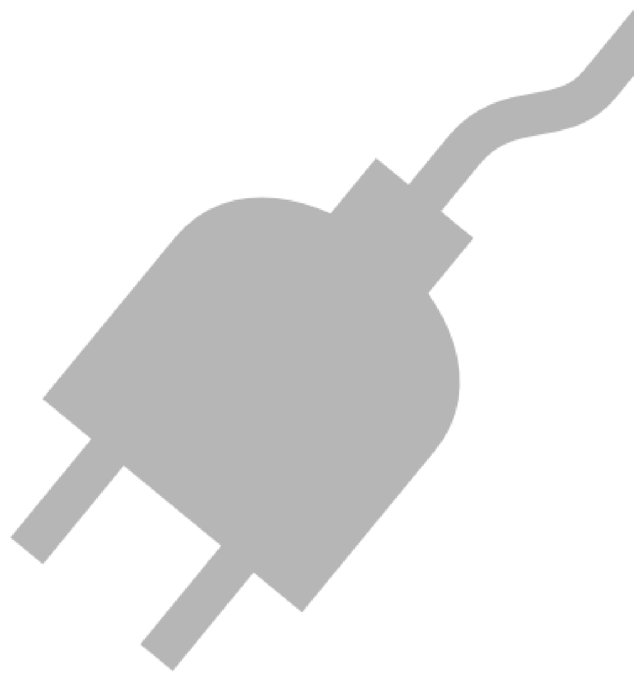
AVF Plug-in User's Manual

Symbolic Nuclear Analysis Package (SNAP)



June 2008

Applied Programming Technology, Inc.

240 Market St., Suite 208
Bloomsburg PA 17815-1951

AVF Plug-in Users Manual

Applied Programming Technology, Inc.
by Ken Jones and Dustin Vogt
Copyright © 2008

***** Disclaimer of Liability Notice ******

The Nuclear Regulatory Commission and Applied Programming Technology, Inc. provide no express warranties and/or guarantees and further disclaims all other warranties of any kind whether statutory, written, oral, or implied as to the quality, character, or description of products and services, its merchantability, or its fitness for any use or purpose. Further, no warranties are given that products and services shall be error free or that they shall operate on specific hardware configurations. In no event shall the US Nuclear Regulatory Commission or Applied Programming Technology, Inc. be liable, whether foreseeable or unforeseeable, for direct, incidental, indirect, special, or consequential damages, including but not limited to loss of use, loss of profit, loss of data, data being rendered inaccurate, liabilities or penalties incurred by any party, or losses sustained by third parties even if the Nuclear Regulatory Commission or Applied Programming Technology, Inc. have been advised of the possibilities of such damages or losses.

Table of Contents

1. Introduction 1
    1.1. Legacy Tools 1
    1.2. SNAP and the AVF Plug-in 2
2. Files Bundled with AVF 3
3. AVF Model Components 4
    3.1. Regression 4
    3.2. AV Script 5
    3.3. Component Validation 15
4. Submitting Jobs 16
    4.1. Regression Jobs 16
    4.2. Report Jobs 18
    4.3. AVScript Jobs 19
5. Other Features 25
    5.1. Importing Legacy AVScript Inputs 25
    5.2. Importing TRACE ATF 26
    5.3. Editing Graphs from AptPlot 28
    5.4. Figure Templates 28
A. Report Generation 29
    A.1. Creating Reports 29
    A.2. Report Definition 30
    A.3. Statistics File 33
B. Developing AVF Plug-ins 35
    B.1. CaseSupport 35
    B.2. CaseDefinition 36
Index 39

Chapter 1. Introduction

The Automated Validation Framework (AVF) plug-in is a SNAP plug-in designed to be a generic replacement for assessment and validation tools such as the TRACE ATF and AVScript. This section discusses the legacy tools and how AVF provides largely identical functionality.

1.1. Legacy Tools

1.1.1. AVScript

The Automated Validation Script (AVScript) is a Perl application used to automate running codes, generating plots, and computing figures of merit. It supports running RELAP5, TRACE, and TRAC-B, as well as loading data directly from ASCII and NRC Databank files. AVScript also allowed axial plots to be created easily with AcGrace.

AVScript is incorporated directly into the AVF plug-in. This functionality has been made generic and pluggable so that additional codes can be supported without any modifications to the AVF.

1.1.2. AcGrace and AptPlot

AcGrace and AptPlot are WYSIWYG, scriptable plotting applications with support for analysis-code binary plot-files. The former is a legacy application written primarily for UNIX and UNIX-like systems. Due to the difficulty of building and running the application on Windows, AcGrace has been deprecated in favor AptPlot, a functional clone rewritten in Java. AptPlot can be run on any platform with a Java 5.0 or newer Runtime Environment.

Either AcGrace or AptPlot are used by AVScript to create plots and extract data. Custom batch files are generated for the program based on the case, figure, data, and page definitions. These scripts load, plot, and export data. The AVScript component of the AVF utilizes AptPlot to perform these tasks in much the same way as the legacy system.

Only AptPlot can be used with the AVF plug-in, as it utilizes functionality not found in AcGrace. AptPlot is not distributed with the AVF plug-in. To download it, visit www.aptplot.com.

1.1.3. ACAP

The Automated Code Assessment Program (ACAP) is described in its documentation as "a tool to provide quantitative comparisons between nuclear reactor systems (NRS) code results and experimental measurements." AVScript uses ACAP to generate figures of merit. AVScript inputs define an ACAP 'config' file (essentially a script without data)

and the data to compare. AVScript extracts the data from the source, fills out the config file with these values, and runs ACAP with the complete script.

The console version of this utility has been ported to Java so as to run on any platform with a 1.5 compatible Java Runtime Environment. The Java port of ACAP is included with the AVF distribution.

1.1.4. TRACE ATF

The TRACE Automated Testing Framework (ATF) is a collection of Perl scripts, input files, and AVScript inputs used to analyze TRACE code versions. The framework divides testing into three tiers of varying purpose: measuring consistency between code versions (regression), code health (robustness), and code accuracy (assessment). Each of these tiers are broken into several *suites*: related input groups.

Regression testing is the process of running a large number of inputs with two versions of a code and comparing the differences. This is achieved in the ATF by using GNU make to run the codes (make is chosen for its ability to run multiple inputs concurrently). Once all the results have been arranged, a Perl script parses the results, diffs various files, and generates HTML reports that list the comparisons and catalog the diffs. This functionality is supported in the AVF through a set of components that define regression suites, several submit options for running regression jobs on a Calculation Server, and a generic report generator.

Robustness and Assessment compare results between code versions and data utilizing AVScript along with several utility Perl applications. There is no distinction between Robustness, Assessment, and standard AVScript runs in the AVF. This functionality can be implimented by creating distinct AVScript components and selecting only the scripts of interest at submit time.

1.2. SNAP and the AVF Plug-in

The Symbolic Nuclear Analysis Package (SNAP) consists of a suite of integrated applications designed to simplify the process of performing thermal-hydraulic analysis. SNAP provides a highly flexible framework for creating and editing input for engineering analysis codes as well as extensive functionality for submitting, monitoring and interacting with the analysis codes. The modular plug-in design of the software allows functionality to be tailored to the specific requirements of each analysis code.

The Automated Validation Framework (AVF) is a SNAP plug-in designed to replace the TRACE ATF and AVScript while providing a pluggable system for additional analysis codes. The AVF plug-in uses a client/server model that should be familiar to many SNAP users. Regression suites and AVScript definitions can be designed in the Model Editor and submitted to a Calculation Server. The resulting jobs and their results can then be observed with JobStatus.

Chapter 2. Files Bundled with AVF

A number of files are installed with the AVF plug-in. The most obvious is the plug-in JAR required to use the framework in SNAP. The AVF distribution also includes a Java version of the ACAP command-line utility available in the SNAP `bin` folder. In addition, an avf folder is created in the SNAP installation folder that contains the following files and folders.

- `bin` - The folder where all executables and scripts employed by post-case and pre-figure commands must be located here if not in the user's PATH.
- `plugins` - AVF pluggable components are installed here. Several plug-ins are included with the AVF distribution. For more information on developing AVF plug-ins, see Appendix B, *Developing AVF Plug-ins*.
- `reportDefs` - the report definition files that determine how to generate reports for a specific code type. Each definition is named after the plug-in ID of the code with the `.xml` file extension (TRACE.xml, RELAP.xml, etc.).
- `templates` - the HTML templates used to generate regression reports.

Note Modifying the resources in the `avf` directory is unsupported. However, it is possible to adjust the layout and contents of reports by modifying definitions and templates. If you intend to do this, *always back-up the `avf` directory before making modifications*. Updating the AVF plug-in overwrites its contents. Furthermore, errors in the XML and template files can break AVF functionality.

Chapter 3. AVF Model Components

An AVF model contains several categories of components, all of which are used to define test sets. Components are organized into two main categories: those associated with *Regression* (and Report) jobs, and *AV Script* jobs. The following sections define the purpose and usage of each component.

3.1. Regression

Regression components are fairly simple to define: they consist of input models arranged into suites and suites organized into sets.

3.1.1. Input Models

Input Model components specify code inputs for regression jobs. An input model component is a file *stub*: the component represents a file without indicating that file's location on the machine. These stubs allow suites to refer to models without knowing exactly where they are. During regression job submission, when the parent location of the input models is known, each stub is replaced with the complete path.

Input Model components have the following noteworthy attributes:

- *File Name*. This is the case-specific name of the file. Each name must be unique among input models.
- *Type*. The type of input. This value must be a valid SNAP plug-in ID. The identified plug-in must be installed on the machine that runs the regression job. The editor for this value is an editable drop-down menu. The ID can be entered manually or selected from the drop-down list. Only those case types loaded as AVF plug-ins will be listed.
- *Restart File*. An optional restart reference to another input model. All restart models are displayed in the Navigator with a modified label: if input model *A* restarts *B*, it will be listed as "*A* > *B*".

3.1.2. Suites and Suite Sets

Suite components are named lists of input models. When submitting a regression run, suites are selected to determine which inputs to run. On the server where a regression job is run, the suites define the folders used to organize the large number of files generated by the job.

Suites Sets are named lists of suits. Sets provide another layer of organization on top of suites. Suite sets may be chosen in place of suites when submitting regression runs: this has the same effect as selecting all suites represented by the requested sets. Sets have no effect on the organization of regression job files.

3.2. AV Script

The *AV Script* category contains components associated with running AVScript jobs on a Calculation Server. *Executable* components define code version stubs filled in at submit time. Each *Script* component is composed of cases, figures, and data traces that define how to run inputs and generate plots. Scripts may optionally contain pages and ACAP definitions. The Navigator displays these definitions in sub-categories under each script component. All of these sub-definitions support copy and paste. The properties of these components can be edited in the Property View (see Figure 3.1, “AV Script Components in the Navigator and Property View”).

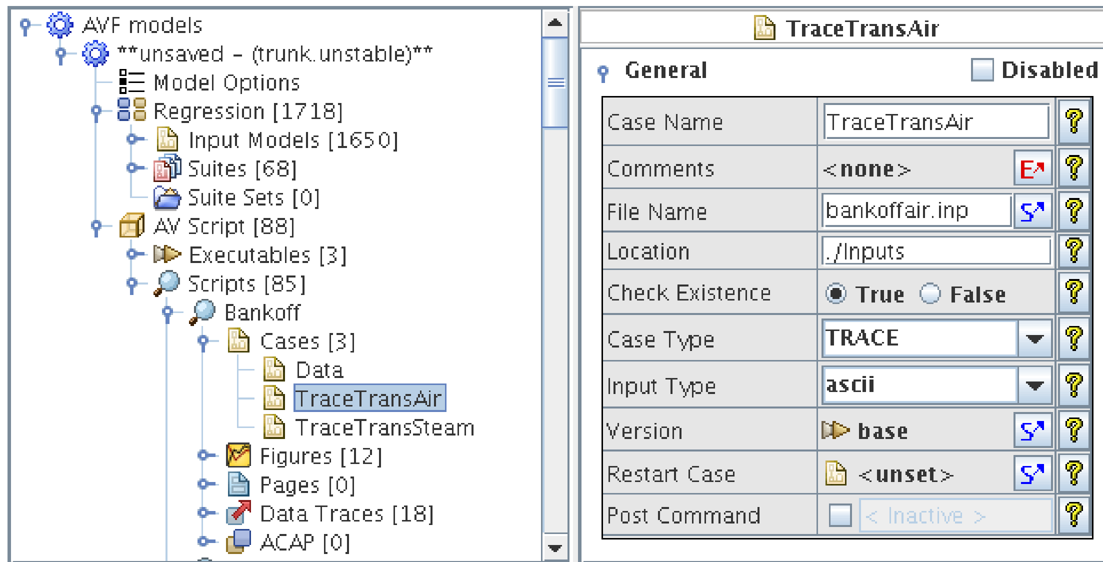


Figure 3.1. AV Script Components in the Navigator and Property View

3.2.1. Executables

Executable components specify named code executables for AVScript runs. Similar to input model components, each executable is a *stub* that takes the place of an executable defined at submit time. This allows AVScript components to reference specific code versions without knowing where the executable is located.

Executable components have the following noteworthy attributes:

- *Executable Name*. The name of the executable. Each name must be unique among executables.
- *Executable Type*. The type of executable. This value must be a valid SNAP plug-in ID.

3.2.2. Changes From Legacy AVScript

For those experienced with the legacy AVScript application, it is important to clearly identify which input values are *not* necessary or have different semantics in the new

system. This section highlights changes to the way scripts are defined. New users can skip this section entirely.

Most values in the path definitions are unneeded. The paths definition file indicates the paths to a number of required applications and resources. Several of these values are defined in the Calculation Server, such as the location of AptPlot and the demultiplexers. For applications bundled with the AVF plug-in, such as ACAP, the location is known in advance. No equivalent to *topdir* is necessary: the locations of inputs and outputs have been decoupled and are specified when the script is submitted. Executables are also defined at submit time (see Section 4.3, “AVScript Jobs”). When importing legacy AVScript inputs, code versions are parsed into Executable stubs with the indicated name and type.

All Location properties define input locations in Calculation Server path syntax. When submitting AVScripts, a mounted folder on a Calculation Server is selected as an input location. On the server, AVF looks for all input files *underneath* this input folder. As an example, a Case input file *a.inp* is located at *C:\trunk\Assessment\WALL\inputs*. The folder *C:\trunk* is mounted on this server with the name *TRUNK*. During submit, the user selects */TRUNK/Assessment* as the input folder. For a case to correctly locate this file, its *File Name* must be set to *a.inp* and its *Location* set to *WALL/inputs*. Note the lack of prefix and the use of the slash character (/) as a folder separator. Constructs such as *../* cannot be used to reach a directory outside the selected input folder. The *Location* field is kept separate from the file name to facilitate multi-edit.

Case files that might not exist before running a post-case or pre-figure command are not prefixed with an ampersand. Instead, set the *Check Existence* property to *false*.

Case file names and ACAP config-file names must include the file extension. These are no longer assumed, except when importing legacy AVScript inputs.

Legend coordinates should never be prefixed by a 'W'. By doing so, the legacy AVScript marked the legend coordinates as world-coordinates on an AptPlot graph. Instead, set the figure's *Legend Coordinate Type* value to *World*.

Annotations are now defined as a component of a figure. Each figure has an *Annotations* property. The editor launches a dialog that can be used to add, remove, and edit annotations. This naturally represents the way annotations related to figures in the legacy input format.

Ellipse annotations are no longer defined in the form: X_{center}, Y_{center}, *Width, Height*. Instead, they follow the standard convention of X_{min}, Y_{min}, X_{max}, Y_{max}.

Data definition expressions should not be prefixed by an equals sign. Instead, set the *X Variable Type* or *Y Variable Type* value to *Expression*.

Variables in axial data definitions no longer use ZZ or ZZZ to indicate the mesh index segment. Instead, they use the %2N, %3N, etc. format defined for AptPlot's built-in axial plot functionality. The old format is automatically converted to the new format on import.

Figure and page names must be unique between both. Files generated for figures and pages (AptPlot batch files, images, etc.) are created in common directories. If a figure and a page in the AVScript component share a common name, the figure files will be overwritten, and unexpected errors may arise.

Figures and pages no longer have a file location value. Instead, generated files are automatically sorted into a predefined directory structure.

3.2.3. Cases

Cases represent code inputs and data sources. A case component defines an input or data file, its type, a code version used to run the input (if appropriate), and any restart information.

Cases have the following properties:

- *Case Name*. The ID of this case. Each *Case Name* should be unique among cases in the parent AVScript component.

- *File Name*. The name of the file referenced by this case.

- *Location*. The *Calculation Server path* where the input or data file is located. This location should use the slash character (/) as a folder separator. Do not prefix the path with separators. Constructs such as ../ will not work here.

- *Check Existence*. If set to true, AVScript verifies that the file exists before running the script. This should only be set to false if a post-case or pre-figure command is going to generate the file.

- *Case Type*. The type of this case. The type determines what codes are used to process inputs, how data is extracted from files, etc.. The editor for this value allows selecting from the supported case types *found on the client machine*. If the AVF installation where the script will be submitted contains support for additional codes, their types can be entered manually.

  Several case types for data files are always supported. These include the following.
  - *ASCII Data*. A text file with values in space-delimited columns.
  - *DATABANK Data*. An NRC Databank binary data file.

- *Input Type*. Further defines the type of input for this case type. For example, *ASCII Data* can be specified in a number of formats, including *xy*, *xydx*, *xydy*, etc.. Like the case type, these may be entered manually if the pluggable component for the required code is not installed on the client machine.

- *Version*. The executable used to run this case. This is not displayed for known data-file types.

- *Restart File Name*. If activated, this field specifies the optional restart case. This is not displayed for known data-file types.
- *Post Command*. If activated, this field specifies the optional command run immediately following execution of the case.

3.2.4. Figures and Annotations

Figure components define the look of a graph. Each contains properties such as title, subtitle, position on the plot, etc.. Figures may also contain annotations: custom lines, boxes, ellipses, and text rendered on the plot.

The following properties are available to figure components:

- *Figure Name*. The figure ID. Each name should be unique among figures *and pages* in the parent AVScript component.
- *Title Text* and *Subtitle Text*. The optional title and subtitle drawn over the graph.
- *X Axis Label* and *Y Axis Label*. Denotes the labels drawn along the X and Y axes.
- *Scaling Type*. Determines whether the X and Y axes are scaled linearly or logarithmically.
- *X Axis Boundaries* and *Y Axis Boundaries*. Determines if the X and Y axes are scaled automatically or explicitly. Automatically scaled graphs adjust their X and Y bounds to display all data values. Explicit boundaries are defined below.
- *Axis Bounds*. Defines the minimum and maximum values displayed by the figure. This property is only enabled if at least one of *X Axis Boundaries* or *Y Axis Boundaries* is set to *Explicit*.
- *Tick Marks*. Sets the major and minor tick intervals along the X and Y axes.
- *Legend X Coordinate* and *Legend Y Coordinate*. The location of the legend's upper-left corner.
- *Legend Coordinate Type*. Determines if the coordinate described above is a Viewport coordinate or a World coordinate.
- *Legend Length*. The length of the line used in legend entries.
- *Viewport*. The Viewport specifies the placement of the graph on the plot. Valid values always range from 0 to 1 in either direction. If one dimension is longer, that dimension's viewport coordinates extend from 0 to the ratio of the sides. For example, if the graph is 3 inches wide by 2 inches tall, viewport X coordinates extend from 0 to 1.5, and viewport Y coordinates extend from 0 to 1.

- *Character Size*. The default size of text rendered in the plot. This value is a multiplier of AptPlot's default font size. Thus, specifying .75 will produce text that is 75% of the normal size.

- *Symbol Size*. The default size of symbols rendered in the plot. Like *Character Size*, this value is a multiplier of AptPlot's default size.

- *Orientation*. Determines the size of the plot on which the figure is graphed.

- *Page Width* and *Page Height*. The size of the plot in inches. Width and height are only available if the *Orientation* is set to *Custom*.

- *Annotations*. The graphical and textual objects drawn on the figure (see below).

- *Pre Command*. An optional command that will be run before the figure is generated.

To edit a figure's annotations, select the **Edit** button from its *Annotations* property. This displays the *Edit Annotations* dialog as illustrated in Figure 3.2, “Annotation Dialog”. With this dialog, annotations can be created, removed, and edited.

Annotations have the following properties:

- *Annotation Name*. The ID of the annotation. Each name should be unique among annotations in the figure.

- *Object Type*. The type of the annotation: box, ellipse, string of text, or line.

- *Color*. The color of the annotation.

- *Location*. The placement of the annotation in world coordinates. Box, ellipses, and lines two coordinate pairs to specify the absolute placement of the shape. Strings are placed by a single coordinate that places the string based on the *Justification*.

- *Line Style*. The style of the line: solid, dotted, dashed, or a combination. This value is only available for boxes, ellipses, and lines.

- *Line Width*. The width of the line, from 0 to 20. This value is only available for boxes, ellipses, and lines.

- *Fill Color*. The color used to fill a box or ellipse.

- *Fill Pattern*. The pattern used to fill a box or ellipse.

- *Font*. The font used to render a string.

- *Character Size*. The character size used to render a string. This value should range from 0 to 1000, where 100 is 100% of the default character size.

- *Justification*. The placement of a string relative to the *Location* coordinate. Each value is a combination of horizontal placement followed by vertical placement.
- *Rotation*. The rotation of a string around the *Location* coordinate.
- *String Value*. The text of a string. This value may use AptPlot's typesetting constructs.
- *Arrow Heads Placement*. The placement of arrow heads on a line.
- *Arrow Type*. The type of arrow heads placed on a line.
- *Arrow Length*. The length of the arrows placed on a line. Allowable values range from -10 to 10.
- *Arrow d/L Factor*. The width of the arrow head. Allowable values range from 0 to 10.
- *Arrow l/L Factor*. The triangular shape of the arrow head. Allowable values range from -1 to 1.

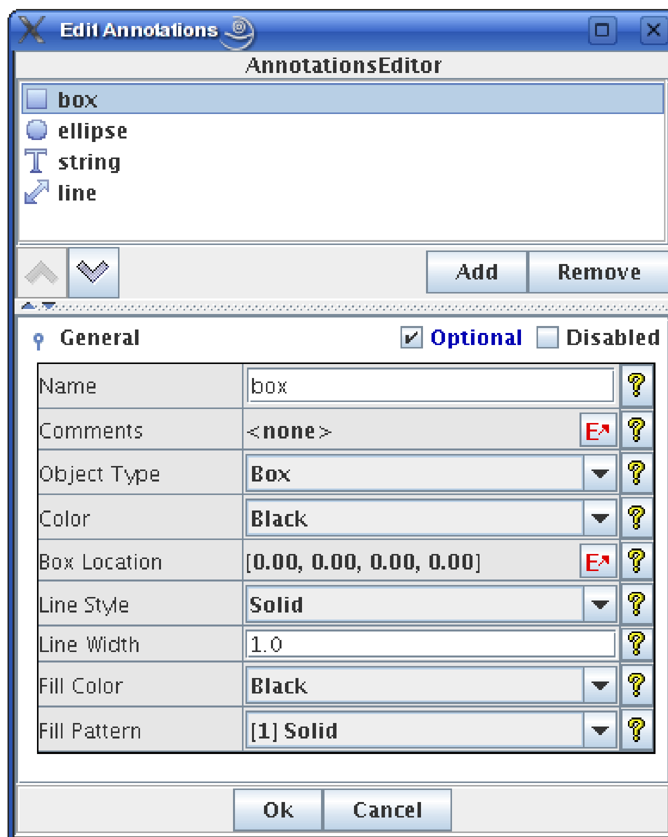


Figure 3.2. Annotation Dialog

3.2.5. Data Traces

Data traces define a data set in a graph. They specify the case from which the data originates, the figure upon which the trace is plotted, the channel or complex expression

that defines the data, the appearance of the line, etc.. Data Trace properties are described below.

In the *General* attribute group.

- *Data Name*. The ID of this data trace. Each name should be unique among data definitions in the AVScript component.

- *Case*. Where the plot data originates. Each data trace must have a valid case reference.

- *Figure*. Where the data is plotted. Each data trace must have a valid figure reference.

- *Plot Type*. The type of plot represented by this data trace. The available options are described below. Unless otherwise noted, any two data traces referencing a particular figure must have identical plot types.
  - *Time*. A simple time-history plot. Only *Y Variable* may specify data in the trace. The *X Variable* is either assumed or taken automatically from the appropriate source (such as the time channel in a binary plot file).
  - *Time Point*. A single point from a time-history plot. Only *Y Variable* may specify data in the trace. These plots must also specify the *Time* value (not to be confused with the *Time* plot type). This indicates which point in the data set should be extracted and plotted. *Time Point* data traces may reside on the same figure as *Time* plots.
  - *Parametric*. Similar to *Time*, but *X Variable* is also defined.
  - *Axial*. An axial profile plot. Only *Y Variable* is specified, but it allows a special construct for substituting incremental channels (described below). *Axial* plots cannot be used with *Databank* cases, and do not allow expressions.

- *X Variable*, *X Variable Type*, *Y Variable*, and *Y Variable Type*. The variable to be plotted on the X or Y axis and whether that value is an expression or a channel. This value can be a channel name, data file column index, or complex channel expression. For ASCII Data, specify the column in the data file to use (column indexes start at 1). Other types may use either a channel name of a complex expression.

  For channel variables, simply enter the name of the channel. The property editor allows selecting channels from a completed job on a Calculation Server. Only those jobs that match the referenced case's *Case Type* will be displayed.

  Expressions may be any valid AptPlot expression, with one caveat: channel names must be surrounded by the construct `${}`. This tells AVScript exactly where the channel begins and ends, so that it can substitute the appropriate file-prefix in AptPlot expressions. If a channel name uses curly braces or backslashes, they can be escaped with a backslash; { becomes `\{`, } becomes `\}`, and \ becomes `\\`. The cash symbol `($)` does not need to be escaped within the braces.

  Axial data traces use the construct %2N, %3N, etc. to indicate the portion of the channel name substituted with axial mesh indexes. The integer in the construct

determines how many padding zeros are applied for indexes less than the ceiling index.

- *Legend String*. Optional text to be used in the figure's legend for this data trace. If left inactive, the legend value will be automatically generated.

- *Units*. If the case type supports it, determines whether values are converted to *SI* or *British* units. This value has no effect on *ASCII Data* values.

- *X Shift*. Shifts all values along the X axis by this value.

- *Y Shift*. Shifts all values along the Y axis by this value.

- *Slope Factor*. Multiplies all Y values by this value after the shift.

In the *Line Properties* attribute group.

- *Symbol*. The type of symbol centered over data points in the figure.

- *Symbol Fill*. Whether the symbols are filled or outlines. This parameter is only enabled if *Symbol* is set to a value other than *None*.

- *Symbol Skip*. How many data points are skipped between rendered symbols. This parameter is only enabled if *Symbol* is set to a value other than *None*.

- *Line Style*. The type of line drawn between points in the data set.

- *Line Color*. The color of the line drawn between points in the data set.

- *Line Width*. The width of the line drawn between points in the data set.

- *Time*. The transient time for which the axial plot or time point information is to be extracted.

In the *Axial Plot Definitions* attribute group.

- *Axial Start Index*. The index of the first axial elevation value provided in the *Axial Locations* property, starting from 1.

- *Axial Locations*. Displays the dialog shown in Figure 3.3, “Edit Axial Locations Dialog”. This dialog can be used to add and remove axial locations, set their value, and enable and set an optional axial index.

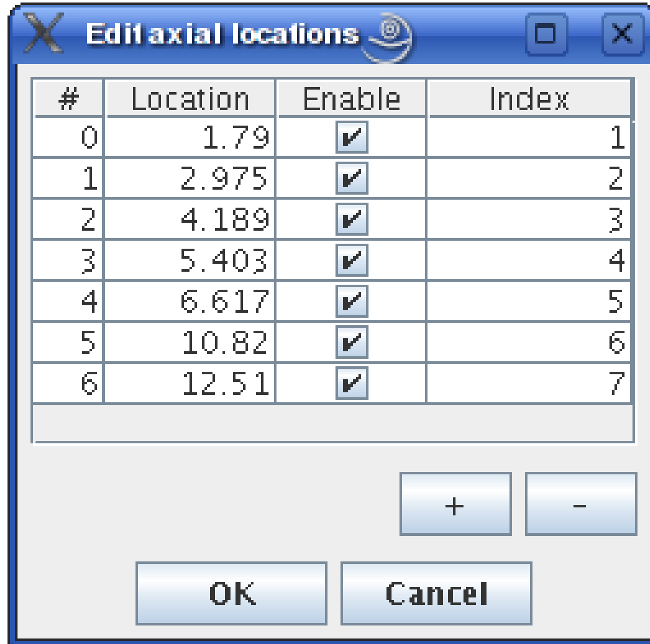


Figure 3.3. Edit Axial Locations Dialog

3.2.6. Pages

Pages allow displaying several figures on a single plot. Each page has the following properties:

- *Page Name*. The page ID. Each name should be unique among pages *and figures* in the parent AVScript component.
- *Figures*. The figures organized by this page definition. Each page should have at least one valid figure reference. The number of figures should match the product of the *Row Count* and *Column Count* properties.
- *Row Count* and *Column Count*. The number of rows and columns that the referenced figures are organized into.
- *Override Title* and *Override Subtitle*. Optional overridden title and subtitle for the organized figures. When activated, only this title and/or subtitle will be displayed; all other figure titles and subtitles will be discarded.
- *Margin*. The margin between the edges of the plot and the arranged figures, specified in viewport units.
- *Column Gap* and *Row Gap*. The margin between columns and rows in the arranged figures, specified in viewport units.
- *Orientation*. The size of the plot on which the page is graphed.

- *Page Width* and *Page Height*. The size of the plot in inches. Width and height are only available if the *Orientation* is set to *Custom*.

3.2.7. ACAP

ACAP definitions are used to produce figures of merit using the ACAP executable. These components have the following properties:

- *ACAP Name*. The ACAP definition ID. Each name should be unique among ACAP definitions in the parent AVScript component.

- *Config File Name*. The name of the config file used to drive the ACAP calculation.

- *Location*. The path, relative to the *Top Directory*, where the config file is located. This location should use the slash character (/) as a folder separator. Do not prefix the path with separators. Constructs such as ../ will not work here.

- *Base Data Set*. The base data set against which all other data sets are compared in the figure of merit computation. Each ACAP definition should have a valid reference for this property.

- *Compared Data Sets*. The data sets compared against the base data set in the figure of merit computation. Each ACAP definition should have at least one valid data set reference for this property.

3.2.8. Spreadsheet Editor

Script components can also be edited in a spreadsheet editor, shown in Figure 3.4, “Script Spreadsheet Editor”. This dialog allows editing the case, figure, page, data trace, and ACAP components in a script definition in a table. There are two ways to access this dialog. First, right-click on a *Script* component in the Navigator and select *Open Table Editor* from the pop-up menu. Second, press the *Edit* button on any of the *Script* component editors for *Cases*, *Figures*, *Pages*, *Data Traces*, and *ACAP*.

Bankoff - unsaved - Table Editor

Name	Case	Figure	Plot Type	X Variable	Y Variable	Legend String	Use SI Units	X Shift	Y Shift	Slope
1A	Data	Fig1A	Parametric	1	2	Data	✓	0	0	1
2A	TraceTransAir	Fig1A	Parametric	cb60	cb58	TRACE	✓	0	0	1
3A	TraceTransAir	Fig2A	Parametric	cb53	cb59	none	✓	0	0	1
7A	Data	Fig3A	Parametric	1	2	Data	✓	0	0	1
8A	TraceTransAir	Fig3A	Parametric	cb59	cb42	TRACE	✓	0	0	1
9A	TraceTransAir	Fig4A	Time		cb60	SQRT(H*sub-f)/C	✓	0	0	1
10A	TraceTransAir	Fig5A	Time		cb58	SQRT(H*sub-g)/C	✓	0	0	1
11A	TraceTransAir	Fig6A	Time		cb60	SQRT(H*sub-f)/C	✓	0	0	1
12A	TraceTransAir	Fig6A	Time		cb42	SQRT(H*sub-g)/C	✓	0	0	1
1S	Data	Fig1S	Parametric	1	2	Data	✓	0	0	1
2S	TraceTransSteam	Fig1S	Parametric	cb60	cb58	TRACE	✓	0	0	1
3S	TraceTransSteam	Fig2S	Parametric	cb53	cb59	none	✓	0	0	1
7S	Data	Fig3S	Parametric	1	2	Data	✓	0	0	1
8S	TraceTransSteam	Fig3S	Parametric	cb59	cb42	TRACE	✓	0	0	1
9S	TraceTransSteam	Fig4S	Time		cb60	SQRT(H*sub-f)/C	✓	0	0	1
10S	TraceTransSteam	Fig5S	Time		cb58	SQRT(H*sub-g)/C	✓	0	0	1
11S	TraceTransSteam	Fig6S	Time		cb60	SQRT(H*sub-f)/C	✓	0	0	1
12S	TraceTransSteam	Fig6S	Time		cb42	SQRT(H*sub-g)/C	✓	0	0	1

Cases [3] | Figures [12] | Pages [0] | Data Traces [18] | ACAP [0]

Figure 3.4. Script Spreadsheet Editor

This dialog allows comparing components while editing individual or multiple values, reordering definitions, adding and removing components, copying and pasting values to and from spreadsheet applications, etc..

3.3. Component Validation

Model validation is supplied for AVF models. Validation checks for unique IDs where appropriate, undefined references, and correct AVScript definitions. Performing a validation check is always recommended after making edits to the model, especially before submitting a job.

Chapter 4. Submitting Jobs

The AVF plug-in allows submitting three types of jobs: regression, report, and AVScript. The first two types of jobs, regression and report, are closely related. A regression job executes a large number of inputs organized into suites. One executable is used for the entire input set. A report job generates a report comparing the two previously submitted regression jobs. An AVScript job is capable of running codes, generating plots, and computing figures of merit.

4.1. Regression Jobs

To submit a regression job, select *Submit Regression Job* from the *Tools* menu. This opens the submit dialog shown in Figure 4.1, "Submit Regression Dialog".

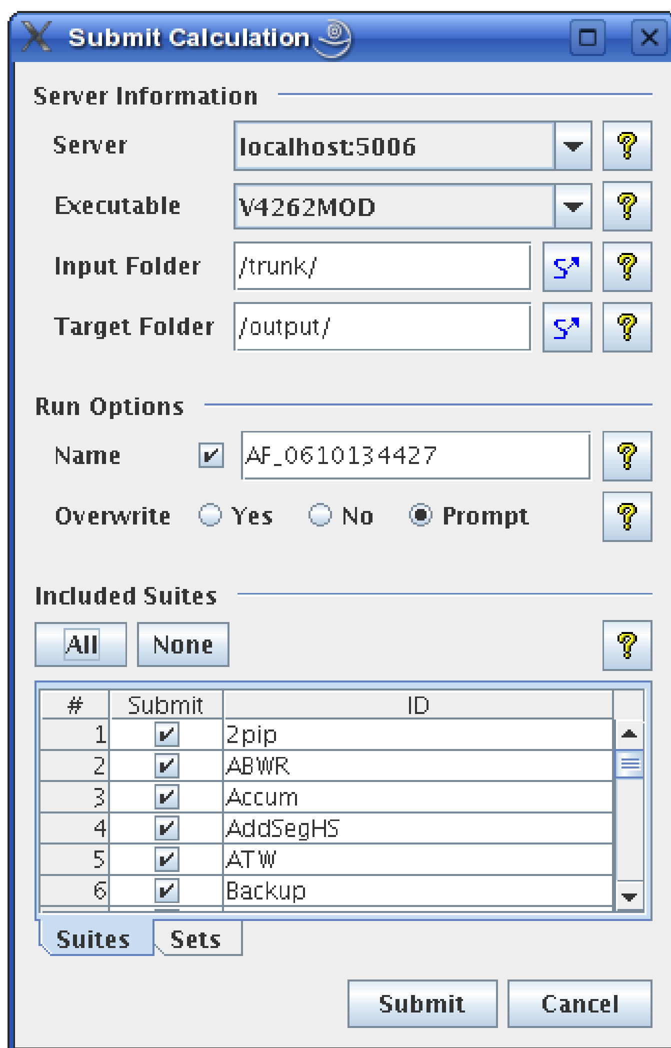


Figure 4.1. Submit Regression Dialog

Server allows selecting the location to run the regression job from a list of known servers. Once a server is selected and a valid connection is established, the *Input Folder* and

Target Folder may be specified. The *Select* button on either opens a dialog used for selecting a mounted folder on the server. With *Input Folder*, the chosen directory must be the location of the input files listed as *Input Models*. The *Target Folder* is where the job will execute and store its results. The *Executable* field allows selecting the executable used to process the inputs, but cannot be specified until the suites have been selected (more on this below).

Name defines the name of the regression job. Any resource associated with this job created in the Target Folder will be labelled after the name. If the check next to the field is unselected, the name will be generated based on the date.

Overwrite determines what to do if a job with a matching name already exists at the target folder: *Yes* will overwrite it silently, *No* will reject the job, and *Prompt* will display a dialog asking the user what to do.

The *Included Suites* section is used to select which suites are submitted. This is broken into two tabs: *Suites* and *Sets*. The first allows picking suites from all suites in the AVF model. The second allows selecting from suite sets, with one caveat: suite sets are not actually submitted. Instead, selecting a set selects all of its referenced suites as selected, and vice versa. A suite set is only shown as selected if all of its referenced suites are. If even one suite in the set is inactive, the set will be shown as unselected. If two suite sets overlap, it is possible that changing selected status of one will modify the other.

The *Executable* can be specified once the list of submitted suites is ready. The available choices are those server executables whose plug-in ID matches the case type of the first input model in the first selected suite. It is for this reason that a suite should never contain input models with mixed type, and the selected suites should only reference models of the same type.

Once all of the above fields have been set to the desired values, the regression job can be sent to the Calculation Server by pressing the **Submit** button. First, as with other plug-ins, this will add the regression job to the server's job queue. This job is a *launcher*: it invokes the AVF Launcher application to handle running the requested inputs. Instead of directly invoking the analysis code, it spawns Calculation Server jobs for each input and lets the server handle their execution. This process is described below.

Note The launcher job does not count toward the maximum number of running jobs specified for the server. This eliminates the potentially dangerous condition of a user submitting multiple regression jobs that never complete, as they wait for the completion of spawned jobs that would never be run.

The launcher job first creates a *top directory* for all regression files. This folder, located in the *Target Folder*, is named after the regression job appended with the suffix *_Jobs*. Next, a folder is created in the top directory for each submitted suite. Finally, the launcher spawns a job in the appropriate suite folder for each input model in each suite. Each of these is a standard job. They're added to the server's queue and started in the order they're added. The launcher job completes after all spawned jobs complete.

4.2. Report Jobs

To submit a report job, select *Submit Report Job* from the *Tools* menu. This opens the submit dialog shown in Figure 4.2, "Submit Report Dialog".

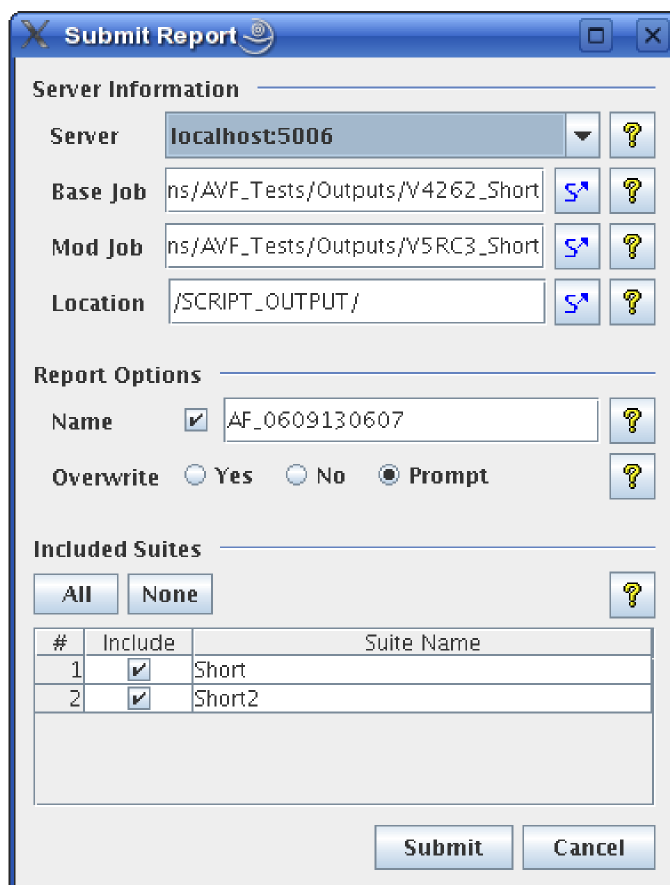


Figure 4.2. Submit Report Dialog

Server allows selecting the location to run the report job from a list of known servers. Once a server is selected and a valid connection is established, the *Base Job*, *Mod Job*, and *Location* may be specified.

Location is the target folder where the report will be generated. The *Select* button to the right of *Location* opens a dialog used for selecting a mounted folder on the server.

Base Job and *Mod Job* define the two previously submitted regression jobs that will be compared in the report. The **Select** button to the right of each will open the *Select Regression Job* dialog shown in Figure 4.3, "Select Regression Job Dialog". The list of suites that were included in the regression job is shown to the right for convenience.

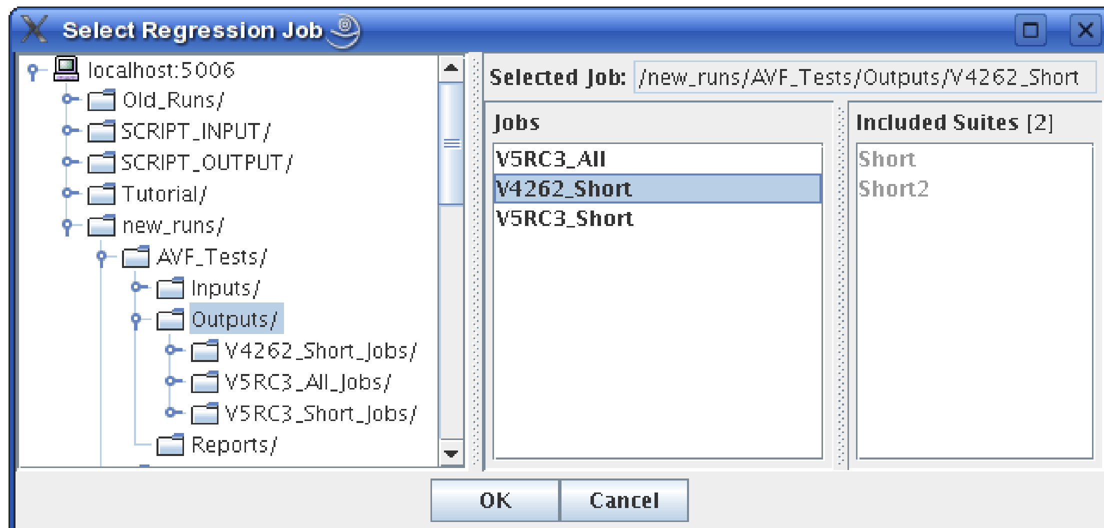


Figure 4.3. Select Regression Job Dialog

Name defines the name of the report job. Any resource associated with this job created in the *Target Folder* will be labelled after the name. If the check next to the field is not selected, the name will be generated based on the date.

Overwrite determines what to do if a job with a matching name already exists at the target folder: *Yes* will overwrite it silently, *No* will reject the job, and *Prompt* will display a dialog asking the user what to do.

The *Included Suites* section displays the suites that are included in both selected regression jobs. Any or all of these suites can be selected to be included in the report.

Once all of the above fields have been set, the report job can be sent to the Calculation Server by pressing the **Submit** button. This will add the report run to the server's job queue. This process compares the statistics, failures, resulting output, etc. and creates an HTML report displaying the results. For more information on how reports are defined, see Appendix A, *Report Generation*.

4.3. AVScript Jobs

To submit an AVScript job, select *Submit AVScript Job* from the *Tools* menu. This opens the submit dialog shown in Figure 4.4, “Submit AVScript: Location Tab”.

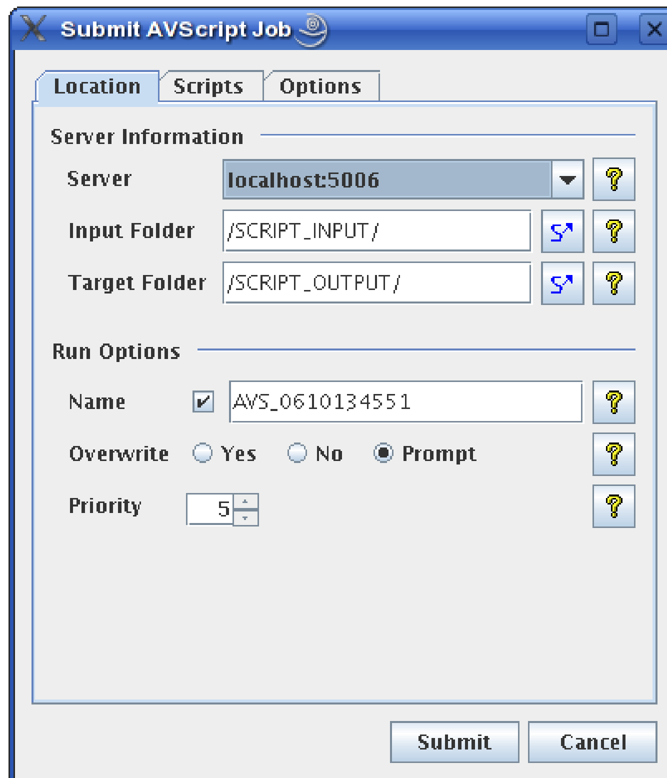


Figure 4.4. Submit AVScript: Location Tab

The *Location* tab specifies where the script is submitted, and where its files originate.

Server allows selecting the location to run the AVScript job from a list of known servers. Once a server is selected and a valid connection is established, the *Input Folder* and *Target Folder* may be specified. The *Select* button to the right of each will open a folder selection dialog for the server. The selected *Input Folder* must be the location of the input files defined by *Cases*. The *Target Folder* is where the job will execute and store its results.

Name defines the name of the AV Script job. Any resource associated with this job created in the *Target Folder* will be labelled after the name. If the check next to the *Name* field is not selected, a name will be generated based on the date.

Overwrite determines what to do if a job with a matching name already exists in the target folder: *Yes* will overwrite it silently, *No* will reject the job, and *Prompt* will display a dialog asking the user what to do.

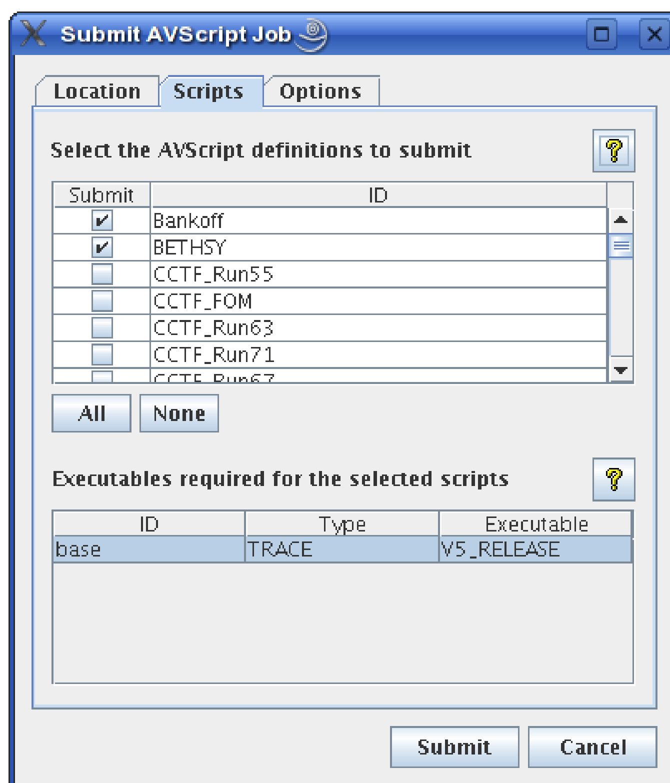


Figure 4.5. Submit AVScript: Scripts Tab

The *Scripts* tab is used to select which script definitions are submitted, and which code versions on the selected *Server* will be used for each *Executable* stub. The list of executables in the bottom table is includes every executable referenced by the scripts selected in the first table. Each executable can be edited by selecting a value from the editor in the *Executable* column.

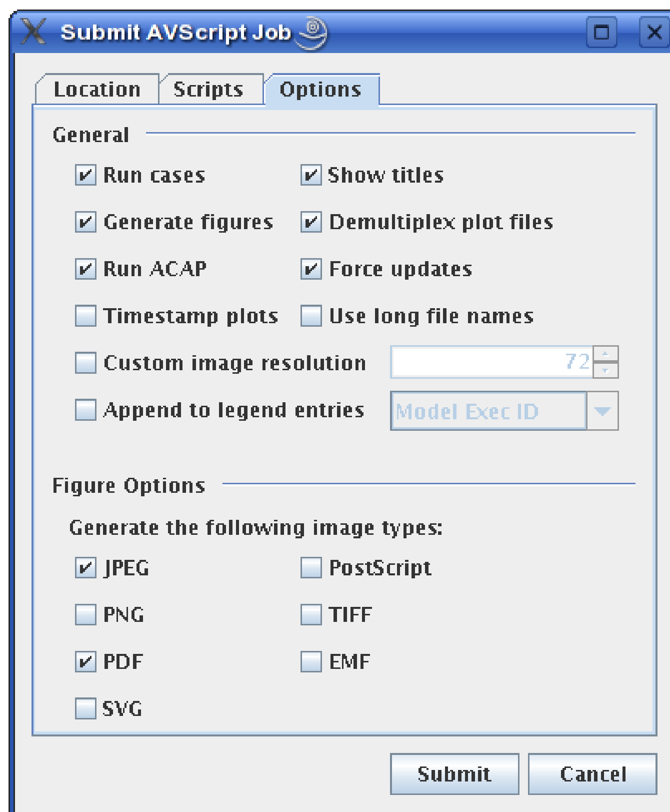


Figure 4.6. Submit AVScript: Options Tab

The *Options* tab is used to specify the manner in which the scripts are executed. It has the following *General* options.

- *Run cases*. If true, the executable cases in the script will be run on the referenced code version. If false, AVScript assumes that the appropriate cases have already been run in the target location for the selected script definitions. If this is not the case, the script will exit during its initialization stage.
- *Generate figures*. Determines whether AptPlot is used to generate figures.
- *Run ACAP*. Determines whether ACAP is used to generate figures of merit.
- *Timestamp plots*. Add a timestamp to each generated figure.
- *Show titles*. Display the titles specified for figures and pages.
- *Demultiplex plot files*. When set to true, all plot files created by running cases will be automatically demultiplexed when the case completes. This incurs an initial expensive read of the entire file, but offers improvements to performance when a case plot file is read by more than one figure or page.
- *Force updates*. Rerun cases, regenerate figures, and rerun ACAPs, even if an output file newer than the inputs exists.

- *Use long file names*. When generating ASCII data dumps of a figure's data traces, use a more explicit file name.

- *Custom image resolution*. A custom image resolution to be used in all images generated for figures.

- *Append to legend entries*. When selected, a label is appended to the legend entries of all data traces on all figures and pages. The choices are as follows.
    - *Model Exec ID*. The name of the executable stub as defined in the AVF model.
    - *Server Exec ID*. The ID of the executable on the Calculation Server.
    - *Server Exec Desc*. The description of the executable on the Calculation Server.

Note At least one of *Run cases*, *Generate figures*, and *Run ACAP* must be selected before a submit is allowed.

When *Generate figures* is selected, the *Figure Options* become available. These indicate the types of images generated for each figure. If generating figures, at least one image type must be selected before the script can be submitted.

Once all of the above fields have been set to the desired values, the AV Script job can be sent to the Calculation Server by pressing the **Submit** button. This will add the report run to the server's job queue. Once the script job begins execution, it takes the following actions.

1. Each script's cases are submitted, in order, to the Calculation Server in the *Runs* folder (see below). Only one case will be submitted at a time for each script.

2. Begin generating figures. Repeat the following for each figure in each script: generate the AptPlot batch script used to create the figure, then run it in AptPlot.

3. Each script's figures are processed, in order.

   The AptPlot batch file is generated and then executed using AptPlot.

4. Each script's pages are processed, in order.

   The AptPlot batch file is generated and then executed using AptPlot.

5. Each script's ACAP definitions are processed, in order.

   First, the AptPlot batch files are created to extract the required data. Then, the scripts are executed with AptPlot and the resulting data appended to the ACAP config file. Finally, an ACAP script is generated to load the data and executed with ACAP.

AVScript jobs organize files into a very strict directory hierarchy. A folder is created with a name in the form "*<JOB>_Jobs*". A folder is then created in <JOB>_Jobs to for each submitted AV Script. Each AV Script folder then contains the following folders:

- *Runs*. Case inputs and outputs. To prevent any file-name collisions, each case is placed in a separate folder named after the associated case. Data files are also copied to this location.
- *Batch*. Generated AptPlot batch scripts. Each batch file corresponds to one figure or page in the associated AVScript definition. These files can be used to recreate the final figures and pages in AptPlot, but incur the runtime cost of reading and recomputing data.
- *Figures*. The images created by the scripts in *Batch*. In addition to the binary image files, an AptPlot File (APF) is also created here for each figure. These can be used to load the final figures and pages directly in AptPlot.
- *Data*. ASCII data dumps of plot data created when generating figures.
- *FOMs*. All resources associated with computing figures of merit. The generated ACAP scripts and computed results are all stored here.

Chapter 5. Other Features

This section details additional AVF features.

5.1. Importing Legacy AVScript Inputs

AVScript components can be created by importing legacy input files. Selecting **File** > **Import** > **AVScript** from the main menu will open a file dialog that can be used to select one or more files. Which of the input files are selected is not important; the AVF Plug-in uses the AVScript filename rules to determine which selected files are valid AVScript inputs and which files it should look for. To be precise, the plug-in will import files with the following names:

- Any of the default AVScript input file names:

avcasedefs.txt	case definitions file
avpathdefs.txt	path definitions file
avfigdefs.txt	figure definitions file
avdatadefs.txt	data definitions file
avpagedefs.txt	page definitions file [optional]
avacapdefs.txt	ACAP definitions file [optional]
avannodefs.txt	annotations definitions file [optional]

- Prefixed file names, where <NAME> is a particular prefix.

<NAME>Cases.txt	case definitions file
<NAME>Path.txt	path definitions file
<NAME>Figs.txt	figure definitions file
<NAME>Data.txt	data definitions file
<NAME>Page.txt	page definitions file [optional]
<NAME>Acap.txt	ACAP definitions file [optional]
<NAME>Annotations.txt	annotations definitions file [optional]

For example, suppose the target directory contains the files:

```
Test1Acap.txt    Test1Figs.txt    Test2Data.txt   updates.txt
Test1Cases.txt   Test1Path.txt    Test2Figs.txt
Test1Data.txt    Test2Cases.txt   Test2Path.txt
```

This indicates the directory contains two sets of input: one prefixed by "Test1" and one prefixed by "Test2". Note that the optional ACAP definitions file is present for the first file set. The directory also contains a file named "updates.txt", which is of no interest to the importer. If the user were to open the file dialog and select the files: *Test1Figs.txt*,

Test1Path.txt, *Test2Cases.txt*, and *updates.txt*, the importer will create two AVScript definitions, one named *Test1* populated by definitions from the five corresponding inputs (including the optional ACAP definition), and one named *Test2*, defined by the other four inputs. The importer ignores the *updates.txt* file, as its name does not follow the input file naming conventions. In this sample case, two files were included which matched the "Test1" prefix; the plug-in will still do the right thing when multiple files are selected which match one file set.

If the model's current model is an AVF model, the imported AVScript definitions will be added to that model. Otherwise, a new model will be created with the new definitions.

Any errors encountered while importing the selected inputs will be reported to the ModelEditor's message window.

5.2. Importing TRACE ATF

An entire AVF Model can be created by importing an existing TRACE ATF installation. To do so, select **File** > **Import** > **TRACE ATF** as shown in Figure 5.1, “File Import Sub-Menu”. This displays a file dialog which may be used to select the directory in which the ATF is installed. This dialog also displays a list of check boxes which can be used to select which segments of the ATF are imported (see Figure 5.2, “ATF Import File Dialog”); these options are detailed below.

Note Although this process is described for TRACE ATF installations, it will work for any inputs organized as described.

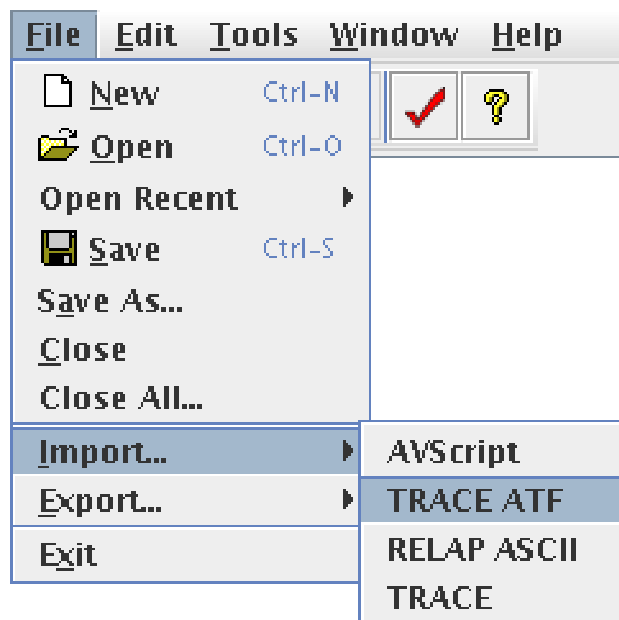


Figure 5.1. File Import Sub-Menu

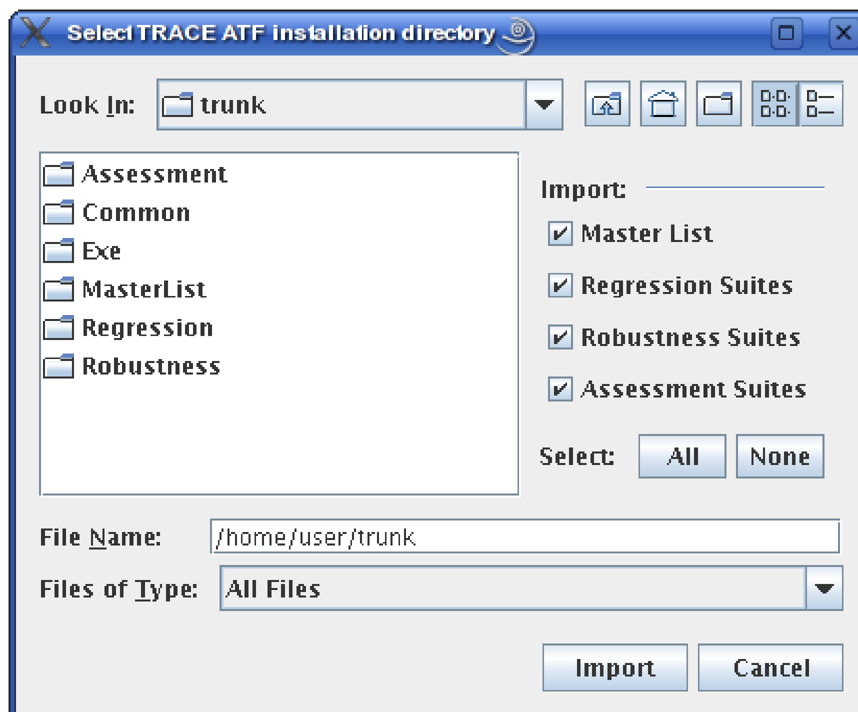


Figure 5.2. ATF Import File Dialog

- **Master List** - imports all files located in the `MasterList` directory of the selected location. Each file with an appropriate file extension is imported as a new input model. The accepted file extensions, and the input types they correspond to, are described below.
  - Files whose names end with the `.inp`, `.tpr`, and `.rst` extensions are imported as TRACE input files. Furthermore, any files ending with *.n*, where *n* is any positive integer, are also imported as a TRACE input file.
  - File names ending with `.i` are imported as RELAP5 input files.

- **Regression Suites** - imports all suite directories located in the `Regression` directory of the selected location. Each suite is imported as a new suite object.

  Only directories containing the file .isSuiteDir file are imported. In such directories, the plug-in looks for a file named `fileList`, which it expects to follow the TRACE ATF file list syntax. This file is scanned to determine which input model definitions should be associated with the newly created suite.

  Note A suite's list of input models references existing input model components only. Thus, if no input model exists with a file name matching that found in the `fileList`, the reference cannot be created.

- **Robustness Suites** - imports all suite input files in the `Robustness` directory of the selected location. Each set of input files is imported as a new AVScript definition.

  Only directories containing the file .isSuiteDir are imported. In such directories, the plug-in examines all files to see if they match the allowed AVScript input file

names (avpathdefs.txt, TestPath.txt, etc.). For each set of inputs found, an AVScript component is created with a name matching the input file prefix, or is left unnamed if the inputs use the default names.

- **Assessment Suites** - same as **Robustness Suites**, but for the `Assessment` directory.

Any errors encountered while importing the selected inputs will be reported to the ModelEditor's message window.

5.3. Editing Graphs from AptPlot

Figures and pages can be edited directly in AptPlot, if the SNAP configuration's *Plotting Tool* property indicates an AptPlot installation. Selecting the **Edit** button for the *Configure Page* property of an AVScript component will display a dialog listing the page definitions for that component. By selecting a page from that dialog, the AVF plug-in will use the page definition and any figures it references to determine the placement and decorations of AptPlot's displayed graphs. When closing AptPlot, the application will prompt for permission to export its current configuration. The plug-in then reads the exported file to configure the page and its referenced figures, creating new figures for any extra graphs. Figures are reconfigured by the properties of visible AptPlot graphs, and Pages are likewise adjusted by the appropriate fields of AptPlot's *Arrange Graph* dialog. Similarly, figures can be edited from the *Configure in AptPlot* property editor.

Note This feature overrides the placement settings of each referenced figure. If you do not want to override them, only edit empty page definitions.

Note This feature will **not** work with AcGrace. It depends on AptPlot extensions that are not present in AcGrace.

5.4. Figure Templates

AVF supports exporting a Figure's properties as a template. Such templates can then be imported with only selected values substituted into the edited figure's properties.

To export a template, select a figure in the Navigator. In the property view, select the **Export Template** button in the *Figure Template* property editor. Select the location to save the file and press **Save**.

To import a template, select the **Import Template** button from the same editor, select the template, and press **Open**. A dialog will be displayed asking which properties to import, and showing the value of each property in the template. Select the desired properties and press the **OK** button to import the values.

Appendix A. Report Generation

Each report compares a base code version to a modified code version (typically abbreviated as *mod*). The report generator assumes that the modified version is, in fact, a progressive revision of the base version. In practice, this only effects how results are sorted. The compared codes can be any two codes or code versions so long as their outputs are comparable.

A.1. Creating Reports

Report generation consists of the following steps.

1. Read the report definition. This file determines the overall structure and content of the report, what files to parse, etc. For more information on the report definition, see Section A.2, “Report Definition”.

2. If success conditions are defined in the report file, locate failed runs and generate the failure report. See Section A.1.1, “Creating the Failure Report”.

3. Gather run statistics. To do so, the statistics file for each run is read and pertinent statistics are tabulated.

4. Write the statistics reports. See Section A.1.2, “Creating Statistics Reports”.

5. If diffs are defined in the report file, diff the appropriate files and generate the differences report. See Section A.1.3, “Creating the Differences Report”.

6. Write the summary report. This file contains general information and links to the other generated files.

A.1.1. Creating the Failure Report

The failure report lists runs that failed for either compared executable. The report generator identifies failures with the given checks in the order listed.

1. If the statistics file is necessary and missing, the run has failed. The report configuration determines whether statistics are necessary (see Section A.2.3, “Element: <success>”). If no other success conditions are defined and the statistics file is available, the run is a success.

2. If the statistics file is present and lists at least one fatal error, the run has failed.

3. If no output file matches a success condition, the run has failed. This can mean either no output files are found, or those that are found fail to meet the success criteria (see Section A.2.3, “Element: <success>”).

Once failures are identified, the report is written to the file `FailureReport.html`. Failures are sorted in the following order:

1. Unexpected: ran in base but failed in mod
2. Expected: failed in both
3. Fixed: failed in base but ran in mod
4. Misc: the run is not present for one version and failed in the other

A.1.2. Creating Statistics Reports

Statistics reports detail differences between the two code versions for a given statistic. Any given run is only included if its statistics file exists for both versions and both contain the requested value. The report generator computes the difference between the value in both the base version and mod version for each input in the given suite. If the difference exceeds the threshold or percentage threshold set in the report configuration, the value is included in the results (see Section A.2.2, “Element: <stat>”). The differences are sorted and printed out in an HTML report.

A.1.3. Creating the Differences Report

The differences report tabulates diffs between various output files. Unless defined in the report configuration, this report is not generated (see Section A.2.4, “Element: <diff>”). Any given run is only diff'd if the corresponding files are present in at least one of the version directories. The report generator runs the defined diff command, substituting the location of the base and mod output files. The results are saved to a file with the following path:

<report directory>/<diff directory>/<suite>/<file name>.diff

If the file size of the diff does not meet or exceed the threshold set in the report configuration, the diff is deleted. Otherwise, a link to the diff is included in the report, which is written to the file `DiffReport.html`.

A.2. Report Definition

The report generator is configured by reading a report definition XML file. This file tells the report generator which statistics are of interest, how to locate runs, how to determine run success, whether to diff files and which files to diff, and a number of other miscellaneous values.

Report files must be both well-formed and valid, or report generation will fail. As in all XML documents, elements, attributes, and attribute values are case-sensitive. The root tag of a report file is the **`report`** element. Elements in the **`report`** block may only appear in the following order: **`stat`**, **`success`**, **`diff`**.

A.2.1. Attributes: <report>

The root **`report`** tag has the following attributes.

`title` - The title of the report. This value must contain at least one non-whitespace character. This value is required.

`description` - A description of the report. This value is optional.

`format` - A Java format-string for numerical values written in reports. This value is optional. For more information about legal format-strings, see the Formatter class in the Java API [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Formatter.html].

`statsExtension` - The extension of statistics files. At least one non-whitespace character must be present. The period delimiter used to separate the file basename and extension cannot be included. This value is optional and defaults to `"xmlstat"`.

A.2.2. Element: <stat>

The **stat** empty element defines a statistic used to generate a report. At least one **stat** element must be defined. It has the following attributes.

`name` - The name of statistics included in the report. Values must contain at least one non-whitespace character. This attribute is *required*.

`file` - The name of the HTML file created for the report. For portability, this value has been restricted to the following. Values must begin with at least one letter or number. Afterwards, the file name may consist of any number of letters, numbers, and periods. This attribute is *required*.

`title` - The title of the report. Values must contain at least one non-whitespace character. This attribute is *required*.

`description` - A description of the report. This attribute is *optional*.

`threshold` - A difference threshold for statistics. The magnitude of the difference between statistics must exceed this value to be considered for the report. A difference within this threshold may still be significant if a `factor` is specified (see below). Only non-negative real numbers may be provided. This value is *optional* and defaults to `"0"` if unspecified.

`factor` - A percentage of the base value that differences must exceed to be considered for the report. This percentage is specified with 1.0 as 100%. Only non-negative real numbers may be provided. This value is *optional*. There is no default value for this attribute, so a percentage-based difference is not considered if `factor` is unspecified.

`prefer` - Indicates the preferred trend in differences. By default, the results of a report are sorted so that favorable results are displayed first; this attribute indicates what kind of differences are improvements. Only `"decrease"` and `"increase"` may be specified for this attribute. This value is *optional* and defaults to `"decrease"`.

A.2.3. Element: <success>

Defines success conditions for code output. Exactly one **success** block may be defined; if unused, the report generator will not generate a failure report.

The **success** element defines one attribute: *stats*. A positive *stats* value indicates that the existence of a statistics file is a necessary indicator of success. Allowable values are "true" and "false". This attribute is *optional* and defaults to "false". The body of the **success** element consists of one or more **condition** elements. Each **condition** element defines a file-set and success criteria through its attributes and children.

The **condition** element defines only one attribute: *extensions*. The value is a comma-separated list of file extensions, sans period, that indicate which output files to check with the **type** criteria. Each **type** has in its body one or more **message** and/or **regex** elements. These values define a string or regular expression that must be found or matched somewhere in a line of an output file for the run to be considered successful. Both elements require that the expression be contained within the body of the element. Furthermore, an optional *case* attribute is available to indicate whether the expression is case-sensitive (allowable values are "true" and the default "false").

Note Regular expressions must be legal as defined in the Java regex API. The allowed syntax is largely identical to Perl 5 (only a few constructs are unavailable), and should be immediately familiar to seasoned regular expression authors. For more information, see the documentation of the Pattern class [http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html].

Special considerations need to be made when designing success conditions. The report generator runs through matching outputs once for each defined **condition**. This means that if two **condition** elements define the same extension, *each corresponding output file will be read twice*. For any given type, only one of the expressions must match for the run to be considered successful, allowing parallel types to be collapsed into a single element. Consider the following example:

A code creates two output files for any given run: one with the extension .info and the other with .out. The message "done" indicates success in both types of files. The regular expression "^complete:" also indicates success if found in .info files. The following conditions block would correctly locate successful runs.

```
<success>
    <condition extensions="out,info">
        <message>done</message>
    </condition>
    <condition extensions="info">
        <regex>^complete:</regex>
    </condition>
</success>
```

As pointed out earlier, this will cause the report generator to read .info files twice. After collapsing the two types, the redundant reads are eliminated.

```
<success>
    <condition extensions="out,info">
        <message>done</message>
        <regex>^complete:</regex>
    </condition>
</success>
```

Unfortunately, this could still lead to problems. If the regular expression matches a line in `.out` files that does *not* indicate success, the report generator can make false positives. The following eliminates any ambiguity.

```
<success>
    <condition extensions="info">
        <message>done</message>
        <regex>^complete:</regex>
    </condition>
    <condition extensions="out">
        <message>done</message>
    </condition>
</success>
```

Each file is now read exactly once with only those success-conditions that apply.

A.2.4. Element: <diff>

Indicates that a diff utility should be run on various code outputs. When present, links to the diffed files are tabulated in a differences report. **`diff`** has the following attributes:

`program` - The differencing application to use. For Windows users, the default value "diff" indicates that the included GNU diff command line utility should be used.

`command` - The diff command arguments. This value supports two symbolic constructs: `%base` and `%mod`. These two values are replaced with the path of the diffed files from the base and mod code output. The attribute value must include both `%base` and `%mod` at least once. If undefined, the value defaults to `"%base %mod"`.

`extensions` - The extensions of files to diff, sans period, in a comma-separated list. This value must contain at least one non-whitespace character. If undefined, it defaults to `"out"`.

`threshold` - The size in bytes that a diff must meet or exceed before included in the report. This value must be a non-negative integer. If undefined, it defaults to `"20480"` (20kB). To include every diffed file, set *`threshold`* to `"0"`.

A.3. Statistics File

The statistics file format stores named numerical values and error descriptions in XML. Like report configuration files, the XML contents must be both well-formed and valid. Any code that outputs statistics files in this format can be used with the report generator.

The statistics file has only a few elements: a root **`statistics`** tag, a block of **`stat`** elements, and an optional block of **`error`** elements. The **`statistics`** element serves

only as the root tag; no attributes are allowed. Each **`stat`** defines a value through two attributes: *`name`* and *`value`*. The constraints on these attributes are straightforward: both are required, *`name`* must contain at least one non-whitespace character, each **`name`** must be unique within the file, and *`value`* must be a real number.

Statistics files may also define any number of empty **`error`** elements after the **`stat`** block. The following attributes are available for **`error`**.

`type` - The type of error. Allowable values are `"input"` and `"runtime"` This value is *required*.

`reason` - Specifies a short description of the error. Values must have at least one non-whitespace character. This value is *required*.

`fatal` - Indicates whether the error is fatal, causing the executable to terminate early. Allowable values are `"true"` and `"false"`. This value is *optional* and has no default.

`componentType` - If the error can be directly associated with a component, that component's type is specified here. Values must have at least one non-whitespace character. This value is *optional* and has no default.

`componentID` - If the error can be directly associated with a component, that component's identity is specified here. Values must have at least one non-whitespace character. Values must have at least one non-whitespace character. This value is *optional* and has no default.

`lineNumber` - This attribute can be utilized to specify a specific line in the input file that caused the error. Only positive integers may be specified. This value is *optional* and has no default.

`time` - Allows specifying the simulated time when the error occurred. Only real numbers may be provided. This value is *optional* and has no default.

Below is an example of the statistics file syntax.

```
<?xml version="1.0"?>
<statistics>

    <stat name="cpu time" value="10.0" />

    <error type="input"
           lineNumber="42"
           reason="first column must be greater than 0"
           fatal="false"
           componentType="pipe"
           componentID="11" />

    <error type="runtime"
           time="10.5"
           reason="divide by 0"
           fatal="true"
           componentType="pipe"
           componentID="11" />

</statistics>
```

Appendix B. Developing AVF Plug-ins

The AVF plug-in provides a plug-in interface for loading case-specific functionality at run time. This allows AVF to support additional codes without any modification to the basic SNAP plug-in. This section details how to construct such a plug-in.

Note To build these plug-ins, you will need to set up a classpath including SNAP and AVF classes.

AVF plug-ins are deployed as a single JAR file. All required class resources must be in the archive. Furthermore, the archive's MANIFEST must indicate the name of the plug-in's CaseSupport implementation via the property `AVScriptCaseSupport-Class`. Once properly packaged, installation is as simple as copying the JAR to the `avf/plugins` directory of a SNAP installation.

Plug-ins only need to implement two classes. CaseSupport defines labels and an icon used to identify the new case type. It also provides a factory method for creating CaseDefinition instances, which define all other necessary information and functionality. Both are abstract classes and must be extended. The methods outlined in the following section are sufficient for all anticipated case types.

B.1. CaseSupport

Implementations of CaseSupport define how to refer to a case type, what icon used to indicate the type in the ModelEditor, and how to create CaseDefinition instances specific to the new type. The full name of this class is `com.apt.avscript.plugin.CaseSupport`.

`public abstract String getID()` - Returns a unique keyword used as a label to identify this code support. Most often this should be the name of the supported analysis code. More specifically, this should be the name of the SNAP plug-in that supports the analysis code.

`public String getExecutableLabel()` - Returns a keyword used as a label to identify executables for this code support. This value will not be used in inputs, but instead as a generic way of referring to the supported code. Most often this should be the same value returned from `getID`.

`public ImageIcon getIcon()` - Returns an icon associated with cases of this type. This icon will be displayed in the Navigator for Input Models whose *Case Type* is set to the value returned from `getID`. The default implementation returns `null`.

`public abstract CaseDefinition createCaseDefinition()` - Instantiates and returns the CaseDefinition implementation associated with the case type.

B.2. CaseDefinition

Implementations of CaseDefinition indicate how to process the case type. CaseDefinition is an abstract class that contains a number of utility methods used in processing cases; only those defined here need to be implemented or overridden. The full name of this class is `com.apt.avscript.plugin.CaseDefinition`.

`public abstract String getServerPluginID()` - Returns the SNAP plug-in ID of the plug-in used to process inputs of this type. In most cases, this will be the same value returned from `getID` in the associated CaseSupport implementation. However, as it is feasible that a CaseSupport implementation might have an ID that does not match the SNAP plug-in ID, this method is provided to guarantee the ability to submit cases to a Calculation Server.

`public boolean isExecutable()` - Returns true if the case can be executed as a Calculation Server job. Most implementations will return `true`. The return value of this method has a number of implications as to which methods must be overridden to provide non-null values. The default implementation returns `false`.

`public abstract String getDataSourceFileExtension( boolean demux )` - Returns the file extension of case data files after the case has been run. As the AVF looks for data files by appending the extension directly to the case name, a period prefix *must* be part of the returned result if the file extension is delimited from the file basename with a period. The `demux` argument is used to indicate whether the application is looking for the standard data or demultiplexed data. Implementations may ignore `demux` if only one extension is appropriate.

`public boolean isBatchOpenCommandRequired()` - Determines whether data files must be opened with a certain type of command before their data is available. Cases that return true must be opened with a command of the form

```
<open command> [file index] [file type] "<file path>"
```

where each token is variable, and tokens between square brackets may be optional. This is the normal form for opening most analysis code plot files in AptPlot; most implementations will return `true`. For example, a RELAP5 demultiplexed file might be opened with the command:

```
RELAP 0 DEMUX "/home/user/relap.dmx"
```

The default implementation returns `false`.

`public void writeCustomExtractBatch( java.io.PrintWriter writer, java.io.File dataFile, DataDefinition dataDef )` - Writes custom AptPlot batch commands to open the case data file. This method must be implemented if `isBatchOpenCommandRequired` returns `false`. The parameters are as follows: `writer` is an open PrintWriter used to write the batch commands, `dataFile` describes the path to the data file, and `dataDef` contains the definition of the data trace to be extracted. The generated batch commands should follow the basic

convention of loading data into the *next available data set in the current graph*. The default implementation is an empty method.

`public String getBatchOpenCommand()` - Returns the AptPlot batch command used to open case data files. This is the `<open command>` portion of the batch command described for `isBatchOpenCommandRequired`. Implementations that return true from `isBatchOpenCommandRequired` must return a non-`null` value. The default implementation returns `null`.

`public String getBatchFileType( boolean demux )` - Returns the `[file type]` portion of the batch open command illustrated for `isBatchOpenCommandRequired` above. In the sample `RELAP` command, the file type `DEMUX` is provided to indicate the file is a demultiplexed file. The semantics of which file type is required (if one is required at all) are unique to each analysis code, and are described in the documentation for AptPlot. Implementations that return `true` from `isBatchOpenCommandRequired` must return a non-`null` value (although the result may be an empty string when a file type is unnecessary). The default implementation returns `null`.

`public String getBatchReadCommand()` - Returns the read command used to extract data after opening a file with a batch open command. This command is used both to add channels to the 'read list' and to initiate an actual read. For example, to read a single TRACE channel named `cb60` at open file 1 in SI units, AVF would generate

```
TREAD 1 "cb60" SIU
TREAD DONE
```

Implementations that return `true` from `isBatchOpenCommandRequired` must return a non-`null` value. The default implementation returns `null`.

`public String getBatchChannelPrefix()` - Returns the channel prefix used to reference case channels in equation interpreter expressions. This is best illustrated by example: the command to load the data channel pn-1A01 into a vector from a TRACE data file at index 1 might look like:

```
CALC "<channelData> = T1_pn-1A01"
```

In the example, the "T" in `T1_pn-1A01` is the channel prefix. Implementations that return `true` from `isBatchOpenCommandRequired` must return a non-`null` value. The default implementation returns `null`.

`public boolean isBatchChannelParenthesized()` - Returns `true` if channel names in equation interpreter expressions must be encapsulated with parentheses. In the sample command shown above for `getBatchChannelPrefix`, the channel name `pn-1A01` is obviously not parenthesized (nor should it be). If parentheses were required, this expression would look like `T1_(pn-1A01)`. The default implementation returns `false`.

`public String[] getInputTypeKeywords()` - Returns a newly allocated array of supported input type keywords. If there is no need to distinguish between

input types for the implemented case, return an array of zero length. The default implementation returns an empty array.

Index

A

AVScript input file names, 25

C

checking the model, 15
creating AVScript inputs, 5
    ACAP, 14
    annotations, 8
    cases, 7
    data traces, 10
    editing in AptPlot, 28
    figures, 8
    importing input files, 25
    pages, 13

E

editing AVF models, 4

S

submitting jobs, 16