Network Working Group
Internet-Draft
Expires: April 27, 2008

N. Freed
S. Vedam
Sun Microsystems
October 25, 2007

Sieve Email Filtering: Representing Sieves and display directives in XML
draft-freed-sieve-in-xml-00

Status of this Memo

Copyright Notice

Abstract

This document describes a way to represent Sieve email filtering language scripts in XML. Representing sieves in XML is intended not as an alternate storage format for Sieve but rather as a means to facilitate manipulation of scripts using XML tools.

The XML representation also defines additional elements that have no counterparts in the regular Sieve language. These elements are

intended for use by graphical user interfaces and provide facilities for labeling or grouping sections of a script so they can be displayed more conveniently. These elements are represented as specially structured comments in regular Sieve format.

Table of Contents

1. Introduction

Sieve [I-D.ietf-sieve-3028bis] is a language for filtering email messages at or around the time of final delivery. It is designed to be implementable on either a mail client or mail server. It is meant to be extensible, simple, and independent of access protocol, mail architecture, and operating system and it is intended to be manipulated by a variety of different user interfaces.

Some user interface environments have extensive existing facilities for manipulating material represented in XML. While adding support for alternate data syntaxes may be possible in most if not all of these environments, it may not be particularly convenient to do so. The obvious way to deal with this issue is to map sieves into XML, possibly on a separate backend system, manipulate the XML, and convert it back to normal Sieve format.

The fact that conversion into and out of XML may be done as a separate operation on a different system argues strongly for defining a common XML representation for Sieve. This way different front end user interfaces can be used with different back end mapping and storage facilities.

Another issue with the creation and manipulation of sieve scripts by user interfaces is that the language is strictly focused on describing email filtering operations. The language contains no mechanisms for indicating how a given script should be presented in a user interface. Such information can be represented in XML very easily so it makes sense to define a framework to do this as part of the XML format. Structured comments can then be used to retain this information when the script is converted to normal Sieve format.

Several Sieve extensions have already been specified [RFC3431] [RFC3598] [RFC3685] [RFC3934] and many more are planned. The set of extensions available varies from one implementation to the next and may even change as a result of configuration choices. It is

therefore essential that the XML representation of Sieve be able to accommodate Sieve extensions without requiring schema changes. It is also desirable that Sieve extensions not require changes to the code that converts to and from the XML representation.

This specification defines an XML representation for sieve scripts and explains how the conversion process to and from XML works. The XML representation is capable of accommodating any future Sieve extension as long as the underlying Sieve grammar remains unchanged. Furthermore, code that converts from XML to the normal Sieve format requires no changes to accommodate extensions, while code used to convert from normal Sieve format to XML only requires changes when

new control commands are added - a rare event. An XML Schema and sample code to convert to and from XML format are also provided in the appendices.

2. Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

3. Grammatical structure of Sieve

The Sieve language is designed to be highly extensible without making any changes to the basic language syntax. Accordingly the syntax of Sieve, defined in section 8 of [I-D.ietf-sieve-3028bis], is entirely structural in nature and employs no reserved words of any sort.

Structurally a sieve script consists of a series of commands. Each command in turn consists of an identifier, zero or more arguments, a optional test or test-list, and finally an optional block containing another series of commands. Commands are further broken down into controls and actions, although this distinction cannot be determined from the grammar.

Some example Sieve controls are:

```
stop;               <-- No arguments, test, or command block
```

```
require "fileinto"; <-- Control with a single argument
if true {stop;}     <-- Control with test and command block
```

Some examples of Sieve actions are:

```
discard;            <-- Action with no arguments, test, or command block
fileinto "folder";  <-- Action with an argument
```

At the time of this writing there are no controls defined that accept both arguments and a test. Similarly, there are currently no defined actions that allow either a test or a command block. Nevertheless, the Sieve grammar allows such constructs to be defined by some future extension.

A test consists of an identifier followed by zero or more arguments, then another test or test-list. Unlike commands, tests cannot be followed by a command block.

Here are some examples of Sieve tests. Note that such tests have to

appear as part of a command in order to be syntactically valid:

```
true                <-- Test with no argument or subordinate test
header "to" "me@example.com" <-- Test with several arguments
```

Command or test arguments can be either string lists, whole numbers or tags. (Tags are simply identifiers preceded by a colon.) Note that although the Sieve grammar treats single strings as a degenerate case of a string list, some tests or actions have arguments that can only be individual strings, not lists.

Here is an example showing the use of both a test-list and a string list:

```
if anyof (not exists ["From", "Date"],
          header :contains "from" "fool@example.edu") {
  discard;
}
```

Extensions can add new controls, actions, tests, or new arguments to existing controls or actions. Extensions can also change how string content is interpreted, although this is not relevant to this

specification. However, it is especially important to note that so far no Sieve extension has added a new control to the language and it seems safe to assume that due to their nature future addition of controls will be rare.

Finally, comments are allowed between lexical elements in a Sieve script. It is very important that comments be preserved in the XML representation.

4. XML Representation of Sieve

Sieve controls and actions are represented in XML as control or action elements respectively. The command's identifier appears as a name attribute on the element itself. This is the only attribute allowed on controls and actions - arguments, tests, test-lists, and nest command blocks are all represented as nested elements. While naming the element after the control or action itself may seem like a better choice, doing so would result in extensions changing the XML schema.

The example Sieve controls shown in the previous section would be represented in XML as:

```
<control name="stop"/>
<control name="require"><str>fileinto</str></control>
```



```
<control name="if"><test name="true"/><control name="stop"/></control>
```

The example Sieve actions shown about would appear in XML as:

```
<action name="discard"/>
<action name="fileinto"><str>folder</str></action>
```

The separation of controls from actions in the XML representation means that conversion from normal Sieve format to XML has to be able to distinguish between controls and actions. This is easily done by maintaining a list of all known controls since experience indicates new controls are rarely added.

Tests are represented in the same basic way as controls and actions, that is, as a test element with a name attribute giving the test

identifier. For example:

```
<test name="true"/>
<test name="header"/><str>to</str><str>me@example.com</str></test>
```

String, number, and tag arguments are represented as str, num, and tag elements respectively. The actual string, number, or tag identifier appears as text inside the element. None of these elements have any defined attributes. Several examples of arguments have already appeared in the preceding control, action and test examples.

String list arguments are represented as a list element which in turn contains one or more str elements. Note that this the distinction between a single string and a string list containing a single string to be preserved. This is not essential since a list containing a single string could simply be mapped to a string, but it seems prudent to maintain the distinction when mapping to and from XML.

Nested command blocks appear as a series of control or action elements inside of outer control or action element. No block element is needed since an inner command block can only appear once and only after any arguments, tests, or test-lists. For example:


```
<control name="if>
  <test name="anyof">
    <test name="not">
      <test name="exists>
        <list><str>From</str><str>Date</str></list>
      </test>
    </test>
    <test name="header">
```

```
        <tag>contains</tag>
        <str>from</str>
        <str>fool@example.edu</str>
      </test>
    </test>
    <action name="discard"/>
  </control>
```

4.1. XML Display Directives

Sometimes graphical user interfaces are a convenient way to provide sieve management functions to users. These interfaces typically summarize/annotate/group/display sieve script(s) in an intuitive way for end users.

To do this effectively, the graphical user interface may require additional information about the sieve script itself. That information or "meta-data" might include, but is not limited to - a sieve name (identifying the current sieve), whether the sieve is enabled or disabled, the order in which the part of the sieve are presented to the user. The graphical user interface may also choose to provide mechanisms to allow the user to modify the script.

It is often useful for a graphical user interface to group related sieve script elements and provide an interface that display these group separately and manage things via these groupings. Some examples include Sieve statementss that together provide vacation responders, blacklists/whitelists and other types of filtering controls.

Some advanced graphical user interfaces may even provide a natural language representation of a sieve script and/or an advanced interface to present sieve statements directly to the user.

A graphical user interface may also choose to support only a subset of action commands in the Sieve language (and its extensions) and so a mechanism to indicate the extent of support and characterize the relationships between those supported action commands and test (with its arguments) is immensely useful and probably required for clients that may not have complete knowledge of sieve grammar and semantics.

The Sieve language contains no mechanisms for indicating how a given

script should be presented in a user interface. The language also does not contain any specific mechanisms to represent other sorts of meta-data about the script. Providing support for such meta-data as part of sieve script is currently totally implementation specific and is usually done by imposing some type of structure on comments.

However, such information can be represented in XML very easily so it makes sense to define a framework to do this as part of the XML format. Implementations may choose to use structured comments to retain this information when the script is converted to normal Sieve format.

This XML representation defines two display directives - displayblock and displaydata - as containers for meta-data needed by graphical user interfaces.

The displayblock element can be used to enclose any number of sieve statements at any level. It is semantically meaningless to the sieve script itself. It allows an arbitrary set of attributes. Implementations MAY use this to provide many simple, display related meta-data for the sieve such as sieve identifier, group identifier, order of processing, etc. This information SHOULD be preserved in structured comments during conversion of XML to the normal Sieve syntax.

The displaydata element supports an any number of arbitrary child elements. Implementations MAY use this represent complex data about that sieve such as a natural language representation of sieve or a way to provide the sieve script directly. Again, this information SHOULD be presered in structured comments when converted.

5. Extended Example

The example sieve script given in section 9 of [I-D.ietf-sieve-3028bis] would be represented in XML as follows:

```
<!--
  Example Sieve Filter
  Declare any optional features or extensions used by the script
-->
<sieve>

  <control name="require">
    <str>fileinto</str>
  </control>
```

```
<!--
  Handle messages from known mailing lists
  Move messages from IETF filter discussion list to filter mailbox
-->
<control name="if">
  <test name="header">
    <tag>is</tag>
    <str>Sender</str>
    <str>owner-ietf-mta-filters@imc.org</str>
  </test>
  <action name="fileinto">
    <str>filter</str>
  </action> <!-- move to "filter" mailbox -->
</control>

<!--
  Keep all messages to or from people in my company
-->
<control name="elsif">
  <test name="address">
    <tag>domain</tag>
    <tag>is</tag>
    <list>
      <str>From</str>
      <str>To</str>
    </list>
    <str>example.com</str>
  </test>
  <action name="keep"/>
</control>

<!--
  Try and catch unsolicited email.  If a message is not to me,
  or it contains a subject known to be spam, file it away.
-->
<control name="elsif">
  <test name="anyof">
    <test name="not">
      <test name="address">
        <tag>all</tag>
        <tag>contains</tag>
        <list>
          <str>To</str>
          <str>Cc</str>
          <str>Bcc</str>
        </list>
```

```
            <str>me@example.com</str>
          </test>
```



```
        </test>
        <test name="header">
          <tag>matches</tag>
          <str>subject</str>
          <list>
            <str>*make*money*fast*</str>
            <str>*university*dipl*mas*</str>
          </list>
        </test>
      </test>
      <action name="fileinto">
        <str>spam</str>
      </action>
    </control>
    <control name="else">
      <!-- Move all other (non-company) mail to "personal"
           mailbox. -->
      <action name="fileinto">
        <str>personal</str>
      </action>
    </control>

  </sieve>
```

The same script could be annotated with grahical display hints in variety of ways. Two possibilities are:

```
  <sieve>

    <control name="require">
      <str>fileinto</str>
    </control>

    <displayblock name="File filter list mail" order="1"
                  group="FILE_TO_FOLDER" enable="true">
      <control name="if">
        <test name="header">
          <tag>is</tag>
          <str>Sender</str>
```

```
          <str>owner-ietf-mta-filters@imc.org</str>
        </test>
        <action name="fileinto">
          <str>filter</str>
        </action>
      </control>
    </displayblock>

    <displayblock name="Keep all company mail" order="2"
```



```
                  group="KEEP_MESSAGE" enable="true">
      <control name="elsif">
        <test name="address">
          <tag>domain</tag>
          <tag>is</tag>
          <list>
            <str>From</str>
            <str>To</str>
          </list>
          <str>example.com</str>
        </test>
        <action name="keep"/>
     </control>
    </displayblock>

    <displayblock name="File suspected spam" order="3"
                  group="FILE_TO_FOLDER" enable="true">
       <control name="elsif">
         <test name="anyof">
           <test name="not">
             <test name="address">
               <tag>all</tag>
               <tag>contains</tag>
               <list>
                 <str>To</str>
                 <str>Cc</str>
                 <str>Bcc</str>
               </list>
               <str>me@example.com</str>
             </test>
           </test>
           <test name="header">
```

```
            <tag>matches</tag>
            <str>subject</str>
            <list>
              <str>*make*money*fast*</str>
              <str>*university*dipl*mas*</str>
            </list>
          </test>
        </test>
        <action name="fileinto">
          <str>spam</str>
        </action>
      </control>
    </displayblock>

    <displayblock name="File noncompany mail as personal" order="4"
                  group="FILE_TO_FOLDER" enable="true">
```



```
      <control name="else">
        <action name="fileinto">
          <str>personal</str>
        </action>
      </control>
    </displayblock>

  </sieve>
```

Note that since displayblock elements are semantically null as far as the script itself is concerned they can be used to group structures like elsif and else that are tied to statements in other groups.

```
 <sieve>

   <displaydata>
     <nls-interpretation>
       If the e-mail header "Sender" is owner-ietf-mta-filters@imc.org
       then file it into the "filter" folder.

       Otherwise if the address in the "From" or "To" has a domain that
       is "example.com" then keep it.

       Otherwise  messages meeting with any of these conditions:
```

```
        (1) None of the addresses in "To" or "Cc"  or "Bcc" contains the
            domain "example.com".

        (2) The "Subject" field matches the pattern *make*money*fast* or
            *university*dipl*mas* then file  it into the "spam" folder.

        If all else fails then file the message in the "personal" folder.
      </nls-interpretation>
     </displaydata>

     ... the actual sieve script ...

  </sieve>
```

6. Security Considerations

Any syntactically valid sieve script can be represented in XML. Accordingly, all security considerations applicable to Sieve and any extensions used also apply to the XML representation.

The use of XML carries its own security risks. Section 7 of RFC 3470 [RFC3470] discussses these risks

Arbitrary data can be placed in the extensible displayblock and displaydata constructs defined in this specification, possibly including entire scripts in languages other than Sieve. Appropriate security precautions should be taken when using these facilities.

7. References

7.1. Normative References

[I-D.ietf-sieve-3028bis]
Guenther, P. and T. Showalter, "Sieve: An Email Filtering Language", draft-ietf-sieve-3028bis-12 (work in progress), February 2007, <http://www.ietf.org/internet-drafts/draft-ietf-sieve-3028bis-12.txt>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3470] Hollenbeck, S., Rose, M., and L. Masinter, "Guidelines for the Use of Extensible Markup Language (XML) within IETF Protocols", BCP 70, RFC 3470, January 2003.

7.2. Informative References

[RFC3431] Segmuller, W., "Sieve Extension: Relational Tests", RFC 3431, December 2002.

[RFC3598] Murchison, K., "Sieve Email Filtering -- Subaddress Extension", RFC 3598, September 2003.

[RFC3685] Daboo, C., "SIEVE Email Filtering: Spamtest and VirusTest Extensions", RFC 3685, February 2004.

[RFC3934] Wasserman, M., "Updates to RFC 2418 Regarding the Management of IETF Mailing Lists", BCP 94, RFC 3934, October 2004.

Appendix A. Schema for Sieves in XML

The following defines a schema for the XML representation of Sieve scripts. Note that aside from defining the displaydata and displayblock elements this schema imposes no constraints on their content.

```
<?xml version="1.0" encoding="UTF-8"?>
```



```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <xsd:element name="sieve">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
          <xsd:element ref="control"/>
          <xsd:element ref="action"/>
          <xsd:element ref="displayblock"/>
          <xsd:element ref="displaydata"/>
```

```
      </xsd:choice>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="command">
  <xsd:sequence>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="str"/>
      <xsd:element ref="num"/>
      <xsd:element ref="list"/>
      <xsd:element ref="tag"/>
    </xsd:choice>
    <xsd:element ref="test" minOccurs="0" maxOccurs="1"/>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="control"/>
      <xsd:element ref="action"/>
      <xsd:element ref="displayblock"/>
      <xsd:element ref="displaydata"/>
    </xsd:choice>
  </xsd:sequence>
  <xsd:attribute use="required" name="name" type="identifier"/>
</xsd:complexType>

<xsd:element name="control" type="command"/>

<xsd:element name="action" type="command"/>

<xsd:element name="test">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="str"/>
        <xsd:element ref="num"/>
        <xsd:element ref="list"/>
        <xsd:element ref="tag"/>
      </xsd:choice>
```



```
      <xsd:element ref="test" minOccurs="0"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute use="required" name="name" type="identifier"/>
```

```
  </xsd:complexType>
</xsd:element>

<xsd:element name="list">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="str" minOccurs="1"
                   maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="tag" type="identifier"/>

<xsd:element name="str" type="xsd:string"/>

<xsd:element name="num" type="xsd:nonNegativeInteger"/>

<xsd:simpleType name="identifier">
  <xsd:restriction base="xsd:token">
    <xsd:pattern value="[A-Za-z_][A-Za-z0-9_]*"/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name="displayblock">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="control"/>
        <xsd:element ref="action"/>
        <xsd:element ref="displayblock"/>
        <xsd:element ref="displaydata"/>
      </xsd:choice>
    </xsd:sequence>
    <xsd:anyAttribute processContents="skip"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="displaydata">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:any processContents="skip"/>
    </xsd:sequence>
  </xsd:complexType>
```

```
    </xsd:element>

  </xsd:schema>
```

Appendix B. Stylesheet for conversion from XML

```
  <?xml version="1.0" encoding="UTF-8"?>

  <!-- Convert Sieve in XML to standard Sieve syntax -->

  <xsl:stylesheet version="1.0"
                  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="text" encoding="UTF-8"
                media-type="application/sieve"/>

    <!-- Only preserve whitespace in str elements -->
    <xsl:strip-space elements="*"/>
    <xsl:preserve-space elements="str"/>

    <!-- Match top level sieve node,
         start processing in sieve mode -->

    <xsl:template match="sieve">
      <xsl:apply-templates
         select="control|action|comment()|displayblock|displaydata"
         mode="sieve">
        <xsl:with-param name="prefix" select="''"/>
      </xsl:apply-templates>
    </xsl:template>

    <!-- Routine to properly literalize quotes in Sieve strings -->

    <xsl:template name="quote-string">
      <xsl:param name="str"/>
      <xsl:choose>
        <xsl:when test="not($str)"/>
        <xsl:when test="contains($str, '&quot;')">
          <xsl:call-template name="quote-string">
            <xsl:with-param name="str"
               select="substring-before($str, '&quot;')"/>
          </xsl:call-template>
          <xsl:text>\&quot;</xsl:text>
          <xsl:call-template name="quote-string">
            <xsl:with-param name="str"
               select="substring-after($str, '&quot;')"/>
```

```
          </xsl:call-template>
```



```
        </xsl:when>
        <xsl:when test="contains($str, '\')">
          <xsl:call-template name="quote-string">
            <xsl:with-param name="str"
               select="substring-before($str, '\')"/>
          </xsl:call-template>
          <xsl:text>\\</xsl:text>
          <xsl:call-template name="quote-string">
            <xsl:with-param name="str"
               select="substring-after($str, '\')"/>
          </xsl:call-template>
        </xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="$str"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

    <!-- Sieve mode processing templates -->

    <xsl:template match="control|action" mode="sieve">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
</xsl:text>
      <xsl:value-of select="$prefix"/>
      <xsl:value-of select="@name"/>
      <xsl:variable name="blockbegin"
              select="generate-id(control|action)"/>
      <xsl:for-each select="*|comment()">
        <xsl:choose>
          <xsl:when test="self::str|self::num|
                          self::list|self::tag|self::test">
            <xsl:apply-templates select="." mode="sieve"/>
          </xsl:when>
          <xsl:when test="generate-id(.) = $blockbegin">
            <xsl:text xml:space="preserve">
</xsl:text>
            <xsl:value-of select="$prefix"/>
            <xsl:text>{</xsl:text>
            <xsl:apply-templates select="." mode="sieve">
```

```
              <xsl:with-param name="prefix"
                 select="concat($prefix, '  ')"/>
            </xsl:apply-templates>
          </xsl:when>
          <xsl:otherwise>
            <xsl:apply-templates select="." mode="sieve">
              <xsl:with-param name="prefix"
                 select="concat($prefix, '  ')"/>
```



```
            </xsl:apply-templates>
          </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>
      <xsl:choose>
        <xsl:when test="count(control|action) &gt; 0">
          <xsl:text xml:space="preserve">
  </xsl:text>
          <xsl:value-of select="$prefix"/>
          <xsl:text>}</xsl:text>
        </xsl:when>
        <xsl:otherwise>
          <xsl:text>;</xsl:text>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:template>

    <xsl:template match="test" mode="sieve">
      <xsl:text xml:space="preserve"> </xsl:text>
      <xsl:value-of select="@name"/>
      <xsl:apply-templates select="str|num|list|tag|comment()"
                 mode="sieve"/>
      <xsl:if test="count(descendant::test) &gt; 0">
        <xsl:text> (</xsl:text>
        <xsl:for-each select="test">
          <xsl:apply-templates select="." mode="sieve"/>
          <xsl:if test="count(following-sibling::test) &gt; 0">
            <xsl:text>,</xsl:text>
          </xsl:if>
        </xsl:for-each>
        <xsl:text> )</xsl:text>
      </xsl:if>
    </xsl:template>
```

```
<xsl:template match="str" mode="sieve">
  <xsl:text> &amp;quot;</xsl:text>
  <xsl:call-template name="quote-string">
    <xsl:with-param name="str" select="text()"/>
  </xsl:call-template>
  <xsl:text>&amp;quot;</xsl:text>
</xsl:template>

<xsl:template match="num" mode="sieve">
  <xsl:text xml:space="preserve"> </xsl:text>
  <!-- Use numeric suffixes when possible -->
  <xsl:choose>
    <xsl:when test="(number(text()) mod 1073741824) = 0">
      <xsl:value-of select="number(text()) div 1073741824"/>
```



```
      <xsl:text>G</xsl:text>
    </xsl:when>
    <xsl:when test="(number(text()) mod 1048576) = 0">
      <xsl:value-of select="number(text()) div 1048576"/>
      <xsl:text>M</xsl:text>
    </xsl:when>
    <xsl:when test="(number(text()) mod 1024) = 0">
      <xsl:value-of select="number(text()) div 1024"/>
      <xsl:text>K</xsl:text>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="text()"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="list" mode="sieve">
  <xsl:text> [</xsl:text>
  <xsl:for-each select="str">
    <xsl:apply-templates select="." mode="sieve"/>
    <xsl:if test="count(following-sibling::str) &amp;gt; 0">
      <xsl:text>,</xsl:text>
    </xsl:if>
  </xsl:for-each>
  <xsl:text> ]</xsl:text>
</xsl:template>
```

```
    <xsl:template match="tag" mode="sieve">
      <xsl:text> :</xsl:text>
      <xsl:value-of select="text()"/>
    </xsl:template>

    <xsl:template match="comment()" mode="sieve">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
  </xsl:text>
      <xsl:value-of select="$prefix"/>
      <xsl:text>/*</xsl:text>
      <xsl:value-of select="."/>
      <xsl:value-of select="$prefix"/>
      <xsl:text>*/</xsl:text>
    </xsl:template>

    <!-- Convert display information into structured comments -->
    <xsl:template match="displayblock" mode="sieve">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
  </xsl:text>
```



```
      <xsl:value-of select="$prefix"/>
      <xsl:text>/* [*</xsl:text>
      <xsl:apply-templates select="@*" mode="copy"/>
      <xsl:text> */</xsl:text>
      <xsl:apply-templates
         select="control|action|comment()|displayblock|displaydata"
         mode="sieve">
        <xsl:with-param name="prefix" select="$prefix"/>
      </xsl:apply-templates>
      <xsl:text xml:space="preserve">
  </xsl:text>
      <xsl:value-of select="$prefix"/>
      <xsl:text>/* *] */</xsl:text>
    </xsl:template>

    <xsl:template match="displaydata" mode="sieve">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
  </xsl:text>
```

```
      <xsl:value-of select="$prefix"/>
      <xsl:text>/* [|</xsl:text>
      <xsl:apply-templates mode="copy">
        <xsl:with-param name="prefix"
                        select="concat($prefix, '  ')"/>
      </xsl:apply-templates>
      <xsl:text xml:space="preserve">
  </xsl:text>
      <xsl:value-of select="$prefix"/>
      <xsl:text>   |] */</xsl:text>
    </xsl:template>

    <!-- Copy mode processing templates -->

    <xsl:template match="*[not(node())]" mode="copy">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
  </xsl:text>
      <xsl:value-of select="$prefix"/>
      <xsl:text>&lt;</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:apply-templates select="@*" mode="copy"/>
      <xsl:text>/&gt;</xsl:text>
    </xsl:template>

    <xsl:template match="*[node()]" mode="copy">
      <xsl:param name="prefix"/>
      <xsl:text xml:space="preserve">
  </xsl:text>
```



```
      <xsl:value-of select="$prefix"/>
      <xsl:text>&lt;</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:apply-templates select="@*" mode="copy"/>
      <xsl:text>&gt;</xsl:text>
      <xsl:apply-templates mode="copy">
        <xsl:with-param name="prefix"
                        select="concat($prefix, '  ')"/>
      </xsl:apply-templates>
      <xsl:if test="*[last()][not(text())]">
        <xsl:text xml:space="preserve">
  </xsl:text>
```

```
        <xsl:value-of select="$prefix"/>
      </xsl:if>
      <xsl:text>&lt;/</xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:text>&gt;</xsl:text>
    </xsl:template>

    <xsl:template match="@*" mode="copy">
      <xsl:text> </xsl:text>
      <xsl:value-of select="name()"/>
      <xsl:text>="</xsl:text>
      <xsl:value-of select="."/>
      <xsl:text>"</xsl:text>
    </xsl:template>

  </xsl:stylesheet>
```

Appendix C. Acknowledgements

The stylesheet copy mode code is loosely based on a sample code posted to the xsl-list list by Americo Albuquerque.

Authors' Addresses

Ned Freed
Sun Microsystems
3401 Centrelake Drive, Suite 410
Ontario, CA 92761-1205
USA

Phone: +1 909 457 4293
Email: ned.freed@mrochek.com

Srinivas Saisatish Vedam
Sun Microsystems

Phone: +91 80669 27577
Email: Srinivas.Sv@Sun.COM

Full Copyright Statement

Intellectual Property

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).