Avatox User Guide

Version 1.8

McKae Technologies

www.mckae.com

Avatox Overview

Avatox ("Ada, Via Asis, To Xml") is an application that generates XML representations of Ada source code. The XML format for the converted source code is called AXF, for Avatox XML Format, and is stored in files with a ".axf" type extension. The user provides the filenames of one or more compilation units that Avatox semantically analyzes using an Ada Semantic Interface Specification (ASIS) toolkit, with the aid of which a corresponding XML representation of the source code is produced. Avatox' additional capabilities include identifying supporting or closure[1] compilation units and generating XML for those related units, generating supplemental annotations for various Ada elements that can be exploited for analysis and transformation, and can immediately apply an XSL (XML Stylesheet Language) stylesheet to the resulting AXF file(s).

Here is an example of the AXF generated for a context clause:

```
with Ada.Calendar.Formatting;
```

becomes:

```
<aClause pedigree="asis" startLine="29" startCol="1" endLine="29" endCol="29">
 <aWithClause pedigree="asis" startLine="29" startCol="1" endLine="29" endCol="29"/>
 <anExpression pedigree="asis" startLine="29" startCol="6" endLine="29" endCol="28">
   <aSelectedComponent pedigree="asis" startLine="29" startCol="6" endLine="29" endCol="28"/>
   <anExpression pedigree="asis" startLine="29" startCol="6" endLine="29" endCol="17">
     <aSelectedComponent pedigree="asis" startLine="29" startCol="6" endLine="29" endCol="17"/>
     <anExpression pedigree="asis" startLine="29" startCol="6" endLine="29" endCol="8">
        <anIdentifier pedigree="asis" ident="Ada" startLine="29" startCol="6" endLine="29"
                                                                                endCol="8"/>
     </anExpression>
     <anExpression pedigree="asis" startLine="29" startCol="10" endLine="29" endCol="17">
        <anIdentifier pedigree="asis" ident="Calendar" startLine="29" startCol="10" endLine="29"
                                                                                endCol="17"/>
     </anExpression>
   </anExpression>
   <anExpression pedigree="asis" startLine="29" startCol="19" endLine="29" endCol="28">
     <anIdentifier pedigree="asis" ident="Formatting" startLine="29" startCol="19" endLine="29"
                                                                                endCol="28"/>
   </anExpression>
 </anExpression>
</aClause>
```

This AXF representation almost exactly corresponds to the sequence of elements encountered while doing a depth-first ASIS traversal of an Ada compilation unit. The AXF representation is obviously far more verbose than the single context clause, which is to be expected since AXF is depicting the detailed

1 Due to bugs in the ASIS-for-GNAT implementation, closure identification is temporarily disabled.

structural information of the source code statement. Thus tools or transformations can be performed at a very detailed level of operation.

To assist with analyses and transformations Avatox can generate annotation elements that supplement the ASIS-based elements. These annotations are presented as axfPoint elements, axfPoint being "AXF POints Of INformation for Transformation". The following AXF fragment includes relevant axfPoint elements (and also has the line and column information removed—via an XSL stylesheet—to declutter the content):

```
<aClause pedigree="asis">
   <aWithClause pedigree="asis"/>
   <anExpression pedigree="asis">
      <aSelectedComponent pedigree="asis"/>
      <anExpression pedigree="asis">
         <aSelectedComponent pedigree="asis"/>
         <anExpression pedigree="asis">
            <anIdentifier pedigree="asis" ident="Ada"/>
            <axfPoint pedigree="axfPoints" axfXref="Ada">
               <axfPoint pedigree="axfPoints" axfXrefName="Ada" axfXrefLevel="0"/>
            </axfPoint>
         </anExpression>
         <anExpression pedigree="asis">
            <anIdentifier pedigree="asis" ident="Calendar"/>
            <axfPoint pedigree="axfPoints" axfXref="Ada.Calendar">
               <axfPoint pedigree="axfPoints" axfXrefName="Ada.Calendar" axfXrefLevel="0"/>
            </axfPoint>
         </anExpression>
      </anExpression>
      <anExpression pedigree="asis">
         <anIdentifier pedigree="asis" ident="Formatting"/>
         <axfPoint pedigree="axfPoints" axfXref="Ada.Calendar.Formatting">
            <axfPoint pedigree="axfPoints" axfXrefName="Ada.Calendar.Formatting" axfXrefLevel="0"/>
         </axfPoint>
      </anExpression>
   </anExpression>
</aClause>
```

The highlighted axfPoint elements provide cross-referencing information about each of the identifiers that name the hierarchy of library packages referenced in the context clause. There are other kinds of axfPoint elements, and those are described in detail in the Avatox XML Format document.

Avatox Usage

Avatox is invoked from the command line and has the following usage profile:

```
avatox Unit(.ads|.adb) ... [-I <include-dir>...]
               [-o <output-file>|-m <directory>]
               [ -k] [ -t] [ -x] [-v] [[-c | -s] [-p]]
               [-e "<regexp>" ...] [-f "<exp>" ...]
               [-ne "<regexp>" ...] [-nf "<exp>" ...]
               [-a] [-axfxr] [-axfsc] [-axftr]
               [-xsl <stylesheet-file>] [-xslto <output-file>]
               [-xslext <file-extension>]
               [-xslp <param=value> ...]
```

The name of one or more Ada source code units are provided on the command line. As Avatox is geared to work specifically with AdaCore's ASIS-for-GNAT distribution, Ada source files must follow the standard GNAT file naming conventions. The “ads” file type extension, for specifications, or “adb”, for unit bodies, must be included as part of the filename. A wildcard specification for selecting files can be used.

The options fall into four categories:

- Unit selection, output, and formatting
- Unit filtering
- axfPoint annotation element generation
- Stylesheet application

For all options that take a directory or filename argument, the space between the option identifier and its argument is optional.

Unit Selection, Output, and Formatting

`-I <include-dir> ...`

Any external directories containing files on which the selected units depend must be specified using the -I directive. Relative or absolute paths can be supplied, and separate include directives must be provided for each included directory.

`-o <output-file>`

If all of the AXF produced for the one or more units specified on the command line is to be placed in a single file, the -o option gives the name of that file. If AXF for multiple units are being generated, the AXF for all the units will be concatenated within a single `codeRepresentation` AXF element.

`-m <directory>`

In contrast to placing all the AXF into a single file via -o, or writing it all to stdout, specifying -m with a directory will store the AXF for each processed unit in that directory, each file named with its source file name to which an “axf” type extension is appended. For example, the filename for the Vatox.Traversal package specification, whose source file is vatox.traversal.ads, is vatox-traversal.ads.axf. (To have Avatox write all the files into the current directory, use “-m.”)

If neither the -o or -m options are given, the generated AXF for all units is concatenated together and written to stdout.

`-k`

“Krunch” the AXF output, removing all line feeds and indentation. This is much harder for a person to

read, but saves space and transmission time for AXF that is normally going to be fed into another XML processing application.

-x

The default capitalization style for AXF documents is ALL CAPITALIZED, with element names that are derived from ASIS identifiers being the 'Image of that identifier, meaning that the words of the identifier are also separated by underscores. The -x option employs a “camelBack” capitalization style and removes all underscores, so that the resulting AXF representation looks more like typical XML-based documents. So:

```
<A_SELECTED_COMPONENT>
```

is represented instead as:

```
<aSelectedComponent>
```

-v

Verbose. Print progress messages as Avatox processes unit files and generates AXF output.

-t

Part of Avatox' processing causes Ada source code units to be compiled to generate their “compilation tree” files, with the ASIS information needed for the AXF then being extracted from these trees using ASIS queries. Normally these intermediate tree files are deleted at the conclusion of Avatox' execution, but by providing this option, those tree files are retained. Normally this option is needed only when either some other ASIS-based application will be processing the same set of source code, and it needs the tree files to be pre-created in order to function.

Avatox needs the tree files to be pre-created when it is going to be processing code using a non-ASCII character set, so the user probably does not want to have to regenerate the tree files between each Avatox invocation. For information on processing non-ASCII character sets, see “Processing UTF-8”.

-s

In addition to generating AXF for the explicitly listed units, process their “Supporting” compilation units. Supporting units are those whose presence is required for the listed unit(s) to compile. These would be generic or regular package and subprogram specifications. The identification of supporting units is transitive, so the supporters of supporting units are collected and corresponding AXF generated for them as well. Supporting units can be collected and processed all the way back to the standard Ada language units (such as package Ada and its children) if desired, see the -p option below. (By default the predefined units are omitted.) This option is incompatible with the -c option.

-c

(The -c, Closure, option is temporarily disabled due to bugs in the GNAT GPL 2006 ASIS-for-GNAT

implementation.)

In addition to generating AXF for the explicitly listed units, process the compilation units making up their Closure. Closure units are those whose presence is required for the listed unit(s) to compile and link, so in this case the explicitly provided units must be legal as Ada main program unit(s). Closure units can be collected and processed all the way back to the standard Ada language units if desired, see the -p option below. (By default the predefined units are omitted.) This option is incompatible with the -s option.

`-p`

Generate AXF for the Predefined Ada language units that are identified when collecting Supporting (-s) or Closure (-c) units. By default AXF is not generated for these units. AXF is never generated for packages System and Standard, since these are built into the compiler and therefore there is no information accessible to ASIS. This option is valid only when either -s or -c is given.

Unit Filtering

One or more filters can be specified in order to process a subset of compilation units from those that have been provided to Avatox for processing. A list of multiple candidate units needing filtering is most likely either the result of a wildcarded file specification, or from the set of supporting or closure units. Two kinds of filters are supported, regular expression and file wildcard style.

The letters in Avatox' filter expressions *should use lower-case letters*. Because Ada is not a case-sensitive programming language, Avatox *converts all unit names to lower-case before applying the filter(s)*. So a filter that contains capitalized letters *will not match on those letters*.

Expressions specified as filters must be defined to match the entire compilation unit name, Avatox filtering does not function as a "substring match". For example, `-f "vatox"` will match only a unit named Vatox, and will match neither "avatox" nor "vatox.traversal". To match those, the filter expression must be "*vatox*" to match both, or "vatox*" to just match the latter one.

`-e "<regexp>"`

Filter the list of compilation units through the regexp (regular expression), with only those matching the expression being processed further. Regular expression syntax is quite powerful and discussion of it is beyond the scope of this user guide. Web searches for "Regular Expression" or "regexp" will provide an abundance of useful information.

`-f "<exp>"`

Filter the list of compilation units through a filename-style wildcarding pattern, i.e, which uses '*', '?', and the '[' ']' bracket pair. Only those units passing the filter will be processed.

`-ne "<regexp>"`

Filter the list of compilation units through the regexp, but with only those **not** matching the pattern being processed.

`-nf`

Filter the list of compilation units through the filename-style wildcarding pattern, with only those units not matching the pattern being processed.

The negation filters, -ne and -nf, can be useful when processing a unit and its closure or supporting units, and one wishes to exclude an entire package hierarchy, e.g., `-ne "asis.*"`. It also works to exclude single units by simply specifying a -ne or -nf and the complete unit name.

Multiple filters, of both wildcarding kinds and both inclusion and exclusion, can be provided, with the list of candidate units being filtered through each in turn in the order in which it appears on the command line.

It is strongly recommended that filter expressions be enclosed in quotes to prevent the command shell from treating them as actual file wildcards and expanding the filtering expression into a list of files. For example, attempting to use `-f vatox*` as a filtering expression, could result in Avatox actually received as its arguments:

```
-f vatox.ads vatox-axf_pedigrees.adb vatox-axf_pedigrees.ads vatox-axf_points.ads
```

Avatox would then use "vatox.ads" as the filter, which is not the name of any compilation unit, and so no units would be processed into AXF.

axfPoint Element Generation

AXF Points Of INformation for Transformation (axfPoint) elements provide annotations supporting analysis and transformation of AXF content. Some axfPoints support source language independence, while others simplify the referencing and cross-referencing of identifiers. More detailed information about each of the axfPoint element types in available in the Avatox XML Format document.

`-axfxr`

Generate cross-reference axfPoint elements to unambiguously specify a particular identifier.

`-axfsc`

Generate "enclosing scope" axfPoint elements to uniquely identify a particular identifier, in a way that cross-reference axfPoint elements can reference.

`-axftr`

Represent language "terminals", i.e., numeric literals and arithmetic operators, using a language-independent notation. Non-base 10 numeric literals are represented using Ada's based literal notation, and arithmetic operators are associated with enumerations, e.g., axfPlus, axfNE, etc.

`-a`

Shorthand option to select all axfPoint annotation element options.

Stylesheet Application

If Avatox is built with support for XML Stylesheet Language (XSL) transformations, the name of a file containing an XSL stylesheet can be specified on the command line and applied to the generated AXF file(s). For each of the one or more AXF files produced (including output to stdout), the stylesheet will be applied to it and a corresponding resulting document generated.

`-xsl <stylesheet-file>`

The name of the file containing a valid XSL stylesheet.

`-xslto <output-file>`

The name of the single output file into which to write the result of applying the stylesheet to the single generated AXF output file (including stdout).

`-xslext <file-extension>`

The file extension, without the ".", to append to a unit's filename when generating a filename for the AXF for a given unit. If this is omitted, the default file extension is "axt" (Avatox Xml Transformed).

`-xslp <"param=value">`

Set any global XSL parameters, "`<xsl:param ... >`", declared in the stylesheet. Each -xslp argument must be of the form "param=value". It is recommended that the entire expression be enclosed in quotes. And recall that when a param is set to a value, if that value is alphanumeric it is treated as an XML element tag, to treat it as a string, it must be enclosed in quotes.

For example:

```
"name=adric"          <!-- Sets the parameter 'name' to reference the element adric -->
```

versus

```
"name='adric'"        <!-- Sets the parameter 'name' to the string 'adric' -->
```

If the expression is not enclosed in quotes, there must be no spaces between the '=' operator and the param and value strings. The best thing to do, though, is just enclose the whole param/value expression in quotes.

Interactions amongst -xslto, -xslext, -o, and -m

These options can only appear in certain combination with each other and stdout, due to where they generate their output.

-o and -m obviously cannot appear together because -o directs that output be written to the specified file, while -m directs that AXF be written into files named after the compilation units that are processed.

Similarly for -xslto and -xslext, -xslto specifies the file where the result(s) of the XSL transformation will be placed, while -xslext is assuming that multiple files are going to be generated, and it's specifying the type extension for them—therefore -xslext is valid only when -m is specified.

When an XSL transformation is being performed:

- If -o is specified, *or neither* -o *nor* -m (which means the AXF is to be written to stdout) is specified, then -xslto names the file into which to write the transformed AXF.
- If none of -o, -m, -xslto, or -xslext are specified, then the transformed document is written to stdout, immediately after the generated AXF is written to stdout.
- If a -m directory and no -xslext extension is specified, the transformed files will be given a name generated by combining the unit's filename and the default extension ("axt"). The transformed units will be placed into the directory specified by -m.
- If a -m directory and a -xslext extension is specified, the transformed files will be given a name generated by combining the unit name with the designated extension, and placed in the -m specified directory.

XSL Transformation Temporary Files

When an XSL transformation is performed on AXF output that is being written to stdout, or the transformed output is written to stdout, temporary files are utilized. Normally these files are automatically deleted at the end of Avatox' execution, but if that execution is interrupted for some reason those files will remain in the current working directory, and would need to be manually deleted. (Leaving them causes no harm, they simply take up disk space.) The names of the files consist of ".avx" to which is appended a date/time string, such as ".avx2007-02-12-01:42:34" for the AXF output, and ".avx", the date/time string, and ".axt" for stylesheet output that was going to stdout.

Building Avatox

Obviously an ASIS implementation must be installed. Avatox is built with the ASIS-for-GNAT distribution. Avatox utilizes Ada 2005 features and some GNAT vendor-supplied utilities.

Some of the Mckae Technologies' McKae components (with version 1.05 or better of XML EZ Out) are utilized, so the necessary subset of that whole collection accompanies the distribution in the mckae subdirectory.

In addition, a subset of Dmitri Kazakov's "Strings_Edit" utilities are included for handling UTF-8 character encoding. The entire Strings_Edit collection is available at http://www.dmitry-kazakov.de/ada/strings_edit.htm.

Avatox can be built with or without support for XSL transformations, depending on whether the libxslt library is available on the target platform.

When building with GNAT GPL 2006 or a comparable compiler, the default project file builds Avatox with XSL transformation support:

```
gnatmake -Pavatox.gpr
```

To omit XSL support, build with:

```
gnatmake -Pavatox_noxsl.gpr
```

To explicitly build Avatox without using a project file, invoke gnatmake with the appropriate options:

```
gnatmake -O2 avatox.adb -Imckae -Istrings_edit_subset -I<path-to-ASIS> \
    -Iwith_xslt -largs -L<path-to-asis-libs> -lasis -lxslt
```

or without XSL support:

```
gnatmake -O2 avatox.adb -Imckae -Istrings_edit_subset -I<path-to-ASIS> \
    -Ino_xslt -largs -L<path-to-asis-libs> -lasis
```

Processing UTF-8

The ASIS-for-GNAT implementation distributed with GNAT GPL 2006 distribution does not support on-the-fly compilation when the source file uses a character encoding other than ASCII, because there is no way to specify the "-gnatW8" switch through the ASIS interface to the compiler[2]. Therefore the corresponding tree files have to be manually generated prior to Avatox' invocation. This is done with the aid of the "-gnatc" and "-gnatt" pair of switches.

The easiest way to do this is by using gnatmake, for example:

```
$ gnatmake -c -gnatc -gnatt -gnatW8 bisiesto.adb
```

This generates the needed tree, ".adt", files that Avatox will then recognize and process (and the -t option can be employed to retain existing adt tree files, so that they do not necessarily have to be regenerated for files unaffected by subsequent source updates):

---

2 Through personal correspondence I was informed that an upcoming release of ASIS-for-GNAT will support the specification of the -gnatW*x* switch. 18 Sep 2006.

```
$ avatox bisiesto.adb -o bisiesto.adb.axf -t
```

If -I, include directory, specifications are required to compile, they can be specified as usual on the gnatmake command line.

Invocation Examples

These examples use a source code base the implementation of Avatox itself. The $GNAT environment variable represents the root of the system's GNAT installation directory, below which the ASIS library resides, in $GNAT/include/asis. When a command line stretches over more than one line, the *nix continuation character, "\", will indicate that the entire line comprises a single Avatox invocation.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis
```

Convert the Avatox main subprogram into AXF and write it to stdout.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis -o avatox.adb.axf
```

Same as above, except the output is written to the file avatox.adb.axf.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis -o avatox.adb.axf \
      -axfsc -axftr
```

Convert the Avatox main program into AXF, and include generation of scope and cross-reference axfPoint elements. The generation of axfPoint elements also implicitly sets the -x option, to force use of the camelBack XML naming style.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis -o avatox_supporters.axf -a -s
```

Convert the Avatox main program into AXF, along with all the compilation units which support its compilation (-s), excluding the language-predefined ones (no -p). Include generation of all axfPoint elements. Write the entire collection of AXF into avatox_supporters.axf.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis -m /tmp/avx_files -a -s -p -v
```

Convert the Avatox main program and its supporting units (including those predefined by the language via the -p option) into AXF, place each unit's AXF in its file in the /tmp/avx_files subdirectory, and show verbose progress messages as this is done.

```
$ avatox avatox.adb -Imckae -I$GNAT/include/asis -m /tmp/avx_files -a -s -p -v \
      -ne ".*asis.*" -nf "*a4g*"
```

Convert the Avatox main program and all its supporting units into AXF, but exclude those whose unit names contain the substrings "asis" or "a4g".

```
$ avatox *ada_refs* -Imckae -I$GNAT/include/asis -m.
```

Process those files matching the "*ada_refs*" wildcard, and output their AXF into individual files within the current directory. (With the Avatox 1.6 distribution, two files are generated: vatox-axf_points-references-ada_refs.ads.axf and vatox-axf_points-references-ada_refs.adb.axf.)

```
$ avatox vatox*.adb -Imckae -Istrings_edit_subset -I$GNAT/include/asis \
      -m axf_repository -s -nf vatox
```

Process all the files in the Vatox library hierarchy and their supporting, non-language defined, units, and excluding the Vatox package itself. Output their AXF into individual files into the axf_repository subdirectory beneath the current directory.

```
$ avatox vatox*.adb -Imckae -Istrings_edit_subset -I$GNAT/include/asis \
      -m axf_repository -s -nf vatox -xsl transforms/deleteLCInfo.xsl
```

Process all the files in the Vatox library hierarchy as above, and then subject each generated axf file to the supplied stylesheet that removes the line and column span information (which declutters the files if line/column information is not needed).

```
$ avatox vatox*.adb -Imckae -Istrings_edit_subset -I$GNAT/include/asis \
      -m axf_repository -s -nf vatox -xsl transforms/deleteLCInfo.xsl \
      -xslext xfm -xslp "param1='value1'"
```

Process all the files in the Vatox library hierarchy as above, subject each generated axf file to the supplied stylesheet, and set the file extension for each to "xfm". Pass in an XSL parameter, giving the parameter "param1" the string value "value1" (the deleteLCInfo.xsl stylesheet contains no parameters, so that parameter association is ignored in this situation).

Sample AXF Queries

Given the generated XML file, one can query and modify it as desired. Here are some sample queries:

1. Retrieve all the comments (these examples assume the default 'Image representation of the ASIS kinds):

    //A_COMMENT/text()

2. Find all referenced operators:

    //AN_OPERATOR_SYMBOL/@operator

3. Find all string literals that appear in a statement:

    //A_STRING_LITERAL[ancestor::A_STATEMENT]/@literal

4. Find all string literals that appear in a variable declaration:

    //A_DECLARATION[A_VARIABLE_DECLARATION]/

descendant::A_STRING_LITERAL/@literal

5. Find the line of a comment that immediately precedes a declaration

    //A_COMMENT[following-sibling::A_DECLARATION][position()=last()]

6. Find every assignment statement

    //AN_ASSIGNMENT_STATEMENT

7. Find every function call invoked as the right hand side of an assignment

    //A_FUNCTION_CALL/../preceding-sibling::AN_ASSIGNMENT_STATEMENT

8. Find all integer literals

    //AN_INTEGER_LITERAL/@literal

The XIA (XPath In Ada) distribution, available at www.mckae.com/xia.html includes a test_xpath application which will load an XML file and apply user-defined queries such as those above to it.

Licensing

Avatox is licensed as GPL, due to it being derived from the GPL-licensed distribution of the Display_Source application that accompanies AdaCore's ASIS distribution.

The supporting McKae components, for command line processing, lexical name transformation, Xsl transformation, and XML EZ Out, are licensed under the GNAT-Modified GPL (GMGPL).

All other components are licensed by their respective copyright holder(s).

Marc A. Criley
Mckae Technologies
www.mckae.com
18 Apr 2007