
::	The WHEN Statement	122
::	The Problem Description	124
::	The Code	124
<b>A Simple Chemical Engineering Flowsheeting Example</b>		<b>144</b>
::	The problem description	144
::	The code	145
<b>The ASCEND predefined collection of models</b>		<b>162</b>
	<i>o system.a4l</i>	162
	<i>o atoms.a4l</i>	162
	<i>o Typical use of library files</i>	163
	<i>o Examples and scripts</i>	163
<b>The ASCEND IV language syntax and semantics</b>		<b>164</b>
::	Preliminaries	165
-	Punctuation	166
	<i>o keywords:</i>	166
	<i>o (* *)</i>	167
	<i>o ( )</i>	167
	<i>o { }</i>	167
	<i>o [ ]</i>	168
	<i>o .</i>	168
	<i>o ..</i>	168
	<i>o :</i>	168
	<i>o ::</i>	168
	<i>o ;</i>	168
-	Basic Elements	168
	<i>o L</i>	169
	<i>o M</i>	169
	<i>o T</i>	169
	<i>o E</i>	169
	<i>o Q</i>	169
	<i>o TMP</i>	170
	<i>o LUM</i>	170
	<i>o P</i>	170
	<i>o S</i>	170
	<i>o C</i>	170
-	Basic Concepts	175

::	Data Type Declarations	178
	o <i>UNIVERSAL</i>	179
-	Models	179
	o <i>MODEL</i>	179
	o <i>foo</i>	179
	o <i>bar</i>	179
	o <i>column(n,s)</i>	180
	o <i>flowsheet</i>	180
-	Sets	181
	o <i>:=</i>	181
	o <i>UNION[setlist]</i>	181
	o <i>+</i>	181
	o <i>INTERSECTION[]</i>	182
	o <i>*</i>	182
	o <i>-</i>	182
	o <i>CARD[set]</i>	182
	o <i>CHOICE[set]</i>	182
	o <i>IN</i>	182
	o <i>SUCH_THAT (* 4 *)</i>	183
	o <i>/</i>	183
-	Constants	184
	o <i>real_constant</i>	184
	o <i>integer_constant</i>	184
	o <i>symbol_constant</i>	184
	o <i>boolean_constant</i>	184
	o <i>:=</i>	185
-	Variables	185
	o <i>ATOM</i>	185
	o <i>DEFAULT, DIMENSION, and DIMENSIONLESS</i>	185
	o <i>real</i>	186
	o <i>integer</i>	186
	o <i>boolean</i>	186
	o <i>symbol</i>	186
	o <i>:=</i>	186
	o <i>DATA (* 4+ *)</i>	187
	o	188
-	Relations	188
	o <i>=, &gt;=, &lt;=, &lt;, &gt;, &lt;&gt;</i>	189
	o <i>MAXIMIZE, MINIMIZE</i>	189
	o <i>+</i>	189
	o <i>-</i>	189
	o <i>*</i>	189
	o <i>/</i>	189
	o <i>^</i>	189
	o <i>-</i>	189

	<i>o</i>	<i>ordered_function()</i>	189
	<i>o</i>	<i>SUM[term set]</i>	189
	<i>o</i>	<i>PROD[term set]</i>	190
	<i>o</i>	<i>MAX[term set]</i>	190
	<i>o</i>	<i>MIN[term set]</i>	190
-		Derivatives in relations (* 4+ *)	190
-		External relations	190
-		Conditional relations (* 4 *)	191
-		Logical relations (* 4 *)	191
-		NOTES (* 4 *)	191
::		Declarative statements	194
	<i>o</i>	<i>IS_A</i>	195
	<i>o</i>	<i>IS_REFINED_TO</i>	195
	<i>o</i>	<i>ALIASES (* 4 *)</i>	195
	<i>o</i>	<i>ALIASES/IS_A (*4*)</i>	195
	<i>o</i>	<i>WILL_BE (* 4 *)</i>	195
	<i>o</i>	<i>ARE_THE_SAME</i>	195
	<i>o</i>	<i>WILL_BE_THE_SAME (* 4 *)</i>	196
	<i>o</i>	<i>WILL_NOT_BE_THE_SAME (* 4 *)</i>	196
	<i>o</i>	<i>ARE_NOT_THE_SAME (* 4+ *)</i>	196
	<i>o</i>	<i>ARE_ALIKE</i>	196
	<i>o</i>	<i>FOR/CREATE</i>	196
	<i>o</i>	<i>FOR/CHECK</i>	196
	<i>o</i>	<i>SELECT/CASE (*4*)</i>	196
	<i>o</i>	<i>CONDITIONAL (*4*)</i>	196
	<i>o</i>	<i>WHEN/CASE (* 4 *)</i>	196
	<i>o</i>	<i>IS_A</i>	197
	<i>o</i>	<i>IS_REFINED_TO</i>	197
	<i>o</i>	<i>ALIASES (* 4 *)</i>	198
	<i>o</i>	<i>ALIASES/IS_A (*4*)</i>	199
	<i>o</i>	<i>WILL_BE (* 4 *)</i>	200
	<i>o</i>	<i>ARE_THE_SAME</i>	200
	<i>o</i>	<i>WILL_BE_THE_SAME (* 4 *)</i>	202
	<i>o</i>	<i>WILL_NOT_BE_THE_SAME (* 4 *)</i>	202
	<i>o</i>	<i>ARE_NOT_THE_SAME (* 4+ *)</i>	202
	<i>o</i>	<i>ARE_ALIKE</i>	202
	<i>o</i>	<i>FOR/CREATE</i>	203
	<i>o</i>	<i>SELECT/CASE (*4*)</i>	204
	<i>o</i>	<i>CONDITIONAL (*4*)</i>	204
	<i>o</i>	<i>WHEN/CASE (* 4 *)</i>	204
::		Procedural statements	204
	<i>o</i>	<i>METHODS</i>	204
	<i>o</i>	<i>ADD METHODS IN type_name; (*4*)</i>	
205			
	<i>o</i>	<i>REPLACE METHODS IN type_name;</i>	
(*4*)			205

	<i>o</i>	<i>ADD METHODS IN DEFINITION</i>	
<i>MODEL;</i>			205
	<i>o</i>	<i>METHOD</i>	205
	<i>o</i>	<i>FOR/DO statement</i>	206
	<i>o</i>	<i>IF</i>	207
	<i>o</i>	<i>SWITCH (* 4 *)</i>	207
	<i>o</i>	<i>CALL</i>	207
	<i>o</i>	<i>RUN</i>	207
::		Parameterized models	208
-		The parameter list	208
-		The WHERE list	210
-		The assignment list	210
-		Refining parameterized types	210
::		Miscellany	211
-		Variables for solvers	211
	<i>o</i>	<i>solver_var</i>	211
	<i>o</i>	<i>lower_bound</i>	211
	<i>o</i>	<i>upper_bound</i>	211
	<i>o</i>	<i>nominal</i>	211
	<i>o</i>	<i>fixed</i>	212
	<i>o</i>	<i>generic_real</i>	212
	<i>o</i>	<i>solver_semi, solver_integer,</i>	
<i>solver_binary</i>			212
	<i>o</i>	<i>ivpsystem.a4l</i>	212
-		Supported attributes	213
	<i>o</i>	<i>(* 4+ *)</i>	213
-		Single operand real functions:	213
	<i>o</i>	<i>exp()</i>	213
	<i>o</i>	<i>ln()</i>	213
	<i>o</i>	<i>sin()</i>	213
	<i>o</i>	<i>cos()</i>	213
	<i>o</i>	<i>tan()</i>	213
	<i>o</i>	<i>arcsin()</i>	213
	<i>o</i>	<i>arccos()</i>	213
	<i>o</i>	<i>arctan()</i>	213
	<i>o</i>	<i>erf()</i>	213
	<i>o</i>	<i>sinh()</i>	213
	<i>o</i>	<i>cosh()</i>	213
	<i>o</i>	<i>tanh()</i>	214
	<i>o</i>	<i>arcsinh()</i>	214
	<i>o</i>	<i>arccosh()</i>	214
	<i>o</i>	<i>arctanh()</i>	214
	<i>o</i>	<i>lnm()</i>	214
	<i>o</i>	<i>abs()</i>	214
-		Logical functions	215
	<i>o</i>	<i>SATISFIED() (*4*)</i>	215



-	UNITS definitions	215
<b>Units library</b>		<b>218</b>
::	Units	218
::	The basic units in an extended SI MKS system	218
::	Units defined in measures.a4l, the default system units library of atoms.a4l.	219
<b>Brief History of ASCEND</b>		<b>232</b>

# CHAPTER 1 A TYPICAL SCENARIO FOR RUNNING THE ASCEND SYSTEM

The ASCEND system is a modeling environment. We have designed it to allow modelers to pose, debug and solve or optimize models described by up to of the order of a hundred thousand nonlinear algebraic equations on a conventional UNIX workstation or PC running Windows NT4 or Windows 95. The ASCEND system comprises three major parts: the ASCEND modeling language for posing models, the ASCEND interactive environment to allow users to compile, debug and execute models, and a suite of solvers and optimizers.

Detailed information on modeling appears in Chapter 15, Chapter 16, Chapter 17, and the Howto-ASCEND documents. The primary aim of this book is to describe the graphic user interface of ASCEND (a moving target if ever there was one) and to serve as a language reference (Chapter 19). You would typically proceed as follows to use the ASCEND GUI for modeling.

1. Using your favorite text editor (e.g., xemacs<sup>1</sup>), you will create a model of the problem you wish to solve in the ASCEND modeling language. ASCEND models are type definitions. Each model typically includes a declaration of the parts from which it is constructed, including variables, instances of previously defined types and arrays of any of these. Each model also includes the equations it adds to the model definition over and above those equations that its parts will provide. Finally if you abide by our advice on model writing, you will also write three or four methods that you will later run interactively on the compiled model instance to prepare it to be solved.

If the model is particularly complex, you will probably create your model using types defined earlier by yourself and others. For chemical process flowsheet models, we provide a library of types. We also provide a file that contains most of the types of variables and constants anyone would use to construct a model.

2. Start up the interactive ASCEND user interface by typing

---

1. Xemacs is a very powerful text editor which is widely used on UNIX workstations. It is available for free for both UNIX workstations and PCs through the WWW (search on xemacs).

'ascend' on the Unix shell command line or double clicking on the ASCEND icon on a PC. A number of different tool sets, each represented by a special window, open up on the screen. The one you will likely focus on at first is the SCRIPT window. In this window under Tools, you will open whichever tool window you want to work with at first, probably the LIBRARY tool set which provides tools to load text files containing ASCEND models. You will likely move and/or resize these windows.

3. Using a tool in the LIBRARY tool set, you will load the files containing the previously defined types of which your model makes use. You will open last the file containing the model you have just written. Unless you are incredibly skilled and/or lucky, you will see several error messages indicating that you have not correctly posed your model as the system attempts to load your new file. Moving back to your favorite editor you will correct syntax errors discovered by this file loading process, attempt again to load, make more corrections, etc.
4. Once your new model description can pass the loading process without errors, you will compile a simulation for it. Again there could be errors.
5. You will export this compiled simulation to the BROWSER tool set so you can look at the model, examining all its parts. If there were compiler errors, you may use tools in this tool set to aid you to find exactly what you have done wrong in posing the model.

For example, if it is a particularly complex model, you will methodically examine it to see if you have configured it as you wanted.

6. Once you are through inspecting the model and have removed all the errors that of which you are aware, you will prepare the model to be solved. This you will do by asking the system to execute the methods you should have written to go with your model description. If you abide by the style of modeling that we strongly advocate, your model will have these methods attached to it -- written before your first attempt to compile it. These methods will be for setting initial values for the variables, for scaling the variables, and for setting the "fixed" flags for a sufficient number of variables to make the model instance well-posed. To be well-posed means a number of things, among them that the model has the same number of variables to be calculated as equations available for calculating them.
7. You will next export the model to the SOLVER tool set. When importing a model, the SOLVER tool set analyzes the model to discover how many variables and equations are in its description.

If it is not an optimization problem, the SOLVER looks to see if it is well-posed and, if not, will issue warning messages and open up an interactive tool provided to aid you to make it well-posed right then and there. What you learn while using this tool you will likely encode right away into the model description so the next time you compile this model, it will become well-posed without this interactive step.

8. You can interactively choose among the available solvers and will most likely choose our nonlinear equation solving solver. With fingers crossed, you will ask the solver to start solving.
9. Whether or not it solves successfully, you will likely return to the BROWSER to inspect the results as you can view the value for every variable and equation residual in the model using the BROWSER. If the solving process fails, you can select tools both in the BROWSER and the SOLVER to look for the likely problem. For example, you may have posed your problem and its initial conditions such that the solution is out of bounds. A tool will tell you if the SOLVER has driven any of the variables in the model to their bounds. Another will tell you if some of your variables are poorly scaled. Yet another will investigate the model to see if it is locally singular, and if it is, that tool will report to you exactly which equations (by name) have given it reason to believe that to be so. (In the near future, this tool will also tell you that you should change what you are fixing and what you are calculating to remove this singularity, if such a move would prove useful. It will give you a list of variables from which to choose for each of these trades.)
10. You may wish to see the output in units different from those currently used. Opening the UNITS tools set will allow you to change wholesale from SI to American Engineering and/or to change individual units such as those for pressure from bars to atm.
11. You may have opened the SCRIPT tool set before loading the model files. Before doing all the above steps within ASCEND, you may then have activated a tool to record all the steps you will subsequently take to load, compile, initialize and solve the model. This tool will construct a script from the steps it sees you taking. You will likely then edit this script, for example to delete some of the missteps you have taken, and then save it. You may also pick out any of the steps in the script and execute them at any time rather than look for the tools in the tool set windows. You would use a script to aid you to repeat all the above steps quickly while you are debugging a model. You will also prepare a script to hand your model to someone else to execute. Indeed, your first

experience with ASCEND may be to run a script that someone else has provided so you can be sure to run your first example successfully.

# CHAPTER 2 GETTING STARTED WITH ASCEND

## 2.1 PHILOSOPHY

Our goal is to create a set of very powerful modeling/solving tools. A side effect is that users can often find uses for the tools we did not anticipate. Another is that, while we have tried to build a user interface that lets every user from beginner to expert use our tools (or theirs combined with ours!) in a comfortable fashion, we have almost certainly erred on the side of giving the user too much control. Knowing that to be the case, the user should fearlessly dive in and try to use the system, at first by doing some of the simple problems we have provided in this documentation. The first step, of course, is to start the system up, the purpose of this section.

This chapter largely out of date. See [howto-ascend.pdf](http://www.cs.cmu.edu/~ascend/howto-ascend.pdf) and instructions on [http://www.cs.cmu.edu/~ascend/ascend\\_download.htm](http://www.cs.cmu.edu/~ascend/ascend_download.htm) instead

### 2.1.1 GETTING THE ASCEND SYSTEM AND INSTALLING IT

ASCEND is available through our Web page. Using your web browser, go the URL

<http://www.cs.cmu.edu/~ascend/Home.html>

Follow the instructions (the ftp link) there to download ASCEND.

#### Installing the UNIX version

If you are downloading a version to run on a UNIX workstation, then find someone who is a UNIX expert to help you. The process will involve transferring the source files for ASCEND along with a MAKE file. The MAKE file will allow a UNIX specialist to compile ASCEND and get it ready for use. There are detailed instructions that come with

this version to help in installing it. (Your expert's expertise may be very minimally required for installing it on most systems.)

### Installing the PC version

If you are downloading to a PC running under either NT or Windows 95, you will be downloading ASCEND4.zip. Uncompress using WinZip, double click on install.exe and follow the instructions.

## **2.1.2 STARTING ASCEND**

### **2.1.2.1 FOR PC USERS ONLY**

On the PC, simply double click on the ASCEND icon.

### **2.1.2.2 FOR UNIX USERS ONLY**

The ASCEND IV interface is an open system written in TCL on top of several libraries of C code. The users are expected to customize it to suit their individual tastes.

We assume users are at least aware of the existence of environment variables and X resources. If you are not, contact your UNIX expert or the person who installed ASCEND on your system.

### Environment Variables

Normally, if you are running on **UNIX** your system administrator will have set up a shell script to let you run ASCEND simply by typing `ascend`. To see if this is true, try typing

```
ascend -h
```

If this doesn't work, you may need to define the following environment variables in your `.login` (or perhaps `.profile`) file, or if you can find the ASCEND binary, it will frequently run without requiring a shell script.

#### **ASCENDDIST**

points to the directory where the ASCEND code has been installed.

```
setenv ASCENDDIST /usr1/ballan/asc4/test
```

#### **ASCENDHELP**

points to the ASCEND help file tree on your system. The tree does not have to reside with the rest of the distribution, though it may. This should have been configured for you when was installed.

#### **ASCENDLIBRARY**

is a colon-separated list of directories where ASCEND looks for files which are required by other files or which are read into ASCEND from a script without giving a complete path name. If you do not define ASCENDLIBRARY, the system will make guesses that usually work.

```
setenv ASCENDLIBRARY $ASCENDDIST/models/examples:$ASCENDDIST/models/libraries
```

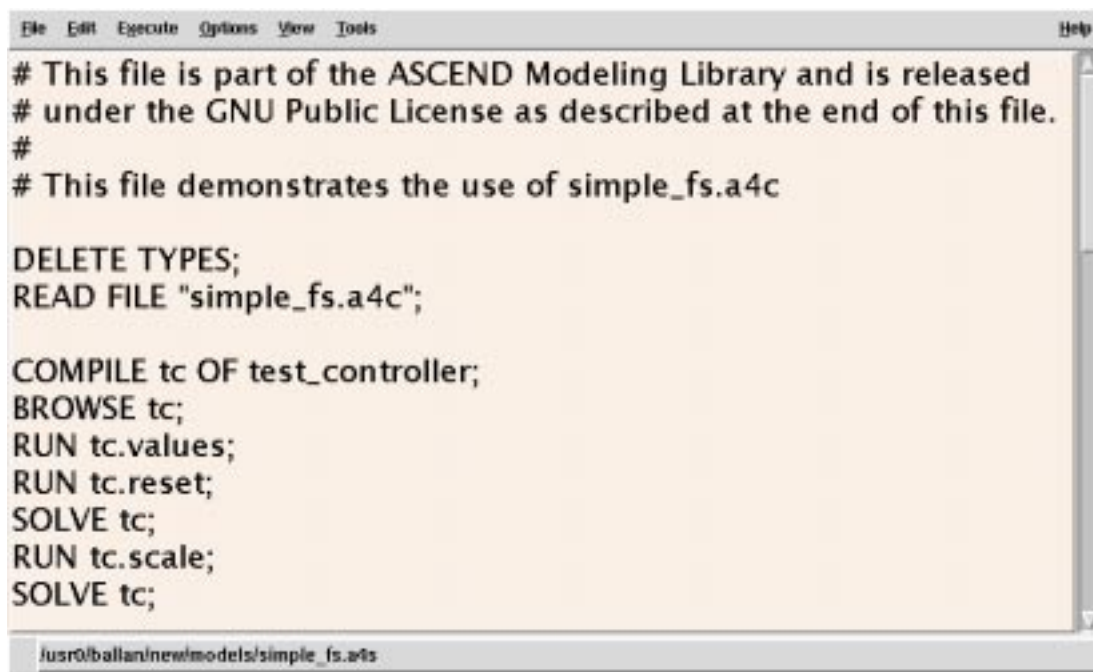






## CHAPTER 3 SCRIPT

The Script Utility (see Figure 3-1) allows us to record the process of solving a model, or any other user interface process. Once this process is recorded in the form of a script, the script can be repeated either fully or in part. The solution process for a given model can be communicated to another modeler by distributing a script saved to a file. Following is an outline of the various menus and buttons on the script window along with a library of the ASCEND commands which can be recorded.



```
# This file is part of the ASCEND Modeling Library and is released
# under the GNU Public License as described at the end of this file.
#
# This file demonstrates the use of simple_fs.a4c

DELETE TYPES;
READ FILE "simple_fs.a4c";

COMPILE tc OF test_controller;
BROWSE tc;
RUN tc.values;
RUN tc.reset;
SOLVE tc;
RUN tc.scale;
SOLVE tc;
```

/usr0/ballan/new/models/simple\_fs.a4s

Figure 3-1 ASCEND's Script Window

## 3.1 THE SCRIPT MENU BAR

### 3.1.1 SCRIPT FILE MENU

The script file menu provides several functions for managing script files. The script utility may contain multiple scripts but will only display one at any given time. Upon startup a scratch workspace is provided

<b>New File</b>	Request a buffer name and creates a new buffer with this name.
<b>Read File</b>	Requests a filename through the file selection box and proceeds to load this file (which is assumed to contain ASCEND Script and/or Tcl statements) into a new Script window buffer. No error checking is performed on the loaded file.
<b>Import File</b>	Requests a filename through the file selection box and appends the text contained in this file to the end of the current buffer.
<b>Exit ASCEND</b>	Exit the ASCEND system. You will asked to confirm that you wish to do this.
<b>Save</b>	Saves the text in the current script buffer window to the current script file (indicated by the filename at the bottom of the script window). The existing file is overwritten.
<b>Save As</b>	Request a filename through the file selection box and saves the text in the current script buffer window to this file. If the specified file exists, it is overwritten.
<b>Buffer List</b>	A list of scripts used in the current ASCEND session is displayed at the bottom of the file menu. A script can be redisplayed in the script window by selecting it from the buffer list. This window contains the words "License-Warranty.tcl" when you first start ASCEND (which is the initial contents of the Script buffer).

Note: if you alter the contents (for example, clear it), the system will restore the modified contents and not the original contents. The file you originally read remains unchanged unless you save to it.

### 3.1.2 SCRIPT EDIT MENU

<b>Record actions</b>	When the record function is activated a log of interface events with defined ASCEND Script commands is appended to the end of the current script window buffer. Most, but not all, interface events have
-----------------------	--

corresponding script commands. The record function can be turned on and off by toggling the pull down button on the grill menu or the record button at the bottom of the script window.

<b>Select all</b>	Selects (highlights) all text in the window. (A known bug exists here -- if you do not place the cursor into the text buffer the first time you use this tool, the highlighting may not occur although the text is in fact highlighted.)
<b>Delete statements</b>	Removes (cuts) the <i>selected</i> text. The removed text is NOT saved for later pasting.
<b>Cut</b>	Cut highlighted text to the computer paste buffer. You can paste this text into any application that supports cut, copy and paste -- e.g., into Framemaker or Excel.
<b>Copy</b>	Copy highlighted text to the computer paste buffer. You can paste this text into any application that supports cut, copy and paste -- e.g., into Framemaker or Excel.
<b>Paste</b>	Paste the contents of the computer paste buffer into the Script buffer at the point of the cursor.

### 3.1.3 SCRIPT EXECUTE MENU

<b>Run statements selected</b>	This button takes the selected text, breaks it into statements delimited by any semicolons (;) that appear in the selection, and executes each statement in the Tcl global environment.
<b>Step through statements selected</b>	This button allows you to single step through the highlighted statements. Two windows open: a small window that allows you to proceed to the next statement (next button), change from single step to running the rest of the script automatically (go button) or stop (stop button) and the Display where, while single stepping, you will see the statement being executed.

### 3.1.4 SCRIPT OPTIONS WINDOW

<b>Save all options and appearances for all windows</b>	This tool saves the complete current appearance of the ASCEND windows and all the options selected anywhere in the system. It writes the text file <i>ascend.ad</i> and a number of text files ending with <i>.a4o</i> into the subdirectory <i>ascdata</i> in your "home" directory. The <i>ascend.ad</i> file is a collection of the information placed into the <i>.a4o</i> files at the time you run this instruction. Its main role is to aid in debugging.
---	--

At any time, you can go into any of the windows and use the appropriate Save “window” appearance button to write a new .a4o file for that window. You can also save new options where such tools exist throughout the system, resulting in other .a4o files. The new information in these .a4o files will not be reflected in ascend.ad file, but it will be what is used to set window positions, etc., overriding what is in the ascend.ad file.

### 3.1.5 SCRIPT VIEW WINDOW

<b>Font</b>	Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.
<b>Save Script appearance</b>	Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a .a4o text file for this window which the system stores in the subdirectory <i>ascdata</i> in your “home” directory.
<b>Save all appearances</b>	Saves a .a4o file for all the ASCEND windows. (See the above instruction to see the purpose of these files.)

### 3.1.6 SCRIPT TOOLS WINDOW

Use the tools in this menu to open other windows in ASCEND when they are closed or iconified.

This menu lists the major ASCEND windows. Selecting one of them will open that window on your screen. See the help manuals for these windows to find out more about them. This menu has almost exactly the same content as the ASCEND toolbox window. See the documentation corresponding to the toolbox for more details.

### 3.1.7 SCRIPT HELP MENU

<b>On SCRIPT</b>	Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.
<b>On getting started with ASCEND</b>	Brings up a text description of where to look for help on getting started with using ASCEND -- the howto book (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look in any local (but perhaps outdated) copy of the documentation.

**About ASCEND IV** Brings up a window telling you briefly about the GNU license and other information about the ASCEND IV software. This same window opens the first time you start ASCEND on your computer.

## 3.2 THE SCRIPT LANGUAGE

### 3.2.1 SUMMARY

Script keywords are commands defined for ASCEND (in CAPS) which may be used on the commandline or in the Script. Keywords are actually Tcl functions which encapsulate one or more of the C primitives and other Tcl procedures, so that the user can conveniently emulate button presses. A working knowledge of tcl is not necessary to benefit from the Script's functionality; however, the tcl literate user will be able to create very powerful scripts.

Each keyword takes 0 or more arguments. The use of arguments is given in the following syntax:

<b>&lt;arg&gt;</b>	indicates the use of <b>arg</b> is required.
<b>&lt;a1,a2&gt;</b>	indicates that the use of either <b>a1</b> or <b>a2</b> is required
<b>&lt;a1 a2&gt;</b>	indicates use of both <b>a1</b> and <b>a2</b> required. Usually written <b>&lt;a1&gt; &lt;a2&gt;</b>
<b>[a1]</b>	indicates the use of <b>a1</b> is optional.
<b>[a,b]</b>	indicates that either <b>a</b> or <b>b</b> is optional, but not both.
<b>qlfdid</b>	is short for 'QuaLiFieD IDentifier'
<b>qlfpid</b>	is short for 'QuaLiFied Procedure IDentifier'

OF, WITH, TO, and other args in all CAPS are modifiers to the keyword which make it do different things.

**{ }** It is generally best to **enclose all object names and units** in {braces} to prevent Tcl from performing string substitution.

### 3.2.2 QUICK REFERENCE:

<b>ASSIGN</b>	Sets the value of something atomic
<b>BROWSE</b>	Exports an object to the browser
<b>CLEAR_VARS</b>	Sets all the fixed flags to FALSE
<b>COMPILE</b>	Compiles a simulation of a given type
<b>DELETE</b>	Deletes a simulation or the type library
<b>DISPLAY*</b>	Displays something
<b>INTEGRATE</b>	Runs an IVP integrator
<b>MERGE</b>	Performs an ARE_THE_SAME
<b>PLOT</b>	Creates a plot file
<b>PRINT</b>	Prints one of the printable windows
<b>PROBE</b>	Exports an object to the probe
<b>READ</b>	Reads in a model, script, or values file.
<b>REFINE</b>	Performs an IS_REFINED_TO
<b>RESTORE*</b>	Reads a simulation from disk.
<b>RESUME</b>	Resumes compiling a simulation
<b>RUN</b>	Runs a procedure
<b>SAVE*</b>	Writes a simulation to disk
<b>SHOW</b>	Calls a unix plot program on a file from PLOT
<b>SOLVE</b>	Runs the solver
<b>WRITE</b>	Writes values in Tcl format to disk

\* Items not yet implemented.

### 3.2.3 COMMANDS

**ASSIGN**                    `ASSIGN <qlfdid> <value> [units]`

Sets the value of atom ‘qlfdid’ from the script. If value is real, it is required to give a set of units compatible with the dimensions of the variable. If the variable has no dimensions yet, ASSIGN will fix the dimensions.

**BROWSE**                    `BROWSE <qlfdid>`

Exports qlfdid to the browser, displaying it as the current instance in the browser.

**CLEAR\_VARS**                `CLEAR_VARS <qlfdid>`

Sets the value of the fixed flag to FALSE for all the variables on qlfdid.

**COMPILE**                    `COMPILE <simname> [OF] <type>`

Build a simulation of the type given with name simname. You can get away with leaving out OF or spelling it wrong.

**DELETE**                    `DELETE <TYPES,simname>`

The modifier TYPES will cause all simulations to be deleted. If a simulation name (simname) is specified only that simulation will be deleted.

**DISPLAY**                    `DISPLAY <kind> [OF] <qlfdid>`

How qlfdid is displayed varies with kind. kinds are: VALUE  
ATTRIBUTES CODE ANCESTRY

**INTEGRATE**                `INTEGRATE {qlfdid args}`

Runs an integrator on qlfdid. There are several permutations on the syntax. It is best to have solved qlfdid before hand to have good initial values.

```
INTEGRATE qlfdid (assumes LSODE and entire range)
INTEGRATE qlfdid WITH (assumes entire range)
INTEGRATE qlfdid FROM n1 TO n2 (assumes lsode)
INTEGRATE qlfdid FROM n1 TO n2 WITH integrator
```



Requires:

- $n1 < n2$
- qlfdid be of an integrable type (a refinement of ivp.)

## **MERGE**

MERGE <qlfdid1> [WITH] <qlfdid2>

ARE\_THE\_SAME qlfdid1 and qlfdid2 if possible.

## **OBJECTIVE**

OBJECTIVE

Semantics of OBJECTIVE that will be supported are unclear as no OBJECTIVE other than the declarative one is yet supported. Not implemented yet

## **PLOT**

PLOT <qlfdid> [filename]

Writes plot data from qlfdid, which must be a plottable instance, to filename.

## **PRINT**

PRINT <PROBE,DISPLAY>

Prints out the text currently in the Probe or Display.

## **PROBE**

PROBE ONE qlfdid

Exports the item qlfdid to the Probe.

PROBE ALL qlfdid

PROBE qlfdid

Exports items found in qlfdid matching the current specifications of Visit in the Browser. By default, all variables and relations.

Items always go to currently selected probe context.

## **READ**

READ [FILE,<VALUES,SCRIPT>] <filename>

Loads a file from disk. Searches for files in directories (Working directory):::\$ASCENDLIBRARY unless a full path name is given for filename.

The modifier FILE is used to indicate that the file contains ASCEND source code (ASCEND source code files normally have a .asc extension).

The modifier **VALUES** is used to indicate that the file contains variable data written by **WRITE VALUES** (These files normally have a .values extension).

The modifier **SCRIPT** is used to indicate that the file is a script file to be loaded at the end of the Script window (Script files normally have a .s extension).

If neither **VALUES** nor **SCRIPT** are found, **FILE** will be assumed.  
Note: You will get quite a spew from the parser if you leave out the **SCRIPT** or **VALUES** modifier by accident.

**REFINE**                    `REFINE <qlfdid> [TO] <type>`

Refines qlfdid to given type if they are conformable.

**RESTORE**                `RESTORE <file>`

Reloads a simulation from disk

**RESUME**                `RESUME <simname>`

Reinvokes compiler on simname.

**RUN**                    `RUN <qlfpid>`

Run the procedure qlfpid as if from the browser Initialize button.

**SAVE**                   `SAVE <sim> [TO] <filename>`

Filename will be assumed to be in Working directory (on utils page) unless it starts with a / or a ~. Not implemented yet.

**SHOW**                  `SHOW <filename, LAST>`

Invokes the plotter program on the filename given or on the file LAST generated by PLOT.

**SOLVE**                `SOLVE <qlfdid> [WITH] [solvername]`

Exports qlfdid to the solver and attempts to solve it with the default solver (usually QRSly) or the solver indicated by the optional solvername argument. Solvername must be given as it appears on the menu buttons. Bugs: Should use current solver rather than default.

**WRITE**                   WRITE <kind> <qlfddid> <file> [args]

Write something (what sort of write indicated by kind) about qlfddid to a file. args may modify as determined by kind. At present only VALUES is supported. SYSTEM (for solver dump) would be nice.

WRITE VALUES filename.

Filename must be a full path name or in the pwd, also known as ‘.’.

### 3.3 SCRIPT WINDOW BINDINGS

In the event binding descriptions that follow, M1 is short for mouse-button-1 (the left mousebutton), M2 is the middle button, and M3 is the right mouse button. On machines with no middle button, M3 is still the right mouse-button and M2 is unavailable.

<b>M1</b>	repositions the cursor.
<b>M1-Drag</b>	selects text.
<b>Shift-M1[-Drag]</b>	extends the selection.
<b>Double-M1</b>	selects the nearest word.
<b>Double-M1-Drag</b>	selects the nearest word and those you drag over, whole words at a time.
<b>Triple-M1</b>	selects the nearest line.
<b>Triple-M1-Drag</b>	selects the nearest line and those you drag over, whole lines at a time.
<b>M2</b>	does nothing.
<b>M2-Held-Down</b>	has an effect similar to the scrollbar.
<b>M3</b>	does nothing.
<b>Control-M1</b>	Starts another part of a disjoint selection.
<u>UNIX bindings:</u>	The text widgets in ASCEND share a common stack of cut/copy/paste text pieces. This is a CMU extension of the text bindings, not default Tk behavior, and it is EMACS-like, but not EMACS (EMACS uses a

ring, not a stack.) When the stack is empty, Paste does nothing. This is a design decision. The Tcl function `ascPopText` can be changed to behave differently.

<b>Control-k</b>	Cuts text to the end of the current line, putting it on the stack.
<b>Control-w</b>	Cuts the selected text, putting it on the stack.
<b>Meta-w</b>	(e.g. diamond-w on most Sun keyboards) Copies the selected text onto the stack.
<b>Control-y</b>	Pastes the most recent text added to the stack, and removes it from the stack.
<b>Meta-y</b>	Not supported.
<u>MSW bindings:</u>	The standard Control-X, Control-C, Control-V bindings of Microsoft Windows clipboard apply to text widgets. The UNIX text stack is not available.



## CHAPTER 4 LIBRARY

The Library window (Figure 4-1) in ASCEND allows the user to read *types* into the ASCEND system from files, *compile* types into instances, and delete types.

Types are the templates used to create simulations. They come in two flavors: ATOM, which has a value associated with the instance name when it is instantiated, and MODEL, which has no value. ATOMs are the variables and constants in ASCEND; MODELs are the complex structures one can build in ASCEND. ATOMs, further, come in vanilla

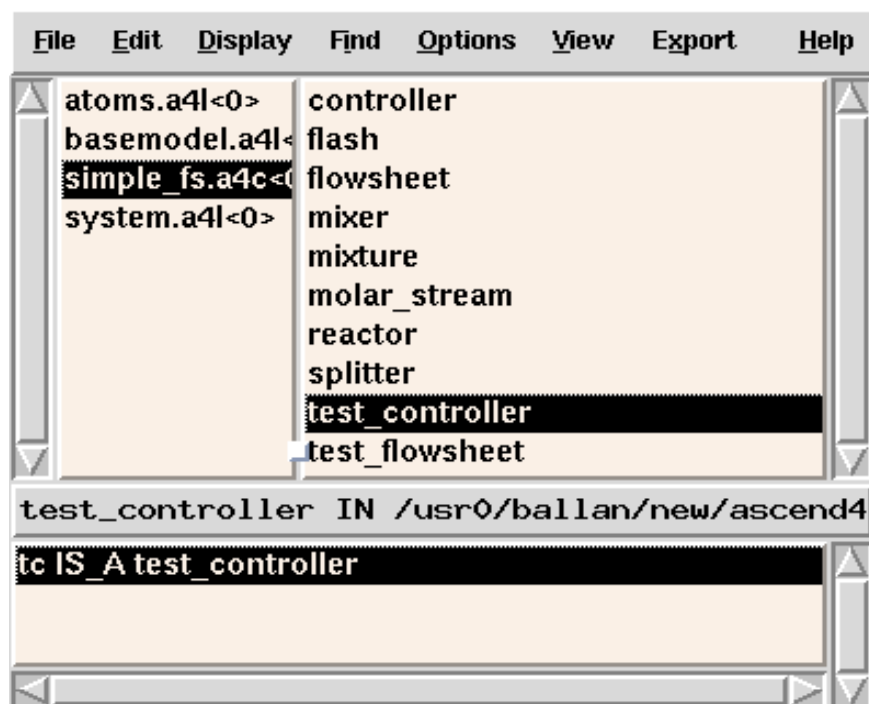


Figure 4-1 ASCEND Library Window.

and UNIVERSAL flavors. Universal atoms have a single compiled instance which is global to all simulations created.

Both ATOMS and MODELS are defined in source files. By convention, source files are named with the endings `.a4c` (ASCEND IV code) and `.a4l` (ASCEND IV library). You are free to use any other ending, but you will find the ASCEND file type filters you use when browsing for files will be ineffectual.

In the ASCEND Library window, source files appears in the upper left box. On the other hand, the types defined in the highlighted source file appear in the upper right box. A double-button2 in either box will compile the highlighted type definition. It doesn't reselect. The upper left box should perhaps have double-button2 bound to reread the selected source module. The ASCEND fundamental type such as integer, real, etc., are not shown in the library window, since their definition is performed internally, not by using a specific source file. The lower box of the ASCEND Library window contains the name of the simulations that have been compiled and can be run.

The data structure used to store type definitions is sketched in Figure 4-2.

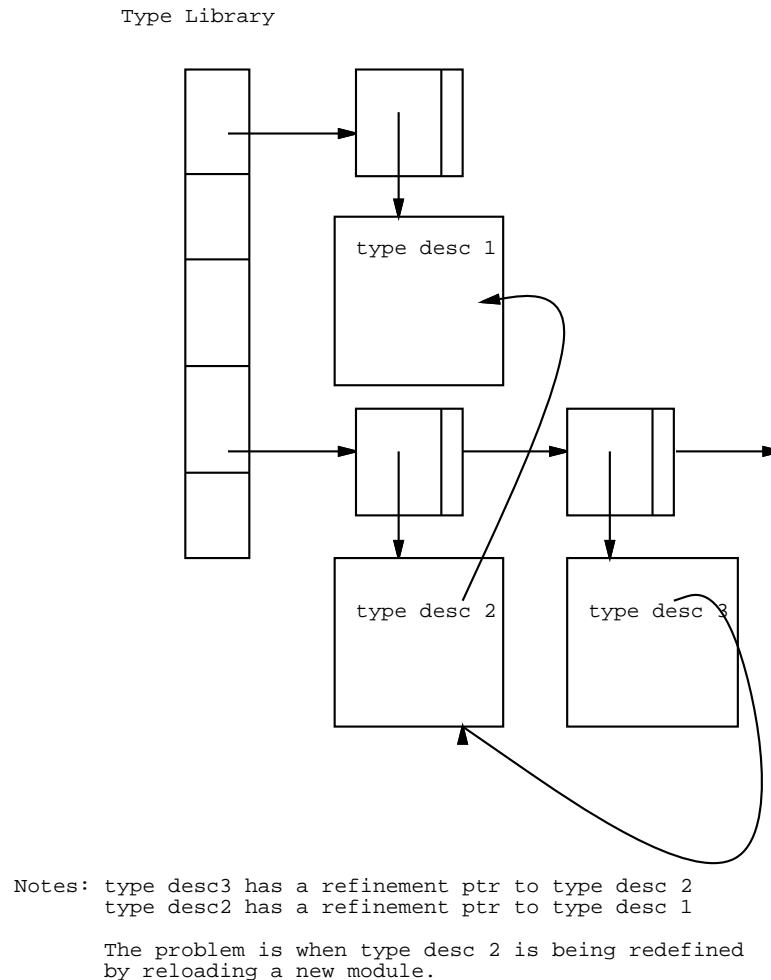


Figure 4-2 Data structure used to store type definitions.

## 4.1 MENU BAR

The menu bar on the Library window has eight entries: File, Edit, Display, Find, Options, View, Export and Help.

### 4.1.1 THE FILE MENU

#### **Read types from file**

This loads type definitions into the system. The file selection dialog is used to select a source file.

The names of types are unique within the system. A new definition of a type overwrites the old definition of a type in all cases. If the new definition and the old definition were read from files of the same name, this overwrite will be done silently. If the new definition comes from a



different file, the overwrite will be done noisily.

**This is incorrect, but perhaps is as it ought to be.** *Existing types which refined or had parts that were of the old type definition will now refine or have parts which are of the new type. e.g. If you reread system.asc (and hence solver\_var) everybody in the interface library who pointed at the old solver\_var type will now point at the new solver\_var type.*

Instances already compiled using the definitions that have been overwritten will continue to point at a copy of the old definition the system has squirreled away somewhere. These squirreled away copies will not necessarily be the same as what is in the interface type library if you have reread a file with a newer type definition. This may cause refinement of the old instance to fail. In general if you redefine a type, you will probably want to reinstantiate things that depend on that type.

#### **Close window**

It closes the ASCEND's Library window. To reopen, use the Tools menu in the Script window or use the SCRIPT tool in the Toolbar window.

#### **Exit ASCEND**

Exit the ASCEND system. You will be asked to confirm that you wish to do this.

### **4.1.2 THE EDIT MENU**

#### **Create simulation**

Create (or instantiate) a simulation based on a type definition. Anytime that the compile button is selected, the compile dialog window shown in Figure 4-3 will ask for the name which will be used to identify the simulation. All simulation created can be seen and in the lower box of the ASCEND Library window. This box can contain any number of simulations.

#### **Suggest methods**

Pick any type in the right window and apply this tool to write suggested methods for that type. See the Howto book, Chapter 2, for a list of the methods we suggest one should write for models. These suggested methods are prototypes which you can cut and paste into your favorite text editor. You should be carefully edit them before adding them to the model type definition.

#### **Delete Simulation**

This tool works to remove previously compiled simulations listed in the bottom subwindow of the Library window. Select a simulation to delete and use this tool to eliminate it from ASCEND. ASCEND will clear the model from the Browser and Solver. Items placed into the Probe window are not deleted.

**Delete all types**

Destroys all simulations and deletes all types. This option has no effect in the fundamental definitions. Deleting all types clears the Browser and Solver windows, but not the Probe window.

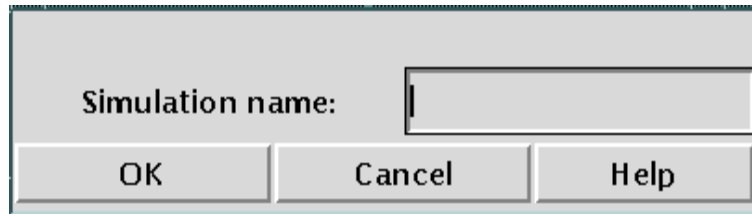


Figure 4-3 The Create Simulation Dialog

### 4.1.3 THE DISPLAY MENU

Most of the options in the Display Menu will be enabled only if a type definition has been selected; this is because the tasks performed in the menu are implicitly associated with a type definition.

**Code**

Displays the source code of the selected type in the ASCEND Display Window.

**Ancestry**

Allows the use of the Type Refinement Hierarchy Window. See Section 4.2 on page 46 documenting this window.

**Refinement hierarchy**

Displays the refinement hierarchy of the selected type in the ASCEND Display window.

**External functions**

Display in the Display window any external function defined from a loaded package library.

**Hide type**

The Browser will not display any type definition which you select to be hidden with this tool. You may select to hide the type or the type and all its refinements. For example, doing the latter with `solver_var` will hide all variables in a compiled instance of the model.

**UnHide type**

Reverses the action of “hiding” a type. Select the type in the right window and select unhide if that tool is lit. The Browser will immediately begin to display this previously hidden type. The default for all the type definitions (except fundamentals) is to be “unhidden”.

Both Hide Type and UnHide Type have two selections as a submenu. The user can ask for the un/hiding of only the **selected type**, or for the un/hiding of the selected **type and its refinements**.

### Hide/Show Fundamentals

This special option is given because fundamental types do not appear as definitions in the ascend libraries, but we still may want to able/enable such types for browsing purposes. When this button is selected, the window shown in Figure 4-4 will be used to perform the desired hiding or unhiding of any of the fundamental types.

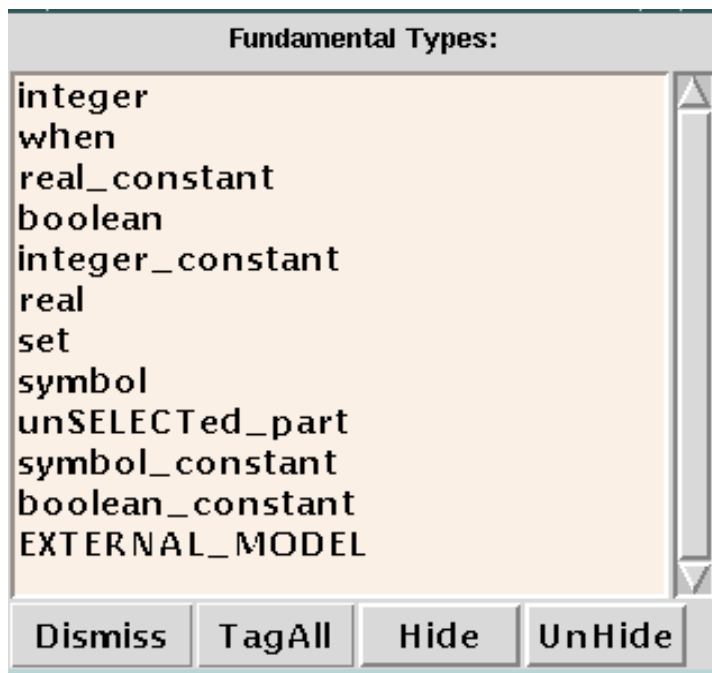


Figure 4-4 Select the fundamental type to Hide or Unhide.

## 4.1.4 THE FIND MENU

### ATOM by units

This extremely handy tool allows you to find all the ATOMS that are currently loaded into the library whose dimensionality conforms to a user specified set of units. For example, if you have loaded the library atoms.a4l into the Library, then you can use this tool to find all atom definitions that could be expressed in ft<sup>3</sup> or in kJ/mol. To use, select the tool. In the window that opens type in the units and select OK. You do have to know the units ASCEND will recognize. Open the Units window and under the Display menu select Show all units for a complete list.

### Type by name

Finds a type by its name. The type will become the current type highlighted in the Library right and left upper windows.

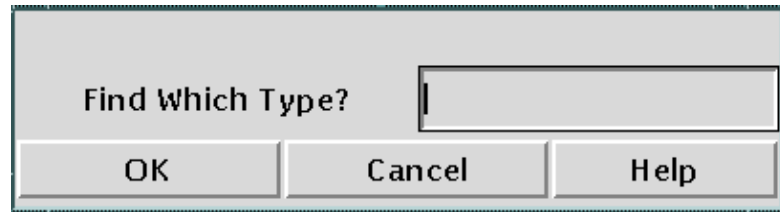


Figure 4-5 The Library's Find Type dialog.

#### **Type by fuzzy name**

Finds all type names currently loaded in the Library window that match a word (provided by the user) in any fuzzy way. For instance, the name *column* would list the following: *demo\_column*, *mw\_demo\_column*, *plot\_column*, etc. if these were currently loaded in the Library. The fuzzy name is defined in a dialog window similar to that used in the Find Type by name option.

#### **Pending statements**

There are three selections under the Pending Statements submenu, these are **To Display**, **To Console**, and **To File**. Pending statements in a simulation are relations that have not yet been fully processed by ASCEND's compiler. It is the modeler's job to correct the pending relations in order to arrive at a fully functional simulation. Corrections may be made by either creating a model which refines the current model or by editing ASCEND code and starting over. This option gives the user access to information about the type and location of the pending statements. Often pending statements arise from a common cause such as an incorrectly qualified or misspelled name for a set.

#### **To Display**

By selecting the **To Display** option, all of the simulation pendings are displayed in the *Display* window.

#### **To Console**

By selecting the **To Console** option, all of the simulation pendings are displayed in the Console window (in UNIX, the Console is the window from which you started ASCEND IV).

#### **To File**

By selecting the **To File** option, the *File select box* is opened and the user is asked to enter the name of the file in which to save the model pendings.

### **4.1.5 THE OPTIONS MENU**

The titles for most of these tools more or less describes their purpose. We will not describe them. With these options, you can turn on or off messages ASCEND will generate while compiling. Turning off warning and error messages will, of course, mean that you will not be told about problems your model may have that we were able to detect.

Since ASCEND will still compile in spite of warning messages (which generally reflect your model does not conform to what we believe to be good modeling practice), you may wish to suppress them. See the [Howto Book, Chapter 2](#) for a discussion of good modeling practice.

We describe only those tools that do not turn on and off compiler messages.

#### **Generate C binary**

If you have a C compiler installed and ASCEND knows about it, then you may elect to have ASCEND compile C code to evaluate equation residuals in ASCEND. The compiler finds and will, when this option is selected and possible, compile only a piece of code for each unique equation type, of which there are very few in any model. The evaluation of residuals will be much faster using compiled C code.

#### **Simplify compiled equations**

This option is on by default. ASCEND will reduce terms in equations such as a product of constants whose values it knows to a single resultant constant when you select this option. Whole terms in equations may disappear if ASCEND finds them multiplied by the constant zero.

#### **Save options**

Save the current setting for all the options selected using items in this menu. ASCEND put the saved information into a text file *library\_opt.a4o* and which it then saves in the *ascdata* subdirectory of your “home” directory.

### **4.1.6 THE VIEW MENU**

#### **Font**

Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.

#### **Open automatically**

Toggles a switch which, if set, will cause this window to open whenever anything is placed into it.

#### **Save appearance**

Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a *.a4o* text file for this window which the system stores in the subdirectory *ascdata* in your “home” directory.

### **4.1.7 THE EXPORT MENU**

There are three selections under this submenu, these are **Simulation to Browser**, **Simulation to Solver**, and **Simulation to Probe**.

<b>Simulation to Browser</b>	By selecting the <b>Simulation to Browser</b> option, the simulation highlighted in the lower box of the Library window is loaded into the <i>Browser</i> . From the <i>Browser</i> , the model can be explored in more detail.
<b>Simulation to Solver</b>	By selecting the <b>Simulation to Solver</b> option, the simulation highlighted in the lower box of the Library window is loaded into the <i>Solver</i> . (Note that exporting to the solver causes a degrees of freedom analysis to be carried out.)
<b>Simulation to Probe</b>	By selecting the <b>Simulation to Probe</b> option, all of the variables of the simulation highlighted in the lower box of the Library window are loaded into the <i>Probe</i> . This is not recommended as there are usually more variables in a model than the user would wish to view at one time. However, if the user does wish to look at all of the variables and their current values, the <b>Simulation to Probe</b> option can be useful.

#### 4.1.8 THE HELP MENU

<b>On LIBRARY</b>	Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.
-------------------	--

## 4.2 TYPE REFINEMENT HIERARCHY WINDOW

The type tree is a directed acyclic graph (DAG) based on the type hierarchy currently defined in the interface Library. Selection of the tool Display Ancestry (mentioned above when describing tools under the Display menu) for any selected type gives the entire refinement hierarchy for that type, by enabling the use of the window shown in Figure 4-6.

The current focus in the hierarchy is indicated by a rectangle around the type name and the Current type.

The buttons on the left in the type window operate on the currently selected type:

‘Atoms’ shows the types of ATOMIC parts in the selected type definition. It also shows the incremental code for the type. You can select from the part types list to look at a different hierarchy.

‘Code’ shows the internally stored code of the selected type. The expressions, both algebraic and logical, are in reverse Polish notation.

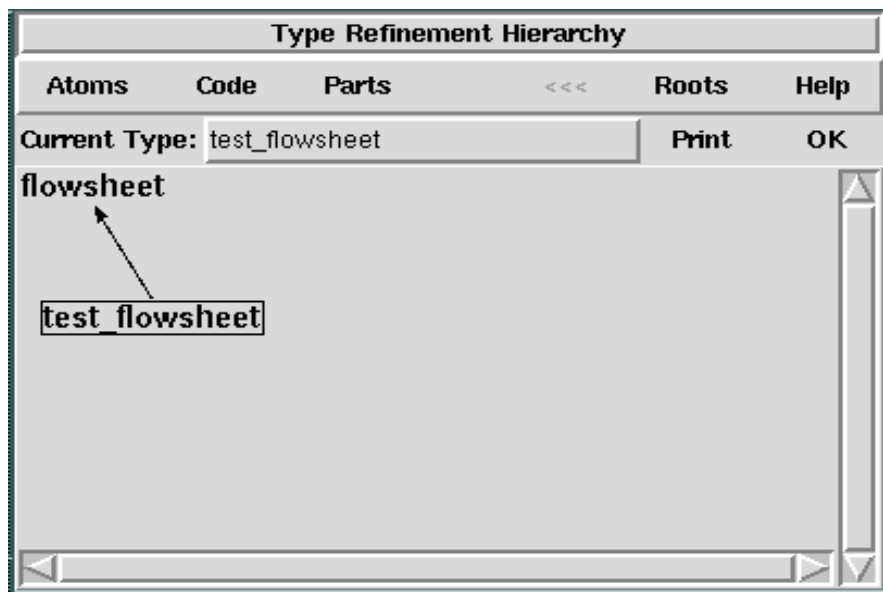


Figure 4-6 The Type Refinement Window.

This is different from the way the code of the Library Display Code button shows it. Comparison of the two is sometimes a useful debugging tool.

'Parts' (Figure 4-7) shows the types of MODEL parts in the selected

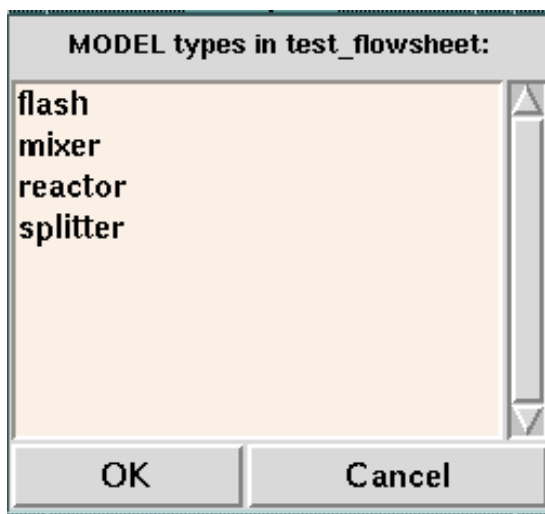


Figure 4-7 The Parts window displays the parts.

type definition. It also shows the incremental code for the type.

The '<<<' (or backtrack) button backs up to the previously displayed type hierarchy, if there is one.

‘Roots’ (Figure 4-8) shows the existing root types, that is, the existing types which are not refinements of anything.

While ASCEND is building the graph, you may see a spew in the window from which ASCEND was started about orphaned types. This means there are types in the Library which are refinements of older types which are no longer in the Library.

While ASCEND is getting the Atom or Model parts list for a type, part types names which are undefined will be spewed.

When an older type is replaced in the Library by a new one of the same name, the old one is squirreled away where types that refined it can still see it. The only way to get current types to look at the new definition without touching the source files for the current types is to delete all types and reread the entire Library.



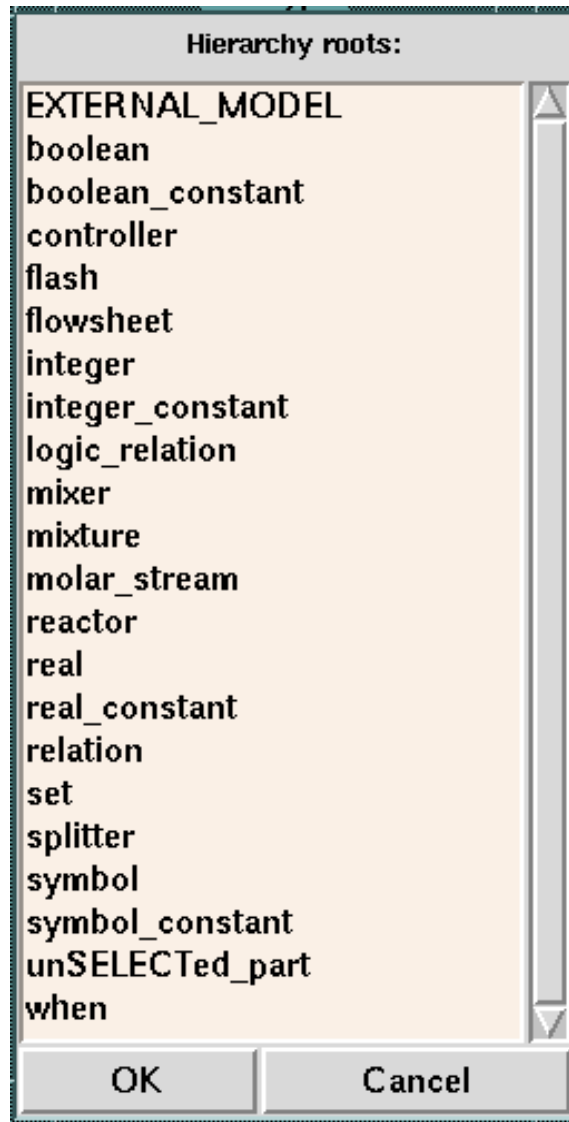


Figure 4-8 The Hierarchy Roots Window.

## CHAPTER 5 BROWSER

The Browser window (Figure 5-1) provides the means with which to view the parts of a simulation. When a simulation is exported to the Browser, the name of the simulation appears in the Browser's upper left box and the child instances of the simulation appear in the upper right box. Selecting a child instance in the right box will move the instance to the bottom of the stack in the left box and display its children in the right box. The instance tree can be traversed in this manner until an atom (usually a variable) resides at the bottom of the stack in the left box and its attributes appear in the right box. Selecting a member of the stack in the left box will clear any lower instances on the stack and display the selected instance's children in the right box.

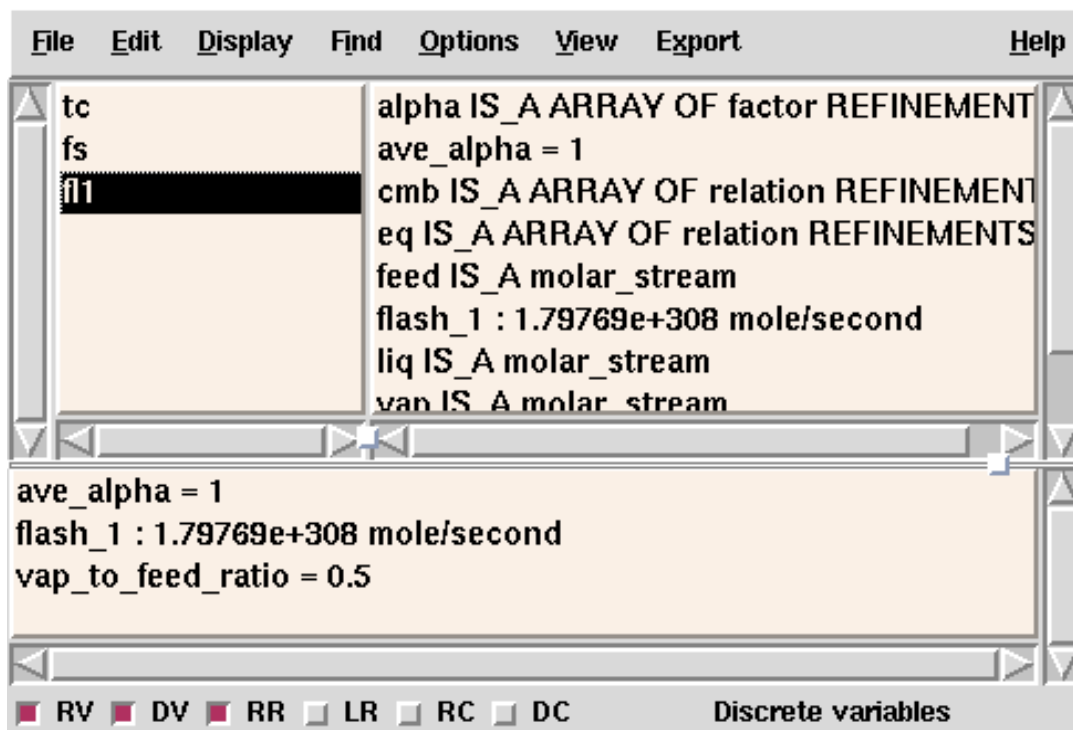


Figure 5-1 ASCEND's Browser window.

A subset of the instances appearing in the upper right box as well as the values of these instances appear in the Browser's lower box. Which subset of instances appears in the lower box is controlled by the user by clicking in some of the options given in the bar at the bottom of the Browser window. In Figure 5-1 RV has been selected. RV stands for **Real Variables**. Therefore, the child instances of fl1 which are real variables and the values of these real instances are shown in the lower box. Other options in the bar at the bottom of the Browser window, which can be simultaneously selected, are DV (Discrete Variable), RR (Real Relations), LR (Logical Relations), RC (Real Constants) and DC (Discrete Constants). Selecting an instance in the lower box with the right button of the mouse will have the same effect as selecting the same instance in the upper right box. On the other hand, selecting an instance in the lower box with the left button of the mouse will bring up the "Set Value" Dialog box, which will give the user the option of modifying the value of the selected instance. More about the "Set Value" option will be given in the following section of this document.

## 5.1 THE MENU BAR

- The menu bar on the Browser window has eight entries: File, Edit, Display, Find, Options, View, Export, and Help.

### 5.1.1 BROWSER FILE MENU

<b>Read values</b>	Reads the values from a file previously saved by Write Values. Values files are read using full path names (including the simulation name). The simulation for which values are being read does not necessarily have to be in the Browser (but it should exist). You may specify that the values are to be read into a different simulation or simulation part than they were originally saved from, provided the old and new locations are compatible. If the original simulation does not exist, you will be asked for a new simulation name.
<b>Write values</b>	Saves the values for the instance in the Browser to a file for later rereading.
<b>Close window</b>	Closes the Browser window. To reopen, go to the Script window and select <i>Instance browser</i> under the tools menu or select the BROWSER on the Toolbox window.
<b>Exit ASCEND</b>	Exit the ASCEND system entirely. You will be asked if you really wish to complete this instruction.

### 5.1.2 BROWSER EDIT MENU

<b>Run method</b>	If the instance in the left box has one or more methods available, Edit ->Run Method will be available for selection. Selecting Run Method
-------------------	--

will display the Methods Selection Window containing a list of available methods for the current Browser instance. A method is selected by clicking its name (only one method can be selected at a time). Depressing the OK button will run the selected method. Depressing the Show button will display the code for the selected method. Depressing the Cancel button will close the Method Selection Window without running any method.

**Clear Vars**

In ASCEND, the type solver\_var and all its refinements constitute a variable for solution purposes. Each variable has a boolean, named “fixed”, as one of its children. When a variable’s fixed boolean, or fixed flag as it is commonly called, is set to False, the variable is considered an output variable (i.e. the solver will determine its value). The Clear Vars method sets the fixed flag of every variable which is a child of the current Browser instance to False.

**Set value**

When the current Browser instance is a real, symbol, integer, or boolean Edit->Set Value will be available for selection. Selecting Set Value displays the Set Value Dialog box. The value (and units in the case of reals) may be set by filing in the value (and units) fields of the Set Value Dialog box and depressing the OK button. Depressing the Cancel button closes the Set Value Dialog box. Booleans are assigned simply by double clicking the mouse button 2 on their name when it appears in the right browser box. Write values

Selecting Edit->Write Values saves the attribute values of all atoms which are descendents of the current instance to a file. A file select box is displayed so a new file may be created or an old file over written. The attribute values are written to the selected file along with their path names relative to the current instance. The first line of the file specifies the path from the simulation to the current instance.

**Refine**

Refines the current Browser instance to a given type. Edit->Refine may only be selected if the Library contains types which are refinements of the current Browser instance type. Selecting Edit->Refine displays the eligible types for the refinement of the current part in the Refinement dialog box. Selecting a type and depressing OK refines the current type to the selected type. Depressing Show displays the ASCEND code for the selected type. Depressing the Cancel button closes the Refinement dialog box without making any refinements.

**Merge**

ARE\_THE\_SAME the current part (left side of the Browser) with another given part. Do not ARE\_THE\_SAME parts from 2 different simulations. You cannot merge parts of atoms (which are atomic) with anything. The dialog box will ask for the name of the instance that you

want to merge with the instance highlighted in the left box of the browser.

<b>Compile</b>	Submenu containing Resume Compilation and Create Part.
<b>Resume Compilation</b>	Attempts to process any pending statements in the simulation in the Browser. It does not matter where in the simulation you have browsed to, Resume always starts at the top.
<b>Create Part</b>	This is a feature of the PASCAL version only. The proper way to add a part to a simulation is to create a refinement of the original model in a new file, read in that definition, and refine the simulation up to that new model.

### 5.1.3 BROWSER DISPLAY MENU

<b>Attributes</b>	Display the attributes of a real variable. Other functionality may be added later to this button.
<b>Relations</b>	Display all the relations at or below the current point in the Browser. Relations get arbitrary names unless explicitly named by the user in code. The arbitrary name, at the moment consists of ParentName_n where n is the number of the nth relation in the MODEL ParentName. If this name is not unique, enough letters a-z get added to make it unique. When the instance highlighted in the left box of the Browser is a real variable, this option will display all of the relation in which such a variable is incident.
<b>Conditional Relations</b>	Display all the conditional relations in or in the children or grandchildren etc., of the current object in the Browser. Conditional Relations do not have to be satisfied. They are used as boundaries in conditional programming. The arbitrary name of a conditional relation is obtained in the same way as any other relation, but in general, the name of a conditional relation must be provided by the user, since the operator SATISFIED requires such a name.
<b>Logical Relations</b>	Display all the logical relations in or in the children or grandchildren etc., of the current object in the Browser. Logical Relations get arbitrary names unless explicitly named by the user in code. The arbitrary name of a Logical Relation follows the same pattern as that of real relations. When the instance highlighted in the left box of the Browser is a boolean variable, this option will display all of the logical relation in which such a boolean variable is incident.

**Conditional  
Logical  
Relations**

Display all the logical relations in or in the children or grandchildren etc., of the current object in the Browser. Conditional Logical Relations do not have to be satisfied. They are used as conditions to check in conditional programming. The arbitrary name of a conditional logical relation is obtained in the same way as any other logical relation, but in general, the name of a conditional logical relation must be provided by the user, since the operator SATISFIED requires such a name.

**Whens**

This option is enabled for instances of models, relations, booleans, symbols, and integers. For the case of a model instance, this button will display not only all the when instances defined as parts of such a model, but also the when instances which include such a model in one of their CASEs. Distinction is made between those two possibilities. For relation, boolean, symbol and integer instances, this option displays the when instances which include such relation, symbol, etc., either in one of their CASEs or in the list of conditional variables. When instances are useful for the conditional configuration of a problem and always get arbitrary names.

**Plot**

Invokes a plotting program, if allowable, on the current object. The type of plot generated is controlled by the Utilities page variables Plot.type and Plot.program. See the relevant chapters in the Howto manual on plotting to find the types which ASCEND will plot. Also see the ASCEND library of models supporting plotting: plot.a4l (and any of the other model files containing the name plot which have examples of plotting within them).

**Statistics**

Prints out some information about the object tree in the Browser starting with the current object and going downward through its children, grandchildren, etc.

### 5.1.4 BROWSER FIND MENU

**By name**

Search for an instance with a given qualified name and go there. The name of the instance to search for is defined in the dialog. This option may be useful for jumping around in the instance tree. Since names can be quite long, you may find this tool most useful when you have found the name elsewhere and can cut and then paste it into the dialogue box that opens for this tool.

**By type**

You can search for objects of any particular type with certain attributes. The default type will list all fixed solver\_vars for the problem. The allowable searches are best explained by examples as shown in Table 5-1 (with the third being the default just mentioned). The search

**Table 5-1** Examples of the performance of the Find by type option

Type	Attribute	Low Value	High Value	Explanation
unit				Find all parts that are units.
solver_var	fixed			Find all refinements of solver_var with a part fixed
solver_var	fixed	TRUE		Find all refinements of solver_var with a part fixed where fixed==TRUE
stream	Ftot	4		Find all streams with part Ftot where value is $4 \pm \text{epsilon}$
stream	Ftot	4	10	Find all streams with part Ftot where $4 \leq \text{Ftot} \leq 10$
relation	VALUE	0		Find all relations with a residual $0 \pm \text{epsilon}$
symbol	VALUE	ACH		Find all symbols where VALUE is 'ACH'
symbol	VALUE	ACH	ACZ	find all symbols where 'ACH' $\leq$ VALUE $\leq$ 'ACZ'
component_constant				Find all parts that are component_constants
symbol_constant	VALUE	UNDEFINED		Find all undefined symbol_constants. Works for all types with a value.

is loosely matched, i.e. any object that is of the type given, OR a refinement of the type given and matches the attribute qualifications, will be on the list of items found.

If there are no matches, there is no results box: just a message in a popup error message.

The results of the Find appear in a box and you can export one or more of the results in the box to the Browser or the Probe by selecting them and clicking on Browser or Probe. When you have finished exporting items to wherever you like, click on OK. Do close this box as the rest of the interface will ignore you until you do.

Notes:

- Clear any of the extra fields not required for your search before you hit OK. We will usually find nothing that matches if there are extra search parameters hanging around that don't make sense.

- VALUE is a special keyword for dealing with atomic types. Variables and symbols have a value internally but not a child named VALUE. Similarly, relations have a residual but not as an accessible part with that name.
- Symbols and integers will be matched exactly if only a low value is given. The matching of symbols given a low and high value is done lexically according to the collating sequence of the machine in use.
- Frequently what you really want to see is the name of a set of things of a given type -- e.g. a case where you want to know what components are in a flowsheet. Find will return the instances though, not their common parent. Simply export one to the Browser and then click up a level to see the set of components in use.
- You can tab between fields in the Find by Type widget.
- You can select a type name in the library. Pick the type in the right window. Its name will appear in the lower middle window. Highlighting the name and use the typical method to copy a set of highlighted characters for your computer (e.g., Ctrl-c on a PC). Then use the typical way to paste into the type slot of the Find by Type Window (Ctrl-v on a PC).
- Epsilon is about  $1e-8$  in terms of the SI units for any real quantity.

## Aliases

Find all the other names that the current object has in the simulation. For example, assume that you have named a simulation as fs. Assume further that the output stream from the mixer, m1, is merged with the input stream for the reactor, r1. Then, that stream is an object with two different names. Suppose you are looking at r1.feed as the current object. Asking for aliases will give the list

fs.r1.feed

fs.m1.output

If you pick one of the aliases, it can be exported to the BROWSER, the SOLVER or the PROBE. Alternate names for objects can also be created by ALIASES statements and by passing them into a parameterized MODEL.

## Where created

Find the other names that the current object was CONSTRUCTED under. If an object is shown as being created under 4 names, it means that once there were 4 objects and that 3 were destroyed in merge



(ARE\_THE\_SAME) statements to reach the current unity. (Merging is expensive).

If you pick one of the names, it can be exported to the BROWSER, the SOLVER or the PROBE. Alternate names for objects can also be created by ALIASES statements or by passing them into a parameterized MODEL, but these names do not appear in the list of creations.

### **Clique**

Find all the instances that ARE\_ALIKE with the current one. The instances shown are bound together so that if the formal type of one is changed, they are all upgraded with the first. Parameterized objects cannot be ARE\_ALIKE'd because when parameters exist the formal type requires outside information (the parameters) in order to check that it is being used in a valid way.

### **Eligible variables**

Find real variables eligible to be fixed. If the solver is occupied by the same simulation, this query is thrown to the Solver. If not, the degrees of freedom are analyzed as if the current model were exported to the Solver.

### **Active Relations**

Find all the relations of the current object. You may tag one, several or all of the relations found and export them to the Probe. You may also export the first tagged relation to the Browser.

### **Operands**

If the current object is a relation, list all the operands in it. One or more of these may then be exported to the Probe. You may also export the first tagged operand to the Browser.

### **Pendings**

List in the Console window all the statements that the compiler failed to compile for the current Browser object.

## **5.1.5 BROWSER OPTIONS MENU**

### **Hide Passed Parts**

Toggles the display of parts which were passed into the Browser object as passed parameters. Note that these shared objects were created by a parent (or grandparent, etc.) of the current Browser object and will appear on the list of parts for that parent.

### **Suppress Atoms**

This button toggles whether or not to show atomic instances in the upper right box of the Browser window.

### **Display Atom Values**

This button toggles whether to display values or to display the types of atoms in the child box (upper right side) of the Browser. For the case of relations, the residual shown with the relation is the last computed by the solver and not the residual at the current values of the variables.

<b>Check Dimensionality</b>	This switch turns warnings about relation inconsistency off and on. In principle it should not be necessary, but for the quick and dirty model it is sometimes handy.
<b>Save Options</b>	Save the current options in this menu. When you restart ASCEND, the system will reset the options to these saved settings.
<b>Hide Names</b>	This option has a similar functionality from that of Hide Types in the ASCEND Library windows. That is, it will hide or unhide instances for browsing purposes. The difference, however, is that this option hides by name, not by type. To clarify, it is quite different to hide instances with the name <code>fs</code> than to hide all instances of type <code>test_flowsheet</code> .
<b>UnHide Names</b>	Reverses the effect of the command Hide Names. A list of hidden names appear from which you can select what to unhide..

### 5.1.6 BROWSER VIEW MENU

<b>Font</b>	Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.
<b>Open automatically</b>	Toggles a switch which, if set, will cause the Browser window to open whenever anything is placed into it by an export command.
<b>Save window appearance</b>	Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a <i>.a4o</i> text file for this window which the system stores in the subdirectory <i>ascdata</i> in your “home” directory.

### 5.1.7 BROWSER EXPORT MENU

<b>to Solver</b>	Checks the model for exportability (must be of type MODEL without any pending compilation) and, if acceptable, sends it to the Solver.
<b>Many to Probe</b>	Sends the child instances of the current part being browsed to the Probe. The types of instances sent to the Probe are selected in the filtering window shown in Figure 5-2. Every switch toggles whether or not to export each of types to the Probe.

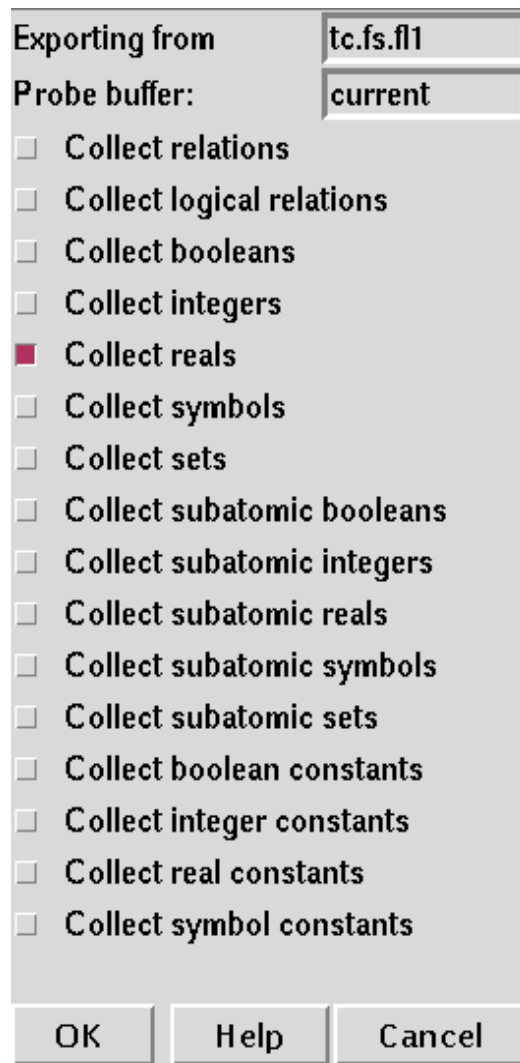


Figure 5-2 Filtering instances sent to the Probe

**Item to Probe** Exports the instance on the left box of the Browser to the Probe.

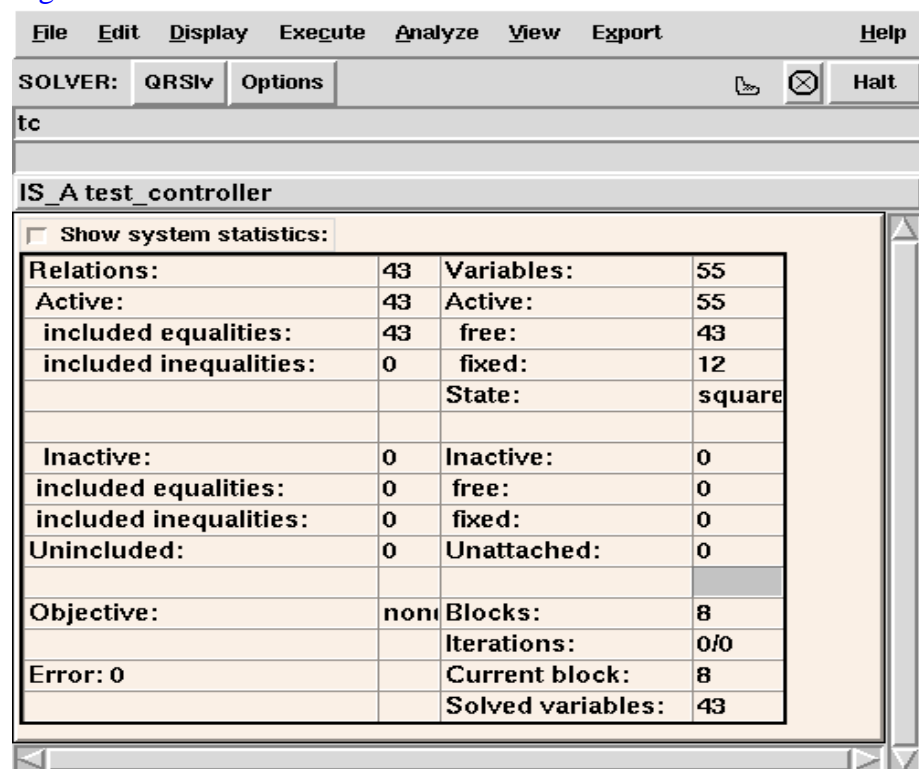
### 5.1.8 BROWSER HELP MENU

**On BROWSER** Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.

## CHAPTER 6 SOLVER

The purpose of the [Solver Utility shown in Figure 6-1](#) is to provide support for the numerical solving and debugging of an imported instance. To this end the Solver allows the user to access numerical solvers and analysis functions and displays statistical and status information for the problem being solved. The upper section of the solver window contains a menu bar; buttons for selecting numerical solvers, solver options, and halting the solver; a label containing the name of the current instance (or problem being solved); and a label containing the type of the current instance. The remainder of the Solver window is devoted to providing statistics about the problems relations and variables along with a description of the problem's state.

Figure 6-1 Solver Window



## 6.1 THE SOLVER MENU BAR

The menu bar on the Solver window has eight entries: File, Edit, Display, Execute, Analyze, View, Export, and Help.


### 6.1.1 SOLVER FILE MENU

<b>Close Window</b>	Close the Solver window. To open it again, go to the Script window and select the Solvers tool under the Tools menu or select Solver on the Toolbox.
<b>Exit ASCEND</b>	Exit the ASCEND system completely. You will be asked to verify that you really do want to exit.

### 6.1.2 SOLVER EDIT MENU

<b>Remove instance</b>	Removes the current instance from the solver.
<b>Select objective</b>	Provides a list of objectives from which one may select. The selected objective will be used in any subsequent optimizations until another objective is selected.

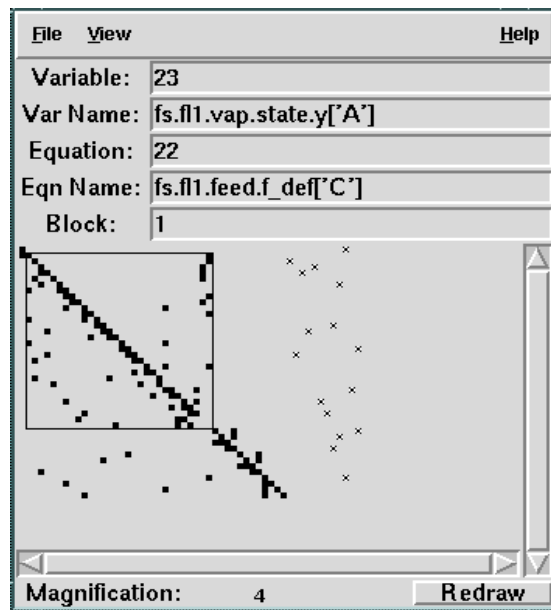
### 6.1.3 SOLVER DISPLAY MENU

<b>Status</b>	Shows the internal status of the Solver along with the largest block scaled residual vector two-norm.	
<b>Unattached variables</b>	Shows variables not incident in any of the relations in the current system being solved. (These variables are not a part of the current problem. Changing their attributes will have no effect on the problem.)	
<b>Unincluded relations</b>	Shows relations not in the current system being solved.	
<b>Incidence matrix</b>	Incidence matrix shows the incidence of variables in relations (See Figure 6-2). Clicking mouse-1 (left button) on the matrix displays the names and numbers of the relation/variable at that coordinate, whether that coordinate is occupied or not. A box is drawn around the partitioned block containing the selected coordinate and the block number is displayed. The selected block or the entire incidence matrix	

may be printed by selecting the PrintBlock or the Print button respectively. The scale of the incidence matrix can be changed by sliding the magnification bar and depressing the Redraw button. Depressing the OK button will close the INCIDENCE window.

Drawing large dense matrices may take a while. Drawing matrices on problems bigger than about 1000x1000 may be prohibitively expensive on slow machines. The row/column ordering is that done by the selected solver, except that fixed vars and unincluded relations are moved to the edges.

Figure 6-2 The Incidence Matrix



## 6.1.4 SOLVER EXECUTE MENU

### **Solve**

Solve the current problem as an algebraic or optimization problem depending on what solver is selected

### **Single step**

Perform a single iteration of the system with the solver in question. In some solvers (e.g. MINOS) there is no iteration mode. For these selecting single step will result in a full solve attempt.

For QRSly, an iteration will be a Newton like step if there are many variables in the current block or if the current block is a blackbox singleton. Singletons not from blackboxes are numerically inverted when being solved.

**Integrate** Invoke the selected integrator (LSODE currently available) on the problem.

### 6.1.5 SOLVER ANALYZE MENU

**Reanalyze** Reanalyzes the current problem to determine if a proper set of variables has been fixed to make the problem well-posed (i.e., square).

**Debugger** Opens a tool which deals with the system as a numbered list of variables and relations. See Section 6.4, "Debugger," on page 69 for more information about the Debugger.

**Overspecified** Finds and displays the variables that can be freed to reduce the degrees of freedom in an overspecified system.

**Find dependent eqns.** Finds structural or numeric dependencies of a system.

**Numeric Dependency.** Doesn't mean much on an unsolved system. This command inverts one block at a time and checks the blocks for numeric dependency using the QRSlv solver. Any non-zero dependency is reported, but those relations with coefficients down around machine epsilon (1e-16) are probably not dependent. Poorly scaled problems can appear more singular than they really are.

**Structural Dependency.** Find the equations or variables involved in a structural dependency. For systems that should be square, this is similar to overspecified, but for DAE's this detects the equations which need to be differentiated to overcome an index problem in the model. The user interface for reporting the data returned is not complete.

**Find unassigned eqns.** Shows the equations which cannot be assigned by the structural analysis.

**Evaluate unincluded eqns.** Evaluates the residuals of unincluded relations and checks them for convergence. This may or may not be a wise idea, depending on why the relations have been excluded.

**Find vars near bounds** This will write variable names passing the test

$$\text{abs}(\text{value-bound})/\text{nominal} < \text{epsilon} \quad (6.1)$$

to the console. The test is performed first for lower bounds and then for upper bounds and the results are clearly marked. This tool can be used for locating variables which may yield a more tractable problem when moved to the bound and fixed while freeing other variables. The value

of Epsilon can be set on the Solvers General parameter page accessible through the Options button in the Solver window.

**Find vars far  
from nominal**

This tool will write variable names passing test

$$\text{abs}(\text{value-nominal})/\text{nominal} > \text{bignum} \quad (6.2)$$

to the console. This test can be used for locating variables which are poorly scaled and for evaluating where model initialization methods need improvement. The value of bignum can be set using the Solvers General parameter page accessible through the Options button in the Solver window.

### 6.1.6 SOLVER VIEW MENU

**Font**

Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.

**Open  
automatically**

Toggles a switch which, if set, will cause this window to open whenever anything is placed into it.

**Save Solver  
appearance**

Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a *.a4o* text file for this window which the system stores in the subdirectory *ascdata* in your “home” directory.

### 6.1.7 SOLVER EXPORT MENU

**to Browser**

This button sends the instance currently in the Solver to the Browser.

**to Probe**

This button sends the instance currently in the Solver to the Probe piecewise; that is, all the variables and relations get shipped, not the instance name itself.

## 6.2 SOLVER BUTTON BAR

The solver button bar, which is located just below the solver menu, contains three buttons, the solver select button, the solver options button, and the halt button.

**Solver Select  
Button**

This button contains the name of the currently selected solver. Depressing this button reveals a menu of available solvers which can be selected.



### Solver Options Button

The Options menu on the solver allows the user to view and to change the settings for the parameters associated with ASCEND's solvers. A solver's parameters may be changed even when the solver is empty or another solver is selected. Depressing the options button reveals a list of parameter pages which can be selected for viewing (and editing).

Below, we discuss using the parameter pages and the general solver parameters; solver specific parameters are discussed below in Section 6.3, "Available Solvers," on page 67.

### Halt Button

Halts the solver and returns control to the interface as soon as possible. Not all solvers connected to ASCEND will respond to the halt signal.

## 6.2.1 GENERAL PARAMETERS PAGE

Selecting General under Options will display the General Parameter Page (See Figure 6-3). This is where we keep items relevant to the interface and to the way mathematical specialty functions and utilities are handled in ASCEND. Following, we will discuss the parameters that appear on this page.

Figure 6-3 General Parameter Page

iterations before screen update	10
cpu sec before screen update	3
modified log epsilon	1e-08
bound check epsilon	1e-3
far from nom bignum	10e3
integrator state log	y.dat
integrator observation log	obs.dat
integrator log SI units	display
integrator log columns	variable
<input checked="" type="checkbox"/> overwrite integrator logs	
<input type="checkbox"/> check numeric rank after solving	
<input type="checkbox"/> show block summary	
<div>OK</div> <div>Help</div>	

### Iterations before screen update

Because the interface update is sometimes rather time consuming (or more accurately when the window manager is slow, the interface holds up the solver) this specifies how many iterations to stay down in the solver algorithm before returning to the user interface to update statistics. In the case of floating point errors or solution completion

before the limit is reached, the return and update will happen immediately rather than waiting for the limit to be reached. For solvers that don't truly iterate in an accessible fashion (e.g. MINOS) this parameter is ignored.

CPU seconds before screen update

For solvers which do offer access to status information between iterations, this is the maximum number of cpu seconds before an interface update. If, while still not done with the number of iterations given in "iterations before screen update," the solver algorithm detects that the cpu seconds limit has expired, then it will return early to update the interface. At least one iteration will be completed before the clock is checked.

Modified log epsilon

This parameter controls the value for epsilon in the "lnm" function. Lnm can be used instead of natural log (ln) when the argument is likely to be very small or to go negative in the solution process. This avoids a host of floating point errors in initialization and solving of many kinds of models.

The modified natural log function  $f$  is defined as

$$f(x) = \begin{cases} \ln(x) \forall (x > \epsilon) \\ \frac{x}{\epsilon} - 1 + \ln(\epsilon) \forall (x \leq \epsilon) \end{cases}$$

The first derivative of this function is continuous. The second derivative has a jump from 0 to  $-1/\epsilon^2$  at  $x = \epsilon$ .

Bound check epsilon

This is the epsilon parameter used in the [link: to obvious location/Find vars near bounds] utility under the Solvers Analyze Menu.

Far from nom bignum

This is the bignum parameter used in the [link: to obvious location/Find vars far from nom] utility under the Solvers Analyze Menu.

Integrator state log

This is the name of the file for integrator state variable output. It defaults to y.dat in the current directory.

Integrator observation log

This is the name of the file for user defined observation output during integration. It defaults to obs.dat in the current directory.

Integrator log SI units

This switch causes the output to be written in SI units or in the user's selected interface units.

<u>Integrator log columns</u>	This option selects how the state and observation logs should be formatted. We can produce fixed or variable width formats suitable for import into nearly any other software package.
<u>Overwrite integer logs</u>	This switch lets the user control whether integration log files should be appended or replaced with each run.
<u>Check numeric rank after solving</u>	When selected the numeric rank will be checked at the solution and a message will be displayed if the system is rank deficient.
<u>Show block summary</u>	When selected the cost statistics (cpu, iterations, evaluations) for all blocks of significant size will be listed to the screen after each solve.

## 6.3 AVAILABLE SOLVERS

Here is the list of solvers that at one time or another have been connected to ASCEND:

- Slv
- QRSlv
- LSODE
- MINOS
- LSSlv
- Opt (SQP)
- CONOPT
- Make MPS

All of these solvers may not be available in your installation of ASCEND. A brief description of ASCEND's primary solver, QRSlv, follows.

### 6.3.1 QRSLV

QRSlv is a nonlinear algebraic equation solver based on the paper "A Modified Least Squares Algorithm for Solving Sparse NxN Sets of Nonlinear Equations" by A. Westerberg and S. Director (EDRC TECH REPORT 06-5-79).

### 6.3.1.1 PARAMETERS

Following is an incomplete list of control parameters for the QRSlv algorithm. Most users will only change the time limit, iteration limit, and maximum residual as the default parameter values work quite well.

<u>Time limit</u>	The total number of seconds allowed in 1 push of the Solve button.
<u>Iteration limit</u>	Total number of iterations in for any single partition in the problem.
<u>Minimum pivot (epsilon)</u>	the smallest pivot value allowed in the linear solution of a subproblem.
<u>Pivot tolerance</u>	pivot selection criterion.
<u>Maximum residual</u>	This is the maximum absolute error that QRSlv is allowed to consider an equation as solved. Self scaling equations will more easily satisfy this than those that aren't. E.g. an energy balance (with terms the size of $10^8$ ) will have a far harder time meeting this convergence criterion if you do not divide them through by an appropriate constant.
<u>Partitioning</u>	If off, entire problem will be solved as a block. Divergence is usually the result on nonlinear problems of any size above 25 or so.
<u>Detailed info</u>	QRSlv spews all sorts of info if you turn this switch on. The utility of such info is often as much for the authors of slv as for the user. The volume of info is large. Most of the spew (that to do with singletons (1x1 blocks) is suppressed if the switch 'show singletons details' is off.
<u>Auto-resolve</u>	When complete, the solver is supposed to rerun itself for changes of significance made in the interface if this switch is on.
<u>write to file SlvLinsol.dat</u>	If this switch is on, a whole set of files named SlvLinsol.dat.X where X is integer are produced during the solution of the problem. The X increments for each successive linear system inversion or solution. The files contain Jacobian and rhs data in machine readable forms for import to stand-alone solver tools. There are generally quite a lot of them. X always starts at 0 for a given ascend session and goes up from there.
<u>show singletons details</u>	When the 'detailed solving info required' switch is ON this switch controls whether or not full singleton solving information is shown. In particular, if this is off all direct solve spew is cancelled, leaving that which usually of interest, the NxN block solution iterations, to be displayed.

bipartial pivoting

An experimental option for stabilizing the RANKI algorithm on hard problems. It enables searching of both current row and column during linear factorization. It is somewhat more expensive in terms on fill and CPU time, but can lead to solution of otherwise unsolvable problems. The modification is due to Joe Zaher. This option is likely to be replaced by a choice of several linear routines eventually. The original motivation came from distillation models which become illconditioned as tray number grows.

## 6.4 DEBUGGER

The Debugger shown in Figure 6-4 is an aid for examining the variables and relations in the Solver. The debugger is often used in tandem with the incidence matrix because the debugger is queried using the solvers's internal relation/variable indexing (which starts at 0). When a variable (relation) number is typed in the variable (relation) entry box the variable (relation) Name and Attribute buttons may be clicked to obtain information about the variable (relation). The information is printed to the console window. The variable (relation) may also be exported to either the Browser or Probe by making the appropriate selection under the export pull down menu.

Figure 6-4 The Debugger Window

Variable:
<input type="text"/>
Name
Attributes
Export
Equation:
<input type="text"/>
Name
Attributes
Export
Block:
<input type="text"/>
Variables
Equations
Export to probe
System:
Variables
Export to probe
OK Help

When a variable or relation number is entered in the debugger, the corresponding partitioned block number appears in the 'block' entry box. Statistics on the number of rows and columns in the block are displayed just below the block entry box. Note that a block number can also be entered directly into the block entry box. The Variables (Equations) pull down menu below the block entry box contains the selections Values, Attributes, and Probe (and Find Dependent). Selecting Values or Attributes will write the requested information to the console for each variable (equation) in the block. Selecting Probe will export the block's variables (equations) to the probe. Selecting Find Dependent under the Equations pull down menu will write the name of any dependent equations within the block to the console. Selecting the Export to Probe button will export both the block's variables and equations to the probe.

The debugger also responds to requests relating to the overall system, or current solver instance. A Variables pull down menu and an Export to Probe button are located beneath the 'System' label on the debugger. The Variables menu contains the selections Values, Attributes, Reset Values, and Reset Nominals. Selecting Values or Attributes will write the requested information to the console for each variable in the system. Selecting Reset Values will reset the system's variables to their nominal values. Selecting Reset Values will reset the system's nominals to their current variable values.

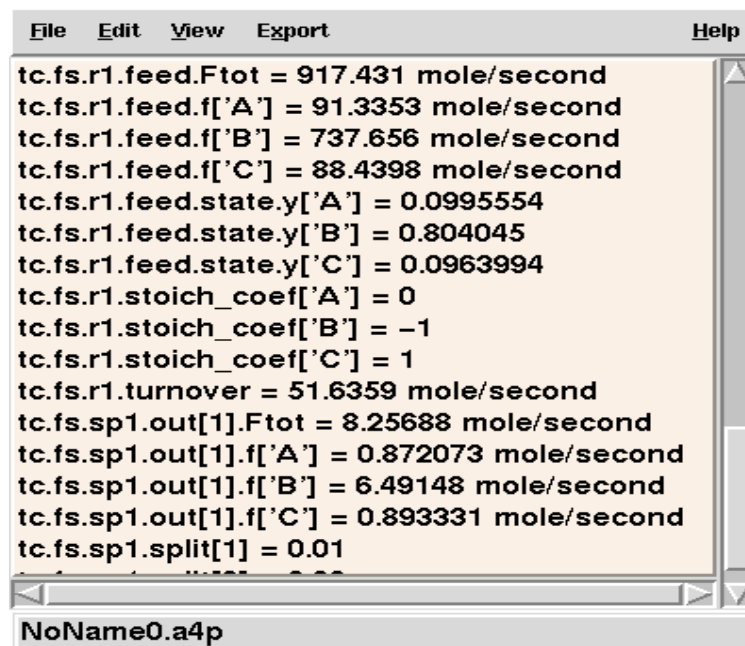
# CHAPTER 7 THE DATA PROBE WINDOW

## 7.1 OVERVIEW

The [data probe shown in Figure 7-1](#) is a window which manages collections of names from the ASCEND simulation universe. Each collection is kept in a buffer, and the user can switch among as many buffers as are needed for convenience. For example, the first buffer may be used as a set of bookmarks to store the names of interesting submodels within a large simulation, a second buffer can be used to monitor a set of key variables, and a third can be used to monitor specifications. The browser provides a two-level view of information - the probe provides a random access view. (A new GIF is required for this window -- the Help button is in the wrong place)



Figure 7-1 Probe window



Names are imported to any collection buffer from the other parts of the user interface or from a previously saved file of names. Once collected, a name remains in the buffer until the user removes it, even if the type library and simulations are deleted. This way the set of names is preserved when the user makes a small modification to a MODEL and rebuilds it.

Names in probe buffers are displayed with their corresponding values or other attributes as appropriate. When a name is not well defined (perhaps because the simulation it came from has been deleted temporarily) the attribute displayed is “UNCERTAIN.” As soon as the name becomes well-defined again by having a corresponding simulation object built, the correct attribute will appear. Names of atomic objects (reals, integers, sets, symbols, booleans) which have not yet been assigned a value will be shown as “UNDEFINED” until some operation assigns them a value.

## 7.2 THE FILE MENU

### New buffer

This starts another collection of names, which is initially empty. Each buffer receives a standard name when it is created, NoNameX.a4p, where X is the number of the buffer. These buffer names appear at the bottom of the File menu.

<b>Read file</b>	This appends a file full of names into the current buffer and will automatically attempt to associate them with the simulations in the system. This way the name list can be reloaded from a prior work session. The file name is not associated with the buffer.
<b>Save</b>	This will save the names in the current buffer to a file with the buffer's menu name. If you wish to save with a more meaningful name, use "Save as." Values are not saved with these names. To save the values, use the Print command.
<b>Save as</b>	This allows you to specify the directory and file name in which to save the names in the current buffer.
<b>Print</b>	This lets you print the current buffer to a printer or a file. This prints what you see in the buffer window, including the values. The printer setup dialog will pop up for you to set the destination.
<b>Close window</b>	Close this window. To reopen use the Data Probe tool under the Tools menu in the Script or select the PROBE button in the Toolbox.
<b>Exit ASCEND</b>	Exit the ASCEND system. You will be asked to verify that you really want to exit.
<b>Buffer list</b>	You can create several different Probe buffers. A list of the buffers appears at the bottom of the File menu.

## 7.3 THE EDIT MENU

<b>Highlight all</b>	Select all items in the displayed buffer.
<b>Remove selected names</b>	This option removes all highlighted lines in the window. The selection in the probe can be set in a disjointed fashion using Ctrl while selecting. One can select contiguous items by dragging or by holding down the Shift key when selecting the first and last item to be highlighted.
<b>Remove all names</b>	This option removes all names in the current buffer window.
<b>Remove UNCERTAIN names</b>	This removes all names that are not well defined. These are the names displayed as "name = UNCERTAIN."
<b>Copy</b>	This copies all the selected items in the current buffer to the computer clipboard. The items may then be pasted into any application that supports the clipboard -- for example, another ASCEND window, a Frame document and Excel.

## 7.4 THE VIEW MENU

<b>Font</b>	Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.
<b>Open automatically</b>	Toggles a switch which, if set, will cause the Browser window to open whenever anything is placed into it by an export command.
<b>Save window appearance</b>	Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a <i>.a4o</i> text file for this window which the sytem stores in the subdirectory <i>ascdata</i> in your “home” directory.

## 7.5 THE EXPORT MENU

<b>to Browser</b>	This option sends the first selected name in the probe to the browser.
<b>to Display</b>	This options sends some form of the selected names in the probe to the Display slave window, replacing whatever used to be in the display.

## 7.6 THE PROBE FILTER

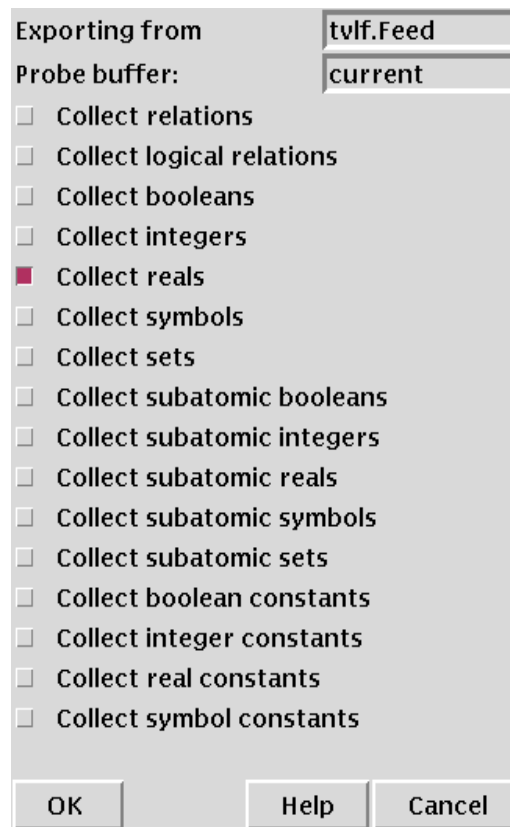
A class or classes of object can be imported to the probe en masse. The [import filter shown in Figure 7-2](#) lets you select which collection of names (probe buffer) is to receive the imported names which are of the types checked. Currently the probe filter window is accessible only from the Browser Export button.

The filtering import can also be executed from the Script using the PROBE command. The list of ones and zeros required for the PROBE command is ordered in the same way as the list of types in the import filter window. The easiest way to set the list of ones and zeros is to use the Script recording feature and the Browser Export Many to probe button.

### 7.6.1 THE HELP MENU

Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.

Figure 7-2 Probe import filter



The image shows a 'Probe import filter' dialog box. It has two text fields at the top: 'Exporting from' with the value 'tvlf.Feed' and 'Probe buffer:' with the value 'current'. Below these are 17 checkboxes for different data types. The 'Collect reals' checkbox is checked, indicated by a red square. The other checkboxes are unchecked. At the bottom are three buttons: 'OK', 'Help', and 'Cancel'.

Exporting from	tvlf.Feed	
Probe buffer:	current	
<input type="checkbox"/>	Collect relations	
<input type="checkbox"/>	Collect logical relations	
<input type="checkbox"/>	Collect booleans	
<input type="checkbox"/>	Collect integers	
<input checked="" type="checkbox"/>	Collect reals	
<input type="checkbox"/>	Collect symbols	
<input type="checkbox"/>	Collect sets	
<input type="checkbox"/>	Collect subatomic booleans	
<input type="checkbox"/>	Collect subatomic integers	
<input type="checkbox"/>	Collect subatomic reals	
<input type="checkbox"/>	Collect subatomic symbols	
<input type="checkbox"/>	Collect subatomic sets	
<input type="checkbox"/>	Collect boolean constants	
<input type="checkbox"/>	Collect integer constants	
<input type="checkbox"/>	Collect real constants	
<input type="checkbox"/>	Collect symbol constants	
OK	Help	Cancel

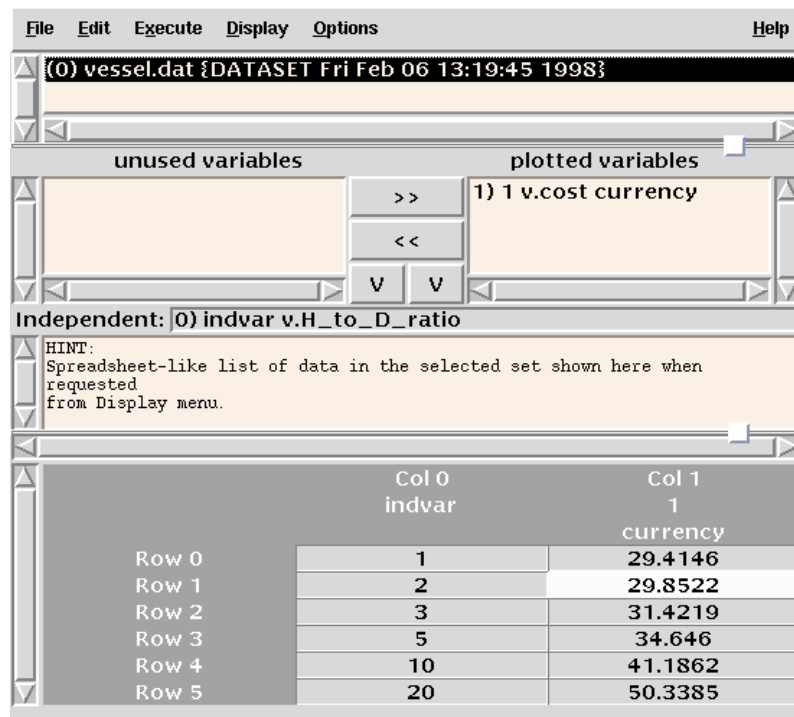


## CHAPTER 8 ASCPLOT

### 8.1 PLOT MAKER

The following contains a description of the options available in each of the *Ascend Plot* menus. The *Ascend Plot* window shown in Figure 8-1

Figure 8-1 The Ascend Plot Window



is a result of clicking on the ascplot button from the Toolbox with the left mouse button.

### 8.1.1 THE EDIT MENU

From the **Edit** Menu, the following options are available when a data set has not yet been loaded: **Load data set** and **Select grapher**. The **Save data set**, **Unload data set**, and **Merge data sets** options are available after one or more data sets have been loaded into the plot window.

#### 8.1.1.1 LOAD DATA SET

Selecting **Load data set** opens the *File select box* window. This window is used to select the file that contains the data generated from the dynamic simulation. The default file is obs.dat. This file contains the observation variables as set forth in the dynamic library models. After having selected the appropriate file, press the OK button and return to the *Ascend Plot* window.

#### 8.1.1.2 SAVE DATA SET

This option is currently not functional.

#### 8.1.1.3 UNLOAD DATA SET

By highlighting the desired data set and selecting **Unload data set** from the **File** menu, the user can remove the data set from the *Ascend Plot* window. The *Delete these data sets?* window appears to verify that the user wants to remove the indicated data sets.

#### 8.1.1.4 MERGE DATA SETS

#### 8.1.1.5 SELECT GRAPHER

Currently, the only supported grapher is Xgraph (or its tk flavored version tkxgraph). Other possible graphers are XMGR and gnuplot. Since these graphers are not distributed with the ASCEND distribution, they are also not supported.

### 8.1.2 THE EXECUTE MENU

To plot the variables in the plotted variables section, select **View plot file** from the **Execute** menu.

### 8.1.2.1 VIEW PLOT FILE

This option will plot the variables displayed in the plotted variables section of the *Ascend Plot* window.

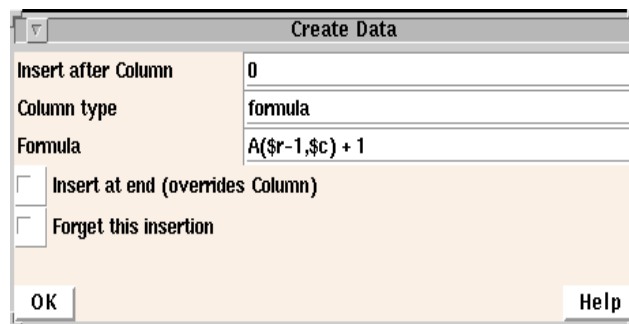
### 8.1.2.2 WRITE PLOT FILE

To save the output in its graphical representation, select **Write plot file** from the **Execute** menu. Selecting this option opens the *File select box*. Enter the name of the file to be saved and press the OK button. The default extension for the graph is .xgraph.

### 8.1.2.3 INSERT COLUMN

Selecting the **Insert column** option from the **Execute** menu opens the *Create Data* window. This window is shown in Figure 8-2.

Figure 8-2 The Create Data Window



There are several options available from the *Create Data* window.

#### 8.1.2.3.1 Insert after Column

This can be any number between 0 and the maximum number of variables in the observation file. For example, if the user wishes to add a column after the third column, the user should enter a 3 in this space.

#### 8.1.2.3.2 Column type

The default value for Column type is data, however by placing the cursor over the data box and pressing the left mouse button, another option is revealed. The other option is formula. The user should select data if no formula can be used to describe the information to be added to the spreadsheet. The user should select formula if that is appropriate. In this case, a column was inserted after Column 0 and we are using the formula Column type.



#### 8.1.2.3.3 Formula

If the data option was selected in the previous section, then this does not apply. However, if the formula option was selected, then the user can edit the default formula. The default formula takes the value of the variable in the current row (\$r) and the column before the new column (\$c-1) and adds one (+1) to it.

#### 8.1.2.3.4 Insert at end (overrides Column)

The user can select this box to place the new column after the last column in the spreadsheet. This will override anything in the Insert after Column line.

#### 8.1.2.3.5 Forget this insertion

The user can select this box to ignore the changes made to the spreadsheet.

#### 8.1.2.4 RECALCULATE COLUMN

This option is currently not functional.

#### 8.1.2.5 INSERT ROW

The insert row option has the same options as the Insert Column option. Note that the formula take the value from the row immediately before it (\$r-1) and the current column (\$c) and adds one (+1) to it.

#### 8.1.2.6 RECALCULATE ROW

This option is currently not functional.

### 8.1.3 THE DISPLAY MENU

The **Display** menu has various features which include showing and hiding the data in the spreadsheet, setting plot titles, loading old plots, updating existing plots, and deleting plots.

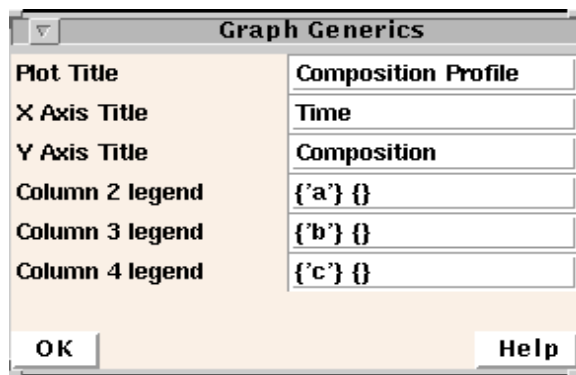
#### 8.1.3.1 SHOW DATA / HIDE DATA

Selecting the **Show data** option from the **Display** menu loads the data into the spreadsheet in the bottom section of the *Ascend Plot* window. This option then toggles to **Hide data**. Selecting this option will hide the data just loaded into the spreadsheet section of the window.

### 8.1.3.2 SET PLOT TITLES

Selecting the **Set plot titles** option from the **Display** menu opens the *Graph Generics* window. This window is shown in Figure 8-3.

Figure 8-3 The Graph Generics Window



There are several options within this window depending on the number of variables being plotted.

#### 8.1.3.2.1 Plot Title

The user can change the default title (AscPlot) to something that is more descriptive and meaningful for the given data. In this case, we set the title to be Composition Profile since we are plotting the mole fractions of the components in the system.

#### 8.1.3.2.2 X Axis Title

The user can change the default title (X) to something more descriptive. In this case, we are plotting the time on the x-axis.

#### 8.1.3.2.3 Y Axis Title

The user can change the default title (Y) to something more descriptive. In this case, we are plotting the Composition on the y-axis.

#### 8.1.3.2.4 Column # legend

In this case, (#) is the number of the variable being plotted. If variables 2, 3, and 4 are being plotted, there will be entries in the *Graph Generics* window entitled Column 2 legend, Column 3 legend, and Column 4 legend. These entries can be changed to something less descriptive than the default. Usually the default for this field is a bit much. In this case, the legend was changed to 'a', 'b', and 'c'.

### 8.1.3.3 LOAD OLD PLOT

This option is currently not functional.

### 8.1.3.4 UPDATE PLOT

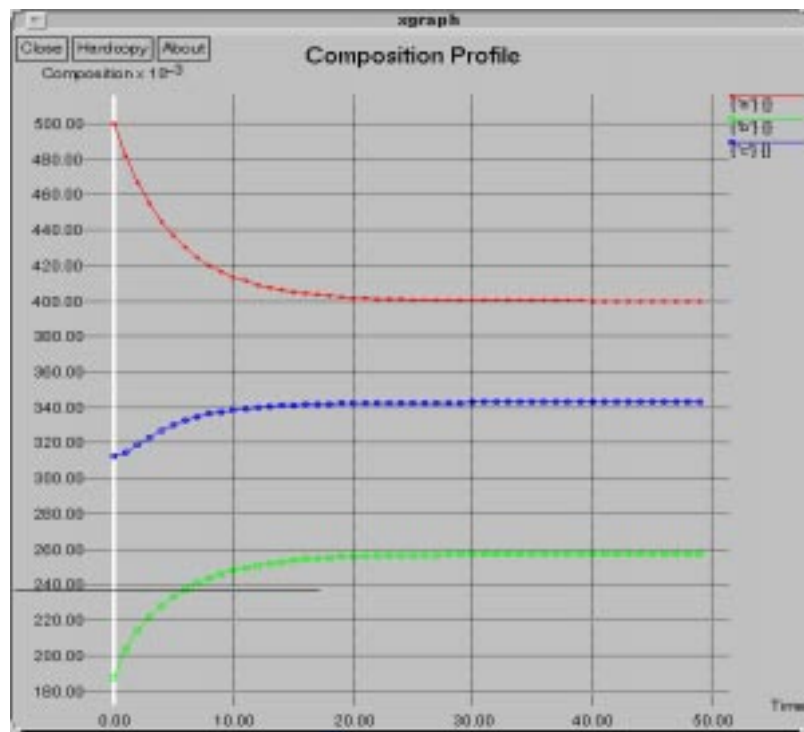
This option is currently not functional.

### 8.1.3.5 DELETE PLOT

This option is currently not functional.

The plot of the completed graph is shown in Figure 8-4.

Figure 8-4 Complete Plot



### 8.1.3.6 THE GRILL



The grill is located directly to the right of PLOT MAKER. Clicking on this button with the left mouse button opens the *XGraph Control* window. By clicking on the More button located at the bottom of the window, the user can scroll through numerous available options for the graphs. Some of these options include line color, fonts, graph type (i.e. log or semilog), and marker types. These are left to the user to explore.

## 8.2 NAVIGATION

Open a file using the **Load data sets** option from the **File** menu. You will notice that the selected file is now displayed in the top section of the *Ascend Plot* window. By double-clicking on the file name with the left mouse button, the observation variables are now placed in the section entitled unused variables. The unused variables are the list of variables that the user does not want to look at in the current graph.

To select a variable to plot, highlight the desired variables using the left mouse button and click on the (>>) button. This will move the variable from the unused variables list to the plotted variables list. Once this is done, you can now plot the variable.

The two buttons separating the unused variables section and the plotted variables section are used to add (>>) and remove (<<) variables to and from the plotted variables list.

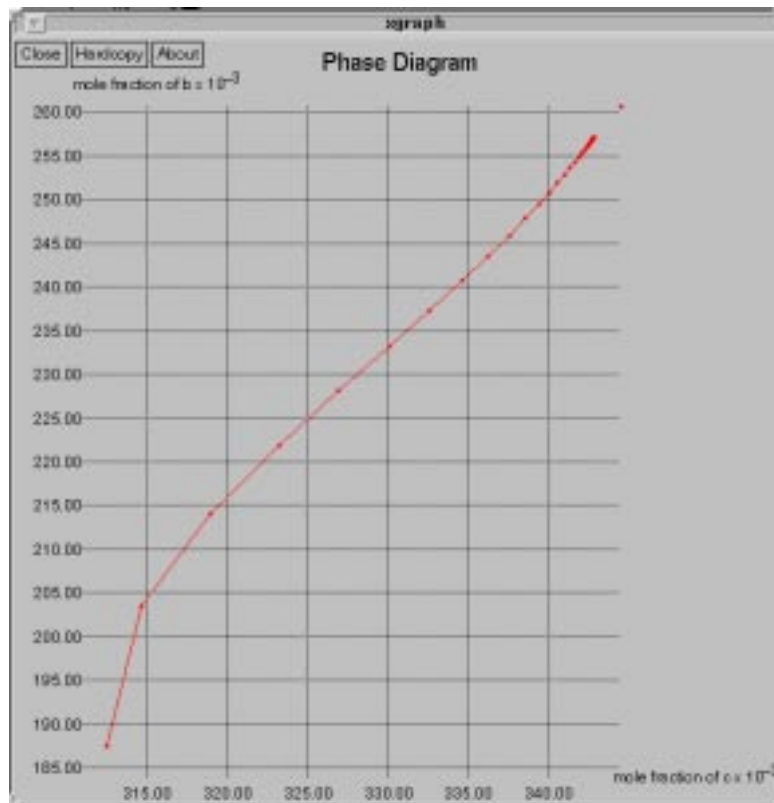
You will notice that there is a section of the *Ascend Plot* window entitled Independent. Here the independent variable is time. This was set in the dynamic library file. If the user desires to look at the phase plot of two of the compositions, the user must move one of the compositions into the Independent variable position.

To do this, let's assume that all of the variables are currently in the unused variables list and we wish to plot the composition of component 'c' versus the composition of component 'b'. Thus, component 'c' is now going to be our independent variable. Highlight component 'c' in the unused variables list and press the (V) button. This button is one of two buttons located directly under the (>>) and (<<) buttons. The (V) button on the left is used to move variables between the unused variables list and the Independent variable list while the (V) button on the right is used to move variables between the plotted variables list and the Independent variable list. Therefore, we are going to use the (V) button on the left.

By doing this, we see that the composition of component 'c' is now the independent variable and the time is now an unused variable. Select the composition of component 'b' and press the (>>) button to move the variable from the unused variables list to the plotted variables list. The only remaining task is to edit the plot title and axes using the **Set plot titles** option from the **Display** menu. Assuming we have done this as described above, the resulting graph is shown in Figure 8-5.

The remaining section of the *Ascend Plot* window is the HINT: section. This section contains a brief description of the various buttons and sections of the Ascend Plot window.

Figure 8-5 Phase Diagram

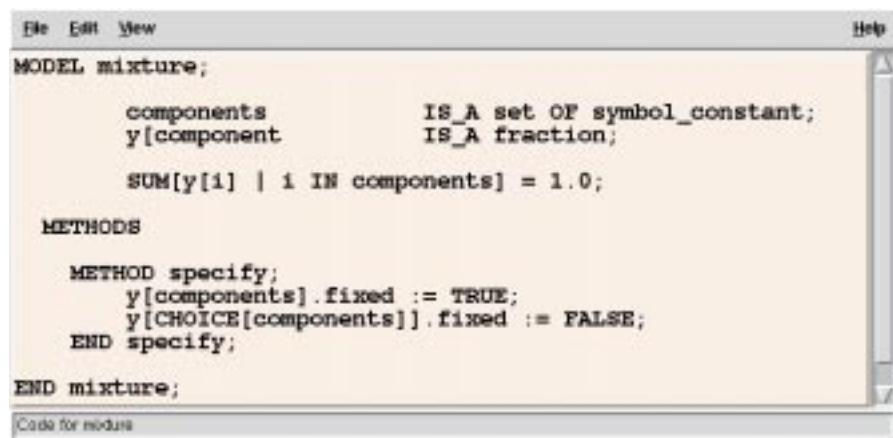


# CHAPTER 9 DISPLAY SLAVE

## 9.1 OVERVIEW

The display slave window (Figure 9-1) functions as a dumping ground for information which is too complex to display in other ways in the Library, Browser or other primary windows. It has rudimentary editing abilities so the user can manually adjust the format of displayed information if needed, for example by rearranging a highly nonlinear relation with more than a few variables. Changes to displayed text do not affect the rest of the system in any way.

Figure 9-1 Display slave window

The image shows a screenshot of a software window titled 'Display Slave'. The window has a menu bar with 'File', 'Edit', 'View', and 'Help'. The main area contains a text editor with a model definition for 'mixture'. The text is as follows:

```
MODEL mixture;  
  
    components          IS_A set OF symbol_constant;  
    y[component]        IS_A fraction;  
  
    SUM[y[i] | i IN components] = 1.0;  
  
    METHODS  
    METHOD specify;  
        y[components].fixed := TRUE;  
        y[CHOICE[components]].fixed := FALSE;  
    END specify;  
  
END mixture;
```

At the bottom of the window, there is a status bar that says 'Code for module'.

## 9.2 DISPLAY FILE MENU

<b>Print</b>	This option brings up the default print dialog described in the section Utilities. The print command can be used to save the displayed text to a file.
<b>Close window</b>	The option closes the display window.
<b>Exit ASCEND</b>	Exit the ASCEND system. You will be asked to verify that you really wish to exit ASCEND.

## 9.3 DISPLAY EDIT MENU

<b>Cut</b>	Cut highlighted text to the computer paste buffer. You can paste this text into any application that supports cut, copy and paste -- e.g., into Framemaker or Excel.
<b>Copy</b>	Copy highlighted text to the computer paste buffer. You can paste this text into any application that supports cut, copy and paste -- e.g., into Framemaker or Excel.
<b>Paste</b>	Paste the contents of the computer paste buffer into the Script buffer at the point of the cursor.

## 9.4 DISPLAY VIEW MENU

<b>Show comments in code</b>	<p>This option controls whether or not comments are displayed when code is displayed <b>as read from source files</b>. This setting is not retroactive; that is code already displayed will not be redisplayed when changing this setting.</p> <p>When code is displayed in the machine representation, i.e. with equations and set expressions in postfix (reverse polish) notation, comments are never displayed.</p>
<b>Save Display options</b>	Saves the current settings for this window for its options. These become the default settings when opening this window in the future. These settings are saved in a <i>.a4o</i> text file for this window which the sytem stores in the subdirectory <i>ascdata</i> in your “home” directory.
<b>Font</b>	Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.

**Open  
automatically**

Toggles a switch which, if set, will cause the Browser window to open whenever anything is placed into it by an export command.

**Save window  
appearance**

Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a *.a4o* text file for this window which the system stores in the subdirectory *ascdata* in your “home” directory.

### 9.4.1 FONT

This option brings up the standard font setting dialog so you can change the size, style, and font of the characters in the display window.

### 9.4.2 OPEN AUTOMATICALLY

This option controls whether or not the display slave window opens automatically when it receives information. Sometimes it is easier to send several items to the display and then open it at the end.

### 9.4.3 DISPLAY HELP MENU

Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.

## 9.5 TITLE LINE

The title line at the bottom of the window is set by the last client to export something to the display. The user may edit the title, but the next time new information is displayed, these edits will be lost.





# CHAPTER 10 ASCEND UNITS

## 10.1 THE MENU BAR

The Units Tool Set provides tools to allow the user to change the display units for variables.

### Units vs dimensions

We distinguish between *units* and *dimensions* in ASCEND. The dimensions of acceleration, for example, are  $L/T^2$ , i.e., length/time squared. Units for acceleration are:  $m/s^2$ ,  $ft/hr^2$  and so forth. A chapter in the Howto book tells you how to enter dimensioned equations into ASCEND and includes a useful discussion on units and dimensions. Also see the ASCEND syntax document for a discussion.

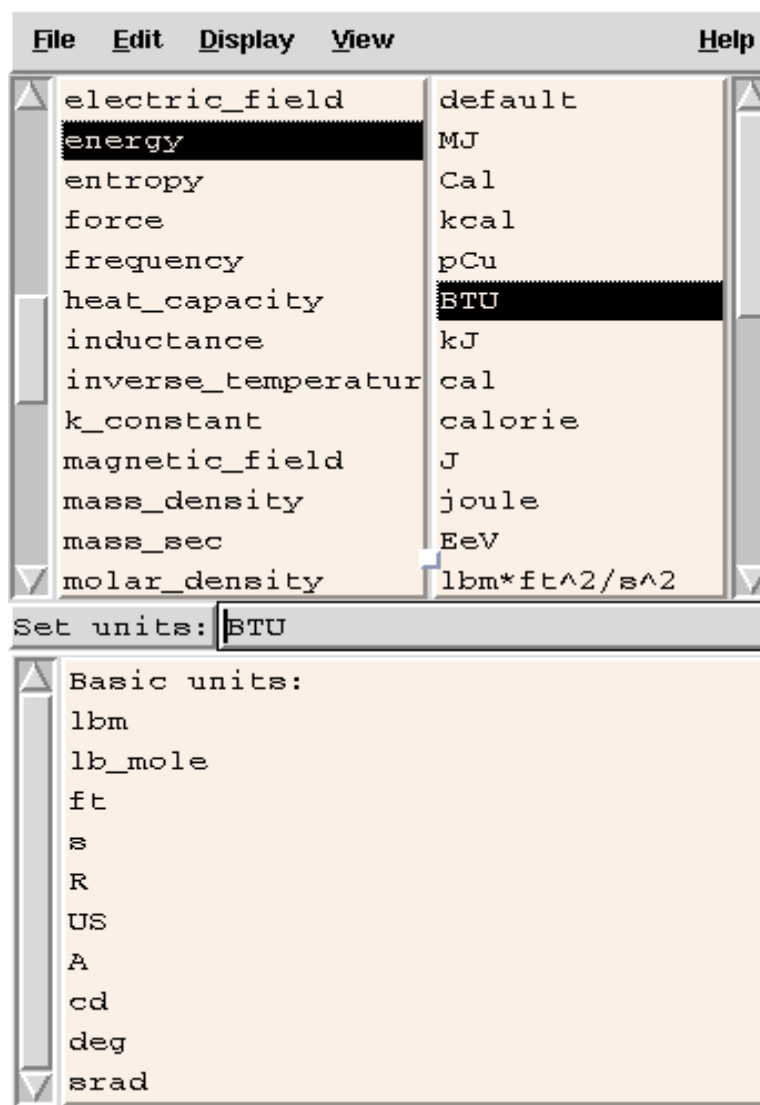
### Typical use

The user will typically first pick the overall system of units such as SI, American Engineering or cgs. Alternatively the user may select to use the *default* display of units for some or all variable types. Displaying in default units means ASCEND will present the units in terms of the ten basic dimensions supported by ASCEND (length, time, temperature, etc.). The user can select the units to be used for each basic dimension. Whichever of these alternatives the user selects, he or she may then also choose the units ASCEND should use to display particular variable types. An example would be to select first SI units, then override the display of energy to be in default units and pressure to be in atm.

Once users have created their favorite choices for display units, they may save them to files for later restoration.

We describe here the various tools available within the Units tool set, as seen in

Figure 10-1 The Units of measure window



### 10.1.1 UNITS FILE MENU

- |                     |  |
|---------------------|--|
| <b>Read file</b>    | Reads in a file previously saved using the “Save file” command. Restores the display units to those previously saved.  |
| <b>Save file</b>    | Writes out (in the current working directory) a plain text version of the user specified display units. Units which are defaulted are not written to this file. One can restore the display units to those currently set by reading this file back in later. |
| <b>Close window</b> | Close this window. To reopen it, select Measuring units in the Tools menu of the Script or select the UNITS button on the Toolbox window.  |

**Exit ASCEND** Exit the ASCEND system. You will be asked to verify that you really wish to exit ASCEND.

### 10.1.2 UNITS EDIT MENU

**Set precision** Use the slider switch for this tool to set the number of digits of precision for displaying variable values to between 4 and 16. Precision is the number of digit displayed when the number is displayed using scientific notation. For example, 0.12345678 e04 for 1234.5678 has a precision of 8 digits.

**Set basic units** Drops a cascading window in which are listed the ten basic dimensions for ASCEND. You can select in which units you wish to see each base dimension to be displayed using this list.

### 10.1.3 UNITS DISPLAY MENU

**Show all units** Causes the Display window to open showing the extensive set of units conversions currently used in ASCEND. The list opens in the Display window.

### 10.1.4 UNITS VIEW MENU

**SI(MKS) set** Pushing this button makes the default display units SI units.

**US Engineering set** Pushing this button makes the default display units US Engineering units.

**CGS set** Pushing this button makes the default display units CGS units.

**Font** Opens the window that lets you reset the fonts for this window. You can select the type of font, the style (bold, etc.) and the size for the font.

**Open automatically** Toggles a switch which, if set, will cause the Browser window to open whenever anything is placed into it by an export command.

**Save window appearance** Saves the current settings for this window for font settings and window size and placement on your computer screen. These become the default settings for opening this window in the future. These settings are saved in a *.a4o* text file for this window which the system stores in the subdirectory *ascdata* in your “home” directory.

### 10.1.5 UNITS HELP MENU

Brings up a text description of where to look for help on this window (i.e., it points to the pdf version of this document on the WWW.) You may, of course, look into the section mentioned in any local (but perhaps outdated) copy of the documentation.

## 10.2 AN ESSAY ON UNITS VS DIMENSIONS

ASCEND stores all numbers in SI (MKS) units internally. The units associated with a dimensionality (as exemplified by some atom) will be used when displaying variables of that dimensionality. These units can be manipulated through the Units window.

Numbers with unrecognized dimensionality (higher derivatives, multipliers, residuals and what not) will be given units consistent with the display units defined for the 10 base dimensions. The display units for the 10 dimensions can be changed through the Units window Display menu if you prefer an alternate default set such as US engineering, and so forth.

We recognize 10 base dimensions in the compiler:

L	distance	meter	m
M	mass	kilogram	kg
T	time	second	s
E	e- current	ampere	A
Q	quantity	mole	mole
TMP	temperature	Kelvin	K
LUM	luminous intensity	candela	cd
P	plane angle	radian	rad
S	solid angle	steradian	srad
C	currency	currency	CR

The units conversions are defined in \$ASCENDDIST/compiler/units\_input, which is not particularly restricted. Units\_input is converted to an efficient binary form (unitsfile.uni) at the time ASCEND is installed.

It can be argued that C is not a fundamental dimension, from a physical standpoint. There is more to life than physics: there is economy, hence engineering, hence an Advanced System for Computations in ENgineering Design.

The dimensions P and S are ‘supplementary’ according to the General Conference, but their use makes the coding of ASCEND much cleaner and easier.

### 10.2.1 ON UNITS

The left box in the Units window lists a set of atom types, each having a unique dimensionality. Selecting an atom in the left box will fill the right box with different possible units that the system knows about to display this type of variable. Dimensionless atoms and wild dimensioned atoms are not shown since they do not have display units. If you do not see an atom you expect here, it is because ASCEND already found another atom of the same dimensionality, e.g. fugacity may show up instead of pressure.

Selecting a unit in the right box sets that unit as the display unit for all variables having the same dimensionality of the selected atom in the left box. Thus picking **atm** for fugacity will also change pressure units to **atm**. Selecting ‘default’ will cause the display to be a combination of the *fundamental* units (a nice way to remind oneself of the fundamental units for energy, for example).

Fundamental units are the units corresponding to single dimensions. These units are chosen on the Display menu under the dimension choices. No atoms with fundamental units are listed in the left box. The current set of fundamental units is always shown at the very bottom of the units window. This set is used whenever a value is displayed which does not have a user specified units set associated with its dimensionality. The fundamental units are created via the `units_input` file mentioned above. If you do not find one you want, ask whoever compiled your version of `unitsfile.uni` to add the missing unit and rebuild the `unitsfile.uni`.

If converting the units for a variable makes the display of that number impossible (e.g., due to overflow). ASCEND will first attempt to display it using its fundamental units. If it still cannot be displayed, it will be displayed in SI units.

You may specify a new combination of existing units (e.g. Pa\*s) using the **Set units** which is the line at the bottom of the window. Type in the combination desired and press RETURN.

Unit strings may not have parentheses in them. For example, kg/(m\*s<sup>2</sup>) is not allowed.



# CHAPTER 11 THE ASCEND TOOLBOX

The [toolbox window shown in Figure 11-1](#) lets the users open and close the various windows for the tool sets available in ASCEND. The toolbox window is a vertical window containing 12 buttons: exit, ascplot, help, utilities, bug report, LIBRARY, BROWSER, SOLVER, PROBE, UNITS, DISPLAY, SCRIPT.

The buttons in the toolbox with names in ALL CAPS (LIBRARY and following) open and close the windows for the corresponding tool sets

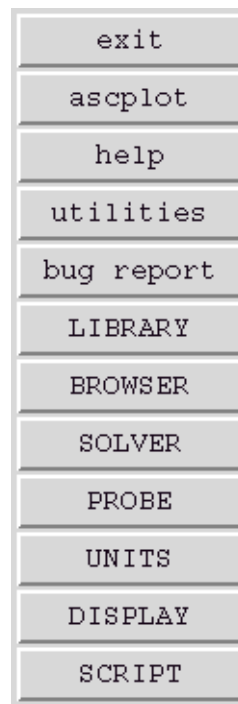


Figure 11-1 The ASCEND Toolbox window.



in ASCEND<sup>1</sup>. As each of these toolsets has its own documentation, we shall not discuss them here. We discuss here only the first set of buttons: Exit, Ascplot, Help, Utilities, and Bug Report.

## 11.1 EXIT

This button shuts down all of ASCEND after making sure you want to quit. ASCEND does no checking to see if there is unsaved work so be certain you have saved what you want of it before selecting this button.

(For more advanced users, we note that, just before exiting, we call the tcl function `user_shutdown` which may be redefined in the `.ascendrc` file in your HOME directory. Under Windows, the `_ascendrc` is the name of the corresponding file.)

## 11.2 ASCPLOT

Selecting this button opens the plotting tool for ASCEND. You can find any file that contains data for a plot and plot it with this tool. Ascplot is described elsewhere.

## 11.3 HELP

Pressing this button will provide access to the Help Documentation for ASCEND. The help system is described elsewhere.

## 11.4 UTILITIES

Selecting this button opens the system utilities window. The system utilities window is described elsewhere.

## 11.5 INTERNALS

The internals window gives catalog access to the global Tcl arrays that maintain and control the GUI and other ASCEND options. It also gives access to the list of ASCEND functions (C callbacks) that are defined and their built-in documentation. Casual users should not use this tool.

---

1. The more advanced user should note that changing the `iconname` of a window (via `ascend.ad`) does not change its toolbox name.

## 11.6 BUG REPORT

The link

<http://www.cs.cmu.edu/~ascend/Email.html>

is connected to the web server for ASCEND at CMU. Alternatively, send a bug report to

[ascend+bugs@cs.cmu.edu](mailto:ascend+bugs@cs.cmu.edu)

if you cannot access this link. We do not have an 800 number, but we usually get to bug reports very quickly.

When submitting a bug report, please try to

1. Duplicate the error.
2. Tell us in excruciating detail how you duplicated it.
3. Report to us the platform and operating system (OS) on which you are running. Also please tell us the distribution number for the ASCEND code on which you are running. This is research software. We are not committed to backward compatibility, and we do not have access to all the platform/OS combinations out there. If the bug you report has been fixed in a newer version, your only fix is to get the new version or fix it yourself. If you are on a platform to which we do not have access, we will consider working out the bugs with you in the hope that you will then give us back a copy for the new platform.
4. Send along any model code you have that is involved in the bug manifestation. It may happen that, in the process of fixing the ASCEND bug, we could fix some of your model bugs. We are not in the business of debugging your model code unless it is also interesting to our research. We often find new applications of ASCEND interesting, however.
5. Subscribe to the ASCEND user mailinglist/bboard: Send mail to [ascend+subscribe@edrc.cmu.edu](mailto:ascend+subscribe@edrc.cmu.edu).



# CHAPTER 12 THE SYSTEM UTILITIES WINDOW

## 12.1 OVERVIEW

The [system utilities window shown in Figure 12-1](#) displays and allows modification of the variables which control the interaction of ASCEND with the operating system and with other programs.

The values of the variables are initialized from the user's environment, from the file `.ascend-config` in the user's HOME directory, and from settings within ASCEND.

If the user chooses to save the system utility settings, ASCEND writes the current values of the variables into file `.ascend-config` in the user's HOME directory. ASCEND will automatically reread those values in the next time it starts.

ASCEND Utility Settings	
Text print command	>> /dev/null cat
PRINTER	foo_bar
TCL_LIBRARY	/usr/local/lib/ascend/lib/
TK_LIBRARY dir	/usr/local/lib/ascend/lib/
WWW restart	rMozilla %U
WWW root URL	/usr/local/lib/ascend/help
WWW startup	rMozilla %U
Spreadsheet command	unset
<div>OK    Save    Read    More    Help</div>	

Figure 12-1 The System Utilities window manages ASCEND's interaction with the operating system and with other programs.

When working with the system utilities window, it is important to remember that changes to the variables propagate *immediately* throughout ASCEND, and that there is no way to undo or cancel changes made to the variables<sup>1</sup>.

## 12.2 VARIABLES

The system utilities window contains the following settings. Settings marked with an asterisk\* are not saved in `.ascend-config`.

To change a variable's value, click in the box to the right of the variable's label and type the new value. This new value is immediately available to the ASCEND system.

### 12.2.1 WWW ROOT URL

ASCEND distributes its help system as HTML documents, and spawns a web browser to view these documents. The variable WWW Root URL gives the root of the ASCEND help tree, and the variables WWW Restart Command and WWW Startup Command contains commands to connect to a running web browser and to start a new web browser, respectively.

WWW Root URL contains the first part of a URL to the ASCEND help tree; it is not necessarily a complete URL. The variable should end in a forward slash (/). The **Help** menus and buttons in ASCEND will append text to this value and invoke WWW Restart Command or WWW Startup Command with the complete URL.

The person who installs ASCEND at a site should set this variable to the root of the directory containing that site's copy of the ASCEND help files, for example:

```
file://localhost/usr/local/lib/ascend/help/  
at CMU ICES.
```

The value

```
http://www.cs.cmu.edu/~ascend/help/  
will connect you to the help pages at the ASCEND web site.
```

---

1. Some variables can be restored to the values in effect the last time the system utilities were saved, but this only works if the user has previously saved the values, and it does not restore every variable.

### 12.2.2 WWW RESTART COMMAND

This is a command to redirect the attention of your already running web browser to a new URL. If this command returns an error code, ASCEND will attempt to start a new browser using the WWW Startup Command.

If your favorite browser does not support restarting, set the value of this variable to `false`. This will cause a new browser to start for every help query from the ASCEND interface.

ASCEND will replace every occurrence of `%U` in this command with the URL to be viewed. The default value of WWW Restart Command is

```
netscape -remote openURL(%U)
```

### 12.2.3 WWW STARTUP COMMAND

This is a command to start your favorite web browser. This command is invoked if the value of WWW Restart Command is `false` or if attempting to start a browser using that command returns an error code.

ASCEND will replace every occurrence of `%U` in this command with the URL to be viewed. The default value of WWW Startup Command is

```
netscape %U
```

### 12.2.4 ASCENDLIBRARY PATH\*

The ASCENDLIBRARY variable contains a list of directories that the Library and Script tools search to find files containing ASCEND models and scripts.

The format of the directory list should resemble the PATH environment variable for your platform: a colon (:) separated list of directories (using forward slashes) on UNIX, a semicolon (;) separated list of directories (using backward slashes) on Windows.

The ASCENDLIBRARY variable is initialized from the user's environment or from the ASCEND binary; its value is not saved in the user's `.ascend-config` file.

### 12.2.5 SCRATCH DIRECTORY

The scratch directory is used to write the temporary and plot files that ASCEND creates. The temporary files are automatically deleted before you leave ASCEND, but the plot files are not (since people often want to save plots). You should periodically remove any plot files from the scratch directory, else you may slowly build up a large collection of past plot files.

Any existing directory you have write access to can be used as the scratch directory. Under UNIX, `/tmp` is the default value of the scratch directory. Under Windows, the directory given in the environment variable `TEMP`, `TMP`, or `TMPDIR` is used as the default value.

### 12.2.6 WORKING DIRECTORY

Typically, this is the directory you start ASCEND from, but it can be any existing directory you have write access to. Our handling of the working directory is a bit “flaky” at the moment because ASCEND’s command line allows the user to change directories without telling the rest of the interface about it. Intermediate files are sometimes written in the working directory.

### 12.2.7 PLOT PROGRAM TYPE

Currently, the only supported plot type is `xgraph plot` (abbreviated `xgraph`). This setting tells the plot window what type of plot file it should generate.

### 12.2.8 PLOT PROGRAM NAME

This is the name of your plotting program. It should accept the plot type listed in Plot Program Type as input.

The default is `xgraph` on UNIX and `tkxgraph` on Windows. Both `xgraph` and `tkxgraph` are available from the ASCEND web site:  
<http://www.cs.cmu.edu/~ascend/>

### 12.2.9 TEXT EDIT COMMAND

This is a command to spawn your favorite text file editor. (Currently, nothing in ASCEND invokes this command.)

The default is `emacs` on UNIX and `runemacs` on Windows.

### 12.2.10 POSTSCRIPT VIEWER

This is a command to spawn a program for viewing Postscript files. (Currently, nothing in ASCEND invokes this command).

The default is `ghostview` on UNIX and on Windows.

### 12.2.11 SPREADSHEET COMMAND

This is a command to spawn your favorite spreadsheet program. (Currently, nothing in ASCEND invokes this command<sup>2</sup>).

### 12.2.12 TEXT PRINT COMMAND

This entry displays the last command generated by the print dialog box. Changing the value of this entry will have no effect on future printing, since the print dialog manages all aspects of printing.

This value is displayed here as a hold-over from previous versions of ASCEND; developers sometimes use it as a check to make sure the print dialog is doing the right thing.

### 12.2.13 PRINTER VARIABLE\*

This entry displays the last printer the user selected in the print dialog box, or the value of the `PRINTER` or `LPDEST` environment variable if the user has not used the print dialog box during this ASCEND session.

Changing the value of this entry will have no effect on future printing<sup>3</sup>, since the print dialog manages all aspects of printing.

This value is not saved in the user's `.ascend-config` file.

### 12.2.14 ASCENDDIST DIRECTORY\*

The value of the `ASCENDDIST` environment variable is the directory containing the installed ASCEND distribution. If a user can see this

- 
2. Nothing invokes this command because there is no ASCEND code that supports it. Someone needs to write code that will write out the desired variables as columns of numbers suitable for importing into any spreadsheet. If you want to be that someone, let us know and we'll be happy to consult. We have some pseudocode for this already; contact us at `ascend@cs.cmu.edu`.
  3. This is not entirely true. This entry will change the value of the `PRINTER` environment variable (but not the `LPDEST` environment variable). Any command you invoke from ASCEND command prompt that depends on the `PRINTER` environment variable will use the value displayed in this entry.



variable inside the system utilities window, it means its value is correct. Changing the value will most likely cause things to break.

The person who installs ASCEND at a site is typically the only person who needs to be concerned with its value.

The ASCENDDIST variable is initialized from the user's environment or from the ASCEND binary; its value is not saved in the user's `.ascend-config` file.

### **12.2.15 TCL\_LIBRARY ENVIRONMENT VARIABLE\***

The value of the TCL\_LIBRARY environment variable is the directory containing the installed `*.tcl` files required by Tcl. If a user can see this variable inside the system utilities window, it means its value is correct. Changing the value will most likely cause things to break.

The person who installs ASCEND at a site is typically the only person who needs to be concerned with its value.

The TCL\_LIBRARY variable is initialized from the user's environment or from the ASCEND binary; its value is not saved in the user's `.ascend-config` file.

### **12.2.16 TK\_LIBRARY ENVIRONMENT VARIABLE\***

The value of the TK\_LIBRARY environment variable is the directory containing the installed `*.tcl` files required by Tk. If a user can see this variable inside the system utilities window, it means its value is correct. Changing the value will most likely cause things to break.

The person who installs ASCEND at a site is typically the only person who needs to be concerned with its value.

The TK\_LIBRARY variable is initialized from the user's environment or from the ASCEND binary; its value is not saved in the user's `.ascend-config` file.

## **12.3 BUTTONS**

The actions associated with the buttons on the system utilities window are:

### 12.3.1 OK

This button closes the system utilities window. Closing will fail if the scratch directory and working directory are not writable by the user.

### 12.3.2 SAVE

This button writes the current value of most of the variables in the system utilities window to a file called `.ascend-config` in your HOME directory<sup>4</sup>. ASCEND will read this file on startup to get your preferred values.

The variables whose names are in ALL CAPS (i.e., ASCENDLIBRARY, PRINTER, ASCENDDIST, TCL\_LIBRARY, TK\_LIBRARY) are **not** saved to `.ascend-config`. These are environment variables that are set as part of the login process. You may change them interactively, but their interactive values are not saved.

### 12.3.3 READ

The button causes the system utilities window to reread the values stored in `.ascend-config` in your HOME directory. This is useful for editing `.ascend-config` outside of ASCEND while running ASCEND, or for verifying that the changes you saved were properly saved.

### 12.3.4 MORE

The button rotates you through the pages of options.

### 12.3.5 HELP

The button should direct your web browser to this document.

---

4. Under Windows, you can set your HOME directory by setting the HOME environment variable by opening the Control Panel, double clicking the System icon, clicking the Environment tab, and adding the HOME variable to the list of user environment variables.



# CHAPTER 13 FONT SELECTION DIALOG

## 13.1 OVERVIEW

The [font selection dialog](#) (Figure 13-1) is used to select the font for the window from which it is opened. There is no way through the interface to change the font for every ASCEND window.

Currently, the fonts you select are not remembered across invocations of ASCEND. This is a feature we will be adding in a future release.

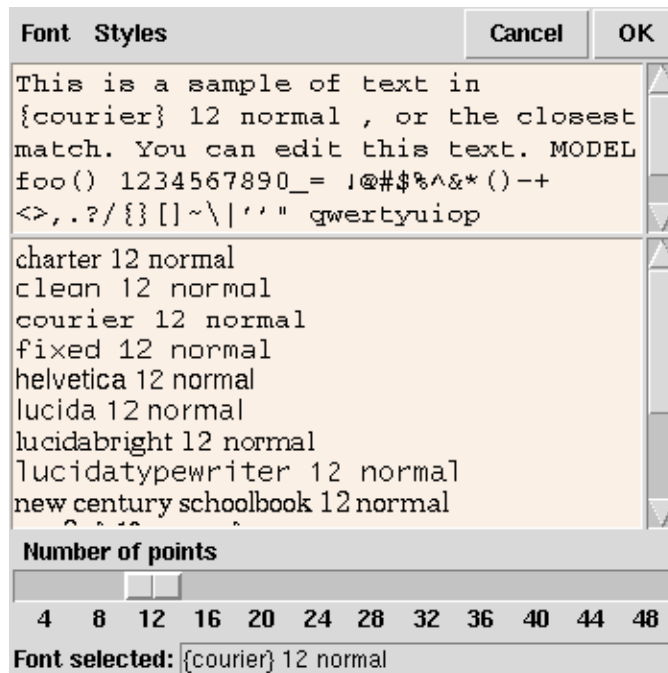


Figure 13-1 The font selection dialog.

To change the default fonts for ASCEND, see Setting the Default Font later in this chapter.

The font which the font selection dialog displays when it is opened is independent of the current window's font. This is actually a feature. When you close the font selection dialog (by pressing either the OK Button or the Cancel Button) and reopen it, it will display the same font as when it was closed. This way, once you find a font you like, you can change other ASCEND windows to this same font by simply opening the font selection dialog and pressing OK. As a default, the very first time you open the font dialog in an ASCEND session, the font is set to `Courier 12 normal`.

The font selection dialog has eight parts: Font Menu, Style Menu, Cancel Button, OK Button, Current Font Sample, Font Sampler Area, Point Size Slider, Current Font Selection.

## **13.2 FONT MENU**

The Font menu displays the fonts available for your platform (e.g., Helvetica, Courier). Selecting one of these fonts will update the Current Font Sample and Current Font Selection areas of the window.

## **13.3 STYLE MENU**

The Style menu allows you to specify attributes (e.g., Bold, Italic) for the selected font. As you add and remove attributes, the Current Font Sample and the Current Font Selection will reflect the changes.

## **13.4 CANCEL BUTTON**

The Cancel button closes the font selection window without changing the fonts of the window.

## **13.5 OK BUTTON**

The OK button closes the font selection window and sets the font of the window to the font listed in the Current Font Selection area.

## 13.6 CURRENT FONT SAMPLE

This area of the font selection window shows a sample of text in font, style, and size you have currently selected.

If you want to see what your current selection does to particular characters, you may type into this area. Note that your additions will be deleted when you change any aspect of the font (style, size, font).

## 13.7 FONT SAMPLER AREA

This area of the font selection window shows you a sample of the fonts available for your platform. You may make one of the listed fonts the current selection by clicking the font with the left mouse button. The currently selected styles and sizes remain in effect.

## 13.8 POINT SIZE SLIDER

This slider lets you choose the point size of the font. The text displayed in the Current Font Sample updates immediately.

## 13.9 CURRENT FONT SELECTION

This area displays the Tcl name for the font (including the size and style(s)) that you have currently selected. You may type in this area, but doing so will have no effect on the font.

## 13.10 SETTING THE DEFAULT FONT

To have ASCEND use the same font each time you run it, you need to do the following steps.

1. Use the font selection dialog to choose a font you like. Make a note of the Tcl name for the font; this name is displayed in the Current Font Selection area of the window.
2. Open the system utilities window and make a note of the value of ASCENDDIST.
3. Exit ASCEND.
4. Under the ASCENDDIST directory, there should be a directory

called TK, and in this directory a file called `ascend.ad`. Copy this file to your HOME directory<sup>1</sup> and name it `.ascend.ad`<sup>2</sup>.

5. Add the following lines at the end of `.ascend.ad`, replacing `courier 11 normal` with the font you noted in Step 1.

```
Global font courier 11 normal
Global labelfont courier 11 normal
Toolbox font courier 11 normal
Library font courier 11 normal
Display font courier 11 normal
Browser font courier 11 normal
Probe font courier 11 normal
Units font courier 11 normal
Script font courier 11 normal
Solver font courier 11 normal
Debugger font courier 11 normal
```

6. Save `.ascend.ad`, and restart ASCEND.

Note that this file also contains the default size and position for most ASCEND windows. To change the position or size of a window, edit the lines containing geometry; the format for the geometry is `WWxHH+xx+yy` where *WW* is the width of the window, *HH* is its height, *xx* is the distance between the left edge of the screen and the left edge of the window, and *yy* is the distance between the screen's top edge and the window's top edge.

---

1. To set your HOME directory under Windows, open the Control Panel, double click the System icon, select the Environment tab, and set the HOME environment variable to a directory you want to consider "home".
2. Under Windows, the name `_ascend.ad` also works.

# CHAPTER 14 THE PRINT DIALOG

## 14.1 OVERVIEW

The [print dialog shown in Figure 14-1](#) allows the user to modify the settings which control the printing of information from within ASCEND.

## 14.2 SETTINGS

### 14.2.1 DESTINATION

This is a pop-up menu that allows you to select one of the following options for printing: Print, Write to file, Append to file, Enscript, or Custom.

Destination	Print
Printer	mirage
Name of file	Noname
Enscript flags	-2rG
User print command	>> /dev/null cat
OK Help Cancel	

Figure 14-1 The print dialog.



### 14.2.1.1 PRINT

On UNIX machines, this option sends the window's contents to the printer specified in the Printer field (*PRINTER*). Under SystemV systems<sup>1</sup>, the command

```
lp -dPRINTER
```

is used as the interface to the printer; on all other UNIX systems, the command

```
lpr -PPRINTER
```

is used.

Under Windows, the command

```
notepad /p
```

is used to send the window's contents to the user's default printer. This will not print the Matrix display and other graphic displays correctly because those generate PostScript. Text windows (Display, Probe, Script, Library) will print correctly.

### 14.2.1.2 WRITE TO FILE

Under UNIX, this option writes the contents of the window to the file listed in the Name of file field. If a file with the same name exists, ASCEND will overwrite the file after verifying that the user wants to overwrite the file.

This option is not available on the Windows platform.

This option is another version of Save As and will likely go away in future releases of ASCEND.

### 14.2.1.3 APPEND TO FILE

Under UNIX, this option appends the contents of the window to the file listed in the Name of file field. If the file does not exist, it will be created.

This option is not available on the Windows platform.

This option will likely go away in future releases of ASCEND.

### 14.2.1.4 ENSCRIPT

On UNIX, this option uses the `enscript` program to queue the window's contents to the printer specified in the Printer field

---

1. HP-UX, SGI IRIX, and Solaris 2.x.

(*PRINTER*). On SystemV systems, the command `enscript -dPRINTER enscript-flags` is used; on all other UNIX systems, the command `enscript -PPRINTER enscript-flags` is used. The value of `enscript-flags` is the value specified in the Enscript flags field.

This option is not available on the Windows platform.

#### **14.2.1.5 CUSTOM**

This option allows the user to specify a custom print command. The user should type their custom command in the User print command field; the command should accept a file name as its final argument.

This option is not available on the Windows platform.

### **14.2.2 PRINTER**

Under UNIX, this field specifies the printer to send the document to when the Destination is Print or Enscript.

This field has no effect under Windows.

### **14.2.3 NAME OF FILE**

Under UNIX, this field contains the name of the file used by the Write to file and Append to file options.

This field has no effect under Windows.

### **14.2.4 ENSCRYPT FLAGS**

Under UNIX, this field contains the options sent to the `enscript` program when the Destination is Enscript.

This field has no effect under Windows.

### **14.2.5 USER PRINT COMMAND**

Under UNIX, this field contains the command used to “print” the window’s contents when the Destination is Custom. This command should accept the name of a file (a temporary file containing the contents of the window) as its final argument.

This field has no effect under Windows.

## **14.3 BUTTONS**

### **14.3.1 OK**

Pressing this button accepts the settings, sends the document to the printer or to the specified file, and closes the print dialog. The values displayed in the Text print command and the PRINTER fields in the system utilities window will change to reflect the new settings.

### **14.3.2 HELP**

Pressing this button should cause your web browser to display this document.

### **14.3.3 CANCEL**

This button ignores any changes you may have made to the settings and closes the print dialog. The file is not printed.

# CHAPTER 15 SOLVED SIMPLE MODELING PROBLEMS WITH ASCEND

In this chapter we present two simple modeling problems for which we then show you our ASCEND models for solving them. Modeling is a matter of style, and we will start to show you what we believe to be good styles for modeling. We assume you have not used the ASCEND system before. These problems are very generic and should be readily followed by anyone with a modest technical background.

One purpose is to show you some of the different ways you can use ASCEND. Specifically we want to show you that you can use ASCEND to setup and solve the simple types of problems that you might have solved using a spreadsheeting program. Indeed, we use ASCEND to solve homework problems quite often. When you factor in the powerful debugging tools, you might find it faster to use ASCEND, especially as the models get more complex. And no one would want (we think) to solve a 20,000 simultaneous nonlinear equation model using a spreadsheeting program.

A major advantage of using ASCEND is that once you have written, debugged and learned to solve such a model, you can interactively alter the "fixed" flags for the variables, changing which variables are to be fixed and which to be calculated. You can then immediately solve or optimize the new problem, using the previously solved problem as the initial guess.

## 15.1 ROOTS OF A POLYNOMIAL

In this problem you wish to find the roots of a polynomial. Assume you do not wish to keep the code. You could readily use a spreadsheet

program with its "root finder" routine to solve this type of problem, but you can as readily use ASCEND.

### 15.1.1 PROBLEM STATEMENT

Numerically compute the roots of  $(x-1)(x-5)(x+7)(x^2+1)=0$ . (Given in this form the roots are obviously 1, 5, and -7. Two roots are complex, and ASCEND will not find them.)

### 15.1.2 ANSWER

You can find the roots by guessing initial points after typing in, loading and compiling the following model. You would use any text editor to enter this model into the computer. If you use a "WYSIWYG" (what you see is what you get) editor such as Word or Framemaker, be sure to save the file as a **text only** file. If possible, use a simpler text editor.

```
MODEL polynomial_roots;                                1
    x          IS_A generic_real;                      2
    (x-1)*(x-5)*(x+7)*(x^2+1) = 0;                     3
END polynomial_roots;                                  4
```

This simple model is a stand-alone model. You need no other predefined library models to support it. Load and compile an instance of this model (using tools in the LIBRARY tool set), browse it (using the BROWSER tool set) to see if it appears to have compiled correctly, and then pass it to the SOLVER tool set.

This model involves a single equation in the single unknown variable,  $x$ . The ASCEND solver treats a single equation in one unknown in a special manner when asked to solve it. The solver first attempts to rearrange the equation by simple algebraic manipulations to isolate the unknown on the left hand side of the equation in the form  $x =$  expression not involving  $x$ . In this form, solving is simply evaluating the expression on the right hand side once. Here the solver would fail as there is no way to isolate  $x$  on the left hand side as the equation is a fifth order polynomial in  $x$ . When rearrangement fails, the solver uses bisection to locate a root in the range between the lower and upper bound on the variable. You can see the bounds,  $x.lower$  and  $x.upper$ , using the BROWSER. The default values for these bounds are plus and minus  $10^{20}$  respectively, which gives a very large range in which to

look for the root. You should change the bounds<sup>1</sup> to more realistic ones. Selecting bounds to be -10 to 0 and then solving will find the root  $x=-7$ . Selecting other values will find the other roots.

This model illustrates that you can quickly set up and solve simple problems using ASCEND. Note that you would have been required to place bounds on  $x$  had you used a goal seeking tool in a spreadsheeting program if you wanted to control which root to locate.

## 15.2 NUMERICAL INTEGRATION OF TABULAR DATA

This problem is similar to the previous one in that it is very easy to set up and solve. It adds in the notion of units (e.g., ft, m, hr, atm) which ASCEND handles in a straight-forward manner, relieving modelers from thinking about converting among the many units they might use when expressing the data for a problem.

Again we are talking about producing throw-away code. All we are really concerned with here is the answer which we intend to put into a report. We are using ASCEND as a "calculator."

### 15.2.1 PROBLEM STATEMENT

Given the following velocity data vs. time, estimate numerically the distance one has traveled between time equal to zero and 100 seconds.

**Table 1: Velocity data to be integrated**

data point number	time, s	velocity, ft/min
1	0	100
2	10	120
3	20	130
4	30	135
5	40	140
6	50	160
7	60	180

- 
1. To set the value for a variable interactively, select the variable when it is display in the right window or in the lower window with the RIGHT mouse button (the other button). A window for changing its value opens.

**Table 1: Velocity data to be integrated**

data point number	time, s	velocity, ft/min
8	70	210
9	80	240
10	90	220
11	100	200

**15.2.2 ANSWER**

(This example will be solved using variables whose types are defined in the file *atoms.a4l*. You must load this file first before loading the file with the code below, else you will experience a number of diagnostic messages indicating missing type definitions. See Chapter 18 for a discussion of libraries on ASCEND.)

The distance traveled is the integral of the velocity over time. We can use Simpson's rule to carry out this integration for evenly spaced points.

$$d = ((v[1] + 4 v[2] + v[3]) + (v[3] + 4 v[4] + v[5]) + \dots + (v[N-2] + 4 v[N-1] + v[N])) * \Delta t / 6 \quad (15.1)$$

where  $d$  is the distance covered when traveling at the velocities,  $V[k]$ , listed. This formula requires there to be an odd number of 3 or more evenly spaced data points, which is fine here as we have eleven velocity points evenly spaced in time. (If there had been an even number of points, we could use Simpson's rule for all but the last time interval and use a simple trapezoidal rule to integrate it.)

An ASCEND model to evaluate this distance is as follows. The types definitions for *speed*, *time* and *distance* are in the file *atoms.a4l*.

```

MODEL travel_distance;                                1
    kmax                IS_A integer_constant;        2
    v[1..2*kmax+1]      IS_A speed;                   3
    delta_time           IS_A time;                    4
    d                    IS_A distance;                 5
                                                            6

```

```

        d = SUM[v[2*k-1]+4*v[2*k]+v[2*k+1] SUCH_THAT k IN
              [1..kmax]]*delta_time/6;
END travel_distance;

MODEL test_travel_distance REFINES travel_distance;
    kmax      := 5;

METHODS

    METHOD specify;
        v[1..2*kmax+1].fixed := TRUE;
        delta_time.fixed     := TRUE;
    END specify;

    METHOD values;
        v[1]      := 100 {ft/min};
        v[2]      := 120 {ft/min};
        v[3]      := 130 {ft/min};
        v[4]      := 135 {ft/min};
        v[5]      := 140 {ft/min};
        v[6]      := 160 {ft/min};
        v[7]      := 180 {ft/min};
        v[8]      := 210 {ft/min};
        v[9]      := 240 {ft/min};
        v[10]     := 220 {ft/min};
        v[11]     := 200 {ft/min};
        delta_time := 10 {s};
    END values;
END test_travel_distance;

```

If you look carefully at this model, you will note that we did NOT account for the conversion factors required because velocities are in ft/min while the time increment is in seconds. ASCEND understands these units and makes all the needed conversions. When you run this model, you can ask for the distance to be displayed to you in any supported length units you would prefer (e.g., ft, mile, m, cm, angstroms, lightyears). The distance traveled, when reported using SI units, is 42.84 m.





# CHAPTER 16 A CONDITIONAL MODELING EXAMPLE: REPRESENTING A SUPERSTRUCTURE

To give an example of the application of the conditional modeling tool in ASCEND -the WHEN statement-, we developed a simplified model for the superstructure given in Figure 16-1. The code listed below exists in a file in the ASCEND models subdirectory entitled *when\_demo.a4c*. You could run this example by loading this file and using it and its corresponding script *when\_demo.a4s*.

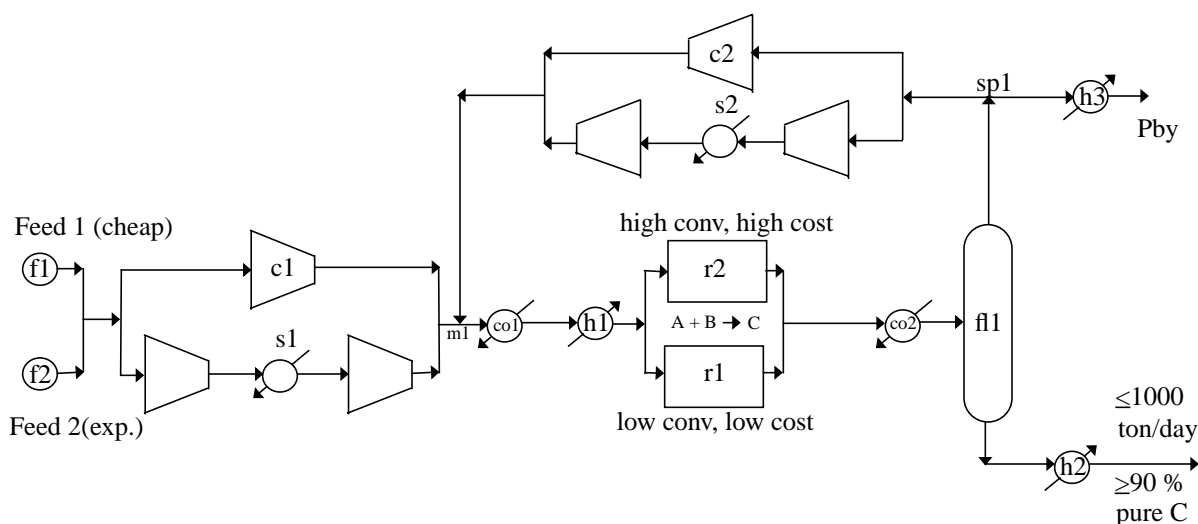


Figure 16-1 Superstructure used in the example of the application of the when statement

## 16.1 THE WHEN STATEMENT

Before showing the example, we want to start by giving a brief explanation about the semantics of the WHEN statement, a tool which allows ASCEND to represent conditional models efficiently.

In the WHEN statement, we take advantage of the fact that ASCEND is based on object oriented concepts where model definitions can contain parts that contain parts to any level. Furthermore, in ASCEND, a simple

relation is treated as an object by itself and can have a name. Based on these ideas, the syntax for the WHEN statement is:

```

WHEN (list_of_variables)
  CASE list_of_values_1:
    USE name_of_equation_1;
    USE name_of_model_1;
  CASE list_of_values_2:
    USE name_of_equation_2;
    USE name_of_model_2;
  CASE list_of_values_nminus1:
    USE name_of_equation_nminus1;
    USE name_of_model_nminus1;
  OTHERWISE:
    USE name_of_equation_n;
    USE name_of_model_n;
END;
```

The following are important observations about the implementation:

- 1 The WHEN statement does not mean conditional compilation. We create and have available the data structures for all of the variables and equations in each of the models. This is actually a requirement for the solution algorithms of conditional models. All the models and equations whose name is given in each of the cases should be declared inside the model which contains the WHEN statement.
- 2 The variables in the list of variables can be of any type among boolean, integer or symbol or any combination of them. That is, we are not limited to the use of boolean variables. Obviously, The list of values in each case must be in agreement with the list of variables in the number of elements and type of each element. In other words, order matters in the list of variables of the WHEN statement, and parentheses are enclosing this list to make clear such a feature.
- 3 Names of arrays of models or equations are also allowed inside the scope of each CASE.

The WHEN statement represents an important contribution to modeling: it allows the user to define the domain of validity of both *models* and *equations* inside the cases of a WHEN statement. This feature enormously increases the scope of modeling in an equation based modeling environment.

Mainly, there are two different ways in which the WHEN statement can be used.:

- First, the WHEN statement can be used to select a configuration of a problem among several alternative configurations.
- Second, in combination with logical relations, the WHEN statement can be used for conditional programming. That is, a problem in which the system of equations to be solved depends on the solution of the problem. A typical example of this situation is the laminar-turbulent flow transition. The selection of the equation to calculate the friction factor depends on the value of the Reynolds number, which is an unknown in the problem.

## 16.2 THE PROBLEM DESCRIPTION

In the example, there are two alternative feedstocks, two possible choices of the reactor and two choices for each of the compression systems. The user has to make 4 decisions (for example, using either the cheap feed or the expensive feed), therefore, there are  $2^4 = 16$  feasible configurations of the problem. All these 16 configurations are encapsulated in one ASCEND model containing 4 WHEN statements which depend on the value of 4 boolean variables.

The value of the four boolean variables will determine the structure of the problem to be solved. In this example, those values are defined by the modeler, but they also could be defined by some logic inference algorithm which would allow the automatic change of the structure of the problem.

The following section gives the code for this model. The first models correspond to the different types of unit operations existing in the superstructure. Those model are very simplified. You may want to skip them and analyze only the model *flowsheet*, in which the use and syntax of the WHEN statement as well as the configuration of the superstructure become evident.

## 16.3 THE CODE

As the code is in our ASCEND examples subdirectory, it has header information that we required of all such files included as one large comment extending over several lines. Comments are in the form (\* comment \*). The last item in this header information is a list of the files one must load before loading this one, i.e., *system.a4l* and *atoms.a4l*.

```

REQUIRE "atoms.a4l";                                     35
(* --> measures,system *)                                36
PROVIDE "when_demo.a4c";                                  37
( *****\                                              38
    when_demo.a4c                                         39
    by Vicente Rico-Ramirez                               40
    Part of the Ascend Library                             41
                                                         42

This file is part of the Ascend modeling library.         43
                                                         44

The Ascend modeling library is free software; you can redistribute 45
it and/or modify it under the terms of the GNU General Public License as 46
published by the Free Software Foundation; either version 2 of the      47
License, or (at your option) any later version.           48
                                                         49

The Ascend Language Interpreter is distributed in hope that it will be 50
useful, but WITHOUT ANY WARRANTY; without even the implied warranty of 51
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU       52
General Public License for more details.                   53
                                                         54

You should have received a copy of the GNU General Public License along with55
the program; if not, write to the Free Software Foundation, Inc., 675    56
Mass Ave, Cambridge, MA 02139 USA.  Check the file named COPYING.       57
                                                         58

Use of this module is demonstrated by the associated script file        59
when_demo.a4s.                                                         60
\ *****\                                              61
                                                         62
( *****\                                              63
    $Date: 1998/05/14 21:39:44 $                                     64
    $Revision: 1.5 $                                                 65
    $Author: rv2a $                                                  66
    $Source: /afs/cs.cmu.edu/project/ascend/Repository/models/when_demo.a4c,v $67
\ *****\                                              68
                                                         69
( *                                                                    70
    This model is intended to demonstrate the degree of flexibility      71
    that the use of conditional statements -when statement- provides      72
    to the representation of superstructures. We hope that this          73
    application will become clear by looking at the MODEL flowsheet,     74
    in which the existence/nonexistence of some of the unit operations    75
    is represented by when statements. A particular combination of       76
    user defined boolean variables -see method values, configuration2,    77
    configuration3- will a define a particular configuration of the       78
    problem.                                                            79
                                                         80

```

```

This model requires:
    "system.a4l"
    "atoms.a4l"
*)

(* ***** *)

MODEL mixture;

    components          IS_A set OF symbol_constant;
    Cpi[components]     IS_A heat_capacity;
    y[components]       IS_A fraction;
    P                   IS_A pressure;
    T                   IS_A temperature;
    Cp                  IS_A heat_capacity;

    SUM[y[i] | i IN components] = 1.0;
    Cp = SUM[Cpi[i] * y[i] | i IN components];

METHODS

    METHOD default_self;
    END default_self;

    METHOD specify;
        Cpi[components].fixed := TRUE;
        P.fixed := TRUE;
        T.fixed := TRUE;
        y[components].fixed := TRUE;
        y[CHOICE[components]].fixed := FALSE;
    END specify;

END mixture;

(* ***** *)

MODEL molar_stream;

    state          IS_A mixture;
    Ftot,f[components] IS_A molar_rate;
    components     IS_A set OF symbol_constant;
    P              IS_A pressure;
    T              IS_A temperature;
    Cp             IS_A heat_capacity;

```

```

components, state.components      ARE_THE_SAME;      127
P, state.P                        ARE_THE_SAME;      128
T, state.T                        ARE_THE_SAME;      129
Cp, state.Cp                      ARE_THE_SAME;      130
                                   131
FOR i IN components CREATE        132
    f_def[i]: f[i] = Ftot*state.y[i]; 133
END FOR;                          134
                                   135
METHODS                           136
                                   137
METHOD default_self;              138
END default_self;                 139
                                   140
METHOD specify;                   141
    RUN state.specify;            142
    state.y[components].fixed := FALSE; 143
    f[components].fixed := TRUE;    144
END specify;                      145
                                   146
END molar_stream;                 147
                                   148
( * ***** *)                   149
                                   150
                                   151
MODEL cheap_feed;                 152
    stream          IS_A molar_stream; 153
    cost_factor      IS_A cost_per_mole; 154
    cost             IS_A cost_per_time; 155
                                   156
    stream.f['A'] = 0.060 {kg_mole/s}; 157
    stream.f['B'] = 0.025 {kg_mole/s}; 158
    stream.f['D'] = 0.015 {kg_mole/s}; 159
    stream.f['C'] = 0.00 {kg_mole/s}; 160
    stream.T = 300 {K};            161
    stream.P = 5 {bar};            162
                                   163
    cost = cost_factor * stream.Ftot; 164
METHODS                           165
                                   166
METHOD default_self;              167
END default_self;                 168
                                   169
METHOD specify;                   170
    RUN stream.specify;           171
    stream.f[stream.components].fixed := FALSE; 172

```

```

        cost_factor.fixed := TRUE;
        stream.T.fixed := FALSE;
        stream.P.fixed := FALSE;
    END specify;

END cheap_feed;

( * ***** *)

MODEL expensive_feed;
    stream          IS_A molar_stream;
    cost_factor      IS_A cost_per_mole;
    cost             IS_A cost_per_time;

    stream.f['A'] = 0.065 {kg_mole/s};
    stream.f['B'] = 0.030 {kg_mole/s};
    stream.f['D'] = 0.05  {kg_mole/s};
    stream.f['C'] = 0.00  {kg_mole/s};
    stream.T = 320 {K};
    stream.P = 6 {bar};

    cost = 3 * cost_factor * stream.Ftot;

METHODS

    METHOD default_self;
    END default_self;

    METHOD specify;
        RUN stream.specify;
        stream.f[stream.components].fixed := FALSE;
        cost_factor.fixed := TRUE;
        stream.T.fixed := FALSE;
        stream.P.fixed := FALSE;
    END specify;

END expensive_feed;

( * ***** *)

MODEL heater;
    input,output      IS_A molar_stream;
    heat_supplied      IS_A energy_rate;
    components         IS_A set OF symbol_constant;

```



```

cost                IS_A cost_per_time;                219
cost_factor         IS_A cost_per_energy;              220
                                                            221
components,input.components,output.components ARE_THE_SAME; 222
FOR i IN components CREATE                                223
    input.state.Cpi[i], output.state.Cpi[i] ARE_THE_SAME; 224
END FOR;                                                  225
                                                            226
FOR i IN components CREATE                                227
    input.f[i] = output.f[i];                             228
END FOR;                                                  229
                                                            230
input.P = output.P;                                       231
                                                            232
heat_supplied = input.Cp *(output.T - input.T) * input.Ftot; 233
                                                            234
cost = cost_factor * heat_supplied;                      235
                                                            236
METHODS                                                    237
                                                            238
METHOD default_self;                                       239
END default_self;                                         240
                                                            241
METHOD specify;                                           242
    RUN input.specify;                                     243
    cost_factor.fixed := TRUE;                             244
    heat_supplied.fixed := TRUE;                           245
END specify;                                              246
                                                            247
METHOD seqmod;                                            248
    cost_factor.fixed := TRUE;                             249
    heat_supplied.fixed := TRUE;                           250
END seqmod;                                              251
                                                            252
END heater;                                              253
                                                            254
                                                            255
( * ***** *)                                           256
                                                            257
MODEL cooler;                                             258
                                                            259
input,output        IS_A molar_stream;                   260
heat_removed        IS_A energy_rate;                    261
components          IS_A set OF symbol_constant;        262
cost                IS_A cost_per_time;                  263
cost_factor         IS_A cost_per_energy;                264

```

```

components,input.components,output.components      ARE_THE_SAME;      265
FOR i IN components CREATE                          266
    input.state.Cpi[i],output.state.Cpi[i]          ARE_THE_SAME;      267
END FOR;                                             268
                                                    269
                                                    270
FOR i IN components CREATE                          271
    input.f[i] = output.f[i];                       272
END FOR;                                             273
                                                    274
input.P = output.P;                                275
heat_removed = input.Cp *(input.T - output.T) * input.Ftot; 276
cost = cost_factor * heat_removed;                  277
                                                    278
METHODS                                             279
                                                    280
METHOD default_self;                                281
END default_self;                                  282
                                                    283
METHOD specify;                                     284
    RUN input.specify;                              285
    cost_factor.fixed := TRUE;                      286
    heat_removed.fixed := TRUE;                    287
END specify;                                       288
                                                    289
METHOD seqmod;                                      290
    cost_factor.fixed := TRUE;                      291
    heat_removed.fixed := TRUE;                    292
END seqmod;                                       293
                                                    294
END cooler;                                       295
                                                    296
(* ***** *)                                     297
                                                    298
MODEL single_compressor; (* Adiabatic Compression *) 299
                                                    300
    input,output      IS_A molar_stream;           301
    components        IS_A set OF symbol_constant; 302
    work_supplied     IS_A energy_rate;             303
    pressure_rate     IS_A factor;                  304
    R                 IS_A gas_constant;            305
    cost              IS_A cost_per_time;           306
    cost_factor       IS_A cost_per_energy;         307
                                                    308
    components,input.components,output.components    ARE_THE_SAME; 309
    FOR i IN components CREATE                      310

```

```

    input.state.Cpi[i],output.state.Cpi[i]          ARE_THE_SAME;          311
END FOR;                                           312
                                                    313
FOR i IN components CREATE                        314
    input.f[i] = output.f[i];                    315
END FOR;                                           316
                                                    317
pressure_rate = output.P / input.P;              318
                                                    319
output.T = input.T * (pressure_rate ^ (R/input.Cp) ); 320
                                                    321
work_supplied = input.Ftot * input.Cp * (output.T - input.T); 322
                                                    323
cost = cost_factor * work_supplied;              324
                                                    325
METHODS                                           326
                                                    327
METHOD default_self;                             328
END default_self;                                329
                                                    330
METHOD specify;                                  331
    RUN input.specify;                           332
    cost_factor.fixed := TRUE;                    333
    pressure_rate.fixed := TRUE;                  334
END specify;                                     335
                                                    336
METHOD seqmod;                                   337
    cost_factor.fixed := TRUE;                    338
    pressure_rate.fixed := TRUE;                  339
END seqmod;                                     340
                                                    341
END single_compressor;                          342
                                                    343
( * ***** *)                                344
                                                    345
MODEL staged_compressor;                        346
                                                    347
    input,output                                IS_A molar_stream;          348
    components                                IS_A set OF symbol_constant; 349
    work_supplied                            IS_A energy_rate;            350
    heat_removed                             IS_A energy_rate;            351
    T_middle                                IS_A temperature;              352
    n_stages                                IS_A factor;                   353
    pressure_rate                            IS_A factor;                   354
    stage_pressure_rate                      IS_A factor;                   355
    R                                         IS_A gas_constant;            356

```

```

cost                IS_A cost_per_time;          357
cost_factor_work    IS_A cost_per_energy;        358
cost_factor_heat    IS_A cost_per_energy;        359
                                                         360
components,input.components,output.components    ARE_THE_SAME; 361
FOR i IN components CREATE                          362
    input.state.Cpi[i],output.state.Cpi[i]        ARE_THE_SAME; 363
END FOR;                                           364
                                                         365
FOR i IN components CREATE                          366
    input.f[i] = output.f[i];                     367
END FOR;                                           368
                                                         369
output.T = input.T;                               370
                                                         371
pressure_rate = output.P / input.P;               372
                                                         373
stage_pressure_rate =(pressure_rate)^(1.0/n_stages); 374
                                                         375
T_middle = input.T * (stage_pressure_rate ^ (R/input.Cp)); 376
                                                         377
work_supplied = input.Ftot * n_stages * input.Cp * 378
    (T_middle - input.T);                          379
                                                         380
heat_removed = input.Ftot * (n_stages - 1.0) *      381
    input.Cp * (T_middle - input.T);               382
                                                         383
cost = cost_factor_work * work_supplied +          384
    cost_factor_heat * heat_removed;               385
                                                         386
METHODS                                           387
                                                         388
METHOD default_self;                             389
END default_self;                                390
                                                         391
METHOD specify;                                  392
    RUN input.specify;                            393
    n_stages.fixed := TRUE;                        394
    cost_factor_heat.fixed := TRUE;                395
    cost_factor_work.fixed := TRUE;                396
    pressure_rate.fixed := TRUE;                   397
END specify;                                     398
                                                         399
METHOD seqmod;                                   400
    n_stages.fixed := TRUE;                        401
    cost_factor_heat.fixed := TRUE;                402

```

```

        cost_factor_work.fixed := TRUE;           403
        pressure_rate.fixed := TRUE;             404
    END seqmod;                                   405
                                                406
END staged_compressor;                          407
                                                408

( * ***** *)                                409
                                                410

MODEL mixer;                                    411
                                                412

    components          IS_A set OF symbol_constant; 413
    n_inputs            IS_A integer_constant;        414
    feed[1..n_inputs], out IS_A molar_stream;         415
    To                  IS_A temperature;             416
                                                417

    components,feed[1..n_inputs].components,        418
    out.components      ARE_THE_SAME;                419
    FOR i IN components CREATE                      420
        feed[1..n_inputs].state.Cpi[i],out.state.Cpi[i] ARE_THE_SAME; 421
    END FOR;                                         422
                                                423

    FOR i IN components CREATE                      424
        cmb[i]: out.f[i] = SUM[feed[1..n_inputs].f[i]]; 425
    END FOR;                                         426
                                                427

    SUM[(feed[i].Cp *feed[i].Ftot * (feed[i].T - To))|i IN [1..n_inputs]]= 428
        out.Cp *out.Ftot * (out.T - To);           429
                                                430

    SUM[( feed[i].Ftot * feed[i].T / feed[i].P )|i IN [1..n_inputs]] = 431
        out.Ftot * out.T / out.P;                   432
                                                433

METHODS                                           434
                                                435

    METHOD default_self;                           436
    END default_self;                              437
                                                438

    METHOD specify;                                439
        To.fixed := TRUE;                          440
        RUN feed[1..n_inputs].specify;              441
    END specify;                                   442
                                                443

    METHOD seqmod;                                  444
        To.fixed := TRUE;                          445
    END seqmod;                                    446
                                                447

END mixer;                                        448

```

```

449
450 (* ***** *)
451
452 MODEL splitter;
453
454     components          IS_A set OF symbol_constant;
455     n_outputs           IS_A integer_constant;
456     feed, out[1..n_outputs] IS_A molar_stream;
457     split[1..n_outputs] IS_A fraction;
458
459     components, feed.components,
460         out[1..n_outputs].components ARE_THE_SAME;
461     feed.state,
462     out[1..n_outputs].state ARE_THE_SAME;
463
464     FOR j IN [1..n_outputs] CREATE
465         out[j].Ftot = split[j]*feed.Ftot;
466     END FOR;
467
468     SUM[split[1..n_outputs]] = 1.0;
469
470 METHODS
471
472     METHOD default_self;
473     END default_self;
474
475     METHOD specify;
476         RUN feed.specify;
477         split[1..n_outputs-1].fixed:=TRUE;
478     END specify;
479
480     METHOD seqmod;
481         split[1..n_outputs-1].fixed:=TRUE;
482     END seqmod;
483
484 END splitter;
485
486
487 (* ***** *)
488
489 MODEL cheap_reactor;
490
491     components          IS_A set OF symbol_constant;
492     input, output       IS_A molar_stream;
493     low_turnover        IS_A molar_rate;
494     stoich_coef[input.components] IS_A factor;
495     cost_factor         IS_A cost_per_mole;

```

```

cost                                IS_A cost_per_time;                495
                                                                           496
components,input.components, output.components    ARE_THE_SAME;        497
FOR i IN components CREATE                498
    input.state.Cpi[i], output.state.Cpi[i]        ARE_THE_SAME;    499
END FOR;                                     500
                                           501
FOR i IN components CREATE                502
    output.f[i] = input.f[i] + stoich_coef[i]*low_turnover;        503
END FOR;                                     504
                                           505
input.T = output.T;                        506
(* ideal gas constant volume *)            507
input.Ftot * input.T / input.P = output.Ftot * output.T/output.P; 508
                                           509
cost = cost_factor * low_turnover;         510
                                           511
METHODS                                     512
                                           513
METHOD default_self;                       514
END default_self;                          515
                                           516
METHOD specify;                            517
    RUN input.specify;                     518
    low_turnover.fixed:= TRUE;              519
    stoich_coef[input.components].fixed:= TRUE; 520
    cost_factor.fixed := TRUE;              521
END specify;                               522
                                           523
METHOD seqmod;                             524
    low_turnover.fixed:= TRUE;              525
    stoich_coef[input.components].fixed:= TRUE; 526
    cost_factor.fixed := TRUE;              527
END seqmod;                               528
                                           529
END cheap_reactor;                         530
                                           531
                                           532
(* ***** *)                            533
                                           534
MODEL expensive_reactor;                   535
                                           536
components                                IS_A set OF symbol_constant; 537
input, output                            IS_A molar_stream;          538
high_turnover                            IS_A molar_rate;             539
stoich_coef[input.components]            IS_A factor;                 540

```

```

cost_factor          IS_A cost_per_mole;          541
cost                 IS_A cost_per_time;          542
                                                            543
components,input.components, output.components    ARE_THE_SAME;  544
FOR i IN components CREATE                          545
    input.state.Cpi[i], output.state.Cpi[i]        ARE_THE_SAME;  546
END FOR;                                           547
                                                            548
FOR i IN components CREATE                          549
    output.f[i] = input.f[i] + stoich_coef[i]*high_turnover;      550
END FOR;                                           551
                                                            552

input.T = output.T;                                553
(* ideal gas constant volume *)                    554
input.Ftot * input.T / input.P = output.Ftot * output.T/output.P; 555
                                                            556

cost = cost_factor * high_turnover;                557
                                                            558

METHODS                                             559
                                                            560

METHOD default_self;                               561
END default_self;                                  562
                                                            563

METHOD specify;                                    564
    RUN input.specify;                             565
    high_turnover.fixed:= TRUE;                     566
    stoich_coef[input.components].fixed:= TRUE;     567
    cost_factor.fixed := TRUE;                      568
END specify;                                       569
                                                            570

METHOD seqmod;                                     571
    high_turnover.fixed:= TRUE;                     572
    stoich_coef[input.components].fixed:= TRUE;     573
    cost_factor.fixed := TRUE;                      574
END seqmod;                                       575
                                                            576

END expensive_reactor;                             577
                                                            578

(* ***** *)                                     579
                                                            580

MODEL flash;                                       581
                                                            582

    components          IS_A set OF symbol_constant;  583
    feed,vap,liq        IS_A molar_stream;           584
    alpha[feed.components] IS_A factor;              585
    ave_alpha           IS_A factor;                 586

```



```

vap_to_feed_ratio      IS_A fraction;                                587
                                                                    588
components,feed.components,                                       589
vap.components,                                                 590
liq.components          ARE_THE_SAME;                             591
FOR i IN components CREATE                                       592
    feed.state.Cpi[i],                                           593
    vap.state.Cpi[i],                                           594
    liq.state.Cpi[i]      ARE_THE_SAME;                           595
END FOR;                                                         596
                                                                    597
vap_to_feed_ratio*feed.Ftot = vap.Ftot;                          598
                                                                    599
FOR i IN components CREATE                                       600
    cmb[i]: feed.f[i] = vap.f[i] + liq.f[i];                     601
    eq[i]:  vap.state.y[i]*ave_alpha = alpha[i]*liq.state.y[i];  602
END FOR;                                                         603
                                                                    604
feed.T = vap.T;                                                  605
feed.T = liq.T;                                                  606
feed.P = vap.P;                                                  607
feed.P = liq.P;                                                  608
                                                                    609
METHODS                                                         610
                                                                    611
METHOD default_self;                                             612
END default_self;                                               613
                                                                    614
METHOD specify;                                                 615
    RUN feed.specify;                                           616
    alpha[feed.components].fixed:=TRUE;                         617
    vap_to_feed_ratio.fixed:= TRUE;                             618
END specify;                                                    619
                                                                    620
METHOD seqmod;                                                  621
    alpha[feed.components].fixed:=TRUE;                         622
    vap_to_feed_ratio.fixed:= TRUE;                             623
END seqmod;                                                     624
                                                                    625
END flash;                                                      626
                                                                    627
( * ***** *)                                                628
                                                                    629

```

Next, the model *flowsheet* is presented. This model represents one of the applications of the WHEN statement. Namely, selecting among

alternative configurations of the problem. Note that in each of the WHEN statements we define the conditional existence of complete ASCEND models. A specific combination for each of the conditional variables -boolean\_vars in the example- will define a specific configuration of the problem. Once a configuration has been selected, it will be kept until the user decides to change it. Note that the user does not have to recompile the model to switch among alternative configurations. The reconfiguration of the system can be done automatically by simply changing the values of the conditional variables. An obvious application of this would be the synthesis of process networks. While running the script *when\_demo.a4s*, note the changes in the number of active equations, active variables and fixed variables for the different configurations. For example, the configuration defined by one of the feeds, two single-stage compressors and one of the reactors contains 169 active equations.

```
( * ***** *) 630
631
MODEL flowsheet; 632
633
(* units *) 634
635
f1      IS_A cheap_feed; 636
f2      IS_A expensive_feed; 637
638
c1      IS_A single_compressor; 639
s1      IS_A staged_compressor; 640
641
c2      IS_A single_compressor; 642
s2      IS_A staged_compressor; 643
644
r1      IS_A cheap_reactor; 645
r2      IS_A expensive_reactor; 646
647
col,co2 IS_A cooler; 648
h1,h2,h3 IS_A heater; 649
f11     IS_A flash; 650
sp1     IS_A splitter; 651
m1      IS_A mixer; 652
653
(* boolean variables *) 654
655
select_feed1      IS_A boolean_var; 656
select_single1    IS_A boolean_var; 657
select_cheapr1    IS_A boolean_var; 658
select_single2    IS_A boolean_var; 659
```

```

660
(* define sets *) 661
662
m1.n_inputs :=2; 663
sp1.n_outputs := 2; 664
665
(* wire up flowsheet *) 666
667
f1.stream, f2.stream, c1.input, s1.input ARE_THE_SAME; 668
c1.output, s1.output, m1.feed[2] ARE_THE_SAME; 669
m1.out, col.input ARE_THE_SAME; 670
col.output, h1.input ARE_THE_SAME; 671
h1.output, r1.input, r2.input ARE_THE_SAME; 672
r1.output, r2.output, co2.input ARE_THE_SAME; 673
co2.output, fl1.feed ARE_THE_SAME; 674
fl1.liq, h2.input ARE_THE_SAME; 675
fl1.vap, sp1.feed ARE_THE_SAME; 676
sp1.out[1], h3.input ARE_THE_SAME; 677
sp1.out[2], c2.input, s2.input ARE_THE_SAME; 678
c2.output, s2.output, m1.feed[1] ARE_THE_SAME; 679
680
681
(* Conditional statements *) 682
683
WHEN (select_feed1) 684
CASE TRUE: 685
USE f1; 686
CASE FALSE: 687
USE f2; 688
END WHEN; 689
690
WHEN (select_single1) 691
CASE TRUE: 692
USE c1; 693
CASE FALSE: 694
USE s1; 695
END WHEN; 696
697
WHEN (select_cheapr1) 698
CASE TRUE: 699
USE r1; 700
CASE FALSE: 701
USE r2; 702
END WHEN; 703
704
WHEN (select_single2) 705

```

```

        CASE TRUE:
            USE c2;
        CASE FALSE:
            USE s2;
    END WHEN;

METHODS

METHOD default_self;
END default_self;

METHOD seqmod;
    RUN c1.seqmod;
    RUN c2.seqmod;
    RUN s1.seqmod;
    RUN s2.seqmod;
    RUN co1.seqmod;
    RUN co2.seqmod;
    RUN h1.seqmod;
    RUN h2.seqmod;
    RUN h3.seqmod;
    RUN r1.seqmod;
    RUN r2.seqmod;
    RUN fl1.seqmod;
    RUN sp1.seqmod;
    RUN m1.seqmod;
END seqmod;

METHOD specify;
    RUN seqmod;
    RUN f1.specify;
    RUN f2.specify;
END specify;

END flowsheet;

( * ***** *)

MODEL test_flowsheet REFINES flowsheet;

    f1.stream.components := ['A','B','C','D'];

METHODS

METHOD default_self;

```

```

END default_self;                                     752
                                                         753
METHOD values;                                       754
                                                         755
    (* Initial Configuration *)                       756
    select_feed1 := TRUE;                             757
    select_single1 := TRUE;                           758
    select_cheapr1 := TRUE;                           759
    select_single2 := TRUE;                           760
                                                         761
    (* Fixed Values *)                               762
                                                         763
(* Physical Properties of Components *)              764
                                                         765
    f1.stream.state.Cpi['A'] := 0.04 {BTU/mole/K};    766
    f1.stream.state.Cpi['B'] := 0.05 {BTU/mole/K};    767
    f1.stream.state.Cpi['C'] := 0.06 {BTU/mole/K};    768
    f1.stream.state.Cpi['D'] := 0.055 {BTU/mole/K};   769
                                                         770
(* Feed 1 *)                                         771
    f1.cost_factor := 0.026 {dollar/kg_mole};         772
                                                         773
(* Feed 2 *)                                         774
    f2.cost_factor := 0.033 {dollar/kg_mole};         775
                                                         776

(* Cooler 1 *)                                       777
    col.cost_factor := 0.7e-06 {dollar/kJ};           778
    col.heat_removed := 100 {BTU/s};                  779
                                                         780

(* Cooler 2 *)                                       781
    co2.heat_removed := 150 {BTU/s};                  782
    co2.cost_factor := 0.7e-06 {dollar/kJ};           783
                                                         784

(* Heater 1 *)                                       785
    h1.heat_supplied := 200 {BTU/s};                  786
    h1.cost_factor := 8e-06 {dollar/kJ};              787
                                                         788

(* Heater 2 *)                                       789
    h2.heat_supplied := 180 {BTU/s};                  790
    h2.cost_factor := 8e-06 {dollar/kJ};              791
                                                         792

(* Heater 3 *)                                       793
    h3.heat_supplied := 190 {BTU/s};                  794
    h3.cost_factor := 8e-06 {dollar/kJ};              795
                                                         796

(* Flash *)                                         797

```

```

    fl1.alpha['A'] := 12.0; 798
    fl1.alpha['B'] := 10.0; 799
    fl1.alpha['C'] := 1.0; 800
    fl1.alpha['D'] := 6.0; 801
    fl1.vap_to_feed_ratio := 0.9; 802
    803
(* Splitter *) 804
    spl.split[1] := 0.05; 805
    806
(* Mixer *) 807
    m1.To := 298 {K}; 808
    809
(* Single Compressor 1 *) 810
    c1.cost_factor := 8.33333e-06 {dollar/kJ}; 811
    c1.pressure_rate := 2.5; 812
    813
(* Single Compressor 2 *) 814
    c2.cost_factor := 8.33333e-06 {dollar/kJ}; 815
    c2.pressure_rate := 1.5; 816
    817
(* Staged Compressor 1 *) 818
    s1.cost_factor_work := 8.33333e-06 {dollar/kJ}; 819
    s1.cost_factor_heat := 0.7e-06 {dollar/kJ}; 820
    s1.pressure_rate := 2.5; 821
    s1.n_stages := 2.0; 822
    823
(* Staged Compressor 2 *) 824
    s2.cost_factor_work := 8.33333e-06 {dollar/kJ}; 825
    s2.cost_factor_heat := 0.7e-06 {dollar/kJ}; 826
    s2.pressure_rate := 1.5; 827
    s2.n_stages := 2.0; 828
    829
(* Reactor 1 *) 830
    r1.stoich_coef['A'] := -1; 831
    r1.stoich_coef['B'] := -1; 832
    r1.stoich_coef['C'] := 1; 833
    r1.stoich_coef['D'] := 0; 834
    r1.low_turnover := 0.0069 {kg_mole/s}; 835
    836
(* Reactor 2 *) 837
    r2.stoich_coef['A'] := -1; 838
    r2.stoich_coef['B'] := -1; 839
    r2.stoich_coef['C'] := 1; 840
    r2.stoich_coef['D'] := 0; 841
    r2.high_turnover := 0.00828 {kg_mole/s}; 842
    843

```

```

        (* Initial Guess *)                                844
                                                            845
(* Flash *)                                                846
    fl1.ave_alpha:= 5.0;                                    847
                                                            848
END values;                                                849
                                                            850
METHOD configuration2;                                     851
(* alternative configuration *)                             852
    select_feed1 := FALSE;                                  853
    select_single1 := FALSE;                                854
    select_cheapr1 := FALSE;                                855
    select_single2 := FALSE;                                856
END configuration2;                                         857
                                                            858
METHOD configuration3;                                     859
(* alternative configuration *)                             860
    select_feed1 := FALSE;                                  861
    select_single1 := TRUE;                                  862
    select_cheapr1 := TRUE;                                  863
    select_single2 := FALSE;                                864
END configuration3;                                         865
                                                            866
END test_flowsheet;                                        867
                                                            868
(* ***** *)                                             869
                                                            870

```

# CHAPTER 17 A SIMPLE CHEMICAL ENGINEERING FLOWSHEETING EXAMPLE

In this example we shall examine a model for a simple chemical engineering process flowsheet. The code listed below exists in the file in the ASCEND examples subdirectory entitled *simple\_fs.asc*. Except for some formatting changes to make it more presentable here, it is exactly as it is in the library version. Thus you could run this example by loading this file and using it and its corresponding script *simple\_fs.s*.

## 17.1 THE PROBLEM DESCRIPTION

This model is of a simple chemical engineering flowsheet. Studying it will help to see how one constructs more complex models in ASCEND. Models for more complex objects are typically built out of previously defined types each of which may itself be built of previously defined parts, etc. A flowsheet could, for example, be built of units and streams. A distillation column could itself be built out of trays and interconnecting streams.

Lines 40 to 56 in the code below give a diagram of the flowsheet we would like to model. This flowsheet is to convert species B into species C. B undergoes the reaction.



The available feed contains 5 mole percent of species A, a light contaminant that acts as an inert in the reactor. We pass this feed into the reactor where only about 7% of B converts per pass. Species C is much less volatile than B which is itself somewhat less volatile than A. Relative volatilities are 12, 10 and 1 respectively for A, B and C. Species A will build up if we do not let it escape from the system. We propose to do this by bleeding off a small portion (say 1 to 2%) of the B we recover and recycle back to the reactor.

The flowsheet contains a mixer where we mix the recycle with the feed, a reactor, a flash unit, and a stream splitter where we split off and remove some of the recycled species B contaminated with species A



Our goal is to determine the impact of the bleed on the performance of this flowsheet. We would also like to see if we can run the flash unit to get us fairly pure C as a bottom product from it.

The first type definitions we need for our simple flowsheet are for the variables we would like to use in our model. The ones needed for this example are all in the file *atoms.a4l*. Thus we will need to load *atoms.a4l* before we load the file containing the code for this model.

The following is the code for this model. We shall intersperse comments on the code within it.

## 17.2 THE CODE

As the code is in our ASCEND models directory, it has header information that we require of all such files included as one large comment extending over several lines. Comments are in the form (\* comment \*).

To assure that appropriate library files are loaded first, ASCEND has the REQUIRE statement, such as appears on line 61:

```
REQUIRE atoms.a4l
```

This statement causes the system to load the file *atoms.a4l* before continuing with the loading of this file. *atoms.a4l* in turn has a require statement at its beginning to cause *system.a4l* to be loaded before it is.

```
( *****\
    simple_fs.asc
    by Arthur W. Westerberg
    Part of the Ascend Library

This file is part of the Ascend modeling library.

Copyright (C) 1994

The Ascend modeling library is free software; you can redistribute
it and/or modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

The Ascend Language Interpreter is distributed in hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
```

General Public License for more details. 18

You should have received a copy of the GNU General Public License along 19  
with the program; if not, write to the Free Software Foundation, Inc., 20  
675 Mass Ave, Cambridge, MA 02139 USA. Check the file named COPYING. 21  
22

Use of this module is demonstrated by the associated script file 23  
simple\_fs.s. 24

\\*\*\*\*\*) 25

(\*\*\*\*\*\ 26

\$Date: 97/02/20 18:54:21 \$ 27

\$Revision: 1.5 \$ 28

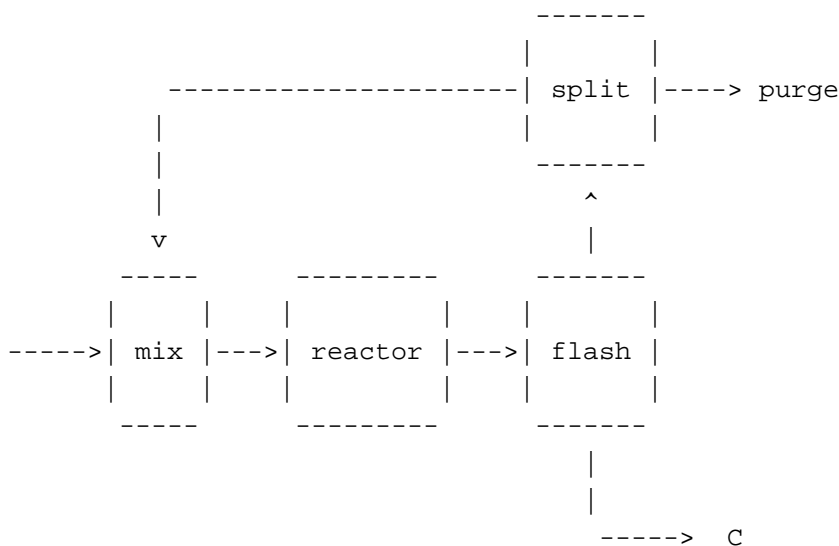
\$Author: mthomas \$ 29

\$Source: /afs/cs.cmu.edu/project/ascend/Repository/models/examples/  
simple\_fs.asc,v \$ 30

\\*\*\*\*\*) 31

(\* 32

The following example illustrates equation based modeling using the 33  
ASCEND system. The process is a simple recycle process. 34  
35



This model requires: "system.a4l" 36  
"atoms.a4l" 37

\*) 38

REQUIRE atoms.a4l 39

The first model we shall define is for defining a stream. In the document entitled “Equation-based Process Modeling” we argue the need to define a stream by maximizing the use of intensive variables and the equations interrelating them. Our problem here requires only the molar flows for the components as the problem definition provides us with all the physical properties as constants. Nowhere for this simple model do we seem to need temperatures, fugacities, etc. To maximize the use of intensive variables, we will use mole fractions and total molar flow to characterize a stream. We must include the equation that says the mole fractions add to unity. Our first model we call *mixture*.

```
( * ***** *) 62
63
MODEL mixture; 64
65
    components          IS_A set OF symbol_constant; 66
    y[components]       IS_A fraction; 67
68
    SUM[y[i] | i IN components] = 1.0; 69
70
METHODS 71
    METHOD clear; 72
        y[components].fixed := FALSE; 73
    END clear; 74
75
    METHOD specify; 76
        y[components].fixed := TRUE; 77
        y[CHOICE[components]].fixed := FALSE; 78
    END specify; 79
80
    METHOD reset; 81
        RUN clear; 82
        RUN specify; 83
    END reset; 84
85
END mixture; 86
87
```

Line 66 of the model for mixture defines a set of symbol constants. We will later include in this set one symbol constant giving a name for each of the species in the problem (A, B and C). Line 67 defines one mole fraction variable for each element in the set of components, while line 69 says these mole fractions must add to 1.0.

We add a methods section to our model to handle the flag setting which we shall need when making the problem well-posed -- i.e., as a problem having an equal number of unknowns as equations. We first have a method called `clear` which resets all the “fixed” flags for all the variables in this model to `FALSE`. This method puts the problem into a known state (all flags are `FALSE`). The second method is our selection of variables that we wish to fix if we were to solve the equations corresponding to a mixture model. There is only one equation among all the mole fraction variables so we set all but one of the flags to `TRUE`. The `CHOICE` function picks arbitrarily one of the members of the set *components*. For that element, we reset the fixed flag to `FALSE`, meaning that this one variable will be computed in terms of the values given to the others.

The reset method is useful as it runs first the `clear` method to put an instance of a mixture model into a known state with respect to its fixed flags, followed by running the `specify` method to set all but one of the fixed flags to `TRUE`.

These methods are not needed to create our model. To include them is a matter of modeling style, a style we consider to be good practice. The investment into writing these methods now has always been paid back in reducing the time we have needed to debug our final models.

The next model we write is for a stream, a model that will include a part we call *state* which is an instance of the type *mixture*.

```
( * ***** *) 88
MODEL molar_stream; 89
  components      IS_A set OF symbol_constant; 90
  state           IS_A mixture; 91
  Ftot,f[components] IS_A molar_rate; 92
  components, state.components ARE_THE_SAME; 93
  FOR i IN components CREATE 94
    f_def[i]: f[i] = Ftot*state.y[i]; 95
  END; 96
METHODS 97
  METHOD clear; 98
    RUN state.clear; 99
    Ftot.fixed := FALSE; 100
  101
  102
  103
  104
  105
  106
```

```

        f[components].fixed:= FALSE;                                107
    END clear;                                                        108
                                                                    109
    METHOD seqmod;                                                    110
        RUN state.specify;                                           111
        state.y[components].fixed:= FALSE;                          112
    END seqmod;                                                       113
                                                                    114
    METHOD specify;                                                    115
        RUN seqmod;                                                  116
        f[components].fixed:= TRUE;                                  117
    END specify;                                                       118
                                                                    119
    METHOD reset;                                                      120
        RUN clear;                                                   121
        RUN specify;                                                 122
    END reset;                                                         123
                                                                    124
    METHOD scale;                                                       125
        FOR i IN components DO                                       126
            f[i].nominal := f[i] + 0.1{mol/s};                      127
        END;                                                         128
        Ftot.nominal := Ftot + 0.1{mol/s};                          129
    END scale;                                                         130
                                                                    131
END molar_stream;                                                    132
                                                                    133

```

We define our stream over a set of components. We next include a part which is of type mixture and call it *state* as mentioned above. We also include a variable entitled *Ftot* which will represent the total molar flowrate for the stream. For convenience -- as they are not needed, we also include the molar flows for each of the species in the stream. We realize that the components defined within the part called *state* and the set of components we just defined for the stream should be the same set. We force the two sets to be the same set with the `ARE_THE_SAME` operator.

We next write the equations that define the individual molar flows for the components in terms of their corresponding mole fractions and the total flowrate for the stream. Note, the equations that says the mole fractions add to unity in the definition of the state forces the total of the individual flowrates to equal the total flowrate. Thus we do not need to include an equation that says the molar flowrates for the species add up to the total molar flowrate for the stream.

We again write the methods we need for handling flag setting. We leave it to the reader to establish that the specify method produces a well-posed instance involving the same number of variables to be computed as equations available to compute them. The scale method is there as we may occasionally wish to rescale the nominal values for our flows to reflect the values we are computing for them. Poor scaling of variables can lead to numerical difficulties for really large models. This method is there to reduce the chance we will have poor scaling.

Note that the nominal values for the remaining variables -- the mole fractions -- are unity. It does not need to be recomputed as unity is almost always a good nominal value for each of them.

Our next model is for the first of our unit operations. Each of these will be built of streams and equations that characterize their behavior. The first models a mixer. It can have any number of feed streams, each of which is a molar stream. We require the component set for each of the feed streams and the output stream from the unit to be the same set. Finally we write a component material balance for each of the species in the problem, where we sum the flows in each of the feeds to give the flow in the output stream, *out*.

```
( * ***** *) 134
135
MODEL mixer; 136
137
    n_inputs          IS_A  integer_constant; 138
    feed[1..n_inputs], out  IS_A  molar_stream; 139
140
    feed[1..n_inputs].components, 141
    out.components          ARE_THE_SAME; 142
143
    FOR i IN out.components CREATE 144
        cmb[i]: out.f[i] = SUM[feed[1..n_inputs].f[i]]; 145
    END; 146
147
METHODS 148
149
    METHOD clear; 150
        RUN feed[1..n_inputs].clear; 151
        RUN out.clear; 152
    END clear; 153
154
    METHOD seqmod; 155
    END seqmod; 156
157
```

```

METHOD specify;                                158
    RUN seqmod;                                159
    RUN feed[1..n_inputs].specify;            160
END specify;                                   161
                                                162
METHOD reset;                                  163
    RUN clear;                                 164
    RUN specify;                              165
END reset;                                     166
                                                167
METHOD scale;                                  168
    RUN feed[1..n_inputs].scale;              169
    RUN out.scale;                            170
END scale;                                    171
                                                172
END mixer;                                    173

```

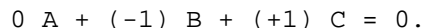
The *METHOD clear* sets all the fixed flags for the parts of this model to false by running each of their clear methods (i.e., for all the feeds and for the stream out). If this model had introduced any new variables, their fixed flags would have been set to FALSE here.

We will implement the method to make the model well posed into two parts: *seqmod* (stands for “sequential modular” which is the mindset we use to get a unit well-posed) and *specify*. The first we shall use within any unit operation to fix exactly enough fixed flags for a unit such that, if we also make the feed streams to it well-posed, the unit will be well-posed. For a mixer unit, the output stream results simply from mixing the input streams; there are no other variables to set other than those for the feeds. Thus the *seqmod* method is empty. It is here for consistency with the other unit operation models we write next. The *METHOD specify* makes this model well-posed by calling the *seqmod* method and then the *specify* method for each of the feed streams. No other flags need be set to make the model well-posed.

*METHOD reset* simply runs *clear* followed by *specify*. Running this sequence of method will make the problem well-posed no matter which of the fixed flags for it are set to TRUE before running *reset*. Finally, flowrates can take virtually any value so we can include a *scale* method to scale the flows based on their current values.

The next model is for a very simple ‘degree of conversion’ reactor. The model defines a turnover rate which is the rate at which the reaction as written proceeds (e.g., in moles/s). For example, here our reaction will be  $B \rightarrow C$ . A turnover rate of 3.7 moles/s would mean that 3.7 moles/s of B would convert to 3.7 moles/s of C. The vector *stoich\_coef* has one

entry per component. Here there will be three components when we test this model so the coefficients would be 0, -1, 1 for the reaction



Reactants have a negative coefficient, products a positive one. The material balance to compute the flow out for each of the components sums the amount coming in plus that created by the reaction.

```
( * ***** *) 174
175
MODEL reactor; 176
177
    feed, out          IS_A molar_stream; 178
    feed.components, out.components ARE_THE_SAME; 179
180
    turnover IS_A molar_rate; 181
    stoich_coef[feed.components] IS_A factor; 182
183
    FOR i IN feed.components CREATE 184
        out.f[i] = feed.f[i] + stoich_coef[i]*turnover; 185
    END; 186
187
METHODS 188
189
    METHOD clear; 190
        RUN feed.clear; 191
        RUN out.clear; 192
        turnover.fixed := FALSE; 193
        stoich_coef[feed.components].fixed := FALSE; 194
    END clear; 195
196
    METHOD seqmod; 197
        turnover.fixed := TRUE; 198
        stoich_coef[feed.components].fixed := TRUE; 199
    END seqmod; 200
201
    METHOD specify; 202
        RUN seqmod; 203
        RUN feed.specify; 204
    END specify; 205
206
    METHOD reset; 207
        RUN clear; 208
        RUN specify; 209
    END reset; 210
```



```

METHOD scale;
    RUN feed.scale;
    RUN out.scale;
    turnover.nominal := turnover.nominal+0.0001 {kg_mole/s};
END scale;
END reactor;

```

The *METHOD clear* first directs all the parts of the reactor to run their *clear* methods. Then it sets the fixed flags for all variables introduced in this model to FALSE.

Assume the feed to be known. We introduced one stoichiometric coefficient for each component and a turnover rate. To make the output stream well-posed, we would need to compute the flows for each of the component flows leaving. That suggests the material balances we wrote are all needed to compute these flows. We would, therefore, need to set one fixed flag to TRUE for each of the variables we introduced, which is what we do in the *METHOD seqmod*. Now when we run *seqmod* and then the *specify* method for the feed, we will have made this model well-posed, which is what we do in the *METHOD specify*.

The flash model that follows is a constant relative volatility model. Try reasoning why the methods attached are as they are.

```

( * ***** *)
MODEL flash;

    feed,vap,liq      IS_A      molar_stream;

    feed.components,
    vap.components,
    liq.components    ARE_THE_SAME;

    alpha[feed.components],
    ave_alpha          IS_A      factor;

    vap_to_feed_ratio  IS_A      fraction;

    vap_to_feed_ratio*feed.Ftot = vap.Ftot;

    FOR i IN feed.components CREATE
        cmb[i]: feed.f[i] = vap.f[i] + liq.f[i];

```

```

        eq[i]:  vap.state.y[i]*ave_alpha = alpha[i]*liq.state.y[i];      239
    END;                                                                    240
                                                                    241
METHODS                                                                    242
                                                                    243
    METHOD clear;                                                            244
        RUN feed.clear;                                                    245
        RUN vap.clear;                                                     246
        RUN liq.clear;                                                     247
        alpha[feed.components].fixed      := FALSE;                      248
        ave_alpha.fixed                    := FALSE;                      249
        vap_to_feed_ratio.fixed            := FALSE;                      250
    END clear;                                                              251
                                                                    252
    METHOD seqmod;                                                           253
        alpha[feed.components].fixed      := TRUE;                       254
        vap_to_feed_ratio.fixed            := TRUE;                       255
    END seqmod;                                                            256
                                                                    257
    METHOD specify;                                                         258
        RUN seqmod;                                                         259
        RUN feed.specify;                                                  260
    END specify;                                                           261
                                                                    262

    METHOD reset;                                                           263
        RUN clear;                                                          264
        RUN specify;                                                       265
    END reset;                                                             266
                                                                    267

    METHOD scale;                                                           268
        RUN feed.scale;                                                     269
        RUN vap.scale;                                                      270
        RUN liq.scale;                                                      271
    END scale;                                                             272
                                                                    273
END flash;                                                                274
                                                                    275
( * ***** *)                                                           276
                                                                    277

```

The final unit operation model is the splitter. The trick here is to make all the states for all the output streams the same as that of the feed. This move makes the compositions all the same and introduces only one equation to add those mole fractions to unity. The rest of the model should be evident.

```
MODEL splitter; 278
279
    n_outputs      IS_A integer_constant; 280
    feed, out[1..n_outputs] IS_A molar_stream; 281
    split[1..n_outputs] IS_A fraction; 282
283
    feed.components, out[1..n_outputs].components ARE_THE_SAME; 284
285
    feed.state, 286
    out[1..n_outputs].state ARE_THE_SAME; 287
288
    FOR j IN [1..n_outputs] CREATE 289
        out[j].Ftot = split[j]*feed.Ftot; 290
    END; 291
292
    SUM[split[1..n_outputs]] = 1.0; 293
294
METHODS 295
296
    METHOD clear; 297
        RUN feed.clear; 298
        RUN out[1..n_outputs].clear; 299
        split[1..n_outputs-1].fixed:=FALSE; 300
    END clear; 301
302
    METHOD seqmod; 303
        split[1..n_outputs-1].fixed:=TRUE; 304
    END seqmod; 305
306
    METHOD specify; 307
        RUN seqmod; 308
        RUN feed.specify; 309
    END specify; 310
311
    METHOD reset; 312
        RUN clear; 313
        RUN specify; 314
    END reset; 315
316
    METHOD scale; 317
        RUN feed.scale; 318
        RUN out[1..n_outputs].scale; 319
    END scale; 320
321
END splitter; 322
323
```

```
( * ***** *) 324
325
```

Now we shall see the value of writing all those methods for our unit operations (and for the models that we used in creating them). We construct our flowsheet by saying it includes a mixer, a reactor, a flash unit and a splitter. The mixer will have two inputs and the splitter two outputs. The next few statements configure our flowsheet by making, for example, the output stream from the mixer and the feed stream to the reactor be the same stream.

The methods are as simple as they look. This model does not introduce any variables nor any equations that are not introduced by its parts. We simply ask the parts to clear their variable fixed flags.

To make the flowsheet well-posed, we ask each unit to set sufficient fixed flags to TRUE to make itself well posed were its feed stream well-posed (now you can see why we wanted to create the methods *seqmod* for each of the unit types.) Then the only streams we need to make well-posed are the feeds to the flowsheet, of which there is only one. The remaining streams come out of a unit which we can think of computing the flows for it.

```
MODEL flowsheet; 326
327
    m1              IS_A  mixer; 328
    r1              IS_A  reactor; 329
    fl1             IS_A  flash; 330
    spl             IS_A  splitter; 331
332
(* define sets *) 333
334
    m1.n_inputs     ==2; 335
    spl.n_outputs   ==2; 336
337
(* wire up flowsheet *) 338
339
    m1.out, r1.feed  ARE_THE_SAME; 340
    r1.out, fl1.feed ARE_THE_SAME; 341
    fl1.vap, spl.feed ARE_THE_SAME; 342
    spl.out[2], m1.feed[2] ARE_THE_SAME; 343
344
METHODS 345
346
    METHOD clear; 347
        RUN m1.clear; 348
```

```

        RUN r1.clear;                                349
        RUN f11.clear;                               350
        RUN sp1.clear;                               351
    END clear;                                       352
                                                    353
    METHOD seqmod;                                   354
        RUN m1.seqmod;                               355
        RUN r1.seqmod;                               356
        RUN f11.seqmod;                              357
        RUN sp1.seqmod;                              358
    END seqmod;                                    359
                                                    360
    METHOD specify;                                  361
        RUN seqmod;                                  362
        RUN m1.feed[1].specify;                      363
    END specify;                                    364
                                                    365
    METHOD reset;                                    366
        RUN clear;                                   367
        RUN specify;                                 368
    END reset;                                       369
                                                    370
    METHOD scale;                                    371
        RUN m1.scale;                                372
        RUN r1.scale;                                373
        RUN f11.scale;                               374
        RUN sp1.scale;                               375
    END scale;                                       376
                                                    377
END flowsheet;                                     378
                                                    379
( * *****)                                         380
                                                    381

```

We have created a flowsheet model above. If you look at the reactor model, we require that you specify the turnover rate for the reaction. We may have no idea of a suitable turnover rate. What we may have an idea about is the conversion of species B in the reactor; for example, we may know that about 7% of the B entering the reactor may convert. How can we alter our model to allow for us to say this about the reactor and not be required to specify the turnover rate? In a sequential modular flowsheeting system, we would use a “computational controller.” We shall create a model here that gives us this same functionality. Thus we call it a “controller.” There are many ways to construct this model. We choose here to create a model that has a flowsheet as a part of it. We introduce a variable conv which will

indicate the fraction conversion of any one of the components which we call the `key_component` here. For that component, we add a material balance based on the fraction of it that will convert. We added one new variable and one new equation so, if the flowsheet is well-posed, so will our controller be well-posed. However, we want to specify the conversion rather than the turnover rate. The *specify* method first asks the flowsheet `fs` to make itself well-posed. Then it makes this one trade: fixing `conv` and releasing the turnover rate.

```

MODEL controller;                                     382

fs                                                     IS_A  flowsheet;   383
conv                                                  IS_A  fraction;   384
key_components                                       IS_A  symbol_constant;  385
fs.r1.out.f[key_components] = (1 - conv)*fs.r1.feed.f[key_components]; 386
                                                    387
METHODS                                               388
                                                    389
METHOD clear;                                         390
  RUN fs.clear;                                       391
  conv.fixed:=FALSE;                                  392
END clear;                                           393
                                                    394
METHOD specify;                                       395
  RUN fs.specify;                                     396
  fs.r1.turnover.fixed:=FALSE;                        397
  conv.fixed:=TRUE;                                   398
END specify;                                         399
                                                    400
METHOD reset;                                         401
  RUN clear;                                          402
  RUN specify;                                        403
END reset;                                           404
                                                    405
METHOD scale;                                         406
  RUN fs.scale;                                       407
END scale;                                           408
                                                    409
END controller;                                       410
                                                    411
( * ***** *)                                     412
                                                    413
                                                    414

```

We now would like to test our models to see if they work. How can we write test for them? We can create test models as we do below.

To test the flowsheet model, we create a `test_flowsheet` model that refines our previously defined flowsheet model. “To refine the previous model” means this model includes all the statements made to define the flowsheet model plus those statements that we now provide here. So this model is a flowsheet but with its components specified to be ‘A’, ‘B’, and ‘C’. We add a new method called *values* in which we specify values for all the variables we intend to fix when we solve. We can also provide values for other variables; these will be used as the initial values for them when we start to solve. We see all the variables being given values with the units specified. The units must be specified in ASCEND. ASCEND will interpret the lack of units to mean the variable is unitless. If it is not, then you will get a diagnostic from ASCEND telling you that you have written a dimensionally inconsistent relationship.

Note we specify the molar flows for the three species in the feed. Given these flows, the equations for the stream will compute the total flow and then the mole fractions for it. Thus the feed stream is fully specified with these flows.

We look at the `seqmod` method for each of the units to see the variables to which we need to give values here.

```

MODEL test_flowsheet REFINES flowsheet;                                415
                                                                           416
    m1.out.components:=[ 'A' , 'B' , 'C' ];                            417
                                                                           418
METHODS                                                                419
                                                                           420
METHOD values;                                                         421
    m1.feed[1].f[ 'A' ]          := 0.005 {kg_mole/s};                422
    m1.feed[1].f[ 'B' ]          := 0.095 {kg_mole/s};                423
    m1.feed[1].f[ 'C' ]          := 0.0 {kg_mole/s};                  424
                                                                           425
    r1.stoich_coef[ 'A' ]         := 0;                                426
    r1.stoich_coef[ 'B' ]         := -1;                               427
    r1.stoich_coef[ 'C' ]         := 1;                                428
    r1.turnover                   := 3 {kg_mole/s};                    429
                                                                           430
    f11.alpha[ 'A' ]              := 12.0;                             431
    f11.alpha[ 'B' ]              := 10.0;                             432
    f11.alpha[ 'C' ]              := 1.0;                              433
    f11.vap_to_feed_ratio         := 0.9;                              434
    f11.ave_alpha                 := 5.0;                              435
                                                                           436
    sp1.split[1]                  := 0.01;                              437

```

```

f11.liq.Ftot:=m1.feed[1].f['B'];
END values;
END test_flowsheet;
( * ***** *)

```

Finally we would like to test our controller model. Again we write our test model as a refinement of the model to be tested. The test model is, therefore, a controller itself. We make our fs model inside our test model into a test\_flowsheet, making it a more refined type of part than it was in the controller model. We can do this because the test\_controller model is a refinement of the flowsheet model which fs was previously. A test\_flowsheet is, as we said above, a flowsheet. We create a values method which first runs the values method we wrote for the test\_flowsheet model and then adds a specification for the conversion of B in the reactor.

```

MODEL test_controller REFINES controller;

fs      IS_REFINED_TO  test_flowsheet;
key_components := 'B';

METHODS

METHOD values;
  RUN fs.values;
  conv          := 0.07;
END values;

END test_controller;
( * ***** *)

```





# CHAPTER 18 THE ASCEND PREDEFINED COLLECTION OF MODELS

The ASCEND system has a main directory of libraries and application models: `ascend4/models`. One can examine and execute these examples when learning how to model in ASCEND.

## **system.a4l**

The file called *system.a4l* in the libraries must always be loaded first in the ASCEND system. It is automatically loaded when one starts the ASCEND system. However, the *delete all types* command will delete all type definitions including the ones in this file. If you have deleted all types, always reload this file first using the *Read* instruction in the Library tool set.

## **atoms.a4l**

The simplest collection of previously defined types are those which define the kinds of constants, parameters and variables we are likely to use in constructing an engineering or scientific model. A file called *atoms.a4l* located in the libraries subdirectory has over 125 types of constants, parameters and variables. Following are three of the definitions it contains.

```

CONSTANT critical_compressibility REFINES
    real_constant DIMENSIONLESS;                                1
                                                                    2
UNIVERSAL CONSTANT speed_of_light                               3
REFINES real_constant := 1{LIGHT_C};                             4
                                                                    5
ATOM volume REFINES solver_var                                  6
    DIMENSION L^3                                                7
    DEFAULT 100.0{ft^3};                                          8
    lower_bound := 0.0{ft^3};                                    9
    upper_bound := 1e50{ft^3};                                  10
    nominal := 100.0{ft^3};                                      11
END volume;                                                      12

```

Note that the first and third include a statement of the dimensionality of the item being defined. For example critical compressibility is dimensionless while the dimensions for volume are  $L^3$  (i.e., length cubed). The ASCEND system supports nine basic dimensions as defined for the standards defining the SI system of units. Dimensions

differ from units in that *length* is a dimension while *feet* is a set of units one may use to express a length. Dimensions in ASCEND are L (length: typical units being ft, m), M (mass: kg, lbm), T (time: s, yr), E (electric current: amp), Q (quantity: mole), TMP (temperature: K, R), LUM (luminous intensity: candela), P (plane angle: radian) and S (solid angle: steradian). We have also included the tenth dimension C (currency: USdollar) so one can express cost. If you wish to express cost in a variety of different currencies (e.g., USdollars, UKpounds), you will have to define the conversion rates.

(See the manual entitled The ASCEND IV language syntax and semantics for more information on dimensionality and units.)

### **Typical use of library files**

One will typically create models in the ASCEND system by including one or more of the library files available. Almost certainly the file *atoms.a4l* will become a part of any engineering or scientific model.

It would be useful for you to view this and a few of the other library files using a text editor such as xemacs to see what libraries we do have available.

### **Examples and scripts**

The examples subdirectory in ASCEND has a number of complete ASCEND models. Each model is in two parts: the .a4c file containing the model definition and the .a4s file containing a script which one can use to execute the model. An example is the model *simple\_fs.a4c* along with its script *simple\_fs.a4s*.

Each of the example files indicates which of the library files one must load and the order in which to load them before loading the example file. If you fail to load a library file, you will experience a large number of diagnostic messages indicating there are missing type definitions.

# CHAPTER 19 THE ASCEND IV LANGUAGE

## SYNTAX AND SEMANTICS

Benjamin Allan<sup>1</sup>

Arthur W. Westerberg<sup>1</sup>

Department of Chemical Engineering  
and the Engineering Design Research Center /  
Institute for Complex Engineered Systems

Carnegie Mellon University

We shall present an informal description of the ASCEND IV language. Being informal, we shall usually include examples and descriptions of the intended semantics along with the syntax of the items. At times the inclusion of semantics will seem to anticipate later definitions. We do this because we would also like this chapter to be used as a reference for the ASCEND language even after one generally understands it. Often one will need to clarify a point about a particular item and will not wish to have to search in several places to do so.

*Syntax* is the form or structure for the statements in ASCEND, where one worries about the exact words one uses, their ordering, the punctuation, etc. *Semantics* describe the meaning of a statement.

To distinguish between syntax and semantics, consider the statement

```
y IS_A fraction;
```

Rules on the syntax for this statement tell us we need a user supplied instance name, *y*, followed by the ASCEND operator *IS\_A*, followed by a type name (*fraction*). The statement terminates with a semicolon. The statement semantics says we are declaring the existence of an instance,

---

1. The ASCEND language has evolved from the combined efforts of several generations of users and implementors. We wish to particularly acknowledge the contributions of ASCEND III implementors Kirk Abbott, Tom Epperly, Peter Piela, Boyd Safrit, Karl Westerberg, and Joe Zaher, and of the ASCEND IV crew: Duncan Coffey, Chad Farschman, Jennifer Perry, Vicente Rico-Ramirez, Mark Thomas and Ken Tyner.

locally named  $y$ , of the type fraction as a part within the current model definition and it is to be constructed when an instance of the current model definition is constructed.

The syntax for a computer language is often defined by using a Bachus-Naur formal (BNF) description. The complete YACC and FLEX description of the language described (as presently implemented) is available by FTP<sup>2</sup> and via the World Wide Web<sup>3</sup>. The semantics of a very high level modeling language such as ASCEND IV are generally much more restrictive than the syntax. For this reason we do not include a BNF description in this paper. ASCEND IV is an experiment. The language is under constant scrutiny and improvement, so this document is under constant revision. Contact the authors for the latest version.

## 19.1 PRELIMINARIES

We will start off with some background information and some tips that make the rest of the chapter easier to read. ASCEND is an object-oriented (OO) language for hierarchical modeling that has been somewhat specialized for mathematical models. Most of the specialization is in the implementation and the user interface rather than the language definition.

We feel the single most distinguishing feature of mathematical models is that solving them efficiently requires that the solving algorithms be able to address the entire problem either simultaneously or in a decomposition of the natural problem structure that the algorithm determines is best for the machine(s) in use. In the ASCEND language object-orientation is used to organize natural structures and make them easier to understand. It is not used to hide the details of the objects. The user (or machine) is free to ignore uninteresting details, and the ASCEND environment provides tools for the runtime suppression of these.

ASCEND is well into its 4th generation. Some features we will describe are not yet implemented (some merely speculative) and these are clearly marked (\* 4+ \*). Any feature not marked (\* 4+ \*) has been completely implemented, and thus any mismatch between the description given here and the software we distribute is a bug we want you to tell us about.

The syntax and semantics of ASCEND may seem at first a bit unusual. However, do not be afraid to just try what comes naturally if what we write here is unclear. The parser and compiler of ASCEND IV really will help you get things right. Of course if what

---

2. In the directory [ftp.cs.cmu.edu/project/ascend/gnu-ascend/](ftp://ftp.cs.cmu.edu/project/ascend/gnu-ascend/) see the file README.  
3. <http://www.cs.cmu.edu/afs/cs.cmu.edu/project/ascend/ftp/gnu-ascend/README>

we write here is unclear, please ask us about it because we aim to continuously improve both this document and the language system it describes.

We will describe, starting in Section 19.1.2, the higher level concepts of ASCEND, but first some important punctuation rules.

ASCEND is cAsE sensitive!

The keywords that are shown capitalized (or in lower case) in this chapter are that way because ASCEND is case sensitive. IS\_A is an ASCEND keyword; isa, Is\_a, and all the other permutations you can think of are NOT equivalent to IS\_A. In declaring new types of models and variables the user is free to use any style of capitalization he or she may prefer, however, they must remain consistent or undefined types and instances will result.

This case restriction makes our code very readable, but hard to type without a smart editor. We have kept the case-sensitivity because, like all mathematicians, we find ourselves running out of good variable names if we are restricted to a 26 letter alphabet. We have developed smart add-ins for two UNIX editors, EMACS and vi, for handling the upper case keywords and some other syntax elements. The use of these editors is described in another chapter.

The ASCEND IV parser is very picky and pedantic. It also tries to give helpful messages and occasionally even suggestions. New users should just dive in and make errors, letting the system help them learn how to avoid errors.

### 19.1.1 PUNCTUATION

This section covers both the punctuation that must be understood to read this document and the punctuation of ASCEND code.

**keywords:**

ASCEND keywords and type names are given in the left column in **bold** format. It is generally clear from the main text which are keywords and which are type names.

Minor items:

Minor headings that are helpful in finding details are given in the left column in underline format.

Tips:

Special notes and hints are sometimes placed on the left.

\*3\*:

This indicates that what follows is specific to ASCEND IIIc and may disappear in a future version of ASCEND IV. Generally ASCEND IV will provide some equivalent functionality at 1/10th of the ASCEND III price.

\*4\*

This indicates that what follows is specific to ASCEND IV and may not be available in ASCEND IIIc. Generally ASCEND III may provide some very klugey equivalent functionality, often at a very high price in terms of increased compilation time or debugging difficulty.

\*4+\*

ASCEND IV functionality that is not fully implemented at the time of this writing. The precise syntax of the final implementation may vary slightly from what is presented here. A revision of this document will be made at the time of implementation.

LHS:

Left Hand Side. Abbreviation used frequently.

RHS:

Right Hand Side. Abbreviation used frequently.

Simple Names:

In ASCEND simple names are made of the characters a through z, A through Z, , (\*4+\*: \$). The underscore is used as a letter, but it cannot be the first letter in a name. The "\$" character is used exclusively as the first character in the name of system defined built-in parts. "\$" is explained in more detail in Section 19.6.2. Simple names should be no more than 80 characters long.

Compound names:

Compound names are simple names strung together with dots (.). See the description of "." below.

Groupings:<< >>

In documentation optional fields are surrounded by these markers.

( \* \* )

Comment. \*3\* Anything inside these is a comment. Comments DO NOT nest in ASCEND IIIc. Comments may extend over many lines. \*4\* Comments DO nest in ASCEND IV.

( )

Rounded parentheses. Used to enclose arguments for functions or models where the order of the arguments matters. Also used to group terms in complex arithmetic, logical, or set expressions where the order of operations needs to be specified.

Efficiency tip:

The compiler can simplify relation definitions in a particularly efficient manner if constants are grouped together.

{ }

Curly braces. Used to enclose units. For example, 1 {kg\_mole/s}. Also used to enclose the body of annotations. **Note:** Curly braces are also used in TCL, the language of the ASCEND user interface, about which we will say more in another chapter.

- [ ] Square brackets. Used to enclose sets or elements of sets.  
Examples: `my_integer_set := [1,2,3]`, demonstrates the use of square brackets in the assignment of a set. `My_array[1]` demonstrates the use of square brackets in naming an array object indexed over an integer set which includes the element 1.
- .
- Dot. The dot is used, as in PASCAL and C, to construct the names of nested objects. Examples: if object `a` has a part `b`, then the way to refer to `b` is as `a.b`. `Tray[1].vle` shows a dot following a square bracket; here `Tray[1]` has a part named `vle`.
- .. Dot-dot or double dot. Integer range shorthand. For example, `my_integer_set := [1,2,3]` and `my_integer_set := [1..3]` are equivalent. If `..` appears in a context requiring `()`, such as the `ALIASES/IS_A` statement, then the range is expanded and ordered as we would naturally expect.
- :
- Colon. A separator used in various ways, principally to set the name of an arithmetic relation apart from the definition.
- ::
- Double colon. A separator used in the methods section for accessing methods defined on types other than the type the method is part of. Explained in Section 19.4.
- ;
- Semicolon. The separator of statements.

### 19.1.2 BASIC ELEMENTS

- Boolean value TRUE or FALSE. Can't get much simpler, eh? In the language definition TRUE and FALSE do not map to 1 and 0 or any other type of numeric value. (In the implementation, of course, they do.)
- User interface tip: The ASCEND user interface programmers have found it very convenient, however, to allow T/F, 1/0, Y/N, and other obvious boolean conventions as interactive input when assigning boolean values. We are lazy users.
- Integer value A signed whole number up to the maximum that can be represented by the computer on which one is running ASCEND. MAX\_INTEGER is machine dependent. Examples are:  
  
123  
-5  
MAX\_INTEGER, typically 2147483647.



Real value

ASCEND represents reals almost exactly as any other mathematically oriented programming language does. The mantissa has an optional negative sign followed by a string of digits and at most one decimal point. The exponent is the letter *e* or *E* followed by an integer. The number must not exceed the largest the computer is able to handle. There can be no blank characters in a real. MAX\_REAL is machine dependent. The following are legitimate reals in ASCEND:

```
-1
1.2
1.3e-2
7.888888e+34
.6E21
MAX_REAL, typically about 1.79E+308.
```

while the following are not:

```
1. 2 (*contains a blank within it*)
1.3e2.0 (*exponent has a decimal in it*)
+1.3 (* illegal unary + sign. x = +1.3 not allowed*)
```

Reals stored in SI units

We store all real values as double precision numbers in the MKS system of units. This eliminates many common errors in the modeling of physical systems. Since we also place the burden of scaling equations on system routines and a simple modeling methodology, the internal units are not of concern to most users.

Dimensionality:

Real values have dimensionality such as length/time for velocity. Dimensionality is to be distinguished from the units such as ft/s. ASCEND takes care of mapping between units and dimensions. A value without units (this includes integer values) is taken to be dimensionless. Dimensionality is built up from the following base dimensions:

Name	<u>definition</u>	<u>typical units</u>
<b>L</b>	length	meter, m
<b>M</b>	mass	kilogram, kg
<b>T</b>	time	second, s
<b>E</b>	electric current	ampere, A
<b>Q</b>	quantity	mole, mole

<b>TMP</b>	temperature	Kelvin, K
<b>LUM</b>	luminous intensity	candela, cd
<b>P</b>	plane angle	radian, rad
<b>S</b>	solid angle	steradian, sr
<b>C</b>	currency	currency, CR

The atom and constant definitions in the library illustrate the use of dimensionality.

Dimensions may be any combination of these symbols along with rounded parentheses, (), and the operators \*, ^ and /. Examples include  $M/T$  or  $M * L^2 / T^2 / TMP$  {this latter means  $(M * (L^2) / (T^2)) / TMP$ }. The second operand for the “to the power” operator, ^, must be an integer value (e.g., -2 or 3) because fractional powers of dimensional numbers are physically undefined.

If the dimensionality for a real value is undefined, then ASCEND gives it a wild card dimensionality. If ASCEND can later deduce its dimensionality from its use in a model definition it will do so. For example consider the real variable  $a$ , suppose  $a$  has wild card dimensionality,  $b$  has dimensionality of  $L/T$ . Then the statement:

Example of a dimensionally consistent equation.

$$a + b = 3 \text{ {ft/s}};$$

requires that  $a$  have the same dimensionality as the other two terms, namely,  $L/T$ . ASCEND will assign this dimensionality to  $a$ . The user will be warned of dimensionally inconsistent equations.

### Unit expression

A unit expression may be composed of any combination of unit names defined by the system and any numerical constants combined with times (\*), divide(/) and “to the power” (^) operators. The RHS of ^ must be an integer. Parentheses can be used to group subexpressions EXCEPT a divide operator may not be followed by a grouped subexpression.

So, {kg/m/s} is fine, but {kg/(m\*s)} is not. Although the two expressions are mathematically equivalent, it makes the system programming and output formatting easier to code and faster to execute if we disallow expressions of the latter sort.

The units understood by the system are defined in Chapter 20. Note that several “units” defined are really values of interesting constants

in SI, e.g.  $R := 1\{\text{GAS\_C}\}$  yields the correct value of the thermodynamic gas constant. Users can define additional units.

### Units

A unit expression must be enclosed in curly braces  $\{\}$ . When a real number is used in a mathematical expression in ASCEND, it must have a set of units expressed with it. If it does not, ASCEND assumes the number is dimensionless, which may not be the intent of the modeler. An example is shown in the dimensionally consistent equation above where the number 3 has the units  $\{\text{ft/s}\}$  associated with it.

Examples:

```
{kg_mole/s/m} same as {(kg_mole/s)/m}
{m^3/yr}
{3/100*ft} same as {0.03*ft}
{s^-1} same as {1/s}
```

Illegal unit examples are

```
{m/(K*kg_mole)} grouped subexpression used in the denominator;
should be written {m/K/kg_mole}.
{m^3.5} power of units or dimensions must be integer.
```

### Symbol Value

The format for a symbol is that of an arbitrary character string enclosed between two single quotes. There is no way to embed a single quote in a symbol: we are not in the escape sequence business at this time. The following are legal symbols in ASCEND:

```
'H2O'
'r1'
'Bill said,"foo" to whom?'
```

while the following are not legal symbol values:

```
"ethanol" (double quotes not allowed)
water (no single quotes given)
'i can't do this' (no embedded quotes)
```

There is an arbitrary upper limit to the number of characters in a symbol (something like 10,000) so that we may detect a missing close quote in a bad input file without crashing.

Sets values

Set values are lists of elements, all of type `integer_constant` or all of type `symbol_constant`, enclosed between square brackets []. The following are examples of sets:

```
['methane', 'ethane', 'propane']
[1..5, 7, 15]
[2..n_stages]
[1, 4, 2, 1, 16]
[]
```

We will say more about sets in 19.2.2.

The value range `1..5` is an allowable shorthand for the integers 1, 2, 3, 4 and 5 while the value range `2..n_stages` (where `n_stages` must be of type `integer_constant`) means all integers from 2 to `n_stages`. If `n_stages` is less than 2, then the third set is empty. The repeated occurrence of 1 in the fourth set is ignored. The fifth set is the empty set.

We use the term *set* in an almost pure mathematical sense. The elements have no order. One can only ask two things of a set: (1) if an element is a member of it and (2) its cardinality (`CARD(set)`). Repeated elements used in defining a set are ignored. The elements of sets **cannot** themselves be sets in ASCEND; i.e., there can be no sets of set.

Sets are unordered.

A set of integers may appear to be ordered to the modeler as the natural numbers have an order. However, it is the user imposing and using the ordering, not ASCEND. ASCEND sees these integers as elements in the set with NO ordering. Therefore, there are no operators in ASCEND such as successor or precursor member of a set.

Arrays

An array is a list of instances indexed over a set, in computer-speak, an associative array of objects. The instances are all of the same *base* type (as that is the only way they can be defined). An individual member of a list may later be more refined than the other members (we shall illustrate that possibility). The following are arrays in ASCEND.

```
stage[1..n_stages]
y[components]
column[areas][processes]
```

where `components`, `areas` and `processes` are sets. For example `components` could be the set of symbols

```
['ethylene', 'propylene'], areas the set of symbols
['feed_prep', 'prod_purification'] while
```

processes could be the set [ 'alcohol\_manuf' ,  
'poly\_propropylene\_manuf' ]. Note that the third example  
(column) is a list of lists (the way that ASCEND permits a multiply  
subscripted array).

The following are elements in the above arrays:

```
stage[1]
y[ 'ethylene' ]
column[ 'feed_prep' ][alcohol_manuf' ]
```

provided that n\_stages is 1 or larger.

There can be any number of subscripts for an array. We point out, however, that in virtually every application of arrays requiring more than two subscripts, there is usually a some underlying concept that is much better modeled as an object than as part of a deeply subscripted array. In the following jagged array example, there are really the concepts of unit operation and stream that would be better understood if made explicit.

#### Arrays can be jagged

(\* 4 \*) Arrays can be 'sparse' or jagged. For example:

```
process[1..3] IS_A set OF integer;
process[1] := [2];
process[2] := [7,5,3];
process[3] := [4,6];
FOR i in [1..3] CREATE
  FOR j IN process[i] CREATE
    flow[i][j] IS_A mass;
  END FOR;
END FOR;
```

process is an array of sets (not to be confused with a set of sets which ASCEND does not have) and flow is an array with six elements spread over three rows:

```
flow[1][2]
flow[2][7], flow[2][3], flow[2][5]
flow[3][4], flow[3][6]
```

Sparse arrays of models and variables are new to ASCEND IV.

#### Arrays are also instances

Each array is itself an object. That is, when you write  
"a[1..2] IS\_A real;" three objects get created: a[1],  
a[2], and a. a is an array instance which has parts named [1]

and [2] that are `real` instances. When a parameterized model requires an array, you pass it the single item `a`, not the elements `a[1..2]`.

### Not contiguous storage

Just in case you still have not caught on, ASCEND arrays are not blocks of memory such as are seen in low-level languages like C, FORTRAN, and Matlab. The *modeling* language does not provide things like `MatMult`, `Transpose`, and `Inverse` because these are *procedural* solving tools. If you are dedicated, you could write `METHODs` that implement matrix algebra, but this is a really dumb idea. We aim to structure our software so that it can interact openly with separate, dedicated tools (such as Matlab) when those tools are needed.

### Index variable

One can introduce a variable as an index ranging over a set. Index variables are local to the statements in which they occur. An example of using an index variable is the following FOR statement:

```
FOR i IN components CREATE
  VLE_equil[i]: y[i] = K[i]*x[i];
END FOR;
```

In this example `i` implicitly is of the same type as the values in the set `components`. If another object `i` exists in the model containing the FOR loop, it is ignored while executing the statements in that loop. This may cause unexpected results and the compiler will generate warnings about loop index shadowed variables.

### Label:

One can label statements which define arithmetic relationships (objective functions, equalities, and inequalities) in ASCEND. Labeling is highly recommended because it makes models much more readable and more easily debugged. Labels are also necessary for relations which are going to be used in conditional modeling or differentiation functions. A label is a sequence of alphanumeric characters ending in a colon. An example of a labeled equation is:

```
mass_balance: m_in = m_out;
```

An example of a labeled objective function is:

```
obj1: MAXIMIZE revenue - cost;
```

If a relation is defined within a FOR statement, it must have an array indexed label so that each instance created using the statement is distinguishable from the others. An example is:

```
FOR i IN components CREATE
  equil[i]: y[i] = K[i]*x[i];
END FOR;
```

The ASCEND interactive user interface identifies relationships by their labels. If one has not provided such a label, the system generates the label:

`modelname_equationnumber`

where *modelname* and *equationnumber* are the name of the model and the equation number in the model. An example is

`mixture_14`

for the unlabeled 14<sup>th</sup> relation in the mixture definition. If there is a conflict caused with an existing name, the generated name has enough letters added after *equationnumber* to make it a unique name. Remember that each model in a refinement hierarchy inherits the equations of its less refined ancestors, so the first equation appearing in the source code of a refining model may actually be the n<sup>th</sup> relation in that model.

### Lists

Often in a statement one can include a list of names or expression. A name list is one or more names where multiple list entries are separated from each other by commas. Examples of a list of names are:

```
T1, inlet_T, outlet_T
y[components], y_in
stage[1..n_stages]
```

### Ordered lists:

If the ordering of names in a list matters, that list is enclosed in (). Order matters in: calling externally defined methods or models, calling most real-valued functions, passing parameters to ASCEND models or methods, and declaring the controlling parameters that SELECT, SWITCH, and WHEN statements make decisions on.

## 19.1.3 BASIC CONCEPTS

### Instances and types

This is an opportune time to emphasize the distinction between the terms *instance* and *type*. A *type* in ASCEND is what we define when we declare an ASCEND model or atom. It is the formal definition of the attributes (parts) and attribute default values that an

object will have if it is created using the type definition. Methods are associated with types.

In ASCEND there are two meanings (closely related) of an instance.

- An *instance* is a *named part* that exists within a type definition.
- An *instance* is a compiled object.

If one is in the context of the ASCEND interface, the system compiles an instance of a model type to create an object with which one carries out computations. The system requires the user to give a simple name for this simulation instance. This name given is then the first part of the qualified name for all the parts of the compiled object.

### Implicit types

It is possible to create an instance that does not have a corresponding type definition in the library. The type of such an instance is said to be *implicit*. (Some people use the word *anonymous*. However, no computable type is anonymous and the implicit type of an instance is theoretically computable). The simplest example of an implicit type is the type of an instance compiled from the built-in definition `integer_constant`. For example:

```
i, j IS_A integer_constant;  
i := 2;  
j := 3;
```

Instances `i` and `j`, though of the same formal type, are implicit type incompatible because they have been assigned distinct values.

Instances which are either formally or implicitly type incompatible cannot be merged. This will be discussed further in Section 19.3.

### Parsing

Most errors in the declaration of an ASCEND model can be caught at parse time because the object type of any well-formed name in an ASCEND definition can be resolved or proved ambiguous. We cannot prove at parse time whether a specific array element will exist, but we can know that should such an element exist, it must be of the type with which the array is defined.



Ambiguity is warned about loudly because it is caused by either misspelling or poor modeling style. The simplest example of ambiguity follows.

Assume a type, `stream`, and a refinement of `stream`, `heat_stream`, which adds the new variable `H`. Now, if we write:

```
MODEL mixer;
input[1..2] IS_A stream;
output IS_A heat_stream;
input[1].H + input[2].H = output.H;
END mixer;
```

We see the parser can find the definition of `H` in the type `heat_stream`, so `output.H` is well defined. The author of the mixer model may intend to refine `input[1]` and `input[2]` to be objects of different types, say `steam_stream` and `electric_stream`, where each defines an `H` suitable for use in the equation. The parser cannot read the author's mind, so it warns that `input[1].H` and `input[2].H` are ambiguous in the mixer definition. The mixer model is not highly reusable except by the author, but sometimes reusability is not a high priority objective. The mixer definition is allowed, but it may cause problems in instantiation if the author has forgotten the assumption that is not explicitly stated in the model and neglects to refine the input streams appropriately.

### Instantiation

Creating an simulation based on a type definition is a multi-phase process called compiling (or instantiation). When an instantiation cannot be completed because some structural parameter (a `symbol_constant`, `real_constant`, `boolean_constant`, `integer_constant`, or `set`) does not have a value there will be `PENDING` statements. The user interface will warn that something is incomplete.

In phase 1 all statements that create instance structures or assign constant values are executed. This phase theoretically requires an infinite number of passes through the structural statements of a definition. We allow a maximum of 5 and have never needed more than 3. There may be pending statements at the end of phase 1. The compiler or interface will issue warnings about pending statements, starting with warnings about unassigned constants.

Phase 2 compiles as many real arithmetic relation definitions as possible. Some relations may be impossible to compile because the constants or sets they depend on do not have values assigned. Other

relations may be impossible because they reference variables that do not exist. This is determined in a single pass.

Phase 3 compiles as many logical arithmetic relation definitions as possible. Some relations may be impossible to compile because the constants or sets they depend on do not have values assigned. Other relations may be impossible because they reference real arithmetic relations that do not exist. This is determined in a single pass.

Phase 4 compiles as many conditional programming statements (WHENs) as possible. Some WHEN relations may be impossible to compile because the discrete variables, models, or relations they depend on do not exist. This is determined in a single pass.

Phase 5 executes the variable defaulting statements made in the declarative section of each model IF AND ONLY IF there are no pending statements from phases 1-4 anywhere in the simulation.

#### default\_self

After all phases are done, the method *default\_self* is called in the top-most model of the simulation, if this method exists.

The first occurrence of each impossible statement will be explained during a failed compilation. Impossible statements include:

- Relations containing undefinable variables (often misspellings).
- Assignments that are dimensionally inconsistent or containing mismatched types.
- Structure building or modifying statements that refer to model parts which cannot exist or that require a type-incompatible argument, refinement, or merge.

## 19.2 DATA TYPE DECLARATIONS

In the spectrum of OO languages, ASCEND is best considered as being class-based, though it is rather more a hybrid. We have atom and model definitions, called *types*, and the compiled objects themselves, called *instances*. ASCEND instances have a record of what type they were constructed from.

Type qualifiers:**UNIVERSAL**

Universal is an optional modifier of all ATOM, CONSTANT, and MODEL definitions. If UNIVERSAL precedes the definition, then ALL instances of that type will actually refer to the first instance of the type that is created. This saves memory and ensures global consistency of data.

Examples of universal type definitions are

```
UNIVERSAL MODEL methane REFINES
generic_component_model;
```

```
UNIVERSAL CONSTANT circle_constant REFINES
real_constant ::= 1{PI};
```

```
UNIVERSAL ATOM counter_1 REFINES integer;
```

Tip: Don't use  
UNIVERSAL variables in  
relations.

It is important to note that, because variables must store information about which relations they occur in, it is a very bad idea to use UNIVERSAL typed variables in relations. The construction and maintenance of the relation list becomes very expensive for universal variables. UNIVERSAL constants are fine to use, though, because there are no relation links for constants.

**19.2.1 MODELS****MODEL**

An ASCEND model has a declarative part and an optional procedural part headed by the METHODS word. Models are essentially containers for variables and relations. We will explain the various statements that can be made within models in Section 19.3 and Section 19.4.

Simple models:**foo**

```
MODEL foo;
  (* statements about foo go here*)
METHODS
  (* METHODS for foo go here*)
END foo;
```

**bar**

```
MODEL bar REFINES foo;
  (*additional statements about foo *)
METHODS
  (* additional METHODS for bar *)
```

```
END bar;
```

### Parameterized Models

(\* 4 \*) Parameterizing models makes them easier to understand and faster for the system to compile. The syntax for a parameterized model vaguely resembles a function call in imperative languages, but it is NOT. When constructing a reusable model, all the constants that determine the sizes of arrays and other structures should be declared in the parameter list so that

- the user knows what is required to reuse the model.
- the compiler knows what values must be set before it should bother attempting to compile the model.

There is no reason that other items could not also go in the parameter list, such as key variables which might be considered inputs or outputs or control parameters in the mathematical application of the model. A simple example of parameterization would be:

#### **column(n,s)**

```
MODEL column(
  ntrays WILL_BE integer_constant;
  components IS_A set of symbol_constant;
);
  stage[1..ntrays] IS_A simple_tray;
END column;
```

#### **flowsheet**

```
MODEL flowsheet;
  tower4size IS_A integer_constant;
  tower4size := 22;
  ct IS_A column(tower4size,['c5','c6']);
  (* additional flowsheet statements *)
END flowsheet;
```

In this example, the column model takes the first argument, `ntrays`, by reference. That is, `ct.ntrays` is an alias for the flowsheet instance `tower4size`. `tower4size` must be compiled and assigned a value before we will attempt to compile the column model instance `ct`. The second argument is taken by value, `['c5','c6']`, and assigned to `components`, a column part that was declared with `IS_A` in the parameter list. There is only one name for this set, `ct.components`. Note that in the flowsheet model there is no part that is a set of `symbol_constant`.

The use of parameters in ASCEND modeling requires some thought, and we will present that set of thoughts in Section 19.5.

Beginners may wish to create new models without parameters until they are comfortable using the existing parameterized library definitions. Parameters are intended to support model reuse and efficient compilation which are not issues in the very earliest phase of developing novel models.

## 19.2.2 SETS

Arrays in ASCEND, as already discussed in Section 19.1.2, are defined over sets. A set is simply an instance with a set value. The elements of sets are NOT instances or sets.

### Set Declaration:

A set is made of either `symbol_constants` or `integer_constants`, so a set object is declared in one of two ways:

```
my_integer_set IS_A set OF integer_constant;
or
my_symbol_set IS_A set OF symbol_constant;
```

**:==**

A set is assigned a value like so:

```
my_integer_set := [1,4];
```

The RHS of such an assignment must be either the name of another set instance or an expression enclosed in square brackets and made up of only set operators, other sets, and the names of `integer_constants` or `symbol_constants`. Sets can only be assigned once.

### Set Operations

#### **UNION[setlist]**

A function taken over a list of sets. The result is the set that includes all the members of all the sets in the list. Note that the result of the UNION operation is an unordered set and the argument order to the union function does not matter. The syntax is:

**+**

```
UNION[list_of_sets]
```

A+B is shorthand for  
UNION[A,B]

Consider the following sets for the examples to follow.

```
A := [1, 2, 3, 5, 9];
B := [2, 4, 6, 8];
```

Then UNION[A, B] is equal to the set [1, 2, 3, 4, 5, 6, 8, 9] which equals [1..6, 8, 9] which equals [[1..9] - [7]].

<b>INTERSECTION[ ]</b>	INTERSECTION[list of set expressions]. Finds the intersection (and) of the sets listed.
<b>*</b>	Equivalent to INTERSECTION[list_of_sets].
A*B is shorthand for INTERSECTION[A,B]	For the sets A and B defined just above, INTERSECTION[A, B] is the set [ 2 ]. The * shorthand for intersection is NOT recommended for use except in libraries no one will look at.
<u>Set difference:</u>	One can subtract one set from another. The result is the first set less any members in the set union of the first and second set. The syntax is
<b>-</b>	<pre>first_set - second_set</pre> <p>For the sets A and B defined above, the set difference A - B is the set [1, 3, 5, 9] while the set difference B - A is the set [ 4 , 6 , 8 ].</p>
<b>CARD[ set ]</b>	Cardinality. Returns an integer constant value that is the number of items in the set.
<b>CHOICE[ set ]</b>	<p>Choose one. The result of running the CHOICE function over a set is an arbitrary (but consistent: for any set instance you always get the same result) single element of that set.</p> <p>Running CHOICE[A] gives any member from the set A. The result is a member, not a set. To make the result into a set, it must be enclosed in square brackets. Thus [ CHOICE[A] ] is a set with a single element arbitrarily chosen from the set A. Good modelers do not leave modeling decisions to the compiler; they do not use CHOICE[ ]. We are stuck with it for historical reasons.</p> <p>To reduce a set by one element, one can use the following</p> <pre>A_less_one IS_A set OF integer; A_less_one ::= A - [CHOICE[A]];</pre>
<b>IN</b>	<p>lhs IN rhs can only be well explained by examples. IN is used in index expressions. If lhs is a simple and not previously defined name, it is created as a temporary loop index which will take on the values of the rhs set definition. If lhs is something that already exists, the result of lhs IN rhs is a boolean value; stare at the model set_example below which demonstrates both IN and SUCH_THAT. If you still are not satisfied, you might examine [[westerbergksets]].</p>

**SUCH\_THAT (\* 4 \*)** Set expressions can be rather clever. We will give a detailed example from chemistry because unordered sets are unfamiliar to most people and set arithmetic is quite powerful. In this example we see arrays of sets and sparse arrays.

```
MODEL set_example;
  (* we define a sparse matrix of reaction coefficient information
  * and the species balance equations. *)
  rxns IS_A set OF integer_constant;
  rxns := [1..3];
  species IS_A set OF symbol_constant;
  species := ['A', 'B', 'C', 'D'];

  reactants[rxns] IS_A set OF symbol_constant; (* species in each rxn_j *)
  reactants[1] := ['A', 'B', 'C'];
  reactants[2] := ['A', 'C'];
  reactants[3] := ['A', 'B', 'D'];

  reactions[species] IS_A set OF integer_constant;
  FOR i IN species CREATE (* rxns for each species i *)
    reactions[i] := [j IN rxns SUCH_THAT i IN reactants[j]];
  END FOR;
  (* Define sparse stoichiometric matrix. Values of eta_ij set later. *)
  FOR j IN rxns CREATE
    FOR i IN reactants[j] CREATE
      (* eta_ij --> mole i/mole rxn j *)
      eta[i][j] IS_A real_constant;
    END FOR;
  END FOR;
  production[species] IS_A molar_rate;
  rate[rxns] IS_A molar_rate; (* mole rxn j/time *)
  FOR i IN species CREATE
    gen_eqn[i]: production[i] =
      SUM[eta[i][j]*rate[j] | j IN reactions[i]];
  END FOR;
END set_example;
```

"|" is shorthand for  
SUCH\_THAT.

The array eta has only 8 elements, and we defined those elements in a set for each reaction. The equation needs to know about the set of reactions for a species i, and that set is calculated automatically in the model's first FOR/CREATE statement.

|

The | symbol is the ASCEND III notation for SUCH\_THAT. We noted that "|" is often read as "for all", which is different in that "for all" makes one think of a FOR loop where the loop index is on the

left of an IN operator. For example, the j loop in the SUM of `gen_eqn[i]` above.

### 19.2.3 CONSTANTS

ASCEND supports real, integer, boolean and character string constants. Constants in ASCEND do not have any attributes other than their value. Constants are scalar quantities that can be assigned exactly once. Constants may only be assigned using the `:=` operator and the RHS expression they are assigned from must itself be constant. Constants do not have subparts. Integer and symbol constants may be used in determining the definitions of sets.

Explicit refinements of the built-in constant types may be defined as exemplified in the description of `real_constant`. Implicit type refinements may be done by instantiating an incompletely defined constant and assigning its final value.

Sets could be considered constant because they are assigned only once, however sets are described separately because they are not quite scalar quantities.

#### **real\_constant**

Real number with dimensionality. Note that the dimensionality of a real constant can be specified via the type definition without immediately defining the value, as in the following pair of definitions.

#### CONSTANT declaration example:

```
CONSTANT molar_weight REFINES real_constant DIMENSION
M/Q;
CONSTANT hydrogen_weight REFINES molar_weight :=
1.004{g/mole};
```

#### **integer\_constant**

Integer number. Principally used in determining model structure. If appearing in equations, integers are evaluated as dimensionless reals. Typical use is inside a MODEL definition and looks like:

```
n_trays IS_A integer_constant;
n_trays := 50;
tray[1..n_trays] IS_A vl_equilibrium_tray;
```

#### **symbol\_constant**

Object with a symbol value. May be used in determining model structure.

#### **boolean\_constant**

Logical value. May be used in determining model structure.



Setting constants**:=**

Constant and set assignment operator.

It is suggested, but not required, that names of all types that refine the built-in constant types have names that end in `_constant`.

```
LHS_list ::= RHS;
```

Here it is required that the one or more items in the LHS be of the same constant type and that RHS is a single-valued expression made up of values, operators, and other constants. The `:=` is used to make clear to both the user and the system what scalar objects are constants.

**19.2.4 VARIABLES**

There are four built-in types which may be used to construct variables: symbol, boolean, integer, and real. At this time symbol types have special restrictions. Refinements of these variable base types are defined with the ATOM statement. Atom types may declare attribute fields with types real, integer, boolean, symbol, and set. These attributes are NOT independent objects and therefore cannot be refined, merged, or put in a refinement clique (ARE\_ALIKED).

**ATOM**

The syntax for declaring a new atom type is

```
ATOM atom_type_name REFINES variable_type
  «DIMENSION dimension_expression»
  «DEFAULT value»; (* note the ; *)
  «initial attribute assignment;»
END atom_type_name;
```

**DEFAULT,  
DIMENSION, and  
DIMENSIONLESS**

The DIMENSION attribute is for variables whose base type is real. It is an optional field. If not defined for any atom with base type real, the dimensions will be left as undefined. Any variable which is later declared to be one of these types will be given *wild card* dimensionality (represented in the interactive display by an asterisk (\*)). The system will deduce the dimensionality from its use in the relationships in which it appears or in the declaring of default values for it, if possible.

`solver_var` is a special case of ATOM and we will say much more about it in Section 19.6.1.

```
ATOM solver_var REFINES real DEFAULT 0.5 {?};
  lower_bound IS_A real;
  upper_bound IS_A real;
  nominal      IS_A real;
  fixed        IS_A boolean;
```

```

fixed := FALSE;
lower_bound := -1e20 {?};
upper_bound := 1e20 {?};
nominal := 0.5 {?};
END solver_var;

```

The default field is also optional. If the atom has a declared dimensionality, then this value must be expressed with units which are compatible with this dimensionality. In the `solver_var` example, we see a DEFAULT value of 0.5 with the unspecified unit `{?}` which leaves the dimensionality wild.

### **real**

Real valued variable quantity. At present, all variables that you want to be attended to by solver tools must be refinements of the type `solver_var`. This is so that modifiable parametric values can be included in equations without treating them as variables. Strictly speaking, this is a characteristic of the solver interface and not the ASCEND language. Each tool in the total ASCEND system may have its own semantics that go beyond the ASCEND object definition language.

### **integer**

Integer valued variable quantity. We find these mighty convenient for use in certain procedural computations and as attributes of `solver_var` atoms.

### **boolean**

Truth valued variable quantity. These are principally used as flags on `solver_vars` and relations. They can also be used procedurally and as variables in logical programming models, subject to the logical solver tool's semantics. (Compare `solver_boolean` and `boolean_var` in Section 19.6.)

### **symbol**

*\*4\** Symbol valued variable quantity. We do not yet have operators for building symbols out of other symbols.

### Setting variables

#### **:=**

Procedural equals differs from the ordinary equals (=) in that it means the left-hand-side (LHS) variables are to be assigned the value of the right-hand-side (RHS) expression when this statement is processed. Processing happens in the last phase of compiling (INSTANTIATION on page 177) or when executing a method interactively through the ASCEND user interface. The order the system encounters these statements matters, therefore, with a later result overwriting an earlier one if both statements have the same the same LHS variable.

Note that variable assignments (also known as “defaulting statements”) written in the declarative section are executed only after an instance has been fully created. This is a frequent source of confusion and errors, therefore we recommend that you **DO NOT ASSIGN VARIABLES IN THE DECLARATIVE SECTION**.

Note that `:=` IS NOT `=`.

We use an ordinary equals (`=`) when defining a real valued equation to state that the LHS expression is to equal the RHS expression at the solution for the model. We use `==` for logical equations.

### Tabular assignments

(\* 4+ \*) Assigning values en masse to arrays of variables that are defined associatively on sets without order presents a minor challenge. The solution proposed in ASCEND IV (but not yet implemented as we’ve not had time or significant user demand) is to allow a tabular data statement to be used to assign the elements of arrays of variables or constants. The DATA statement may be used to assign variables in the declarative or methods section of a model (though we discourage its use declaratively for variable initialization) or to assign constant arrays of any type, including sets, in the declarative section. Here are some examples:

#### **DATA (\* 4+ \*)**

```
MODEL tabular_ex;
lset,rset,cset IS_A set OF integer_constant;
rset := [1..3];
cset := rset - [2];
lset := [5,7];
a[rset][cset] IS_A real;
b[lset][cset][rset] IS_A real_constant;

(* rectangle table *)
DATA FOR a:
COLUMNS 1,3; (*order last subscript cset*)
UNITS {kg/s}, {s}; (* columnar units *)
(* give leading subscripts *)
[1] 2.8, 0.3;
[2] 2.7, 1.3;
[3] 3.3, 0.6;
END DATA;

(* 2 layer rectangle table *)
CONSTANT DATA FOR b:
COLUMNS 1..3; (* order last subscript rset *)
(* UNITS omitted, so either the user gives value in the
table or values given are DIMENSIONLESS. *)
(* ordering over [lset][cset] required *)
[5][1] 3 {m}, 2{m}, 1{m};
```

```
[5][3] 0.1, 0.2, 0.3;
[7][1] -3 {m/s}, -2{m/s}, -1{m/s};
[7][3] 4.1 {1/s}, 4.2 {1/s}, 4.3 {1/s};
END DATA;
```

```
END tabular_ex;
```

For sparse arrays of variables or constants, the COLUMNS and (possibly) UNITS keywords are omitted and the array subscripts are simply enumerated along with the values to be assigned.

## 19.2.5 RELATIONS

Mathematical expression: The syntax for a mathematical expression is any legal combination of variable names and arithmetic operators in the normal notation. An expression may contain any number of matched rounded parentheses, (), to clarify meaning. The following is a legal arithmetic expression:

$$y^2 + (\sin(x) - \tan(z)) * q$$

Each additive term in a mathematical expression (terms are separated by + or - operators) must have the same dimensionality.

An expression may contain an index variable as a part of the calculation if that index variable is over a set whose elements are of type integer. (See the FOR/CREATE and FOR/DO statements below.) An example is:

$$\text{term}[i] = a[i] * x^{(i-1)};$$

### Numerical relations

The syntax for a numeric relation is either

```
optional_label: LHS relational_operator RHS;
or
optional_label: objective_type LHS;
```

*Objective\_type* is either MAXIMIZE or MINIMIZE. RHS and LHS must be one or more variables, constants, and operators in a normal algebraic expression. The operators allowed are defined below and in Section 19.6.3. Variable integers, booleans, and symbols are not allowed as operands in numerical relations, nor are boolean constants. Integer indices declared in FOR/CREATE loops are allowed in relations, and they are treated as integer constants.

Relational operators:

**=, >=, <=, <, >, <>**

These are the numerical relational operators for declarative use.

```
Ftot*y['methane'] = m['methane'];
y['ethanol'] >= 0;
```

Equations must be dimensionally correct.

**MAXIMIZE,  
MINIMIZE**

Objective function indicators.

Binary Operators:

+, -, \*, /, ^. We follow the usual algebraic order of operations for binary operators.

**+**

Plus. Numerical addition or set union.

**-**

Minus. Numerical subtraction or set difference.

**\***

Times. Numerical multiplication or set intersection.

**/**

Divide. Numeric division. In most cases it implies real division and not integer division.

**^**

Power. Numeric exponentiation. If the value of y in  $x^y$  is not integer, then x must be greater than 0.0 and dimensionless.

Unary Operators:

**-**, *ordered\_function()*

**-**

Unary minus. Numeric negation. There is no unary + operator.

***ordered\_function()***  
**)**

unary real valued functions. The unary real functions we support are given in section Section 19.6.3.

Real functions of sets of real terms:

**SUM[term set]**

Add all expressions in the function's list.

For the SUM, the base type real items can be arbitrary arithmetic expressions. The resulting items must all be dimensionally compatible.

An examples of the use is:

```
SUM[y[components]] = 1;
```

or, equivalently, one could write:

```
SUM[y[i] | i IN components] = 1;
```

Empty SUM[] yields wild 0.

When a SUM is compiled over a list which is empty it generates a wild dimensioned 0. This will sometimes cause our dimension checking routines to fail. The best way to prevent this is to make sure the SUM never actually encounters an empty list. For example:

```
SUM[Q[possibly_empty_set], 0{watt}];
```

In the above, the variables  $Q[i]$  (if they exist) have the dimensionality associated with an energy rate. When the set is empty, the 0 is the only term in the SUM and establishes the dimensionality of the result. When the set is NOT empty the compiler will simplify away the *trailing* 0 in the sum.

**PROD[term set]**

Multiply all the expressions in the product's list. The product of an empty list is a dimensionless value, 1.0.

Possible future functions:

**(Not implemented - only under confused consideration at this time.)** The following functions only work in methods as they are not smooth function and would destroy a Newton-based solution algorithm if used in defining a model equation:

**MAX[term set]**

(\* 4+ \*) maximum value on list of arguments

**MIN[term set]**

(\* 4+ \*) minimum value on list of arguments

## 19.2.6 DERIVATIVES IN RELATIONS (\* 4+ \*)

Simply put, we would like to have general partial and full derivatives usable in writing equations, as there are many mathematically interesting things that can be said about both. We have not implemented such things yet for lack of time and because with several implementations (see gPROMS and OMOLA, among others) already out there we can't see too many research points to be gained by more derivative work.

## 19.2.7 EXTERNAL RELATIONS

We cannot document these at the present time. The only reference for them is [[abbottthesis]].

### 19.2.8 CONDITIONAL RELATIONS (\* 4 \*)

The syntax is `CONDITIONAL list_of_relation_statements END CONDITIONAL`;

A `CONDITIONAL` statement can appear anywhere in the declarative portion of the model and it contains only relations to be used as boundaries. That is, these real arithmetic equations are used in expressing logical condition equations via the `SATISFIED` operator. See `LOGICAL FUNCTIONS` on page 215.

### 19.2.9 LOGICAL RELATIONS (\* 4 \*)

#### Logical expression

An expression whose value is `TRUE` or `FALSE` is a logical expression. Such expressions may contain boolean variables. If `A`, `B`, and `laminar` are boolean, then the following is a logical expression:

`A + (B * laminar)`

as is (and probably more clearly)

`A OR (B AND laminar)`

The plus operator acts like an `OR` among the terms while the times operator acts like an `AND`. Think of `TRUE` being equal to 1 and `FALSE` being equal to 0 with the  $1+1=0+1=1+0=1$ ,  $0+0=0$ ,  $1*1=1$  and  $0*1=1*0=0*0=0$ . If  $A = FALSE$ ,  $B = TRUE$  and  $laminar$  is `TRUE`, this expression has the value

`FALSE OR (TRUE AND TRUE) --> TRUE`

or in terms of ones and zeros

`0 + (1 * 1) --> 1.`

Logical relations are then made by putting together logical expressions with the boolean relational operators `==` and `!=`. Since we have no logical solving engine we have not pushed the implementation of logical relations very hard yet.

### 19.2.10 NOTES (\* 4 \*)

Within a `MODEL`(or `METHOD`) definition annotations (hereafter called notes) may be made on a part declared in the `MODEL`, or on

the MODEL (or METHOD) itself. Short notes may be made when defining or refining an object by enclosing the note in "double quotes." Longer notes may be made in a block statement.

Each note is entered in a database with the name of the file, name of MODEL, name of METHOD if applicable, and the language (a kind of keyword) in which the note is written. Users, user interfaces, and other programs may query this database for information on models and simulations. The block notes may include code fragments in other languages that you wish to embed in your MODEL or any other kind of text.

Short notes should be included as you write any model to clarify the roles of parts and variables. All short notes have the language 'inline.' Here are some examples of short notes:

```
L[1..10] "L[i] is the length of the ith rod"
    IS_A distance;
thetaM "angle between horizon and moon",
thetaJ "angle between horizon and jupiter"
    IS_A angle;
car.tires "using car in Minnesota, you betcha"
    IS_REFINED_TO snow_tire;
```

In the second IS\_A statement concerning two angles, we see that a short note in double quotes goes with the name immediately to its left. We also see that the note comes before the comma if the name is part of a list of names. In the third statement, we see that not only simple names but also qualified names may be annotated.

Longer notes are made in block statements of the form below. These blocks can appear in a METHOD or MODEL. These blocks can also be written separately before or after a model as we shall see.

```
NOTES
'language or keyword' list.of, names {
free-form block of text to store in the database
exactly as written.
}
some.other.name {
    this note has the same language or keyword as the
first since we didn't define a new keyword in single
quotes before the name list.
}
'another language' some.other.name {
```



```

    en espanol
  }
  'fortran' SELF {
This model should be solved with subroutine LSODE.
This note demonstrates that "SELF" can be used to
annotate the entire model instead of a named part.
  }
END NOTES;

```

Notes made outside the scope of a model definition look like one of the following:

```

ADD NOTES IN name_of_model;
'language or keyword' list.of, names {
    more text
} (* more than one note may be made in this block if
desired. *)
END NOTES;
ADD NOTES IN name_of_model METHOD name_of_method;
'language or keyword' SELF {
This method proves Fermat's last theorem and makes
toast.
}
'humor' SELF {
ASCEND is not expected to make either proving FLT or
toasting possible.
}
END NOTES;

```

We can add notes to the database before or after defining the annotated model. This is handy for several reasons including:

- Lengthy notes mixed with model and method code can make that code very hard to read.
- Separate notes describing a family of models can be loaded and browsed before loading that library family.
- Users other than the author of a model can annotate that model without fear of introducing typographical errors into the model.

These advantages come with a disadvantage that all documentation has. If you change the model, you ought to change the documentation at the same time. To make finding these documentation locations in need of change easier, the name of the file containing each note is included in the loaded database.

Experience has shown that even documentation embedded directly in models or in other computer programs gets out-dated if the person changing the program is in a hurry and is not required to document properly as part of the task at hand. Neither ASCEND nor any other software system can eliminate the garbage code and documentation that results from undisciplined modeling.

## 19.3 DECLARATIVE STATEMENTS

We have already seen several examples that included declarative statements. Here we will be more systematic in defining things. The statements we describe are legal within the declarative portion of an ATOM or MODEL definition. The declarative portion stops at the keyword METHODS if it is present in the definition or at the end of the definition.

### Statements

Statements in ASCEND terminate with a semicolon (;). Statements may extend over any number of lines. They may have blank lines in the middle of them. There may be several statements on a single line.

### Compound statements

Some statements in ASCEND can contain other statements as a part of them. The declarative compound statements are the ALIASES/IS\_A, CONDITIONAL, FOR/CREATE, SELECT/CASE, and WHEN/CASE statements. The procedural compound statements allowed only in methods are the FOR/DO, FOR/CHECK, SWITCH (\* 4 \*) and the IF statements. Compound statements end with "END *word*;", where *word* matches the beginning of the syntax block, e.g. END FOR. and they can be nested, with some exceptions which are noted later.

### CASE statements are here, finally!

(\*4\*) WHEN/CASE, CONDITIONAL, and SELECT/CASE handle modeling alternatives within a single definition. The easy way to remember the difference is that the first picks which equations to solve WHEN discrete *variables* have certain values, while the second SELECTs which statements to compile based on discrete *constants*. (\* 4 \*) SWITCH statements handle flow of control in methods, in a slightly more generalized form than the C language switch statement.

Type declarations are not compound statements.

MODEL and ATOM type definitions and METHOD definitions are not really compound statements because they require a name following their END word that matches the name given at the beginning of the definition. These definitions cannot be nested.

## ASCEND operator synopses:

We'll start with an extremely brief synopsis of what each does and then give detailed descriptions. It is helpful to remember that an instance may have many names, even in the same scope, but each name may only be defined once.

### **IS\_A**

Constructor. Calls for one or more named instances to be compiled using the type specified. (\* 4 \*) If the type is one that requires parameters, the parameters must be supplied in () following the type name.

### **IS\_REFINED\_TO**

Reconstructor. Causes the already compiled instance(s) named to have their type changed to a more refined type. This causes an incremental recompilation of the instance(s). IS\_REFINED\_TO is not a redefinition of the named instances because refinement can only *add* compatible information. The instances retain all the structure that originally defined them. (\* 4 \*) If the type being refined to requires arguments, these must be supplied, even if the same arguments were required in the IS\_A of the originally less refined declaration of the instance.

### **ALIASES (\* 4 \*)**

Part alternate naming statement. Establishes another name for an instance at the same scope or in a child instance. The equivalent of an ALIASES in ASCEND III is to create another part with the desired name and merge it immediately via ARE\_THE\_SAME with the part being renamed, a rather expensive and unintuitive process.

### **ALIASES/IS\_A (\* 4 \*)**

Creates an array of alternate names for a list of existing instances with some common base type and creates the set over which the elements of the array are indexed. Useful for making collections of related objects in ways the original author of the model didn't anticipate. Also useful for assembling array arguments to parameterized type definitions.

### **WILL\_BE (\* 4 \*)**

Forward declaration statement. Promises that a part with the given type will be constructed by an as yet unknown IS\_A statement above the current scope. At present WILL\_BE is legal only in defining parameters. Were it legal in the body of a model, compiling models would be very expensive.

### **ARE\_THE\_SAME**

Merge. Calls for two or more instances already compiled to be merged recursively. This essentially means combining all the values in the instances into the most refined of the instances and then destroying all the extra, possibly less refined, instances. The remaining instance has its original name and also all the names of the instances destroyed during the merge.

<b>WILL_BE_THE_SAME</b> ( * 4 * )	Structural condition statement restricting objects in a forward declaration. The objects passed to a parameterized type definition can be constrained to have arbitrary parts in common before the parameterized object is constructed.
<b>WILL_NOT_BE_THE_SAME</b> ( * 4 * )	Structural condition statement restricting objects in a forward declaration. We apologize for the length of this key word, but we bet it is easy to remember. The objects passed to a parameterized type definition can be constrained to have arbitrary parts be distinct instances before the parameterized object is constructed. At present the constraint is only enforced when the objects are being passed.
<b>ARE_NOT_THE_SAME</b> ( * 4+ * )	Cannot be merged. We believe it is useful to say that two objects cannot be merged and still represent a valid model. This is not yet implemented, however, mainly for lack of time. The implementation is simple.
<b>ARE_ALIKE</b>	Refinement clique constructor. Causes a group of instances to always be of the same formal type. Refining one of them causes a refinement of all the others. Does not propagate <i>implicit</i> type information, such as assignments to constants or part refinements made from a scope other than the scope of the formal definition.
<b>FOR/CREATE</b>	Indexed execution of other declarative statements. Required for creating arrays of relations and sparse arrays of other types.
<b>FOR/CHECK</b>	Indexed checking of the conditions (WHERE statements) of a parameterized model.
<b>SELECT/CASE</b> ( * 4 * )	Select a subset of statements to compile. Given the values of the specified <i>constants</i> , SELECT compiles all cases that match those values. A name cannot be defined two different ways inside the SELECT statement, but it may be defined outside the case statement and then <i>refined</i> in different ways in separate cases.
<b>CONDITIONAL</b> ( * 4 * )	Describe bounding relations. The relations written inside a CONDITIONAL statement must all be labelled. These relations can be used to define regions in which alternate sets of equations apply using the WHEN statement.
<b>WHEN/CASE</b> ( * 4 * )	When logical <i>variables</i> have certain values, use certain relations or model parts in defining a mathematical problem. The relations are not defined inside the WHEN statement because all the relations must be compiled regardless of which values the logical variables have at any given moment.

**Reminder:**

In the following detailed statement descriptions, we show keywords in capital letters. These words must appear in capital letters as shown in ASCEND statements. We show optional parts to a statement enclosed in double angle brackets (« ») and user supplied names in lower-case *italic* letters. (Remember that ASCEND treats the underscore (\_) as a letter). The user may substitute any name desired for these names. We use names that describe the kind of name the user should use.

Operators in detail:**IS\_A**

This statement has the syntax

```
list_of_instance_names IS_A
model_name «(arguments_if_needed)»;
```

The IS\_A statement allows us to declare *instances* of a given *type* to exist within a model definition. If *type* has not been defined (loaded in the ASCEND environment) then this statement is an error and the MODEL it appears in is irreparably damaged (at least until you delete the type definitions and reload a corrected file). Similarly, if the arguments needed are not supplied or if provably incorrect arguments are supplied, the statement is in error. The construction of the instances does not occur until all the arguments satisfy the definition of *type*.

If a name is used twice in WILL\_BE/IS\_A/ALIASES statements of the same model, ASCEND will complain bitterly when the definition is parsed. Duplicate naming is a serious error. Labels on relations share the same name space as other objects.

**IS\_REFINED\_TO**

This statement has the syntax

```
list_of_instances IS_REFINED_TO
type_name «(arguments_if_needed)»;
```

We use this statement to change the type of each of the instances listed to the type *type\_name*. The modeler has to have defined each member on the list of instances. The *type\_name* has to be a type which refines the types of all the instances on the list.

An example of its use is as follows. First we define the parts called f11, f12 and f13 which are of type flash.

```
f11, f12, f13 IS_A flash;
```

Assume that there exists in the previously defined model definitions the type `adiabatic_flash` that is a refinement of `flash`. Then we can make `fl1` and `fl3` into more refined types by stating:

```
fl1, fl3 IS_REFINED_TO adiabatic_flash;
```

This reconstruction does not occur until the arguments to the type satisfy the definition *type\_name*.

#### **ALIASES (\* 4 \*)**

This statement has the syntax

```
list_of_instances ALIASES instance_name;
```

We use this statement to point at an already existing instance of any type other than `relation`, `logical_relation`, or `when`. For example, say we want a flash tank model to have a variable `T`, the temperature of the vapor-liquid equilibrium mixture in the tank.

```
MODEL tank;
  feed, liquid, vapor IS_A stream;
  state IS_A VLE_mixture;
  T ALIASES state.T;
  liquor_temperature ALIASES T;
END tank;
```

We might also want a more descriptive name than `T`, so `ALIASES` can also be used to establish a second name at the same scope, e.g. `liquor_temperature`.

An `ALIASES` statement will not be executed until the RHS instance has been created with an `IS_A`. The compiler schedules `ALIASES` instructions appropriately and issues warnings if recursion is detected. An array of aliases, e.g.

```
b[1..n], c ALIASES a;
```

is permitted (though we can't think why anyone would want such an array), and the sets over which the array is defined must be completed before the statement is executed. So, in the example of `b` and `c`, the array `b` will not be created until `a` exists and `n` is assigned a value. `b` and `c` will be created at the same time since they are defined in the same statement. This suggests the following rule: if you must use an array of aliases, do not declare it in the same statement with a scalar alias.

The ALIASES RHS can be an element or portion of a larger array with the following exception. The existing RHS instance cannot be a relation or array of relations (including logical relations and whens) because of the rule in the language that a relation instance is associated with exactly one model.

**ALIASES/IS\_A**  
**( \*4\* )**

The ALIASES/IS\_A statement syntax is subject to change, though some equivalent will always exist. We take a set of *symbol\_constant* or *integer\_constant* and pair it with a list of instances to create an array. For the moment, the syntax and semantics is as follows.

```
alias_array_instance[aset] ALIASES
(list_of_instances) WHERE aset IS_A set OF
settype;
```

or

```
alias_array_instance[aset] ALIASES
(list_of_instances) WHERE aset IS_A set OF settype
WITH_VALUE (value_list_matching_settype);
```

*aset* is the name of the set that will be created by the IS\_A to index the array of aliases. If *value\_list\_matching\_set\_type* is not given, the compiler will make one up out of the integers (1..number of names in *list\_of\_instances*) or symbols derived from the individual names given. If the value list is given, it must have the same number of elements as the list of instances does. The value list elements must be unique because they form a set. The list of instances can contain duplicates. If any of these conditions are not met properly, the statement is in error.

ALIASES/IS\_A can be used inside a FOR statement. When this occurs, the definition of *aset* must be indexed and it must be the last subscript of *alias\_array\_instance*. The statement must look like:

```
array_instance[FOR_index][aset[FORindex]]
ALIASES (list_of_instances) WHERE
aset[FORindex] IS_A set OF settype WITH_VALUE
(value_list_matching_settype);
```

Here, as with the unindexed version, the WITH\_VALUE portion is optional.

If this explanation is unclear, just try it out. The compiler error messages for ALIASES/IS\_A are particularly good because we know it is a bit tricky to explain.

**WILL\_BE (\* 4 \*)**

```
instance WILL_BE type_name;
```

The most common use of this forward declaration is as a statement within the parameter list of a model definition. In parameter lists, *list\_of\_instances* must contain exactly one instance. When a model definition includes a parameter defined by WILL\_BE, that model cannot be compiled until a compiled instance at least as refined as the type specified by *type\_name* is passed to it.

(\* 4+ \*) The second potential use of WILL\_BE is to establish that an array of a common base type exists and its elements will be filled in individually by IS\_A or ARE\_THE\_SAME or ALIASES statements. WILL\_BE allows us to avoid costly reconstruction or merge operations by establishing a placeholder instance which contains just enough type information to let us check the validity of other statements that require type compatibility while delaying construction until it is called for by the filling in statements. Instances declared with WILL\_BE are never compiled if they are not ultimately resolved to another instance created with IS\_A. Unresolved WILL\_BE instances will appear in the user interface as objects of type PENDING\_INSTANCE\_*model\_name*. Because of the many implementation and explanation difficulties this usage of WILL\_BE creates, it is not allowed. The ALIASES/IS\_A construct does the same job in a much simpler way.

**ARE\_THE\_SAME**

The format for this instruction is

```
list_of_instances ARE_THE_SAME;
```

All items on the list must have compatible types. For the example in Fig. 1, consider a model where we define the following parts:

```
a1 IS_A A;
b1 IS_A B;
c1 IS_A C;
d1 IS_A D;
e1 IS_A E;
```

Then the following ARE\_THE\_SAME statement is legal

```
a1, b1, c1 ARE_THE_SAME;
```



while the following are not

```
b1, d1 ARE_THE_SAME;
a1, c1, d1 ARE_THE_SAME;
b1, e1 ARE_THE_SAME;
```

When compiling a model, ASCEND will put all of the instances mentioned as being the same into an ARE\_THE\_SAME “clique.” ASCEND lists members of this clique when one asks via the interface for the aliases of any object in a compiled model.

Merging any other item with a member of the clique makes it the same as all the other items in the clique, i.e., it adds the newly mentioned items to the existing clique.

ASCEND merges all members of a clique by first checking that all members of the clique are type compatible. It then changes the type designation of all clique members to that of the most refined member.

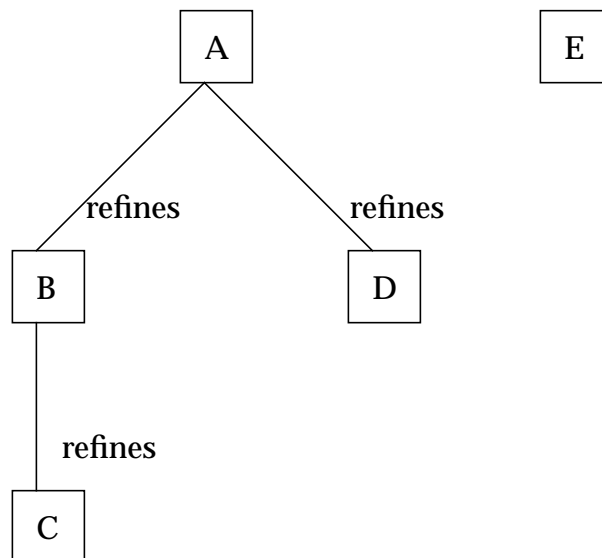


Figure 1. Diagram of the model type hierarchy A,B,C,D,E

It next looks inside each of the instances, all of which are now of the same type, and puts all of the parts with the same name into their respective ARE\_THE\_SAME cliques. The process repeats by processing these cliques until all parts of all parts of all parts, etc.,

are their respective most refined type or discovered to be type incompatible.

There are now lots of cliques associated with the instances being merged. The type associated with each such clique is now either a model, an array, or an atom (i.e., a variable, constant, or set). If a model, only one member of the clique generates its equations. If a variable, it assigns all members to the same storage location.

Note that the values of constants and sets are essentially *type* information, so merging two already assigned constants is only possible if merging them does not force one of them to be assigned a new value. Merging arrays with mismatching ranges of elements is an error.

**WILL\_BE\_THE\_SAME**  
( \* 4 \* )

There is no further explanation of WILL\_BE\_THE\_SAME.

**WILL\_NOT\_BE\_THE\_SAME** ( \* 4 \* )

There is no further explanation of WILL\_NOT\_BE\_THE\_SAME.

**ARE\_NOT\_THE\_SAME**  
( \* 4+ \* )

ARE\_NOT\_THE\_SAME will be documented further when it is implemented.

**ARE\_ALIKE**

The format for this statement is

```
list_of_instance_names ARE_ALIKE;
```

The compiler places all instances in the list into an ARE\_ALIKE clique. It checks that the members are formally type compatible and then it converts each into the most refined type of any instance in the clique. At that point the compiler stops. It does not continue by placing the parts into cliques nor does it merge anything.

There are important consequences of modeling with such a partial merge. The consequences we are about to describe can be much more reliably achieved by use of parameterized types, *when the types are well understood*. When we are exploring new ways of modeling, ARE\_ALIKE still has its uses. When a model and its initial uses are understood well enough to be put into a reusable library, then parameterization and the explicit statement of structural constraints by operators such as WILL\_NOT\_BE\_THE\_SAME should be the preferred method of ensuring correct use.

One consequence of ARE\_ALIKE is to prevent extreme model misuse when configuring models. For example, suppose a modeler creates a new pressure changing model. The modeler is not yet concerned about the type of the streams into and out of the device but does care that these streams are of the same final type. For example, the modeler wants both to be liquid streams if either is or both to be vapor streams if either is. By declaring both to be streams only but declaring the two streams to be alike, the modeler accomplishes this intent. Suppose the modeler merges the inlet stream with a liquid outlet stream from a reactor. The merge operation makes the inlet stream into a liquid stream. The outlet stream, being in an ARE\_ALIKE clique with the inlet stream, also becomes a liquid stream. Any subsequent merge of the outlet stream with a vapor stream will lead to an error due to type incompatibility when ASCEND attempts to compile that merge. Without the ARE\_ALIKE statement, the compiler can detect no such incompatibility unless parameterized models are used.

Another purpose is the propagation of types through a model. Altering the type of the inlet stream through merging it with a liquid stream automatically made the outlet stream into a liquid stream.

If all the liquid streams within a distillation column are alike, then the modeler can make them all into streams with a particular set of components in them and with the same method used for physical property evaluation by merging only one of them with a liquid stream of this type. This is *the primary example* which has been used to justify the existence of ARE\_ALIKE. We have observed that its use makes a column library very difficult to compile efficiently. But since we now have parameterized types to help us keep the column library semantically consistent, ARE\_ALIKE can be left to its proper role: the rapid prototyping of partially understood models. We have yet to see anyone use ARE\_ALIKE in a prototyping context, however.

Finally, because ARE\_ALIKE does not recursively put the parts of ARE\_ALIKEd instances into ARE\_ALIKE cliques, it is possible to ARE\_ALIKE model instances which have compatible formal types but incompatible *implicit* types. This can lead to unexpected problems later and makes the ARE\_ALIKE instruction a source of non-reusability.

#### **FOR/CREATE**

The FOR/CREATE statement is a compound statement that looks like a loop. It isn't, however, necessarily compiled as a loop. What FOR really does is specify an index set value. Its format is:

```
FOR index_variable IN set CREATE
    list_of_statements;
END FOR;
```

This statement can be in the declarative part of the model definition only. Every statement in the list should have at least one occurrence of the index variable, or the statement should be moved outside the FOR to avoid redundant execution. A correct example is

```
FOR i IN components CREATE
    a.y[i], b[i] ARE_THE_SAME;
    y[i] = K[i]*x[i];
END FOR;
```

FOR loops can be nested to produce sparse arrays as illustrated in ARRAYS CAN BE JAGGED on page 173. IS\_A and ALIASES statements are allowed in FOR loops, provided the statements are properly indexed, a new feature in ASCEND IV.

- |                          |  |
|--------------------------|--|
| <b>SELECT/CASE (*4*)</b> | Declarative. Order does not matter. All matching cases are executed. The OTHERWISE is executed if present and no other CASEs match. SELECT is not allowed inside FOR. Writing FOR statements inside SELECT is allowed.   |
| <b>CONDITIONAL (*4*)</b> | Both real and logical relations are allowed in CONDITIONAL statements. CONDITIONAL is really just a shorthand for setting the \$boundary flag on a whole batch of relations, since \$boundary is a write-once attribute invisible through the user interface and methods at this time. |
| <b>WHEN/CASE (* 4 *)</b> | Inside each CASE, relations or model parts to be used are specified by writing, for example, USE mass_balance_1;. The method of dealing with the combined logical/nonlinear model is left to the solver. All matching CASEs are included in the problem to be solved.                  |

## 19.4 PROCEDURAL STATEMENTS

### METHODS

This statement separates the method definitions in ASCEND from the declarative statements. All statements following this statement are to define methods in ASCEND while all before it are for the declarative part of ASCEND. The syntax for this statement is simply

```
METHODS
```

with no punctuation. The next code must be a METHOD or the END of the type being defined. If there are no method definitions, this statement may be omitted.

METHOD definitions for a type can also be added or replaced after the type has been defined. This is to make creating and debugging of methods as interactive as possible. In ASCEND III an instance must be destroyed and recreated each time a new or revised method is added to the type definition. This is a very expensive process when working with models of significant size.

The detailed semantics of method inheritance, addition, and replacement of methods are given at the end of this section.

**ADD METHODS IN**  
**type\_name; (\*4\*)**

This statement allows new methods to be added to an already loaded type definition. The next code must be a METHOD or the END METHODS; statement. If a method of the same name already exists in type\_name, the statement is in error. If other types refine type\_name then the addition follows the method inheritance rules. Any type which inherited methods from type\_name now inherits the methods added to type\_name. If a refinement of type\_name already defines a method ADDED to type\_name, then the existing method in the more refined type is not disturbed.

**REPLACE METHODS**  
**IN type\_name;**  
**(\*4\*)**

This statement allows existing methods to be replaced in an already loaded type definition. The next code must be a METHOD or the END METHODS; statement. If a method of the same name does not exist in type\_name, the statement is in error. If other types refine type\_name then the replacement follows the method inheritance rules. Any type which inherited the old method now inherits the replacement method instead.

**ADD METHODS IN**  
**DEFINITION MODEL;**

This statement allows methods to be added globally. It should be used very sparingly. Library basemodel.a4l contains the example of this statement. Methods in the global model definition are inherited by all models. There is no actual global model definition, but it has a method list for practical purposes.

Initialization routines:

**METHOD**

A method in ASCEND must appear following the METHODS statement within a model. The system executes procedural statements of the method in the order they are written.

At present, there are no local variables or other structures in methods except loop indices. A method may be written recursively,

but there is an arbitrary stack depth limit (currently set to 20 in compiler/initialize.h) to prevent the system from crashing on infinite recursions.

Specifically disallowed in ASCEND III methods are IS\_A, ALIASES, WILL\_BE, IS, IS\_REFINED\_TO, ARE\_THE\_SAME and ARE\_ALIKE statements as these “declare” the structure of the model and belong only in the declarative section.

(\* 4+ \*) In the near future, declarations of local instances (which are automatically destroyed when the method exits) will be allowed. Since methods are imperative, these local structure definitions are processed in the order they are written. Local structures are not allowed to shadow structures in the model context with which the method is called. When local structures are allowed, it will also be possible to define methods which take parameters and return values, thereby making the imperative ASCEND methods a rapid prototyping tool every bit as powerful and easy to use as the declarative ASCEND language.

The syntax for a method declaration is

```
METHOD method_name;
    «procedural statement;» (*one or more*)
END method_name;
```

### Procedural assignment

The syntax is

```
instance_name := mathematical_expression;
or
array_name[set_name] := expression;
or
list_of_instance_names := expression.
```

Its meaning is that the value for the variable(s) on the LHS is set to the value of the expression on the RHS.

DATA statements (DATA (\* 4+ \*) on page 187) can (should, rather) also appear in methods.

### **FOR/DO statement**

This statement is similar to the FOR/CREATE statement except it can only appear in a method definition. An example would be

```
FOR i IN [1..n_stages] DO
    T[i] := T[1] + (i-1)*DT;
    ...
```

```
END FOR;
```

Here we actually execute using the values of *i* in the sequence given. So,

```
FOR i IN [n_stages..1] DO ... END FOR;
```

is an empty loop, while

```
FOR i IN [n_stages..1] DECREASING DO ... END FOR;
```

is a backward loop.

## IF

The IF statement can only appear in a method definition. Its syntax is

```
IF logical_expression THEN
    list_of_statements
ELSE
    list_of_statements
END IF;
```

or

```
IF logical_expression THEN
    list_of_statements
END IF;
```

If the logical expression has a value of TRUE, ASCEND will execute the statements in the THEN part. If the value is FALSE, ASCEND executes the statements in the optional ELSE part. Please use () to make the precedence of AND, OR, NOT, ==, and != clear to both the user and the system.

## SWITCH (\* 4 \*)

Essentially roughly equivalent to the C switch statement, except that ASCEND allows wildcard matches, allows any number of controlling variables to be given in a list, and assumes BREAK at the end of each CASE.

## CALL

External calls are not presently well defined, pending debugging of the EXTERNAL connection prototype originally created by Kirk Abbott.

## RUN

This statement can appear only in a method. Its format is:

```
RUN name_of_method;
```

```

or
RUN part_name.name_of_method;
or
RUN model_type::name_of_method;

```

The named method can be defined in the current model (the first syntax), or in any of its parts (the second syntax). Methods defined in a part will be run in the scope of that part, not at the scope of the RUN statement.

### Type access to methods:

When *model\_type::* appears, the type named must be a type that the current model is refined from. In this way, methods may be defined incrementally. For example:

```

MODEL foo;
    x IS_A generic_real;
METHODS
METHOD specify;
    x.fixed:= TRUE;
END specify;
END foo;

MODEL bar REFINES foo;
    y IS_A generic_real;
METHODS
METHOD specify;
    RUN foo::specify;
    y.fixed := TRUE;
END specify;
END bar;

```

## 19.5 PARAMETERIZED MODELS

Parameterized model definitions have the following general form.

```

MODEL new_type(parameter_list;)
«WHERE (where_list;)»
«REFINES existing_type «(assignment_list;)»»;

```

### 19.5.1 THE PARAMETER LIST

A parameter list is a list of statements about the objects that will be passed into the model being defined when an instance of that model is created by IS\_A or IS\_REFINED\_TO. The parameter list is



designed to allow a complete statement of the necessary and sufficient conditions to construct the parameterized model. The mechanism implemented is general, however, so it is possible to put less than the necessary information in the parameter list if one seeks to confuse the model's reusers. To make parameters easy to understand for users with experience in other computer languages (and to make the implementation much simpler), we define the parameter list as ordered. All the statements in a parameter list, including the last one, must end with a ";". A parameter list looks like:

```
MODEL test (
  x WILL_BE real;
  n IS_A integer_constant;
  p[1..n] IS_A integer_constant;
  q[0..2*n-1] WILL_BE widget;
);
```

Each WILL\_BE statement corresponds to a single object that the user must create and pass into the definition of `test`. We will establish the local name `x` for the first object passed to the definition of `test`. `n` is handled similarly, and it must precede the definition of `p[1..n]`, because it defines the set for the array `p`. Constant types can also be defined with WILL\_BE, though we have used IS\_A for the example `test`.

Each IS\_A statement corresponds to a single constant-valued instance or an array of constant-valued instances that we will create as part of the model we are defining. Thus, the user of `test` must supply an array of constants as the third argument. We will check that the instance supplied is subscripted on the set `[1..n]` and copy the corresponding values to the array `p` we create local to the instance of `test`.

WILL\_BE statements can be used to pass complex objects (models) or arrays of objects. Both WILL\_BE and IS\_A statements can be passed arguments that are *more* refined than the type listed. If an object that is *less* refined than the type listed, the instance of parameterized model `test` will not be compiled. When a parameterized model type is specified with a WILL\_BE statement, NO arguments should be given. We are only interested in the formal type of the argument, not how it was constructed.

### 19.5.2 THE WHERE LIST

We can write structural and equation constraints on the arguments in the WHERE list. Each statement is a `WILL_BE_THE_SAME`, a `WILL_NOT_BE_THE_SAME`, an equation written in terms of sets or discrete constants, or a `FOR/CHECK` statement surrounding a group of such statements. Until all the conditions in the WHERE list are satisfied, an object cannot be constructed using the parameterized definition. If the arguments given to a parameterized type in an `IS_A` or `IS_REFINED_TO` statement cannot possibly satisfy the conditions, the `IS_A` or `IS_REFINED_TO` statement is abandoned by the compiler.

We have not created a `WILL_BE_ALIKE` statement because formal type compatibility in ASCEND is not really a meaningful guarantee of object compatibility. Object compatibility is much more reliably guaranteed by checking conditions on the structure determining constants of a model instance.

### 19.5.3 THE ASSIGNMENT LIST

When we declare constant parameters with `IS_A`, we can in a later refinement of the parameterized model assign their values in the assignment list, thus removing them from the parameter list. If an array of constants is declared with `IS_A`, then we must assign values to ALL the array elements at the same time if we are going to remove them from the parameter list. If an array element is left out, the type which assigns some of the elements and any subsequent refinements of that type will not be compilable.

### 19.5.4 REFINING PARAMETERIZED TYPES

Because we wish to make the parameterized model lists represent all the parameters and conditions necessary to use a model of any type, we must repeat the parameters declared in the ancestral type when we make a refinement. If we did not repeat the parameters, the user would be forced to hunt up the (possibly long) chain of types that yield an interesting definition in order to know the list of parameters and conditions that must be satisfied in order to use a model. We repeat all the parameters of the type being refined before we add new ones. The only exception to this is that parameters defined with `IS_A` and then assigned in the `assignment_list` are not repeated because the user no longer needs to supply these values. A refinement of the model `test` given in Section 19.5.1 follows.

```

MODEL expanded_test (
  x WILL_BE real;
  p[1..n] IS_A integer_constant;
  q[0..2*n-1] WILL_BE better_widget;
  r[0..q[0].k] WILL_BE gizmo;
  ms WILL_BE set OF symbol_constant;
) WHERE (
  q[0].k >= 2;
  r[0..q[0].k].giz_part WILL_BE THE SAME;
) REFINES test(
  n ::= 4;
);

```

In `expanded_test`, we see that the type of the array `q` is more refined than it was in `test`. We see that constants and sets from inside passed objects, such as `q[0].k`, can be used to set the sizes of subsequent array arguments. We see a structural constraint that all the gizmos in the array `r` must have been constructed with the same `giz_part`. This condition probably indicates that the gizmo definition takes `giz_part` as a `WILL_BE` defined parameter.

## 19.6 MISCELLANY

### 19.6.1 VARIABLES FOR SOLVERS

#### **solver\_var**

`Solver_var` is the base-type for all *computable* variables in the current ASCEND system. Any instances of an atom definition that refines `solver_var` are considered potential variables when constructing a problem for one of the solvers.

`Solver_var` has wild card dimensionality. (Wild card means that until ASCEND can decide what its dimensionality is, it has none assigned. ASCEND can decide on dimensionality while compiling or executing.) In `system.a4l` we define the following parts with associated initial values for each:

<u>Attributes:</u>	type	default
<b>lower_bound</b>	real	0.0
<b>upper_bound</b>	real	0.0
<b>nominal</b>	real	0.0

**fixed**                      boolean              FALSE

*lower\_bound* and *upper\_bound* are bounds for a variable which are monitored and maintained during solving. The nominal value the value used to scale a variable when solving. The flag *fixed* indicates if the variable is to be held fixed during solving. All atoms which are refinements of *solver\_var* will have these parts. The refining definitions may reassign the default values of the attributes.

The latest full definition of *solver\_var* is always in the file *system.a4l*.

**generic\_real**                      **One should not declare a variable to be of type *solver\_var*.** The nominal value and bound values will get you into trouble when solving. If you are programming and do not wish to declare variable types, then declare them to be of type *generic\_real*. This type has nominal value of 0.5 and lower and upper bounds of -1.0e50 and 1.0e50 respectively. It is dimensionless. *Generic\_real* is the first refinement of *solver\_var* and is also defined in *system.a4l*.

Kluges for MILPs                      Also defined in *system.a4l* are the types for integer, binary, and semi-continuous variables.

***solver\_semi*,**  
***solver\_integer*,**  
***solver\_binary***                      We define basic refinements of *solver\_var* to support solvers which are more than simply algebraic. Various mixed integer-linear program solvers can be fed *solver\_semi* based atoms defining semi-continuous variables, *solver\_integer* based atoms defining integer variables, and *solver\_binary* based atoms defining binary variables.

Integers are relaxable.                      All these types have associated boolean flags which indicate that either the variable is to be treated according to its restricted meaning or it is to be relaxed and treated as a normal continuous algebraic variable.

Kluges for ODEs                      We have an alternate version of *system.a4l* called *ivpsystem.a4l* which adds extra flags to the definition of *solver\_var* in order to support initial value problem (IVP) solvers (integrators). Integration in the ASCEND IV environment is explained in another chapter.

***ivpsystem.a4l***                      Having *ivpsystem.a4l* is a temporary, but highly effective, way to keep people who want to use ASCEND only for algebraic purposes from having to pay for the IVP overhead. Algebraic users load *system.a4l*. Users who want both algebraic and IVP capability load *ivpsystem.a4l* instead of *system.a4l*. This method is temporary because part of the extended definition of ASCEND IV is that

differential calculus constructs will be explicitly supported by the compiler. The calculus is not yet implemented, however.

## 19.6.2 SUPPORTED ATTRIBUTES

( \* 4+ \* )

The solver\_var, and in fact most objects in ASCEND IV, should have built-in support for (and thereby efficient storage of) quite a few more attributes than are defined above. These built-in attributes are not instances of any sort, merely values. The syntax for naming one of these supported attributes is:

*object\_name* . \$*supported\_attribute\_name*.

Supported attributes may have symbol, real, integer, or boolean values. Note that the \$ syntax is essentially the same as the derivative syntax for relations; derivatives are a supported attribute of relations. The supported attributes must be defined at the time the ASCEND compiler is built. The storage requirement for a supported boolean attribute is 1 bit rather than the 24 bytes required to store a run-time defined boolean flag. Similarly, the requirement for a supported real attribute is 4 or 8 bytes instead of 24 bytes.

## 19.6.3 SINGLE OPERAND REAL FUNCTIONS:

<b>exp()</b>	exponential (i.e., $\exp(x) = e^x$ )
<b>ln()</b>	log to the base e
<b>sin()</b>	sine. argument must be an angle.
<b>cos()</b>	cosine. argument must be an angle.
<b>tan()</b>	tangent. argument must be an angle.
<b>arcsin()</b>	inverse sine. return value is an angle.
<b>arccos()</b>	inverse cosine. return value is an angle.
<b>arctan()</b>	inverse tangent. return value is an angle.
<b>erf()</b>	error function (not available from Microsoft Windoze)
<b>sinh()</b>	hyperbolic sine
<b>cosh()</b>	hyperbolic cosine

<b>tanh( )</b>	hyperbolic tangent
<b>arcsinh( )</b>	inverse hyperbolic sine
<b>arccosh( )</b>	inverse hyperbolic cosine
<b>arctanh( )</b>	inverse hyperbolic tangent
<b>lnm( )</b>	<p>modified ln function. This lnm function is parameterized by a constant a, which is typically set to about 1.e-8. lnm(x) is defined as follows:</p> <p><math>\ln(x)</math> for <math>x &gt; a</math></p> <p><math>(x-a)/a + \ln(a)</math> for <math>x \leq a</math>.</p> <p>Below the value a (default setting is 1.0e-8), lnm takes on the value given by the straight line passing through <math>\ln(a)</math> and having the same slope as <math>\ln(a)</math> has at a. This function and its first derivative are continuous. The second derivative contains a jump at a.</p> <p>The lnm function can tolerate a negative argument while the ln function cannot. At present the value of a is controllable via the user interface of the ASCEND solvers.</p>
Operand dimensionality must be correct.	<p>The operands for an ASCEND function must be dimensionally consistent with the function in question. Most transcendental functions require dimensionless arguments. The trigonometric functions require arguments with dimensionality of plane angles, P. ASCEND functions return dimensionally correct results.</p> <p>The operands for ASCEND functions are enclosed within rounded parentheses, (). An example of use is:</p> <p><math>y = A * \exp(-B/T);</math></p>
<u>Discontinuous functions:</u>	Discontinuous functions may destroy a Newton-based solution algorithm if used in defining a model equation. We strongly suggest considering alternative formulations of your equations.
<b>abs( )</b>	absolute value of argument. Any dimensionality is allowed in an abs() function.

### 19.6.4 LOGICAL FUNCTIONS

**SATISFIED( ) (\*4\*)** SATISFIED(relation\_name,tolerance) returns TRUE if the relation named has a residual value less than the real value, tolerance, given. If the relation named is a logical relation, the tolerance should not be specified, since logical relations evaluate directly to TRUE or FALSE.

### 19.6.5 UNITS DEFINITIONS

As noted in 19.1.2, ASCEND will recognize conversion factors when it sees them as {units). These units are built up from the basic units, and new units can be defined by the user. Note that the assignment  $x := 0.5 \{100\}$ ; yields  $x == 50$ , and that there are no 'offset conversions,' e.g.  $F = 9/5C + 32$ . Please keep unit names to 20 characters or less as this makes life pretty for other users

One or more unit conversion factors can be defined with the UNITS keyword. A unit of measure, once defined, stays in the system until the system is shut down. A measuring unit cannot be defined differently without first shutting down the system, but duplicate or equivalent definitions are quietly ignored.

A UNITS declaration can occur in a file by itself, inside a model or inside an atom. UNITS definitions are parsed immediately, they will be processed even if a surrounding MODEL or ATOM definition is rejected. Because units and dimensionality are designed into the deepest levels of the system, a unit definition must be parsed before any atoms or relations use that definition. It is good design practice to keep customized unit definitions in separate files and REQUIRE those files at the beginning of any file that uses them. Unit definitions are made in the form, for example:

```
UNITS (* several unit definitions could be here. *)
      ohm = {kilogram*meter^2/second^3/ampere^2};
END UNITS;
```

The standard units library, measures.a4l, is documented in Chapter 20.







# CHAPTER 20 UNITS LIBRARY

## 20.1 UNITS

This chapter defines the dimensions and units and all the attendant conversion factors. Note that all conversions are simply multiplicative. This information is from the file `models/measures.a4l` in the ASCEND source code.

Note that units can be easily defined to suit the needs of local users. We are always on the lookout for new and interesting units, so if you have some send them in. From `measures.a4l` we have:

## 20.2 THE BASIC UNITS IN AN EXTENDED SI MKS SYSTEM

These units (kilogram, mole, et c.) are associated with the dimensionality listed here (M, Q, et c.) by the ASCEND IV C code. All other units are derived from these by multiplication factors. Use of units other than these requires loading unit definitions, either from `measures.a4l` or from another ASCEND file containing a UNITS declaration. The system rejects loudly any model or variable definition using undefined units.

```
define kilogram    M; (* internal mass unit SI *)
define mole        Q; (* internal quantity unit SI *)
define second      T;  (* internal time unit SI *)
define meter       L;  (* internal length unit SI *)
define Kelvin      TMP; (* internal temperature unit SI *)
define currency    C;  (* internal currency unit *)
define ampere      E;  (* internal electric current unit SI suggested *)
```

```

define candela    LUM; (* internal luminous intensity unit SI *)
define radian     P;   (* internal plane angle unit SI suggested *)
define steradian  S;   (* internal solid angle unit SI suggested *)

```

## 20.3 UNITS DEFINED IN MEASURES.A4L, THE DEFAULT SYSTEM UNITS LIBRARY OF ATOMS.A4L.

### distance

pc = 3.08374e+16\*meter;

parsec = pc;

kpc = 1000\*pc;

Mpc = 1e6\*pc;

km = meter\*1000;

m = meter;

dm = meter/10;

cm = meter/100;

mm = meter/1000;

um = meter/1000000;

nm = 1.e-9\*meter;

kilometer = km;

centimeter = cm;

millimeter = mm;

micron=um;

nanometer = nm;

angstrom = m/1e10;

fermi = m/1e15;

mi = 1609.344\*meter;

yd = 0.914412\*meter;

ft = 0.304804\*meter;

inch = 0.0254\*meter;

mile = mi;

yard = yd;

feet = ft;

foot = ft;

in = inch;

#### mass

metton = kilogram \*1000;

mtton = kilogram \*1000;

kg = kilogram;

g = kilogram/1000;

gram = g;

mg = g/1000;

milligram = mg;

ug= kilogram\*1e-9;

microgram = ug;

ng=kilogram\*1e-12;

nanogram=ng;

pg=kilogram\*1e-15;

picogram=pg;

amu = 1.661e-27\*kilogram;

time

lbm = 4.535924e-1\*kilogram;

ton = lbm\*2000;

oz = 0.028349525\*kilogram;

slug = 14.5939\*kilogram;

yr = 31557600\*second;

wk = 604800\*second;

dy = 86400\*second;

hr = 3600\*second;

min = 60\*second;

sec = second;

s = second;

ms = second/1000;

us = second/1e6;

ns = second/1e9;

ps = second/1e12;

year = yr;

week = wk;

day = dy;

hour = hr;

minute = min;

millisecond = ms;

microsecond = us;

nanosecond = ns;

picosecond = ps;

molecular quantities      kg\_mole=1000\*mole;

g\_mole = mole;

gm\_mole = mole;

kmol = 1000\*mole;

mol = mole;

mmol = mole/1000;

millimole=mmol;

umol = mole/1e6;

micromole=umol;

lb\_mole = 4.535924e+2\*mole;

temperature      K = Kelvin;

R = 5\*Kelvin/9;

Rankine = R;

money      dollar = currency;

US = currency;

USdollar=currency;

CR = currency;

credits=currency;

reciprocal time      rev = 1.0;

(frequency)      cycle = rev;

rpm = rev/minute;

rps = rev/second;

hertz = cycle/second;

Hz = hertz;

area

ha = meter<sup>2</sup>\*10000;

hectare=ha;

acre= meter<sup>2</sup>\*4046.856;

volume

l = meter<sup>3</sup>/1000;

liter = l;

ml = liter/1000;

ul = liter/1e6;

milliliter = ml;

microliter = ul;

hogshead=2.384809e-1\*meter<sup>3</sup>;

cuft = 0.02831698\*meter<sup>3</sup>;

impgal = 4.52837e-3\*meter<sup>3</sup>;

gal = 3.785412e-3\*meter<sup>3</sup>;

barrel = 42.0\*gal;

gallon = gal;

quart = gal/4;

pint = gal/8;

cup = gal/16;

floz = gal/128;

force

N = kilogram\*meter/second<sup>2</sup>;

newton = N;

dyne = N\*1.0e-5;

pn=N\*1e-9;

picoNewton=pn;

lbf = N\*4.448221;

pressure

Pa = kilogram/meter/second^2;

MPa = 1.0e+6\*Pa;

bar =1.0e+5\*Pa;

kPa = 1000\*Pa;

pascal = Pa;

atm = Pa\*101325.0;

mmHg = 133.322\*Pa;

torr = 133.322\*Pa;

psia = 6894.733\*Pa;

psi = psia;

ftH2O = 2989\*Pa;

energy

J = kilogram\*meter^2/second^2;

joule = J;

MJ = J \* 1000000;

kJ = J \* 1000;

mJ=J\*1.0e-3;

uJ=J\*1.0e-6;

nJ=J\*1.0e-9;

milliJoule=mJ;



power

microJoule=uJ;

nanoJoule=nJ;

erg = J\*1.0e-7;

BTU = 1055.056\*J;

pCu = BTU \* 1.8;

cal = J\*4.18393;

calorie = cal;

kcal=1000\*calorie;

Cal=1000\*calorie;

W = J/second;

EW = 1.0e+18\*W;

PW = 1.0e+15\*W;

TW = 1.0e+12\*W;

GW = 1.0e+9\*W;

MW = 1.0e+6\*W;

kW = 1000\*W;

mW = W/1000;

uW = W/1000000;

nW = W/1e9;

pW = W/1e12;

fW = W/1e15;

aW = W/1e18;

terawatt = TW;

gigawatt = GW;

megawatt = MW;

kilowatt = kW;

watt = W;

milliwatt = mW;

microwatt = uW;

nanowatt = nW;

picowatt = pW;

femtowatt = fW;

attowatt = aW;

hp= 7.456998e+2\*W;

#### absolute viscosity

poise = Pa\*s;

cP = poise/100;

#### electric charge

coulomb=ampere\*second;

C = coulomb;

coul = coulomb;

mC = 0.001\*C;

uC = 1e-6\*C;

nC = 1e-9\*C;

pC = 1e-12\*C;

#### miscellaneous electro- magnetic fun

V = kilogram\*meter^2/second^3/ampere;

F = ampere^2\*second^4/kilogram/meter^2;

ohm = kilogram\*meter^2/second^3/ampere^2;

$\text{mho} = \text{ampere}^2 \cdot \text{second}^3 / \text{kilogram} / \text{meter}^2;$

$S = \text{mho};$

$\text{siemens} = S;$

$A = \text{ampere};$

$\text{amp} = \text{ampere};$

$\text{volt} = V;$

$\text{farad} = F;$

$\text{mA} = A / 1000;$

$\text{uA} = A / 1000000;$

$\text{kV} = 1000 * V;$

$\text{MV} = 1e6 * V;$

$\text{mV} = V / 1000;$

$\text{mF} = 0.001 * F;$

$\text{uF} = 1e-6 * F;$

$\text{nF} = 1e-9 * F;$

$\text{pF} = 1e-12 * F;$

$\text{kohm} = 1000 * \text{ohm};$

$\text{Mohm} = 1e6 * \text{ohm};$

$\text{kS} = 1000 * S;$

$\text{mS} = 0.001 * S;$

$\text{uS} = 1e-6 * S;$

$\text{Wb} = V * \text{second};$

$\text{weber} = \text{Wb};$

```
tesla = Wb/m^2;
```

```
gauss = 1e-4*tesla;
```

```
H = Wb/A;
```

```
henry = H;
```

```
mH = 0.001*H;
```

```
uH = 1e-6*H;
```

### numeric constants of some interest

To set a variable or constant to these, the code is (in the declarations):

```
ATOM unspecified_unitwise REFINES real;
END unspecified_unitwise;
MODEL gizmo;
  x IS_A unspecified_unitwise;
  (* if some other atom type is more appropriate, by all
   * means, use it.
   *)
  x := 1 {PI};
  ...
END gizmo;
```

Note that you generally should NOT declare an ATOM and a variable in order to use a constant in an equation. For example, to use GAS\_C in a simple gas law equation, write:

```
P*v = n*1{GAS_C}*T;
(* GAS_C is the thermochemical gas constant "R"*)
```

```
molecule = 1.0;
PI=3.141592653589793;           # Circumference/Diameter ratio
EULER_C = 0.57721566490153286;  # euler gamma
GOLDEN_C = 1.618033988749894;   # golden ratio
HBAR = 1.055e-34*J*second;      # Reduced Planck's constant
PLANCK_C = 2*PI*HBAR;           # Planck's constant
LIGHT_C = 2.99793e8 * meter/second; # Speed of light in vacuum
MU0 = 4e-7*PI*kg*m/(C*C);       # Permeability of free space
EPSILON0 = 1/LIGHT_C/LIGHT_C/MU0; # Permittivity of free space
BOLTZMAN_C = 1.3805e-23 * J/K;   # Boltzman's constant
AVOGADRO_C = 6.023e23 *molecule/mole; # Avogadro's number of molecules
GRAVITY_C = 6.673e-11 * N*m*m/(kg*kg); # Newtons gravitational constant
GAS_C = BOLTZMAN_C*AVOGADRO_C;   # Gas constant
INFINITY=1.0e38;                 # darn big number;
eCHARGE = 1.602e-19*C;           # Charge of an electron
```

```

EARTH_G = 9.80665 * m/(s*s);           # Earth's gravitational field, somewhere
eMASS = 9.1095e-31*kilogram;           # Electron rest mass, I suppose
pMASS = 1.67265e-27*kilogram;          # Proton mass

```

constant based  
conversions

```

eV = eCHARGE * V;

keV = 1000*eV;

MeV = 1e6*eV;

GeV = 1e9*eV;

TeV = 1e12*eV;

PeV = 1e15*eV;

EeV = 1e18*eV;

lyr = LIGHT_C * yr;                    # Light-year

oersted = gauss/MU0;

```

subtly dimensionless  
measures

```

rad = radian;

srad = steradian;

deg = radian*1.74532925199433e-2;

degrees = deg;

grad = 0.9*deg;

arcmin = degrees/60.0;

arcsec = arcmin/60.0;

```

light quantities

```

cd = candela;

lm = candela*steradian;

lumen = lm;

lx = lm/meter^2;

lux= lx;

```

miscellaneous rates

gpm = gallon/minute;

time variant  
conversions

MINIMUMWAGE = 4.75\*US/hr;

SPEEDLIMIT = 65\*mi/hr;

Conversions we'd like to see, but probably won't:

milliHelen = beauty/ship;



## CHAPTER 21 BRIEF HISTORY OF ASCEND

ASCEND is an acronym which stands for **Advanced System for Computations in ENgineering Design**<sup>1</sup>. The name ASCEND first appeared in print in 1978. The ASCEND programs are a series of modeling systems that Arthur Westerberg and his graduate students at Carnegie Mellon University have developed since that time.

### ASCEND I

Dean Benjamin developed the first ASCEND system. It was an interactive system in Fortran. Chemical engineering students at Carnegie Mellon University used this system from about 1978 to 1982 to carry out multicomponent flash calculations. It supported the senior design project.

### ASCEND II

Almost in parallel, Michael Locke developed the ASCEND II simulation system for his PhD thesis [1981]. ASCEND II allowed users to create models by configuring them using predefined types of parts. System maintainers defined the library of types, each in the form of seven handcrafted Fortran subroutines. These routines computed the space needed for the data when instantiating a part, generated numerical values for the partial derivatives and the residuals of the equations that the part instance provided to the overall model, generated proper variable and equation scaling and the like. Michael Locke used this system to create models involving a few thousand equations to test variants of the Sequential Quadratic Programming algorithm. Tom Berna and he developed for optimizing structured engineering systems. Selahattin Kuru also used ASCEND II to generate and test solution algorithms for dynamic simulation that he subsequently developed for his PhD. Two companies used the software architectural design of ASCEND II to create their own internal equation-based modeling systems.

Experience at this time demonstrated that models involving several thousands of equations were solvable and could even be efficiently optimized. The question of interest moved from how to solve large equation-based models to how to aid an engineer to pose them, debug them and get them to solve.

---

1. ASCEND originally stood for “Advanced System for Chemical ENgineering Design” but the second generation system and following are not discipline specific, thus the name change.



In 1983 Dean Benjamin proposed the first version of a modeling language for posing complex models. Larry Gaydos and Art Westerberg further developed this language in the spring of 1984.

### ASCEND III

In 1984 Peter Piela undertook a PhD project with Art Westerberg to “reduce the time needed to create and solve a complex model by one order of magnitude.” He developed what became ASCEND III. He had the help of two Carnegie Mellon University undergraduate students, Tom Epperly and Karl Westerberg, and of Roy McKelvey, a member of the faculty in the Design Department in Fine Arts. This team developed this system on the Apollo workstation and in Pascal. It comprised three parts: a modeling language and compiler, an interactive user interface and a suite of solvers. The language used object-oriented principles, with the exception of hiding of information. Modelers define types to create model definitions. A type (called a model in ASCEND) is a collection of variables and complex parts whose types are previously defined and the definition of the equations that model is to supply. A model can also be the refinement of a previously defined type. The language fully supported associative arrays and sets. For example, a distillation column is an array of trays. It also supported deferred binding by allowing one to reach inside a part and alter one of its parts to be a more refined type. The language and its compiler obviated the need to have a system programmer write the seven subroutines needed in ASCEND II.

The interactive user interface supplied the user with organized access to the many tools in the ASCEND III system. There were tools to load model definitions, to compile them, to browse them, to solve them, to probe them, to manipulate the display units (e.g., ft/s) for reporting variable values, to create reports and to run methods on them. One could even point at a part and ask that it be made into a more refined type (triggering the compiler to restart). As previously solved values were not overwritten, they became the starting point for the more complex model. Thus one could creep up on the solution by solving more and more complex versions of a model. Many of the tools were there specifically to aid the user in debugging their models as they tried to solve them. A tool could tell a user that the model appeared to be singular and why. Another set of tools aided in picking a consistent set of variables to fix before solving. Browsing allowed the user to look at all parts of the model. It was easy to check the configuration of a model. One could ask that parts of a model be solved one at a time.

Experience by Peter Piela, Oliver Smith, Neil Carlberg and Art Westerberg with ASCEND III demonstrated very clearly that *skilled* modelers could develop, debug and solve very complex models much more rapidly than they could with previously available tools, easily

meeting the original target of a order of magnitude reduction in time required.

### ASCEND IIIc

In the fall of 1992, Kirk Abbott and Ben Allan approached Art Westerberg and said they wanted to convert the ASCEND III system from Pascal into C. They would also use Tcl/Tk for the interface. With these changes, the system would then run on most Unix workstations. Tom Epperly and Karl Westerberg had already created a C version for the compiler and solver. Abbott and Allan wanted to do this conversion even after they were warned that converting the system would take time that they could be using to do more apparently relevant work to complete their PhD theses. They insisted<sup>2</sup>. They were aided by Tom Epperly who, although located remotely, worked with them on the compiler. In eight months and putting in excessively long hours, they had a working system that could mimic most of the capabilities of the ASCEND III system.

Several students and a few people outside CMU could now use the system for modeling. Bob Huss and Boyd Safrit performed the hardest testing when they used ASCEND IIIc to model nonideal distillation processes. They developed and solved models involving up to 15,000 equations. Using a rudimentary capability for solving initial value problems, Safrit also solved dynamic models.

With use came the recognition of a need for improvements.

Attempts to teach ASCEND to others showed that it was a great system to speed up the modeling process for experts. Nonexperts found it nearly impossible to reuse models contained in the ASCEND libraries. The library for computing the thermodynamic properties of mixtures was particularly elegant but almost impossible to reuse. Modelers would reinvent their own properties models quickly, unable to use the library models.

Models larger than about 17,000 equations took more space than our largest workstation could provide. The models by Huss and Safrit were pushing the limits. Abbott and Allan established the goal to increase the size possible by a factor of at least ten, i.e., to about a quarter of a million equations. ASCEND needed to solve models more quickly. Without counting the increases coming from faster and larger hardware, the goal here too was a factor of ten. If solving were to be that fast, then compiling would stand out as unacceptably slow. The goal: ten times faster.

Abbott, with Allan, exposed a new style for modeling in ASCEND. He created prototypes of the various repeating types that occur in a model.

---

2. It should be understood that Art Westerberg was thrilled they insisted on doing this conversion.

The compiled equations and other data structures to define these prototypes then became available for all subsequent instances of parts that were of the same type as the prototype. Only the instance data needed to be developed separately. Demonstrated impact on compile times was dramatic.

Abbott, with Allan, looked at how to speed up the solving times. The new twist was to use the model structure as defined by the model definition to expose a global reordering for the model equations before presenting them for solution. The time to solve the linear Newton equations as the inner loop of solving nonlinear equations dropped by factors of 5 to 10.

#### ASCEND IV

Ben Allan has taken a lead role and worked with Mark Thomas, Vicente Rico-Ramirez and Ken Tyner to produce the next version of ASCEND, ASCEND IV. Playing the role of tester, an undergraduate, Jennifer Perry, demonstrated that Allan's introduction of parameterized types dramatically increased the reusability of the model libraries, converting it into an almost automatable exercise. Adding language constructs to permit the modeler to state what constitutes misuse of a model leads to the system generating diagnostics the model itself defines. Allan also completely revised the data structures and the interface between ASCEND and its solvers so that adding new solvers is much less work and so the solvers in ASCEND themselves become separable from ASCEND and usable by others.

Allan also defined the addition of NOTES to ASCEND which are like methods except they are not understood by the ASCEND system itself. Rather they can be passed to programs outside ASCEND. An example includes documentation notes which a documentation manager can use to compose answers to queries about what is in an ASCEND model. Another is a note that contains a bitmap description of a part that an external package could use to draw a symbol of that part.

ASCEND IV can now handle discrete variables and constants (logical, binary, symbolic, and integer). It supports the solver directing that parts of the model be excluded when solving such as when solving using implicit enumeration (dynamic model modification). CONOPT is now attached for optimization. The standard solver is rapidly becoming much more robust. ASCEND IV can generate a GAMS model corresponding the ASCEND model, giving access to solvers GAMS has that ASCEND does not.

While not quite there just yet, the goal to compile and solve 250,000 equations on a 150 megahertz workstation having about 250 megabytes of fast memory in a few tens of minutes is in sight.

# INDEX

## Symbols

\$ 167  
 . 168  
 : label indicator 168

## Numerics

4 167  
 4+ 165

## A

ambiguity 177  
 ARE\_THE\_SAME  
     merging streams 156  
 array 172  
     sparse 173  
 ASCEND III 166  
 associative array 172

## B

BNF 165  
 braces 167, 171  
 brackets 168

## C

colon 168  
 comment 167  
 compiling 177  
 Conditional Modeling 122  
 currency 170

## D

Data Type 178  
 declarative 179  
 default\_self 178  
 dimensionality 169  
     wild card 170  
 dot 168  
 double colon 168  
 double dot 168

## E

efficiency  
     compiling equations 167  
 electric current 169  
 environment variables 23

## F

flash  
     simple 153

FLEX 165  
 Flowsheeting 144  
 FOR 174  
 formal definition 175  
 ftp 165

## G

gas constant 171

## I

implicit (anonymous) type or class 176  
 impossible equations 177  
 impossible statement 178  
 index variable 174  
 instance 175  
 instantiation 177  
 integer range 168  
 integration 118  
 IS\_A 166

## J

jagged array 173

## L

label 174  
 length 169  
 LHS 167  
 limit  
     strings 171  
 luminous intensity 170

## M

mass 169  
 Matlab 174  
 matrix algebra 174  
 MAX\_INTEGER 168  
 MAX\_REAL 169  
 METHODS 179  
 mixer 150  
 MKS 169  
 model 179

## N

name list 175  
 names 167  
     compound 167

## O

ordered  
     sets not 172  
 ordered list 175

**P**

parameterized model 180  
parentheses 167  
pending statements 177  
plane angle 170

**Q**

quantity 169

**R**

reactor  
    simple 152  
reusable 180  
RHS 167

**S**

semantics 164  
semicolon 168  
sets 172  
solid angle 170  
sparse array 173  
splitter  
    simple 155  
structural parameter 177

symbol 171

syntax 164

**T**

tabular data 118  
temperature 170  
time 169  
type 175

**U**

unit  
    syntax 170  
UNIVERSAL 179

**V**

value  
    integer 168  
    real 169

**W**

WHEN 122

**Y**

YACC 165