Digital Signature Service Core Protocols, Elements, and Bindings
Version 2.0

Working Draft 01

05 January 2018

Technical Committee:
   OASIS Digital Signature Services eXtended (DSS-X) TC

Chairs:
   Juan Carlos Cruellas (cruellas@ac.upc.edu), Univ Politecnica de Cataluna
   Stefan Hagen (stefan@hagen.link), Individual

Editor:
   Stefan Hagen (stefan@hagen.link), Individual

Additional artifacts:
   This prose specificatin is one component of a Work Product that also includes:
   • JSON and XML schemas: http://docs.oasis-open.org/dss-x/dss-core/v2.0/csd01/schemas/

Related work:
   This specification replaces or supersedes:
   • Stefan Drees et al., Digital Signature Service Core Protocols, Elements, and Bindings,
     Version 1.0, OASIS Standard, 11 April 2007,
     http://docs.oasis-open.org/dss/v1.0/oasis-dss-core-spec-v1.0-os.pdf
   This specification is related to:
   • Related specifications (hyperlink, if available)

Declared XML namespaces:
   • http://docs.oasis-open.org/dss-x/ns/dss-core/v2.0/dss

Abstract:
   This document defines JSON and XML based request/response protocols for signing and
   verifying documents and other data. It also defines a timestamp format, and a signature property
   for use with these protocols. Finally, it defines transport and security bindings for the protocols.

Status:
   This Working Draft (WD) has been produced by one or more TC Members; it has not yet been
   voted on by the TC or approved as a Committee Draft (Committee Specification Draft or a
   Committee Note Draft). The OASIS document Approval Process begins officially with a TC vote
   to approve a WD as a Committee Draft. A TC may approve a Working Draft, revise it, and re-
   approve it any number of times as a Committee Draft.
   Any machine-readable content (Computer Language Definitions) declared Normative for this
   Work Product must also be provided in separate plain text files. In the event of a discrepancy
   between such plain text file and display content in the Work Product's prose narrative
   document(s), the content in the separate plain text file prevails.

URI patterns:
   Initial publication URI:
   http://docs.oasis-open.org/dss-x/dss-core/v2.0/csd01/dss-core-v2.0-csd01.docx
   Permanent "Latest version" URI:
   http://docs.oasis-open.org/dss-x/dss-core/v2.0/dss-core-v2.0.docx
   (Managed by OASIS TC Administration; please don’t modify.)

Copyright © OASIS Open 2017. All Rights Reserved.
All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the “OASIS IPR Policy”). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
# Table of Contents

1  Introduction ........................................................................................................................................... 10
   1.1 Organization of DSS Core Protocols, Elements, and Bindings ......................................................... 10
   1.2 Terminology ....................................................................................................................................... 10
       1.2.1 Terms and Definitions .................................................................................................................. 10
       1.2.2 Abbreviated Terms ...................................................................................................................... 10
   1.3 Normative References ....................................................................................................................... 10
   1.4 Non-Normative References ................................................................................................................ 12
   1.5 Typographical Conventions ............................................................................................................. 12

2  Design Considerations ............................................................................................................................. 13
   2.1 Construction Principles ..................................................................................................................... 13
   2.2 Domain Models ................................................................................................................................... 13
       2.2.1 Date and Time Model ..................................................................................................................... 13
   2.3 Schema Organization and Namespaces ............................................................................................. 13
   2.4 DSS Overview (Non-normative) ......................................................................................................... 14
   2.5 DSS-X Component Overview ......................................................................................................... 15
       2.5.1 Schema extensions ......................................................................................................................... 15
   2.6 Version 2.0 goal [non-normative] ...................................................................................................... 16
       2.6.1 Circumventing xs: any .................................................................................................................... 16
       2.6.2 Substituting the 'mixed' schema attribute .................................................................................... 17
       2.6.3 Introducing the NsPrefixMappingType component .................................................................... 17
       2.6.4 Imported XML schemes .............................................................................................................. 17
       2.6.5 Syntax variants ............................................................................................................................ 18

3  Structure Models ....................................................................................................................................... 19
   3.1 Structure Models defined in this document ....................................................................................... 19
      3.1.1 Component NsPrefixMapping .................................................................................................... 19
          3.1.1.1 Semantics ................................................................................................................................. 19
          3.1.1.2 XML Syntax ................................................................................................................................ 19
          3.1.1.3 JSON Syntax ............................................................................................................................ 19
      3.1.2 Component Any ............................................................................................................................ 20
          3.1.2.1 Semantics ................................................................................................................................. 20
          3.1.2.2 XML Syntax ................................................................................................................................ 20
          3.1.2.3 JSON Syntax ............................................................................................................................ 21
      3.1.3 Component InternationalString ................................................................................................ 21
          3.1.3.1 Semantics ................................................................................................................................. 21
          3.1.3.2 XML Syntax ................................................................................................................................ 22
          3.1.3.3 JSON Syntax ............................................................................................................................ 22
      3.1.4 Component DigestInfo ................................................................................................................ 23
          3.1.4.1 Semantics ................................................................................................................................. 23
          3.1.4.2 XML Syntax ................................................................................................................................ 23
          3.1.4.3 JSON Syntax ............................................................................................................................ 23
      3.1.5 Component AttachmentReference ............................................................................................. 24
          3.1.5.1 Semantics ................................................................................................................................. 24
          3.1.5.2 XML Syntax ................................................................................................................................ 24
          3.1.5.3 JSON Syntax ............................................................................................................................ 25
3.1.6 Component Base64Data ................................................................. 25
  3.1.6.1 Semantics ............................................................................ 25
  3.1.6.2 XML Syntax ......................................................................... 26
  3.1.6.3 JSON Syntax ......................................................................... 27
3.1.7 Component Result ................................................................. 28
  3.1.7.1 Semantics ............................................................................ 28
  3.1.7.2 XML Syntax ......................................................................... 28
  3.1.7.3 JSON Syntax ......................................................................... 29
3.1.8 Component RequestBase ..................................................... 29
  3.1.8.1 Semantics ............................................................................ 29
  3.1.8.2 XML Syntax ......................................................................... 30
  3.1.8.3 JSON Syntax ......................................................................... 30
3.1.9 Component ResponseBase .................................................. 30
  3.1.9.1 Semantics ............................................................................ 30
  3.1.9.2 XML Syntax ......................................................................... 31
  3.1.9.3 JSON Syntax ......................................................................... 31
3.1.10 Component Info ................................................................. 31
  3.1.10.1 Semantics ........................................................................... 31
  3.1.10.2 XML Syntax ......................................................................... 31
  3.1.10.3 JSON Syntax ......................................................................... 32
3.1.11 Component Description ..................................................... 32
  3.1.11.1 Semantics ........................................................................... 32
  3.1.11.2 XML Syntax ......................................................................... 32
  3.1.11.3 JSON Syntax ......................................................................... 32
3.1.12 Component InputDocuments ............................................. 33
  3.1.12.1 Semantics ........................................................................... 33
  3.1.12.2 XML Syntax ......................................................................... 33
  3.1.12.3 JSON Syntax ......................................................................... 33
3.1.13 Component DocumentBase ................................................ 34
  3.1.13.1 Semantics ........................................................................... 34
  3.1.13.2 XML Syntax ......................................................................... 35
  3.1.13.3 JSON Syntax ......................................................................... 35
3.1.14 Component Document ....................................................... 36
  3.1.14.1 Semantics ........................................................................... 36
  3.1.14.2 XML Syntax ......................................................................... 36
  3.1.14.3 JSON Syntax ......................................................................... 37
3.1.15 Component TransformedData ............................................ 38
  3.1.15.1 Semantics ........................................................................... 38
  3.1.15.2 XML Syntax ......................................................................... 38
  3.1.15.3 JSON Syntax ......................................................................... 40
3.1.16 Component DocumentHash ............................................... 41
  3.1.16.1 Semantics ........................................................................... 41
  3.1.16.2 XML Syntax ......................................................................... 42
  3.1.16.3 JSON Syntax ......................................................................... 42
3.1.17 Component SignRequest .................................................... 44
  3.1.17.1 Semantics ........................................................................... 44
  3.1.17.2 XML Syntax ......................................................................... 44
  3.1.17.3 JSON Syntax ......................................................................... 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.18</td>
<td>Component SignResponse</td>
<td>45</td>
</tr>
<tr>
<td>3.1.18.1</td>
<td>Semantics</td>
<td>45</td>
</tr>
<tr>
<td>3.1.18.2</td>
<td>XML Syntax</td>
<td>45</td>
</tr>
<tr>
<td>3.1.18.3</td>
<td>JSON Syntax</td>
<td>46</td>
</tr>
<tr>
<td>3.1.19</td>
<td>Component SignatureObject</td>
<td>47</td>
</tr>
<tr>
<td>3.1.19.1</td>
<td>Semantics</td>
<td>47</td>
</tr>
<tr>
<td>3.1.19.2</td>
<td>XML Syntax</td>
<td>47</td>
</tr>
<tr>
<td>3.1.19.3</td>
<td>JSON Syntax</td>
<td>48</td>
</tr>
<tr>
<td>3.1.20</td>
<td>Component SignaturePtr</td>
<td>48</td>
</tr>
<tr>
<td>3.1.20.1</td>
<td>Semantics</td>
<td>48</td>
</tr>
<tr>
<td>3.1.20.2</td>
<td>XML Syntax</td>
<td>49</td>
</tr>
<tr>
<td>3.1.20.3</td>
<td>JSON Syntax</td>
<td>49</td>
</tr>
<tr>
<td>3.1.21</td>
<td>Component VerifyRequest</td>
<td>50</td>
</tr>
<tr>
<td>3.1.21.1</td>
<td>Semantics</td>
<td>50</td>
</tr>
<tr>
<td>3.1.21.2</td>
<td>XML Syntax</td>
<td>50</td>
</tr>
<tr>
<td>3.1.21.3</td>
<td>JSON Syntax</td>
<td>51</td>
</tr>
<tr>
<td>3.1.22</td>
<td>Component VerifyResponse</td>
<td>52</td>
</tr>
<tr>
<td>3.1.22.1</td>
<td>Semantics</td>
<td>52</td>
</tr>
<tr>
<td>3.1.22.2</td>
<td>XML Syntax</td>
<td>52</td>
</tr>
<tr>
<td>3.1.22.3</td>
<td>JSON Syntax</td>
<td>52</td>
</tr>
<tr>
<td>3.1.23</td>
<td>Component OptionalInputsBase</td>
<td>53</td>
</tr>
<tr>
<td>3.1.23.1</td>
<td>Semantics</td>
<td>53</td>
</tr>
<tr>
<td>3.1.23.2</td>
<td>XML Syntax</td>
<td>54</td>
</tr>
<tr>
<td>3.1.23.3</td>
<td>JSON Syntax</td>
<td>55</td>
</tr>
<tr>
<td>3.1.24</td>
<td>Component OptionalInputsSign</td>
<td>55</td>
</tr>
<tr>
<td>3.1.24.1</td>
<td>Semantics</td>
<td>55</td>
</tr>
<tr>
<td>3.1.24.2</td>
<td>XML Syntax</td>
<td>56</td>
</tr>
<tr>
<td>3.1.24.3</td>
<td>JSON Syntax</td>
<td>57</td>
</tr>
<tr>
<td>3.1.25</td>
<td>Component OptionalInputsVerify</td>
<td>59</td>
</tr>
<tr>
<td>3.1.25.1</td>
<td>Semantics</td>
<td>59</td>
</tr>
<tr>
<td>3.1.25.2</td>
<td>XML Syntax</td>
<td>61</td>
</tr>
<tr>
<td>3.1.25.3</td>
<td>JSON Syntax</td>
<td>62</td>
</tr>
<tr>
<td>3.1.26</td>
<td>Component OptionalOutputsBase</td>
<td>64</td>
</tr>
<tr>
<td>3.1.26.1</td>
<td>Semantics</td>
<td>64</td>
</tr>
<tr>
<td>3.1.26.2</td>
<td>XML Syntax</td>
<td>65</td>
</tr>
<tr>
<td>3.1.26.3</td>
<td>JSON Syntax</td>
<td>66</td>
</tr>
<tr>
<td>3.1.27</td>
<td>Component OptionalOutputsSign</td>
<td>66</td>
</tr>
<tr>
<td>3.1.27.1</td>
<td>Semantics</td>
<td>66</td>
</tr>
<tr>
<td>3.1.27.2</td>
<td>XML Syntax</td>
<td>66</td>
</tr>
<tr>
<td>3.1.27.3</td>
<td>JSON Syntax</td>
<td>66</td>
</tr>
<tr>
<td>3.1.28</td>
<td>Component OptionalOutputsVerify</td>
<td>67</td>
</tr>
<tr>
<td>3.1.28.1</td>
<td>Semantics</td>
<td>67</td>
</tr>
<tr>
<td>3.1.28.2</td>
<td>XML Syntax</td>
<td>68</td>
</tr>
<tr>
<td>3.1.28.3</td>
<td>JSON Syntax</td>
<td>69</td>
</tr>
<tr>
<td>3.1.29</td>
<td>Component ClaimedIdentity</td>
<td>70</td>
</tr>
<tr>
<td>3.1.29.1</td>
<td>Semantics</td>
<td>70</td>
</tr>
<tr>
<td>3.1.29.2</td>
<td>XML Syntax</td>
<td>71</td>
</tr>
<tr>
<td>3.1.29.3</td>
<td>JSON Syntax</td>
<td>71</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>3.1.42</td>
<td>Component SignedReference</td>
<td>88</td>
</tr>
<tr>
<td>3.1.42.1</td>
<td>Semantics</td>
<td>88</td>
</tr>
<tr>
<td>3.1.42.2</td>
<td>XML Syntax</td>
<td>88</td>
</tr>
<tr>
<td>3.1.42.3</td>
<td>JSON Syntax</td>
<td>88</td>
</tr>
<tr>
<td>3.1.43</td>
<td>Component VerifyManifestResults</td>
<td>89</td>
</tr>
<tr>
<td>3.1.43.1</td>
<td>Semantics</td>
<td>89</td>
</tr>
<tr>
<td>3.1.43.2</td>
<td>XML Syntax</td>
<td>89</td>
</tr>
<tr>
<td>3.1.43.3</td>
<td>JSON Syntax</td>
<td>90</td>
</tr>
<tr>
<td>3.1.44</td>
<td>Component ManifestResult</td>
<td>90</td>
</tr>
<tr>
<td>3.1.44.1</td>
<td>Semantics</td>
<td>90</td>
</tr>
<tr>
<td>3.1.44.2</td>
<td>XML Syntax</td>
<td>91</td>
</tr>
<tr>
<td>3.1.44.3</td>
<td>JSON Syntax</td>
<td>91</td>
</tr>
<tr>
<td>3.1.45</td>
<td>Component UseVerificationTime</td>
<td>92</td>
</tr>
<tr>
<td>3.1.45.1</td>
<td>Semantics</td>
<td>92</td>
</tr>
<tr>
<td>3.1.45.2</td>
<td>XML Syntax</td>
<td>92</td>
</tr>
<tr>
<td>3.1.45.3</td>
<td>JSON Syntax</td>
<td>93</td>
</tr>
<tr>
<td>3.1.46</td>
<td>Component AdditionalTimeInfo</td>
<td>94</td>
</tr>
<tr>
<td>3.1.46.1</td>
<td>Semantics</td>
<td>94</td>
</tr>
<tr>
<td>3.1.46.2</td>
<td>XML Syntax</td>
<td>94</td>
</tr>
<tr>
<td>3.1.46.3</td>
<td>JSON Syntax</td>
<td>95</td>
</tr>
<tr>
<td>3.1.47</td>
<td>Component VerificationTimeInfo</td>
<td>95</td>
</tr>
<tr>
<td>3.1.47.1</td>
<td>Semantics</td>
<td>95</td>
</tr>
<tr>
<td>3.1.47.2</td>
<td>XML Syntax</td>
<td>96</td>
</tr>
<tr>
<td>3.1.47.3</td>
<td>JSON Syntax</td>
<td>96</td>
</tr>
<tr>
<td>3.1.48</td>
<td>Component AdditionalKeyInfo</td>
<td>97</td>
</tr>
<tr>
<td>3.1.48.1</td>
<td>Semantics</td>
<td>97</td>
</tr>
<tr>
<td>3.1.48.2</td>
<td>XML Syntax</td>
<td>97</td>
</tr>
<tr>
<td>3.1.48.3</td>
<td>JSON Syntax</td>
<td>97</td>
</tr>
<tr>
<td>3.1.49</td>
<td>Component ProcessingDetails</td>
<td>98</td>
</tr>
<tr>
<td>3.1.49.1</td>
<td>Semantics</td>
<td>98</td>
</tr>
<tr>
<td>3.1.49.2</td>
<td>XML Syntax</td>
<td>99</td>
</tr>
<tr>
<td>3.1.49.3</td>
<td>JSON Syntax</td>
<td>99</td>
</tr>
<tr>
<td>3.1.50</td>
<td>Component Detail</td>
<td>100</td>
</tr>
<tr>
<td>3.1.50.1</td>
<td>Semantics</td>
<td>100</td>
</tr>
<tr>
<td>3.1.50.2</td>
<td>XML Syntax</td>
<td>101</td>
</tr>
<tr>
<td>3.1.50.3</td>
<td>JSON Syntax</td>
<td>101</td>
</tr>
<tr>
<td>3.1.51</td>
<td>Component SigningTimeInfo</td>
<td>102</td>
</tr>
<tr>
<td>3.1.51.1</td>
<td>Semantics</td>
<td>102</td>
</tr>
<tr>
<td>3.1.51.2</td>
<td>XML Syntax</td>
<td>102</td>
</tr>
<tr>
<td>3.1.51.3</td>
<td>JSON Syntax</td>
<td>103</td>
</tr>
<tr>
<td>3.1.52</td>
<td>Component UpdatedSignature</td>
<td>104</td>
</tr>
<tr>
<td>3.1.52.1</td>
<td>Semantics</td>
<td>104</td>
</tr>
<tr>
<td>3.1.52.2</td>
<td>XML Syntax</td>
<td>104</td>
</tr>
<tr>
<td>3.1.52.3</td>
<td>JSON Syntax</td>
<td>104</td>
</tr>
<tr>
<td>3.1.53</td>
<td>Component ReturnTransformedDocument</td>
<td>105</td>
</tr>
<tr>
<td>3.1.53.1</td>
<td>Semantics</td>
<td>105</td>
</tr>
<tr>
<td>3.1.53.2</td>
<td>XML Syntax</td>
<td>105</td>
</tr>
<tr>
<td>3.1.53.3</td>
<td>JSON Syntax</td>
<td>106</td>
</tr>
</tbody>
</table>
3.1.54 Component TransformedDocument ................................................. 106
  3.1.54.1 Semantics ................................................................................. 106
  3.1.54.2 XML Syntax ............................................................................. 106
  3.1.54.3 JSON Syntax ............................................................................ 107
3.2 Referenced Structure Models from other documents ............................... 107
  3.2.1 Component Transforms .................................................................. 107
    3.2.1.1 Semantics ................................................................................ 107
    3.2.1.2 XML Syntax ............................................................................ 108
    3.2.1.3 JSON Syntax .......................................................................... 108
  3.2.2 Component Transform ..................................................................... 109
    3.2.2.1 Semantics ................................................................................ 109
    3.2.2.2 XML Syntax ............................................................................ 109
    3.2.2.3 JSON Syntax .......................................................................... 109
  3.2.3 Component NameID ........................................................................ 110
    3.2.3.1 Semantics ................................................................................ 110
    3.2.3.2 XML Syntax ............................................................................ 111
    3.2.3.3 JSON Syntax .......................................................................... 111
4 The Processing Model For Signing .............................................................. 113
  4.1 Processing for XML Signatures ............................................................. 113
    4.1.1 Sub process 'process references' .................................................. 113
    4.1.2 Sub process 'create XML signature' .............................................. 114
      4.1.2.1 XML Signatures Variant Optional Input IncludeObject .......... 115
  4.2 Processing for CMS Signatures ............................................................. 116
    4.2.1 Sub process 'process digest' ......................................................... 116
    4.2.2 Sub process 'create CMS signature' .............................................. 116
  4.3 General processing ............................................................................. 117
    4.3.1 Sub process 'add Timestamp' ....................................................... 117
      4.3.1.1 Processing for CMS signatures time-stamping ...................... 118
      4.3.1.2 Processing for XML Timestamps on XML signatures .......... 118
      4.3.1.3 Processing for RFC 3161 Timestamps on XML signatures .... 119
5 The Processing Model For Verification ....................................................... 120
  5.1 Processing for XML Signatures ............................................................. 121
    5.1.1 Sub process 'retrieve XML signature' .......................................... 121
      5.1.1.1 Multi-Signature Verification .................................................. 121
    5.1.2 Sub process 'recalculate references' .......................................... 122
    5.1.3 Sub process 'verify XML signature' .......................................... 123
      5.1.3.1 Processing for RFC 3161 timestamp tokens on XML Signatures 123
      5.1.3.2 Processing for XML timestamp tokens on XML signatures .... 124
  5.2 Basic Processing for CMS Signatures ................................................... 125
    5.2.1 Sub process 'retrieve CMS signature' ......................................... 125
    5.2.2 Sub process 'verify CMS signature' .......................................... 125
      5.2.2.1 Processing for RFC 3161 Timestamp tokens on CMS Signatures 126
  5.3 General processing ............................................................................ 126
    5.3.1 Sub process 'update Signature' .................................................... 126
    5.3.2 Sub process 'timestamp Signature' ............................................. 127
    5.3.3 Task 'build VerifyResponse' ...................................................... 128
1 Introduction

1.1 Organization of DSS Core Protocols, Elements, and Bindings
The specification is split into twelve chapters.

1.2 Terminology
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 [RFC2119].

1.2.1 Terms and Definitions
For the purposes of this document, the following applies:

<table>
<thead>
<tr>
<th>Term</th>
<th>meaning and maybe ref</th>
</tr>
</thead>
</table>

1.2.2 Abbreviated Terms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>— Spelled out</th>
</tr>
</thead>
</table>

1.3 Normative References


(Remark: As used in DSS, all implementations based upon RFC 5652 and previous releases of CMS will suffice. For the sake of simplicity the
"urn:ietf:rfc:3369" is used throughout the document to indicate a CMS message as specified in RFC 5652 or RFC 3369 or any version (including PKCS #7).


1.4 Non-Normative References


1.5 Typographical Conventions

Keywords defined by this specification use this monospaced font. Normative source code uses this paragraph style.

Text following the special symbol (“) – an opening Guillemet (or French quotation mark) – within this specification identifies conformance statements. Every conformance statement is separated from the following text with the special end symbol (”) – a closing Guillemet, and has been assigned a reference that follows that end symbol in the format [DSS-section#-local#].

Some sections of this specification are illustrated with non-normative examples.

Example 1: text describing an example uses this paragraph style

Non-normative examples use this paragraph style.

All examples in this document are non-normative and informative only. Representation-specific text is indented and marked with vertical lines.

**Representation-Specific Headline**

Normative representation-specific text

All other text is normative unless otherwise labeled e.g. like:

**Non-normative Comment:**

This is a pure informative comment that may be present, because the information conveyed is deemed useful advice or common pitfalls learned from implementer or operator experience and often given including the rationale.
2 Design Considerations

Blurb

2.1 Construction Principles

2.2 Domain Models

2.2.1 Date and Time Model

The specific concept of date and time used in this document is defined in this section and noted in subsequent usage as:

DateTime

« All date time values inside a DSS document MUST adhere to the ISO 8601 [ISO8601] basic or extended Format (as given there in section 4.3.2 “Complete representations” and with the addition of decimal fractions for seconds, similar to ibid. section 4.2.2.4 “Representations with decimal fraction” but with the full stop (.) being the preferred separator for DSS). » [DSS-2.2.1-1].

2.3 Schema Organization and Namespaces

The structures described in this specification are contained in the schema file [Core2.0-XSD]. All schema listings in the current document are excerpts from the schema file. In the case of a disagreement between the schema file and this document, the schema file shall take precedence.

This schema is associated with the following XML namespace:

```
urn:oasis:names:tc:dss:2.0:core:schema
```

If a future version of this specification is needed, it will use a different namespace.

Conventional XML namespace prefixes are used in the schema:

- The prefix `dss:` stands for the DSS core namespace [DSS2XSD].
- The prefix `ds:` stands for the W3C XML Signature namespace [XMLDSIG].
- The prefix `xs:` stands for the W3C XML Schema namespace [Schema1].
- The prefix `saml:` stands for the OASIS SAML Schema namespace [SAMLCore1.1].

Applications MAY use different namespace prefixes, and MAY use whatever namespace defaulting/scoping conventions they desire, as long as they are compliant with the Namespaces in XML specification [XML-ns].

The following schema fragment defines the XML namespaces and other header information for the DSS core schema:

```xml
<xs:schema xmlns:dss="urn:oasis:names:tc:dss:2.0:core:schema"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:saml="urn:oasis:names:tc:SAML:1.0:assertion"
targetNamespace="urn:oasis:names:tc:dss:2.0:core:schema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:annotation>
<xs:documentation xml:lang="en">This Schema defines the Digital Signature Service Core Protocols, Elements, and Bindings Committee Draft 1 for Public Review</xs:documentation>
</xs:annotation>
```
2.4 DSS Overview (Non-normative)

This specification describes two request/response protocols:

1. a signing protocol and
2. a verifying protocol.

Using the first protocol a client can send documents (or document hashes) to a server and receive back a signature on the documents. Using the second protocol a client can send documents (or document hashes) and a signature to a server, and receive back an answer on whether the signature is valid or not.

The elements in which the protocols are formulated are provided in a semantic level and also in JSON and XML syntax. Provided are additional mappings from the generic to the specific entities.

These protocol operations could be useful in a variety of contexts – for example, they could allow clients to access a single corporate key for signing press releases, with centralized access control, auditing, and archiving of signature requests. They could also allow clients to create and verify signatures without needing complex client software and configuration.

The signing and verifying protocols are chiefly designed to support the creation and verification of XML signatures [XMLDSIG], XML timestamps (see [DSS1Core], section 5.1), binary timestamps [RFC 3161] and CMS signatures [RFC 3852]. These protocols may also be extensible to other types of signatures and timestamps, such as PGP signatures [RFC 2440].

It is expected that the signing and verifying protocols will be profiled to meet many different application scenarios. In anticipation of this, these protocols have only a minimal set of required elements, which deal with transferring “input documents” and signatures back and forth between client and server. The input documents to be signed or verified can be transferred in their entirety or the client can hash the documents themselves and only send the hash values to save bandwidth and protect the confidentiality of the document content.

All functionality besides transferring input documents and signatures is relegated to a framework of “optional inputs” and “optional outputs”. This document defines a number of optional inputs and outputs. Profiles of these protocols can pick and choose which optional inputs and outputs to support and can introduce their own optional inputs and outputs when they need functionality not anticipated by this specification.

Examples of optional inputs to the signing protocol include: what type of signature to produce, which key to sign with, who the signature is intended for, and what signed and unsigned properties to place in the signature. Examples of optional inputs to the verifying protocol include: the time for which the client would like to know the signature’s validity status, additional validation data necessary to verify the signature (such as certificates and CRLs), and requests for the server to return information such as the signer’s name or the signing time.

The signing and verifying protocol messages must be transferred over some underlying protocol(s) which provide message transport and security. A binding specifies how to use the signing and verifying protocols with some underlying protocol such as HTTP POST or TLS. Section 6 provides an initial set of bindings.

The previous version of specification ([DSS1Core]) defines two elements that are related to these protocols. First, an XML timestamp element is defined in [DSS1Core], section 5.1. The signing and verifying protocols can be used to create and verify both XML and binary timestamps; a profile for doing so is defined in [XML-TSP]. Second, a RequesterIdentity element is defined in (see [DSS1Core],
section 5.2). This element can be used as a signature property in an XML signature, to give the name of the end-user who requested the signature. These elements remain unchanged and are not repeated in this specification.

### 2.5 DSS-X Component Overview

The DSS-X core is designed to be extended by profiles to support additional functionalities. The DSS specification comes with a set of profiles (see [https://www.oasis-open.org/standards#dssv1.0](https://www.oasis-open.org/standards#dssv1.0)). With version 2.0 there will be ‘extensions’ to augment the use cases beyond the ‘sign & verify’ scope of the previous version. The ‘extensions’ will define other requests and responses while using e.g. the 'ResultType'. A sample for an extension is the ChipGateway Protocol (ref...). To support this approach, the DSS-X 2 schema is split into a generic ‘base’ and the more specific ‘core’ schema.

![Diagram showing the relationship between the different building blocks](image)

**Figure 1**

The diagram above shows the relationship between the different building blocks.

### 2.5.1 Schema extensions

Most profiles define additional OptionalInputs or OptionalOutputs. To support a type-safe extension of the set of optional elements it is recommended to use the XML schema ‘redefine’ mechanism to extend the core schema and derive the related JSON schema from it:

```xml
<xs:redefine schemaLocation="core-schema.xsd">
    <xs:complexType name="dss:OptionalOutputsVerifyType">
        <xs:complexContent>
            <xs:extension base="dss:OptionalOutputsVerifyType">
                <xs:group ref="prf:optionalOutputGroup"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:redefine>
```
The snippet above extends the set of sub-components of `OptionalOutputsVerifyType` with the group of elements of the profile.

### 2.6 Version 2.0 goal [non-normative]

The main changes of this version of the DSS/X core document compared to version 1.0 are:

- Consider the set of comments and bug reports arrived since version DSS 1.0 became standard
- Include requirements that became known only after publication of version 1.0
- Simplify the core schema, e.g. by dropping elements seldom used
- Support syntaxes other than XML
- Support other transport formats than SOAP.

Define a semantic model that can be mapped to different syntaxes. In this document the focus is on XML and JSON, but support for other syntaxes should be possible. Therefore, only the common denominator of syntax features can be used:

- Focus on Base64 as the most versatile way to transport documents and signatures
- Avoid the use of XML specifics (like e.g. mixed content)
- Provide namespace / URI for XPath evaluation explicitly
- Avoid `xs:any` by replacing it with an enumeration of possible types, and if that is not feasible, use base64 blobs as a fallback

To support implementors and to ease the use of the protocol with common frameworks the following list of requirements was compiled:

- One unique object model for all transport syntaxes
- Define type and cardinality of `OptionalInputs` and `OptionalOutputs` child elements explicitly
- Rearrange sequences and choices to produce a strongly typed object model

The provided schemes of DSS-X version 2 reflect these requirements. The XML schemes of version 1 and 2 share many similarities but are not compatible.

### 2.6.1 Circumventing `xs:any`

The XML schema type ‘any’ allows an object to contain arbitrary structures. This comes handy for writers of specifications as an extension point because the structures transported don’t need to be defined upfront. But this advantage at the specification stage comes with a price at the implementation stage. The structures intended to be supported by a client or a server system MUST be known to be implementable. But the usual tools for schema support leave the task of handling the content of an any type to the developer. Without extensive testing problems with unexpected content may occur at runtime, even while using typed languages.

A successor of the `OptionalInputs` element (see section 2.7 of version 1.0 of this document) the component `OptionalInputsVerify` (see section 3.1.25) defines its child elements and their cardinality explicitly. When using additional profiles, the relevant components of the core schema can be redefined using the XML schema’s ‘redefine’ element:

```xml
<xs:redefine>
  <xs:complexType name="OptionalInputsVerifyType">
    <xs:complexContent>
      <xs:extension base="dss:OptionalInputsVerifyType"/>
      <xs:group ref="vr:OptionalInputGroup"/>
    </xs:extension>
  </xs:complexType>
</xs:redefine>
```
With this mechanism it is possible to extend the core schema to specific requirements while preserving the advantage of type safety and tool / IDE support. This sample illustrates the use of 'extension', in the same way 'restriction' can be applied. In more complex scenarios (e.g. multiple profiles apply, need for extending and restriction the core schema) the use of other techniques (e.g. XSLT) may be required.

Another usage scenario for 'xs:any' is the transport of unknown data objects. As sample use case is the Property component (see section 3.1.37). This component is intended to contain signature attributes of unknown structure. In this version of the specification the 'xs:any' type is replaced by a structure containing base64-encoded data and meta data (component Any, see section 3.1.2). When using XML as the transport syntax this seems to be a disadvantage. But direct XML fragment copying may introduce namespace problems and security concerns. Most importantly the cherry-picking of transport syntax features would inhibit a transport independent object model, both on the client and the server side. More complex programming and testing would be inevitable.

### 2.6.2 Substituting the 'mixed' schema attribute

Mixing sub-elements and text within a single element is a great advantage of XML. But when XML is applied for serializing an object model this 'markup language' feature is of little use. Other serialization syntaxes (like JSON) don't support such a feature. So there is the need to substitute the 'mixed' construct to become syntax independent. The substitution is done by removing the mixed attribute and introduce an additional 'value' element to contain the textual content.

### 2.6.3 Introducing the NsPrefixMappingType component

Namespaces are an outstanding feature of the XML world. Therefore a replacement is required for all syntaxes that don't such a feature. The use of naming conventions and prefixes are used to avoid naming collisions. A special challenge is the use of XPath-Expression as elements. The XPath expression itself is represented as a simple string. But the expression may depend on namespace/prefix mappings that are defined within the namespace context of the XML element. The NsPrefixMappingType component (see section 3.1.1) represents the required namespace/prefix mapping. It is recommended to use this element for XML syntax, too. This simplifies the handling on the consumer side and circumvents problems with namespace prefix assignments handled by web frameworks.

### 2.6.4 Imported XML schemes

A special challenge is imposed by the imported schemes, like the [XMLDSIG] scheme, that uses features not supportable by the mentioned 'multi-syntx' approach. For example, the [XMLDSIG] type 'Transform' is defined like this:

```xml
<xs:complexType name="TransformType" mixed="true">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:any namespace="##other" processContents="lax"/>
    <!-- (1,1) elements from (0,unbounded) namespaces -->
    <xs:element name="XPath" type="string"/>
  </xs:choice>
  <xs:attribute name="Algorithm" type="xs:anyURI" use="required"/>
</xs:complexType>
```

Most of the restrictions listed above do apply here:

- The complexType may contain mixed content (child elements and text). This concept is not supported by JSON. So the workaround this limitation is to drop the 'mixed' attribute and to introduce a 'value' element.
- The 'choice' construct is mapped in an untyped way by Java's JAXB framework. Therefore, the container element is changed to a 'sequence'.
- The 'any' type is replaced by a base64 encoded blob.
The option to provide arbitrary namespace / prefix mappings to support the evaluation of XPath expression is not available in e.g. JSON syntax. Therefore an element mapping prefixes to namespaces (of type ‘dsb:NsPrefixMappingType’) is added.

```xml
<xs:complexType name="TransformType">
  <xs:sequence>
    <xs:element maxOccurs="1" minOccurs="0" name="value" type="string"/>
    <xs:element maxOccurs="1" minOccurs="0" name="Base64Content" type="xs:base64Binary"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="XPath" type="string"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="NsPrefixMapping" type="dsb:NsPrefixMappingType"/>
  </xs:sequence>
  <xs:attribute name="Algorithm" type="xs:string" use="required"/>
</xs:complexType>
```

To apply the necessary changes to the imported schemes the XML schema language provides the ‘override’ functionality to change existing schemes. But Java’s JAXB framework’s schema compiler does not support ‘override’ so the adapted schemes are provided alongside DSS-X core schemes.

### 2.6.5 Syntax variants

This version of the DSS/X core document handles the representation of requests and response elements according to the JSON and XML syntax. The general semantics of the elements is discussed in the element’s main section. Details of the JSON or XML formats are discussed in specific subsections:

- JSON syntax
- XML syntax
3 Structure Models

3.1 Structure Models defined in this document

3.1.1 Component NsPrefixMapping

3.1.1.1 Semantics

The NsPrefixMapping component allows the mapping of namespace URIs to namespace prefixes. This is required to evaluate XPath expression when using transport syntaxes that don't support namespace.

Below follows a list of the sub-components that MAY be present within this component:

- The NamespaceURI element MUST contain one instance of a URI.
- The NamespacePrefix element MUST contain one instance of a string.

3.1.1.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type NsPrefixMappingType SHALL implement the requirements defined in the NsPrefixMapping component.

The NsPrefixMappingType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="NsPrefixMappingType">
  <xs:sequence>
    <xs:element name="NamespaceURI" type="xs:anyURI"/>
    <xs:element name="NamespacePrefix" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of NsPrefixMappingType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.1.3 JSON Syntax

The NsPrefixMappingType JSON object SHALL implement in JSON syntax the requirements defined in the NsPrefixMapping component.

The NsPrefixMappingType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dsb-NsPrefixMappingType": {
  "type": "object",
  "properties": {
    "uri": {
      "type": "string"
    },
    "pre": {
      "type": "string"
    }
  },
  "required": ["uri", "pre"]
}
```
Properties in the JSON schema above SHALL implement sub-component of \( NsPrefixMapping \) component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NamespaceURI</td>
<td>uri</td>
</tr>
<tr>
<td>NamespacePrefix</td>
<td>pre</td>
</tr>
</tbody>
</table>

### 3.1.2 Component Any

#### 3.1.2.1 Semantics

This element MAY hold a set of base64 encoded arbitrary data. To help the processing of the data it may be qualified by the mime type element.

Below follows a list of the sub-components that MAY be present within this component:

- The `Content` element MUST occur 1 or more times containing sub-components.
- The `Base64Content` element MUST contain one instance of base64 encoded binary data.
- The optional `MimeType` element MUST contain one instance of a string. This element is denoting the type of the arbitrary data.

#### 3.1.2.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:dss:2.0:base:schema` . The XML type `AnyType` SHALL implement the requirements defined in the Any component.

The `AnyType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="AnyType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="1" name="Content">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Base64Content" type="xs:base64Binary"/>
        </xs:sequence>
        <xs:attribute name="MimeType" type="xs:string" use="optional"/>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Each child element of `AnyType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

@ToDo: Decide on the use of `xs:any`!!!

To provide backward compatibility and convenient processing of XML components the XML schema includes an `xs:any` element. This XML-specific mechanism should be used with caution as it is not compatible with the data-format-neutral approach of this specification.
3.1.2.3 JSON Syntax

The AnyType JSON object SHALL implement in JSON syntax the requirements defined in the Any component.

The AnyType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dsb-AnyType": {
  "type": "object",
  "properties": {
    "content": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dsb-AnyType:Content"
      }
    }
  },
  "required": ["content"]
}
"dsb-AnyType:Content": {
  "type": "object",
  "properties": {
    "b64Content": {
      "type": "string"
    },
    "mimeType": {
      "type": "string"
    }
  },
  "required": ["b64Content"]
}
```

Properties in the JSON schema above SHALL implement sub-component of Any component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>content</td>
</tr>
<tr>
<td>Base64Content</td>
<td>b64Content</td>
</tr>
<tr>
<td>MimeType</td>
<td>mimeType</td>
</tr>
</tbody>
</table>

3.1.3 Component InternationalString

3.1.3.1 Semantics

This element attaches an element to a human-readable string to specify the string’s language. Below follows a list of the sub-components that MAY be present within this component:
- The value element MUST contain one instance of a string. The human readable string. In non-XML representations the value element contains the textual content.
- The lang element MUST contain one instance of a ISO language descriptor. This element identifies the language of the value element.

### 3.1.3.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:dss:2.0:base:schema`. The XML type `InternationalStringType` SHALL implement the requirements defined in the `InternationalString component`.

The `InternationalStringType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="InternationalStringType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute ref="xml:lang" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Each child element of `InternationalStringType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name. The element ‘value’ is represented by the component's XML tag text content.

### 3.1.3.3 JSON Syntax

The `InternationalStringType` JSON object SHALL implement in JSON syntax the requirements defined in the `InternationalString component`.

The `InternationalStringType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dsb-InternationalStringType": {
  "type": "object",
  "properties": {
    "value": {
      "type": "string"
    },
    "lang": {
      "type": "string"
    }
  },
  "required": ["lang"]
}
```

Properties in the JSON schema above SHALL implement sub-component of `InternationalString component` mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>lang</td>
<td>lang</td>
</tr>
</tbody>
</table>
### 3.1.4 Component DigestInfo

#### 3.1.4.1 Semantics

The **DigestInfo** component holds a digest value and an identification of the used digest algorithm. The **DigestMethod** isn’t strongly typed intentionally to support a broad variety of identifiers.

Below follows a list of the sub-components that MAY be present within this component:

- The **DigestMethod** element MUST contain one instance of a string. The string describes the digest algorithm in an appropriate way for the server side processing. Depending on the signature format this may be an OID (e.g. ‘2.16.840.1.101.3.4.2.1’), an URI (e.g. ‘http://www.w3.org/2001/04/xmlenc#sha256’) or a descriptive string (‘SHA-256’).
- The **DigestValue** element MUST contain one instance of base64 encoded binary data.

#### 3.1.4.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:dss:2.0:base:schema`. The XML type **DigestInfoType** SHALL implement the requirements defined in the **DigestInfo** component.

The **DigestInfoType** XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DigestInfoType">
  <xs:sequence>
    <xs:element name="DigestMethod" type="xs:string"/>
    <xs:element name="DigestValue" type="xs:base64Binary"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of **DigestInfoType** XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.4.3 JSON Syntax

The **DigestInfoType** JSON object SHALL implement in JSON syntax the requirements defined in the **DigestInfo** component.

The **DigestInfoType** JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dsb-DigestInfoType": {
  "type": "object",
  "properties": {
    "alg": {
      "type": "string"
    },
    "value": {
      "type": "string"
    }
  },
  "required": ["alg", "value"]
}
```
Properties in the JSON schema above SHALL implement sub-component of `DigestInfo` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigestMethod</td>
<td>alg</td>
</tr>
<tr>
<td>DigestValue</td>
<td>value</td>
</tr>
</tbody>
</table>

### 3.1.5 Component AttachmentReference

#### 3.1.5.1 Semantics

Applications MAY support SOAP 1.2 attachment feature [SOAPAtt] or other attachment specifications (e.g. [SOAPMtom]) to transmit documents.

Below follows a list of the sub-components that MAY be present within this component:

- The optional `DigestInfo` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in section 3.1.4. An element of this type can be used to ensure the integrity of the attachment data. If these elements are supplied the server SHOULD compute a message digest using the algorithm given in `DigestMethod` over the binary data in the octet stream and compare it against the supplied `DigestValue`. If the comparison fails then a `RequesterError` qualified by a `GeneralError` and an appropriate message containing the `AttRefURI` is returned.

- The `AttRefURI` element MUST contain one instance of a URI. SOAP 1.2 attachment feature [SOAPAtt] states that any secondary part (“attachment”) can be referenced by a URI of any URI scheme. `AttRefURI` refers to such a secondary part (“attachment”) and MUST resolve within the compound SOAP message. The default encapsulation mechanism is MIME as specified in the WS-I Attachments Profile [WS-I-Att] (cf. swaRef, http://www.ws-i.org/Profiles/AttachmentsProfile-1.0.html#Referencing_Attachments_from_the_SOAP_Envelope).

#### 3.1.5.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type `AttachmentReferenceType` SHALL implement the requirements defined in the `AttachmentReference` component.

The `AttachmentReferenceType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="AttachmentReferenceType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="DigestInfo" type="dsb:DigestInfoType"/>
  </xs:sequence>
  <xs:attribute name="AttRefURI" type="xs:anyURI" use="required"/>
</xs:complexType>
```

Each child element of `AttachmentReferenceType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.
3.1.5.3 JSON Syntax

The `AttachmentReferenceType` JSON object SHALL implement in JSON syntax the requirements defined in the `AttachmentReference` component.

The `AttachmentReferenceType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dsb-AttachmentReferenceType": {
  "type": "object",
  "properties": {
    "di": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dsb-DigestInfoType"
      }
    },
    "attURI": {
      "type": "string"
    }
  },
  "required": ["attURI"]
}
```

Properties in the JSON schema above SHALL implement sub-component of `AttachmentReference` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DigestInfo</td>
<td>di</td>
</tr>
<tr>
<td>AttRefURI</td>
<td>attURI</td>
</tr>
</tbody>
</table>

3.1.6 Component Base64Data

3.1.6.1 Semantics

The `Base64Data` component is a generic holder for arbitrary data. In addition to the data itself it also contains additional elements to qualify the MimeType of the data. It also offers an Id / Reference pair to implement a deduplication strategy, useful especially for bigger data blobs. The content is contained inside the mutually exclusive elements `Value` or `AttRefURI`. The `Value` element or the XML tag’s content MAY be empty. If it is empty, the `AttRefURI` element MUST point to the components content transferred as an attachment. The `Base64Data` component is a generic holder for arbitrary data. In addition to the data itself it also contains additional elements to qualify the MimeType of the data. It also offers an Id / Reference pair to implement a deduplication strategy, useful especially for bigger data blobs. The content is contained inside the mutually exclusive elements `Value` or `AttRefURI`. The `Value` element or the XML tag’s content MAY be empty. If it is empty, the `AttRefURI` element MUST point to the components content transferred as an attachment. The `Base64Data` component is a generic holder for arbitrary data. In addition to the data itself it also contains additional elements to qualify the MimeType of the data. It also offers an Id / Reference pair to implement a deduplication strategy, useful especially for bigger data blobs. The content is contained inside the mutually exclusive elements `Value` or `AttRefURI`. The `Value` element or the XML tag’s content MAY be empty. If it is empty, the `AttRefURI` element MUST point to the components content transferred as an attachment.
or AttRefURI. The Value element or the XML tag’s content MAY be empty. If it is empty, the AttRefURI element MUST point to the components content transferred as an attachment.

Below follows a list of the sub-components that MAY be present within this component:

- The Value element MUST contain one instance of base64 encoded binary data. This element holds an instance of generic content. This could be a document to be signed, a signature, a schema or other data.
- The AttRef element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.1.5. This element allows to reference content that is transferred in a non-inlined way. These mechanisms may take advantage of optimizations (e.g. optimized transfer encodings). The content of MAY be integrity-protected by a message digest.
- The optional MimeType element MUST contain one instance of a string. This element is denoting the type of the arbitrary data in the value element or the referenced attachment.
- The optional ID element MUST contain one instance of a unique identifier. This identifier gives the binary data a unique label within a particular message. Using this identifier and the IDREF element it is possible to avoid redundant content.
- The optional IDREF element MUST contain one instance of a unique identifier reference. This element identifies another binary data element within a particular message.

**Non-normative Comment:**

There are different standards defined for handling and referencing an attachment. Maybe there will be more to come. Therefore the attachment reference mechanism is somehow generic here. Note: If MIME is used as encapsulation mechanism, the MIME content-type is available via a MIME header. However, the MIME headers may not be available to implementations and the SOAP 1.2 attachment feature is not restricted to MIME. Further the MIME header is not secured by the AttachmentReference’s DigestInfo, which is calculated over the binary attachment data (not including the MIME headers). There are different standards defined for handling and referencing an attachment. Maybe there will be more to come. Therefore the attachment reference mechanism is somehow generic here. Note: If MIME is used as encapsulation mechanism, the MIME content-type is available via a MIME header. However, the MIME headers may not be available to implementations and the SOAP 1.2 attachment feature is not restricted to MIME. Further the MIME header is not secured by the AttachmentReference’s DigestInfo, which is calculated over the binary attachment data (not including the MIME headers). There are different standards defined for handling and referencing an attachment. Maybe there will be more to come. Therefore the attachment reference mechanism is somehow generic here.

Note: If MIME is used as encapsulation mechanism, the MIME content-type is available via a MIME header. However, the MIME headers may not be available to implementations and the SOAP 1.2 attachment feature is not restricted to MIME. Further the MIME header is not secured by the AttachmentReference’s DigestInfo, which is calculated over the binary attachment data (not including the MIME headers).

### 3.1.6.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type Base64DataType SHALL implement the requirements defined in the Base64Data component.

The Base64DataType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.
Each child element of `Base64DataType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.6.3 JSON Syntax

The `Base64DataType` JSON object SHALL implement in JSON syntax the requirements defined in the `Base64Data` component.

The `Base64DataType` JSON object SHALL be defined as in JSON Schema file `[JSON SCHEMA FILE NAME]` whose location is detailed in clause `[CLAUSE FOR LINK TO THE JSON SCHEMA FILE]`, and is copied below for information.

```
"dsb-Base64DataType": {  
  "type": "object",  
  "properties": {  
    "ID": {  
      "type": "string"  
    },  
    "value": {  
      "type": "string"  
    },  
    "attRef": {  
      "type": "object",  
      "$ref": "#/definitions/dsb-AttachmentReferenceType"  
    },  
    "mimeType": {  
      "type": "string"  
    },  
    "IDREF": {  
      "type": "string"  
    }  
  },  
  "minProperties": 1
}
```

Properties in the JSON schema above SHALL implement sub-component of `Base64Data` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td><code>value</code></td>
</tr>
<tr>
<td>AttRef</td>
<td><code>attRef</code></td>
</tr>
<tr>
<td>MimeType</td>
<td><code>mimeType</code></td>
</tr>
<tr>
<td>ID</td>
<td><code>ID</code></td>
</tr>
</tbody>
</table>
3.1.7 Component Result

3.1.7.1 Semantics

The Result element is returned with every response message.

Below follows a list of the sub-components that MAY be present within this component:

- The ResultMajor element MUST contain one instance of a URI. Its value is limited to an item of the following set:
  - urn:oasis:names:tc:dss:1.0:resultmajor:Success
  - urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError
  - urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError
  - urn:oasis:names:tc:dss:1.0:resultmajor:InsufficientInformation

The ResultMajor element describes the most significant component of the result code.

- The optional ResultMinor element MUST contain a URI.

- The optional ResultMessage element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.1.3. It represents a message which MAY be returned to an operator, logged, used for debugging, etc.

3.1.7.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type ResultType SHALL implement the requirements defined in the Result component.

The ResultType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="ResultType">
  <xs:sequence>
    <xs:element name="ResultMajor">
      <xs:simpleType>
        <xs:restriction base="xs:anyURI">
          <xs:enumeration value="urn:oasis:names:tc:dss:1.0:resultmajor:Success"/>
          <xs:enumeration value="urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError"/>
          <xs:enumeration value="urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError"/>
          <xs:enumeration value="urn:oasis:names:tc:dss:1.0:resultmajor:InsufficientInformation"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element minOccurs="0" name="ResultMinor" type="xs:anyURI"/>
    <xs:element minOccurs="0" name="ResultMessage" type="dsb:InternationalStringType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of ResultType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.
3.1.7.3 JSON Syntax

The ResultType JSON object SHALL implement in JSON syntax the requirements defined in the Result component.

The ResultType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dsb-ResultType": {
  "type": "object",
  "properties": {
    "maj": {
      "type": "string",
      "enum": ["urn:oasis:names:tc:dss:1.0:resultmajor:Success",
               "urn:oasis:names:tc:dss:1.0:resultmajor:RequesterError",
               "urn:oasis:names:tc:dss:1.0:resultmajor:ResponderError",
               "urn:oasis:names:tc:dss:1.0:resultmajor:InsufficientInformation"],
    "min": {
      "type": "string"
    },
    "msg": {
      "type": "object",
      "$ref": "/definitions/dsb-InternationalStringType"
    }
  }
},
"required": ["maj"]
```

Properties in the JSON schema above SHALL implement sub-component of Result component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultMajor</td>
<td>maj</td>
</tr>
<tr>
<td>ResultMinor</td>
<td>min</td>
</tr>
<tr>
<td>ResultMessage</td>
<td>msg</td>
</tr>
</tbody>
</table>

3.1.8 Component RequestBase

3.1.8.1 Semantics

The RequestBase component is the base structure for request elements defined by the core protocol or profiles.

Below follows a list of the sub-components that MAY be present within this component:
The optional **Profile** element MAY occur zero or more times containing a URI. This element indicates a set of DSS profiles. It is used by the client to select profiles the server supports.

The optional **RequestID** element MUST contain one instance of a string. The **RequestID** element is used to correlate requests with responses. When present in a request, the server MUST return it in the response.

### 3.1.8.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:dss:2.0:base:schema`. The XML type **RequestBaseType** SHALL implement the requirements defined in the **RequestBase** component.

The **RequestBaseType** XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType abstract="true" name="RequestBaseType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="Profile" type="xs:anyURI"/>
  </xs:sequence>
  <xs:attribute name="RequestID" type="xs:string" use="optional"/>
</xs:complexType>
```

Each child element of **RequestBaseType** XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.8.3 JSON Syntax

The component **RequestBase** is abstract and therefore has no JSON definition.

### 3.1.9 Component ResponseBase

#### 3.1.9.1 Semantics

The **ResponseBase** component is the base structure for response elements defined by the core protocol or profiles.

Below follows a list of the sub-components that MAY be present within this component:
• The Result element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.1.7. The Fehler! Verweisquelle konnte nicht gefunden werden. element represents the status of the request.

• The optional AppliedProfile element MAY occur zero or more times containing a URI. This element lists the set of DSS profile applied by the server. This set MAY include the set of profiles requested by the client. But the server MAY use more comprehensive set of profiles and add additional profiles not requested by the client.

• The optional RequestID element MUST contain one instance of a string. The RequestID element is used to correlate this response with its request.

3.1.9.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type ResponseBaseType SHALL implement the requirements defined in the ResponseBase component.

The ResponseBaseType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType abstract="true" name="ResponseBaseType">
  <xs:sequence>
    <xs:element name="Result" type="dsb:ResultType"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="AppliedProfile" type="xs:anyURI"/>
  </xs:sequence>
  <xs:attribute name="RequestID" type="xs:string" use="optional"/>
</xs:complexType>
```

Each child element of ResponseBaseType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.9.3 JSON Syntax

The component ResponseBase is abstract and therefore has no JSON definition.

3.1.10 Component Info

3.1.10.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:

• The Call element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in section 3.1.11.

3.1.10.2 XML Syntax

The XML element is defined in the XML namespace urn:oasis:names:tc:dss:2.0:base:schema. The XML type InfoType SHALL implement the requirements defined in the Info component.

The InfoType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="InfoType">
```

Each child element of InfoType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.10.3 JSON Syntax

The component Info is not used as JSON object directly.

### 3.1.11 Component Description

#### 3.1.11.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:

- The Name element MUST contain one instance of a string.
- The Specification element MUST contain one instance of a URI.
- The Schema element MUST contain one instance of a URI.
- The optional Option element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in section 3.1.11.

#### 3.1.11.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:dss:2.0:base:schema`. The XML type DescriptionType SHALL implement the requirements defined in the Description component.

The DescriptionType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DescriptionType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Specification" type="xs:anyURI"/>
    <xs:element name="Schema" type="xs:anyURI"/>
    <xs:element name="Option" type="dsb:DescriptionType" maxOccurs="unbounded" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of DescriptionType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.11.3 JSON Syntax

The component Description is not used as JSON object directly.
### 3.1.12 Component InputDocuments

#### 3.1.12.1 Semantics

This element is used to send input documents to a DSS server, whether for signing or verifying. An input document can be any piece of data that can be used as input to a signature or timestamp calculation. An input document can even be a signature or timestamp (for example, a pre-existing signature can be counter-signed or timestamped). An input document could also be a `<ds:Manifest>`, allowing the client to handle manifest creation while using the server to create the rest of the signature. Manifest validation is supported by an optional input / output.

Below follows a list of the sub-components that MAY be present within this component:

- The `Document` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.14.
- The `TransformedData` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.15. It contains the binary output of a chain of transforms applied by a client.
- The `DocumentHash` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.16. It contains a set of digest algorithm and the corresponding hashes. Required transformation steps

#### 3.1.12.2 XML Syntax

The XML type `InputDocumentsType` SHALL implement the requirements defined in the `InputDocuments` component.

The `InputDocumentsType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="InputDocumentsType">
  <xs:choice>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="Document" type="dss:DocumentType"/>
    </xs:sequence>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="TransformedData" type="dss:TransformedDataType"/>
    </xs:sequence>
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="DocumentHash" type="dss:DocumentHashType"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Each child element of `InputDocumentsType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.12.3 JSON Syntax

The `InputDocumentsType` JSON object SHALL implement in JSON syntax the requirements defined in the `InputDocuments` component.
The `InputDocumentsType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-InputDocumentsType": { 
  "type": "object", 
  "properties": { 
    "doc": { 
      "type": "array", 
      "items": { 
        "type": "object", 
        "$ref": "/definitions/dss-DocumentType"
      }
    },
    "transformed": { 
      "type": "array", 
      "items": { 
        "type": "object", 
        "$ref": "/definitions/dss-TransformedDataType"
      }
    },
    "docHash": { 
      "type": "array", 
      "items": { 
        "type": "object", 
        "$ref": "/definitions/dss-DocumentHashType"
      }
    }
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of `InputDocuments` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>doc</td>
</tr>
<tr>
<td>TransformedData</td>
<td>transformed</td>
</tr>
<tr>
<td>DocumentHash</td>
<td>docHash</td>
</tr>
</tbody>
</table>

### 3.1.13 Component DocumentBase

#### 3.1.13.1 Semantics

The `DocumentBaseType` inherits its elements to the components `DocumentType`, `TransformedDataType` and `DocumentHashType`. The `DocumentBaseType` contains the basic information shared by the inheriting components and remaining persistent during the process from input document retrieval until digest calculation for the relevant document.

Below follows a list of the sub-components that MAY be present within this component:
- The optional **ID** element MUST contain one instance of a unique identifier. This identifier gives the input document a unique label within a particular request message. Through this identifier, an optional input can refer to a single input document. Using this identifier and the **IDREF** element it is possible to avoid redundant content.

- The optional **RefURI** element MUST contain one instance of a URI. This specifies the value for a `<ds:Reference>` element’s URI attribute when referring to this input document. The **RefURI** element SHOULD be specified. Not more than one **RefURI** element may be omitted in a single signing request.

- The optional **RefType** element MUST contain one instance of a URI. This specifies the value for a `<ds:Reference>` element’s Type attribute when referring to this input document.

- The optional **SchemaRefs** element MUST contain one instance of a unique identifier reference. The identified schemas are to be used to process the ID attributes during parsing and for XPath evaluation. If anything else but Schema are referred to, the server MUST report an error. If a referred to Schema is not used by the XML document instance this MAY be ignored or reported to the client in the **Fehler! Verweisquelle konnte nicht gefunden werden.** subcomponent ResultMessage (for the definition of Schema subcomponent see the specification in section 3.1.30). The Document is assumed to be valid against the first Schema referred to by **SchemaRefs**. If a Schemas element is referred to first by **SchemaRefs** the document is assumed to be valid against the first Schema inside **SchemaRefs**. In both cases, the remaining schemas may occur in any order and are used either directly or indirectly by the first schema. If present, the server MUST use the schemas to identify the ID attributes and MAY also perform complete validation against the schemas.

**Non-normative Comment:**

It is recommended to use `xml:id` as defined in [xml:id] as id in the payload being referenced by a `<ds:Reference>`, because the schema then does not have to be supplied for identifying the ID attributes.

### 3.1.13.2 XML Syntax

The XML type **DocumentBaseType** SHALL implement the requirements defined in the **DocumentBase** component.

The **DocumentBaseType** XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType abstract="true" name="DocumentBaseType">
  <xs:attribute name="ID" type="xs:ID" use="optional"/>
  <xs:attribute name="RefURI" type="xs:anyURI" use="optional"/>
  <xs:attribute name="RefType" type="xs:anyURI" use="optional"/>
  <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
</xs:complexType>
```

Each child element of **DocumentBaseType** XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.13.3 JSON Syntax

The **DocumentBaseType** JSON object SHALL implement in JSON syntax the requirements defined in the **DocumentBase** component.

The **DocumentBaseType** JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.
Properties in the JSON schema above SHALL implement sub-component of `DocumentBase` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>refURI</td>
<td>refURI</td>
</tr>
<tr>
<td>refType</td>
<td>refType</td>
</tr>
<tr>
<td>schemaRefs</td>
<td>schemaRefs</td>
</tr>
</tbody>
</table>

### 3.1.14 Component Document

#### 3.1.14.1 Semantics

The **Document** component contains input data for DSS processing. Below follows a list of the sub-components that MAY be present within this component:

- The **Base64Data** element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.1.6.

A set of sub-components is inherited from component 3.1.13 and is not repeated here.

#### 3.1.14.2 XML Syntax

The XML type **DocumentType** SHALL implement the requirements defined in the **Document** component. The **DocumentType** XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DocumentType">
  <xs:complexContent>
  </xs:complexContent>
</xs:complexType>
```
Each child element of DocumentType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.14.3 JSON Syntax

The DocumentType JSON object SHALL implement in JSON syntax the requirements defined in the Document component.

The DocumentType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-DocumentType": {
  "type": "object",
  "properties": {
    "ID": {
      "type": "string"
    },
    "refURI": {
      "type": "string"
    },
    "refType": {
      "type": "string"
    },
    "schemaRefs": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-DocumentType"
      }
    },
    "b64Data": {
      "type": "object",
      "$ref": "#/definitions/dsb-Base64DataType"
    }
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of Document component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>refURI</td>
<td>refURI</td>
</tr>
<tr>
<td>refType</td>
<td>refType</td>
</tr>
<tr>
<td>SchemaRefs</td>
<td>schemaRefs</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
</tr>
<tr>
<td>Base64Data</td>
<td>b64Data</td>
</tr>
</tbody>
</table>

### 3.1.15 Component TransformedData

#### 3.1.15.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:
- The optional `<Transforms>` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.2.1. This is the sequence of transforms applied by the client. It specifies the value for a `<ds:Reference>` element’s `<ds:Transforms>` child element. In other words, this specifies transforms that the client has already applied to the input document before the server will hash it. This component is required on a `SignRequest`, optional on a `VerifyRequest`.

- The `<Base64Data>` element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.1.6. This element gives the binary output of a sequence of transforms to be hashed at the server side.

- The optional `<WhichReference>` element MUST contain one instance of an integer. As there may be multiple `<TransformedDataType>`/`<DocumentHashType>` components of the same document having the same URI [RFC 2396] and `<RefType>` on a `SignRequest` or `VerifyRequest` - their correspondance to an already existing `<ds:Reference>` however needs to be established on a `VerifyRequest` only. There is a need to disambiguate such cases. This element hence offers a way to clearly identify the `<ds:Reference>` when URI and `<RefType>` match multiple components. The corresponding `<ds:Reference>` is indicated by this zero-based `<WhichReference>` element (0 means the first `<ds:Reference>` in the signature, 1 means the second, and so on). As there may be multiple `<TransformedDataType>`/`<DocumentHashType>` components of the same document having the same URI [RFC 2396] and `<RefType>` on a `SignRequest` or `VerifyRequest` - their correspondance to an already existing `<ds:Reference>` however needs to be established on a `VerifyRequest` only. There is a need to disambiguate such cases. This element hence offers a way to clearly identify the `<ds:Reference>` when URI and `<RefType>` match multiple components. The corresponding `<ds:Reference>` is indicated by this zero-based `<WhichReference>` element (0 means the first `<ds:Reference>` in the signature, 1 means the second, and so on). As there may be multiple `<TransformedDataType>`/`<DocumentHashType>` components of the same document having the same URI [RFC 2396] and `<RefType>` on a `SignRequest` or `VerifyRequest` - their correspondance to an already existing `<ds:Reference>` however needs to be established on a `VerifyRequest` only. There is a need to disambiguate such cases. This element hence offers a way to clearly identify the `<ds:Reference>` when URI and `<RefType>` match multiple components. The corresponding `<ds:Reference>` is indicated by this zero-based `<WhichReference>` element (0 means the first `<ds:Reference>` in the signature, 1 means the second, and so on). As there may be multiple `<TransformedDataType>`/`<DocumentHashType>` components of the same document having the same URI [RFC 2396] and `<RefType>` on a `SignRequest` or `VerifyRequest` - their correspondance to an already existing `<ds:Reference>` however needs to be established on a `VerifyRequest` only. There is a need to disambiguate such cases. This element hence offers a way to clearly identify the `<ds:Reference>` when URI and `<RefType>` match multiple components. The corresponding `<ds:Reference>` is indicated by this zero-based `<WhichReference>` element (0 means the first `<ds:Reference>` in the signature, 1 means the second, and so on). This component is ignored on a `SignRequest`, optional on a `VerifyRequest`.

A set of sub-components is inherited from component 3.1.13 and is not repeated here.
Non-normative Comment:
It may be possible to establish the `<ds:References>`/`TransformedDataType/DocumentHashType` correspondence by comparing the optionally supplied chain of transforms to those of the `<ds:References>` having the same URI and `RefType` in the supplied `<ds:Signature>` if this chain of transform has been supplied. This can be quite expensive and even outnumber the advantages of `TransformedDataType/DocumentHashType`.

3.1.15.2 XML Syntax
The XML type `TransformedDataType` SHALL implement the requirements defined in the `TransformedData` component.

The `TransformedDataType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="TransformedDataType">
  <xs:complexContent>
    <xs:extension base="dss:DocumentBaseType">
      <xs:sequence>
        <xs:element minOccurs="0" name="Transforms" type="ds:TransformsType"/>
        <xs:element name="Base64Data" type="dsb:Base64DataType"/>
      </xs:sequence>
      <xs:attribute name="WhichReference" type="xs:integer" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Each child element of `TransformedDataType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.15.3 JSON Syntax
The `TransformedDataType` JSON object SHALL implement in JSON syntax the requirements defined in the `TransformedData` component.

The `TransformedDataType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-TransformedDataType": {
  "type": "object",
  "properties": {
    "ID": {
      "type": "string"
    },
    "refURI": {
      "type": "string"
    },
    "refType": {
      "type": "string"
    },
    "schemaRefs": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-DocumentType"
      }
    }
  }
}
```
"transforms": {
   "type": "object",
   "$ref": "/definitions/dsig-TransformsType"
 },
"b64Data": {
   "type": "object",
   "$ref": "/definitions/dsb-Base64DataType"
 },
"whichRef": {
   "type": "integer"
 },
"required": ["b64Data"]
}

Properties in the JSON schema above SHALL implement sub-component of TransformedData component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>RefURI</td>
<td>refURI</td>
</tr>
<tr>
<td>RefType</td>
<td>refType</td>
</tr>
<tr>
<td>SchemaRefs</td>
<td>schemaRefs</td>
</tr>
<tr>
<td>Transforms</td>
<td>transforms</td>
</tr>
<tr>
<td>Base64Data</td>
<td>b64Data</td>
</tr>
<tr>
<td>WhichReference</td>
<td>whichRef</td>
</tr>
</tbody>
</table>

3.1.16 Component DocumentHash

3.1.16.1 Semantics

The DocumentHash component represents a document that will not be transported to the server but just the calculated digest of it. This may be useful to limit the amount of data transferred or to ensure privacy of the document.

Below follows a list of the sub-components that MAY be present within this component:
• The optional `Transforms` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.2.1. It specifies the value for a `<ds:Reference>` element's `<ds:Transforms>` child element when referring to this document hash. In other words, this specifies transforms that the client has already applied to the input document before hashing it. This component is required on a `SignRequest`, optional on a `VerifyRequest`. This component is required on a `SignRequest`, optional on a `VerifyRequest`.

• The `DigestInfos` element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in section 3.1.4. This element MAY contain more than one `DigestInfo` sub-component to represent the digest values calculated with different digest algorithms. This may be useful when a requestor doesn’t know upfront which digest algorithms are supported / accepted by the server for signing. In the case of a verification request the client may not be able to parse the signature and instead calculate the digest for a comprehensive set of digest algorithms.

• The optional `WhichReference` element MUST contain one instance of an integer. As there may be multiple `TransformedDataType/DocumentHashType` components of the same document having the same `URI [RFC 2396]` and `RefType` on a `SignRequest` or `VerifyRequest` - their correspondence to an already existing `<ds:Reference>` however needs to be established on a `VerifyRequest` only. There is a need to disambiguate such cases. This element hence offers a way to clearly identify the `<ds:Reference>` when `URI` and `RefType` match multiple components. The corresponding `<ds:Reference>` is indicated by this zero-based `WhichReference` element (0 means the first `<ds:Reference> in the signature, 1 means the second, and so on).

A set of sub-components is inherited from component 3.1.13 and is not repeated here.

### 3.1.16.2 XML Syntax

The XML type `DocumentHashType` SHALL implement the requirements defined in the `DocumentHash` component.

The `DocumentHashType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DocumentHashType">
  <xs:complexContent>
    <xs:extension base="dss:DocumentBaseType">
      <xs:sequence>
        <xs:element minOccurs="0" name="Transforms" type="ds:TransformsType"/>
        <xs:element maxOccurs="unbounded" minOccurs="1" name="DigestInfos" type="dsb:DigestInfoType"/>
      </xs:sequence>
      <xs:attribute name="WhichReference" type="xs:integer" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Each child element of `DocumentHashType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.16.3 JSON Syntax

The `DocumentHashType` JSON object SHALL implement in JSON syntax the requirements defined in the `DocumentHash` component.
The `DocumentHashType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-DocumentHashType": { 
  "type": "object", 
  "properties": { 
    "ID": { 
      "type": "string" 
    }, 
    "refURI": { 
      "type": "string" 
    }, 
    "refType": { 
      "type": "string" 
    }, 
    "schemaRefs": { 
      "type": "array", 
      "items": { 
        "type": "object", 
        "$ref": "#/definitions/dss-DocumentType" 
      } 
    }, 
    "transforms": { 
      "type": "object", 
      "$ref": "#/definitions/dsig-TransformsType" 
    }, 
    "di": { 
      "type": "array", 
      "items": { 
        "type": "object", 
        "$ref": "#/definitions/dsb-DigestInfoType" 
      } 
    }, 
    "whichRef": { 
      "type": "integer" 
    } 
  }, 
  "required": ["di"] 
}
```

Properties in the JSON schema above SHALL implement sub-component of `DocumentHash` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>refURI</td>
<td>refURI</td>
</tr>
<tr>
<td>refType</td>
<td>refType</td>
</tr>
<tr>
<td>schemaRefs</td>
<td>schemaRefs</td>
</tr>
<tr>
<td>transforms</td>
<td>transforms</td>
</tr>
<tr>
<td>di</td>
<td>di</td>
</tr>
</tbody>
</table>
3.1.17 Component SignRequest

3.1.17.1 Semantics

The SignRequest component is sent by the client to request a signature or timestamp on some input documents.

Below follows a list of the sub-components that MAY be present within this component:

- The optional InputDocuments element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.12.
- The optional OptionalInputs element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.24. It is intended to transport additional input elements of the signing request.

A set of sub-components is inherited from component 3.1.8 and is not repeated here.

3.1.17.2 XML Syntax

The XML type SignRequestType SHALL implement the requirements defined in the SignRequest component.

The SignRequestType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xsd:complexType name="SignRequestType">
  <xsd:complexContent>
    <xsd:extension base="dsb:RequestBaseType">
      <xsd:sequence>
        <xsd:element minOccurs="0" name="InputDocuments" type="dss:InputDocumentsType"/>
        <xsd:element minOccurs="0" name="OptionalInputs" type="dss:OptionalInputsSignType"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Each child element of SignRequestType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.17.3 JSON Syntax

The SignRequestType JSON object SHALL implement in JSON syntax the requirements defined in the SignRequest component.

The SignRequestType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignRequestType": {
  "type": "object",
  "properties": {
    "profile": {
      
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of `SignRequest` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile</td>
<td>profile</td>
</tr>
<tr>
<td>RequestID</td>
<td>reqID</td>
</tr>
<tr>
<td>InputDocuments</td>
<td>inDocs</td>
</tr>
<tr>
<td>OptionalInputs</td>
<td>optInp</td>
</tr>
</tbody>
</table>

### 3.1.18 Component SignResponse

#### 3.1.18.1 Semantics

The `SignResponse` component returns the requested signature or timestamp to the requestor.

Below follows a list of the sub-components that MAY be present within this component:

- The optional `OptionalOutputs` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.27. The `OptionalOutputs` element contains additional signing related outputs returned by the server.
- The optional `SignatureObject` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.19.

A set of sub-components is inherited from component 3.1.9 and is not repeated here.

#### 3.1.18.2 XML Syntax

The XML type `SignResponseType` SHALL implement the requirements defined in the `SignResponse` component.

The `SignResponseType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.
Each child element of `SignResponseType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.18.3 JSON Syntax

The `SignResponseType` JSON object SHALL implement in JSON syntax the requirements defined in the `SignResponse` component.

The `SignResponseType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignResponseType": {
  "type": "object",
  "properties": {
    "result": {
      "type": "object",
      "$ref": "#/definitions/dsb-ResultType"
    },
    "profile": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "reqID": {
      "type": "string"
    },
    "optOutp": {
      "type": "object",
      "$ref": "#/definitions/dss-OptionalOutputsSignType"
    },
    "sigObj": {
      "type": "object",
      "$ref": "#/definitions/dss-SignatureObjectType"
    }
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of `SignResponse` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>result</td>
</tr>
</tbody>
</table>
Component SignatureObject

3.1.19.1 Semantics
The SignatureObject component contains a signature or timestamp of some sort. This element is returned in a sign response message, and sent in a verify request message.

Below follows a list of the sub-components that MAY be present within this component:

- The Base64Signature element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.1.6. A base64 encoding of some arbitrary signature, such as a XML signature [XMLDSIG], PGP [RFC 2440] or CMS [RFC 3852] signature. The type of signature is specified by the MimeType element of the Base64DataType component.
- The SignaturePtr element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in the core specification in section 3.1.20. This element is used to point to an XML signature in an input (for a verify request) or output (for a sign response) document in which a signature is enveloped.
- The optional SchemaRefs element MUST contain one instance of a unique identifier reference. The identified schemas are to be used to process the ID attributes during parsing and for XPath evaluation. If anything else but <Schema> are referred to, the server MUST report an error. If a referred to <Schema> is not used by the XML document instance this MAY be ignored or reported to the client in the Fehler! Verweisquelle konnte nicht gefunden werden. subcomponent ResultMessage (for the definition of Schema subcomponent see the specification of 3.1.30).

3.1.19.2 XML Syntax
The XML type SignatureObjectType SHALL implement the requirements defined in the SignatureObject component.

The SignatureObjectType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SignatureObjectType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="Base64Signature" type="dsb:Base64DataType"/>
      <xs:element name="SignaturePtr" type="dss:SignaturePtrType"/>
    </xs:choice>
  </xs:sequence>
  <xs:attribute name="SchemaRefs" type="xs:IDREFS" use="optional"/>
</xs:complexType>
```
Each child element of SignatureObjectType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.19.3 JSON Syntax

The SignatureObjectType JSON object SHALL implement in JSON syntax the requirements defined in the SignatureObject component.

The SignatureObjectType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignatureObjectType": {
  "type": "object",
  "properties": {
    "b64Sig": {
      "type": "object",
      "$ref": "/definitions/dsb-Base64DataType"
    },
    "sigPtr": {
      "type": "object",
      "$ref": "/definitions/dss-SignaturePtrType"
    },
    "schemaRefs": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "/definitions/dss-DocumentBaseType"
      }
    }
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of SignatureObject component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base64Signature</td>
<td>b64Sig</td>
</tr>
<tr>
<td>SignaturePtr</td>
<td>sigPtr</td>
</tr>
<tr>
<td>SchemaRefs</td>
<td>schemaRefs</td>
</tr>
</tbody>
</table>

### 3.1.20 Component SignaturePtr

#### 3.1.20.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:
• The optional NsPrefixMapping element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in section 3.1.1.

• The WhichDocument element MUST contain one instance of a unique identifier reference. This element identifies the input document being pointed at.

• The optional XPath element MUST contain one instance of a string. This element identifies the signature element being pointed at within the selected document. The XPath expression is evaluated from the root node (see section 5.1 of [XPATH]) of the document identified by WhichDocument. The context node for the XPath evaluation is the document’s DocumentElement (see section 2.1 Well-Formed XML Documents [XML]). Regarding namespace declarations for the expression necessary for evaluation see section 1 of [XPATH].

3.1.20.2 XML Syntax

The XML type SignaturePtrType SHALL implement the requirements defined in the SignaturePtr component.

The SignaturePtrType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SignaturePtrType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="NsPrefixMapping" type="dsb:NsPrefixMappingType"/>
  </xs:sequence>
  <xs:attribute name="WhichDocument" type="xs:IDREF" use="required"/>
  <xs:attribute name="XPath" type="xs:string" use="optional"/>
</xs:complexType>
```

Each child element of SignaturePtrType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.20.3 JSON Syntax

The SignaturePtrType JSON object SHALL implement in JSON syntax the requirements defined in the SignaturePtr component.

The SignaturePtrType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignaturePtrType": {
  "type": "object",
  "properties": {
    "nsDecl": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "/definitions/dsb-NsPrefixMappingType"
      }
    },
    "whichDoc": {
      "type": "object",
      "$ref": "/definitions/dss-DocumentBaseType"
    },
    "XPath": {
      "type": "string"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of SignaturePtr component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>NsPrefixMapping</td>
<td>nsDecl</td>
</tr>
<tr>
<td>WhichDocument</td>
<td>whichDoc</td>
</tr>
<tr>
<td>XPath</td>
<td>xPath</td>
</tr>
</tbody>
</table>

### 3.1.21 Component VerifyRequest

#### 3.1.21.1 Semantics

The `VerifyRequest` component is sent by the client to verify a signature or timestamp on some input documents.

Below follows a list of the sub-components that MAY be present within this component:

- The optional `InputDocuments` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.12.
- The optional `OptionalInputs` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.25.
- The optional `SignatureObject` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.19. The `SignatureObject` element contains a signature or timestamp, or else contains a `<SignaturePtr>` that points to an XML signature in one of the input documents.

A set of sub-components is inherited from component 3.1.8 and is not repeated here.

#### 3.1.21.2 XML Syntax

The XML type `VerifyRequestType` SHALL implement the requirements defined in the `VerifyRequest` component.

The `VerifyRequestType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="VerifyRequestType">
<xs:complexContent>
<xs:extension base="dsb:RequestBaseType">
<xs:sequence>
<xs:element minOccurs="0" name="InputDocuments" type="dss:InputDocumentsType"/>
<xs:element minOccurs="0" name="OptionalInputs" type="dss:OptionalInputsVerifyType"/>
<xs:element minOccurs="0" name="SignatureObject" type="dss:SignatureObjectType"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>
```
Each child element of `VerifyRequestType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.21.3 JSON Syntax

The `VerifyRequestType` JSON object SHALL implement in JSON syntax the requirements defined in the `VerifyRequest` component.

The `VerifyRequestType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-VerifyRequestType": {
  "type": "object",
  "properties": {
    "profile": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "reqID": {
      "type": "string"
    },
    "inDocs": {
      "type": "object",
      "$ref": "#/definitions/dss-InputDocumentsType"
    },
    "optInp": {
      "type": "object",
      "$ref": "#/definitions/dss-OptionalInputsVerifyType"
    },
    "sigObj": {
      "type": "object",
      "$ref": "#/definitions/dss-SignatureObjectType"
    }
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of `VerifyRequest` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile</td>
<td>profile</td>
</tr>
<tr>
<td>RequestID</td>
<td>reqID</td>
</tr>
<tr>
<td>InputDocuments</td>
<td>inDocs</td>
</tr>
<tr>
<td>OptionalInputs</td>
<td>optInp</td>
</tr>
<tr>
<td>SignatureObject</td>
<td>sigObj</td>
</tr>
</tbody>
</table>
3.1.22 Component VerifyResponse

3.1.22.1 Semantics
The VerifyResponse component is returned by the server to provide the results of verification. Below follows a list of the sub-components that MAY be present within this component:

- The optional OptionalOutputs element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.28.

A set of sub-components is inherited from component 3.1.9 and is not repeated here.

3.1.22.2 XML Syntax
The XML type VerifyResponseType SHALL implement the requirements defined in the VerifyResponse component.

The VerifyResponseType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="VerifyResponseType">
  <xs:complexContent>
    <xs:extension base="dsb:ResponseBaseType">
      <xs:sequence>
        <xs:element minOccurs="0" name="OptionalOutputs" type="dss:OptionalOutputsVerifyType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Each child element of VerifyResponseType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.22.3 JSON Syntax
The VerifyResponseType JSON object SHALL implement in JSON syntax the requirements defined in the VerifyResponse component.

The VerifyResponseType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-VerifyResponseType": {
  "type": "object",
  "properties": {
    "result": {
      "type": "object",
      "$ref": "/definitions/dsb-ResultType"
    },
    "profile": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "reqID": {
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of VerifyResponse component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Result</td>
<td>result</td>
</tr>
<tr>
<td>AppliedProfile</td>
<td>profile</td>
</tr>
<tr>
<td>RequestID</td>
<td>reqID</td>
</tr>
<tr>
<td>OptionalOutputs</td>
<td>optOutp</td>
</tr>
</tbody>
</table>

### 3.1.23 Component OptionalInputsBase

#### 3.1.23.1 Semantics

The OptionalInputsBase contains a common set of additional inputs associated with the processing of the request. Profiles will specify the allowed optional inputs and their default values. If a server doesn’t recognize or can’t handle any optional input, it MUST reject the request with a ResultMajor code of RequesterError and a ResultMinor code of NotSupported. All request messages can contain an OptionalInputSign or OptionalInputVerify element depending on the method called. The OptionalInputsBase component defines the elements that are common to all input inputs. Several optional inputs are defined in this document, and profiles can define additional ones.

Below follows a list of the sub-components that MAY be present within this component:
The optional `ServicePolicy` element MAY occur zero or more times containing a URI. This element allows the client to define a set of policies under which the server MUST perform the requested operation. The policy may include information on the characteristics of the server that are not covered by the `Profile` element. The `ServicePolicy` element may be used to select a specific policy if a service supports multiple policies for a specific profile, or as a sanity-check to make sure the server implements the policy the client expects.

The optional `ClaimedIdentity` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.29. This element indicates the identity of the client who is making a request. The server may use this to parameterize any aspect of its processing. Profiles that make use of this element MUST define its semantics.

The optional `Language` element MUST contain a ISO language descriptor. The `Language` element indicates which language the client would like to receive `InternationalString` values in. The server should return appropriately localized strings, if possible.

The optional `Schemas` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.30. The `Schemas` element provides a mechanism for transporting XML schemas required for validating an XML document along with the request message.

The optional `AddTimestamp` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.31. The `AddTimestamp` element indicates that the client wishes the server to embed a timestamp token as a property or attribute of the result or the supplied signature. The timestamp token will be applied to the signature value in the case of CMS/PKCS7 signatures or the `<ds:SignatureValue>` element in the case of XML signatures. Note: Procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS AdES profile [DSS-AdES-P] defines procedures for generating timestamps over the content which is about to be signed (sometimes called content timestamps), and the DSS Timestamp profile [DSS-TS-P] defines procedures for handling standalone timestamps.

The optional `Other` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.37. Other may contain arbitrary content that may be specified in a profile and can also be used to extend the Protocol.

### 3.1.23.2 XML Syntax

The XML type `OptionalInputsBaseType` SHALL implement the requirements defined in the `OptionalInputsBase` component. The `OptionalInputsBaseType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType abstract="true" name="OptionalInputsBaseType">
  <xs:sequence>
    <xs:choice>
      <xs:element maxOccurs="unbounded" minOccurs="0" name="ServicePolicy" type="xs:anyURI"/>
      <xs:element maxOccurs="1" minOccurs="0" name="ClaimedIdentity" type="dss:ClaimedIdentityType"/>
      <xs:element maxOccurs="1" minOccurs="0" name="Language" type="xs:language"/>
      <xs:element maxOccurs="1" minOccurs="0" name="Schemas" type="dss:SchemasType"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```
Each child element of OptionalInputsBaseType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.23.3 JSON Syntax

The component OptionalInputsBase is abstract and therefore has no JSON definition.

3.1.24 Component OptionalInputsSign

3.1.24.1 Semantics

The OptionalInputsSign component defines a set of additional inputs associated with the processing of a signing request. The OptionalInputsSign component contains additional inputs associated with the processing of a signing request. Profiles MAY specify the allowed optional inputs and their default values. The definition of an optional input MAY include a default value, so that a client may omit the OptionalInputsSign yet still get service from any profile-compliant DSS server. If a server doesn't recognize or can't handle any optional input, it MUST reject the request with a ResultMajor code of RequesterError and a ResultMinor code of NotSupported.

Below follows a list of the sub-components that MAY be present within this component:
- The optional `SignatureType` element MUST contain a URI. The `SignatureType` element indicates the type of signature or timestamp to produce (such as a XML signature, a XML timestamp, a RFC 3161 timestamp, a CMS signature, etc.). See section 7.1 for some URI references that MAY be used as the value of this element.

- The optional `IntendedAudience` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.32. A hint regarding the target audience of the requested signature.

- The optional `KeySelector` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.33. The `KeySelector` provides details which key or sets of keys the client is expecting to be used.

- The optional `Properties` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.35. The `Properties` element is used to instruct the server to add certain signed or unsigned properties (aka “signature attributes”) into the signature. The client MAY send the server a particular value to use for each property, or leave the value up to the server to determine. The server MAY add additional properties, even if these aren’t requested by the client.

- The optional `IncludeObject` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.38. The `IncludeObject` element is used to request the creation of an XMLSig enveloping signature.

- The optional `IncludeEContent` element MUST contain a boolean. Its default value is 'false'. If the value of the `IncludeEContent` is ‘true’ a CMS signature includes enveloped (or 'encapsulated') content.

- The optional `SignaturePlacement` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.39. The `SignaturePlacement` element is used to request the creation of an XMLSig enveloped signature placed within a document. The resulting document with the enveloped signature is placed in the optional output `DocumentWithSignature`.

- The optional `SignedReferences` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.41. The `SignedReferences` element gives the client greater control over how the `<ds:Reference>` elements of a XMLSig signature are formed.

- The optional `Nonce` element MUST contain an integer. The `Nonce` element MAY be used to provide a large random number to enable the client correlate a timestamp request with the response.

- The optional `SignatureAlgorithm` element MUST contain a string. The `SignatureAlgorithm` element MAY be used to request a specific signing algorithm.

- The optional `SignatureActivationData` element MUST contain a string. The `SignatureActivationData` element is used by the client to supply activation data.

A set of sub-components is inherited from component 3.1.23 and is not repeated here.

### 3.1.24.2 XML Syntax

The XML type `OptionalInputsSignType` SHALL implement the requirements defined in the `OptionalInputsSign component`. 

---

**References**

[7.1] URI references that MAY be used as the value of the `SignatureType` element.
[3.1.32] Target audience of the requested signature.
[3.1.33] Details on which key or sets of keys the client is expecting.
[3.1.35] Using `Properties` to add properties into the signature.
[3.1.38] Creating an enveloping signature using `IncludeObject`.
[3.1.41] Controlling the creation of a XMLSig signature using `SignedReferences`.
The `OptionalInputsSignType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="OptionalInputsSignType">
  <xs:complexTypeContent>
    <xs:extension base="dss:OptionalInputsBaseType">
      <xs:complexContent>
        <xs:choice>
          <xs:element maxOccurs="1" minOccurs="0" name="SignatureType" type="xs:anyURI"/>
          <xs:element maxOccurs="1" minOccurs="0" name="IntendedAudience" type="dss:IntendedAudienceType"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="KeySelector" type="dss:KeySelectorType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="Properties" type="dss:PropertiesHolderType"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="IncludeObject" type="dss:IncludeObjectType"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="IncludeEContent" type="xs:boolean"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SignaturePlacement" type="dss:SignaturePlacementType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SignedReferences" type="dss:SignedReferencesType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="Nonce" type="xs:integer"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SignatureAlgorithm" type="xs:string"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SignatureActivationData" type="xs:string"/>
        </xs:choice>
      </xs:complexContent>
    </xs:extension>
  </xs:complexTypeContent>
</xs:complexType>
```

Each child element of `OptionalInputsSignType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.24.3 JSON Syntax

The `OptionalInputsSignType` JSON object SHALL implement in JSON syntax the requirements defined in the `OptionalInputsSign` component.

The `OptionalInputsSignType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-OptionalInputsSignType": {
  "type": "object",
  "properties": {
    "policy": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "claimedIdentity": {
      "type": "object",
      "$ref": "#/definitions/dss-ClaimedIdentityType"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of OptionalInputsSign component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServicePolicy</td>
<td>policy</td>
</tr>
<tr>
<td>ClaimedIdentity</td>
<td>claimedIdentity</td>
</tr>
<tr>
<td>Language</td>
<td>lang</td>
</tr>
<tr>
<td>Schemas</td>
<td>schemas</td>
</tr>
<tr>
<td>AddTimestamp</td>
<td>addTimestamp</td>
</tr>
<tr>
<td>Other</td>
<td>other</td>
</tr>
<tr>
<td>SignatureType</td>
<td>sigType</td>
</tr>
<tr>
<td>IntendedAudience</td>
<td>aud</td>
</tr>
<tr>
<td>KeySelector</td>
<td>keySel</td>
</tr>
<tr>
<td>Properties</td>
<td>props</td>
</tr>
<tr>
<td>IncludeObject</td>
<td>incObj</td>
</tr>
<tr>
<td>IncludeEContent</td>
<td>incContent</td>
</tr>
<tr>
<td>SignaturePlacement</td>
<td>sigPlacement</td>
</tr>
<tr>
<td>SignedReferences</td>
<td>sigRefs</td>
</tr>
<tr>
<td>Nonce</td>
<td>nonce</td>
</tr>
<tr>
<td>SignatureAlgorithm</td>
<td>sigAlgo</td>
</tr>
<tr>
<td>SignatureActivationData</td>
<td>sad</td>
</tr>
</tbody>
</table>

### 3.1.25 Component OptionalInputsVerify

#### 3.1.25.1 Semantics

The OptionalInputsVerify component defines a set of additional inputs associated with the processing of a verification request. Profiles MAY specify the allowed optional inputs and their default
values. The definition of an optional input MAY include a default value, so that a client may omit the OptionalInputsVerify yet still get service from any profile-compliant DSS server. If a server doesn’t recognize or can’t handle any optional input, it MUST reject the request with a ResultMajor code of RequesterError and a ResultMinor code of NotSupported.

Below follows a list of the sub-components that MAY be present within this component:
The optional `UseVerificationTime` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.45. The `UseVerificationTime` element instructs the server to attempt to determine the signature’s validity at the specified time, instead of a time determined by the server policy.

The optional `ReturnVerificationTimeInfo` element MUST contain a boolean. Its default value is 'false'. This element can be used by the client to obtain the time instant used by the server to validate the signature.

The optional `AdditionalKeyInfo` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.48. This element provides the server with additional data (such as certificates and CRLs) which it can use to validate the signature. These options are not allowed in multi-signature verification.

The optional `ReturnProcessingDetails` element MUST contain a boolean. Its default value is 'false'. This element instructs the server to return a `ProcessingDetails` element. It is not allowed in multi-signature verification.

The optional `ReturnSigningTimeInfo` element MUST contain a boolean. Its default value is 'false'. This element allows the client to instruct the server to return the time instant associated to the signature creation as a `SigningTimeInfo` element.

The optional `ReturnSignerIdentity` element MUST contain a boolean. Its default value is 'false'.

The optional `ReturnUpdatedSignature` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.31. This element allows the client to instruct the server to return an `UpdatedSignature` output, containing a new or updated signature. This document does not define values for this element, but profiles may provide a set of URIs.

The optional `ReturnTransformedDocument` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.53. The `ReturnTransformedDocument` element instructs the server to return an input document to which the XML signature transforms specified by a particular `<ds:Reference>` have been applied. The result of the transformations will be returned as a `TransformedDocument` element.

The optional `ReturnTimestampedSignature` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.31. It indicates that the client wishes the server to update the signature after its verification by embedding a signature timestamp token as an unauthenticated attribute (see "unauthAttrs" in section 9.1 [RFC 3852]) or "unsigned" property (see section 6.2.5 "The `UnsignedSignatureProperties` element" and section 7.3 "The `SignatureTimeStamp` element" [XAdES]) of the supplied signature. The timestamp token will be on the signature value in the case of CMS/PKCS7signatures or the `<ds:SignatureValue>` element in the case of XML signatures.

The optional `VerifyManifests` element MUST contain a boolean. Its default value is 'false'. This element is allowed in multi-signature verification requests.

A set of sub-components is inherited from component 3.1.23 and is not repeated here.

### 3.1.25.2 XML Syntax

The XML type `OptionalInputsVerifyType` SHALL implement the requirements defined in the `OptionalInputsVerify` component.
The `OptionalInputsVerifyType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="OptionalInputsVerifyType">
  <xs:complexContent>
    <xs:extension base="dss:OptionalInputsBaseType">
      <xs:sequence>
        <xs:choice>
          <xs:element maxOccurs="1" minOccurs="0" name="UseVerificationTime" type="dss:UseVerificationTimeType"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="ReturnVerificationTimeInfo" type="xs:boolean"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="AdditionalKeyInfo" type="dss:AdditionalKeyInfoType"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="ReturnProcessingDetails" type="xs:boolean"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="ReturnSigningTimeInfo" type="xs:boolean"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="ReturnSignerIdentity" type="xs:boolean"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="ReturnUpdatedSignature" type="dss:UpdateSignatureInstructionType"/>
          <xs:element maxOccurs="unbounded" minOccurs="0" name="ReturnTransformedDocument" type="dss:ReturnTransformedDocumentType"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="ReturnTimestampedSignature" type="dss:UpdateSignatureInstructionType"/>
          <xs:element default="false" maxOccurs="1" minOccurs="0" name="VerifyManifests" type="xs:boolean"/>
        </xs:choice>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Each child element of `OptionalInputsVerifyType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.25.3 JSON Syntax

The `OptionalInputsVerifyType` JSON object SHALL implement in JSON syntax the requirements defined in the `OptionalInputsVerify` component.

The `OptionalInputsVerifyType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-OptionalInputsVerifyType": {
  "type": "object",
  "properties": {
    "policy": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "claimedIdentity": {
      "type": "object",
      "$ref": "#/definitions/dss-ClaimedIdentityType"
    },
    "lang": {
      "type": "string"
    }
  }
}
```
"type": "string",
},
"schemas": {
  "type": "object",
  "$ref": "#/definitions/dss-SchemasType"
},
"addTimestamp": {
  "type": "object",
  "$ref": "#/definitions/dss-AddTimestampType"
},
"other": {
  "type": "array",
  "items": {
    "type": "object",
    "$ref": "#/definitions/dss-PropertyType"
  }
},
"useVerificationTime": {
  "type": "object",
  "$ref": "#/definitions/dss-UseVerificationTimeType"
},
"returnVerificationTime": {
  "type": "boolean",
  "default": "false"
},
"addKeyInfo": {
  "type": "array",
  "items": {
    "type": "object",
    "$ref": "#/definitions/dss-AdditionalKeyInfoType"
  }
},
"returnProcDetails": {
  "type": "boolean",
  "default": "false"
},
"returnSigningTime": {
  "type": "boolean",
  "default": "false"
},
"returnSigner": {
  "type": "boolean",
  "default": "false"
},
"returnUpdated": {
  "type": "array",
  "items": {
    "type": "object",
    "$ref": "#/definitions/dss-UpdateSignatureInstructionType"
  }
},
"returnTransformed": {
  "type": "array",
  "items": {
    "type": "object",
    "$ref": "#/definitions/dss-ReturnTransformedDocumentType"
  }
},
"returnTimestamped": {
```
"type": "object",
"$ref": "/definitions/dss-UpdateSignatureInstructionType"
},
"verifyManifests": {
  "type": "boolean",
  "default": "false"
}
```

Properties in the JSON schema above SHALL implement sub-component of OptionalInputsVerify component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServicePolicy</td>
<td>policy</td>
</tr>
<tr>
<td>ClaimedIdentity</td>
<td>claimedIdentity</td>
</tr>
<tr>
<td>Language</td>
<td>lang</td>
</tr>
<tr>
<td>Schemas</td>
<td>schemas</td>
</tr>
<tr>
<td>AddTimestamp</td>
<td>addTimestamp</td>
</tr>
<tr>
<td>Other</td>
<td>other</td>
</tr>
<tr>
<td>UseVerificationTime</td>
<td>useVerificationTime</td>
</tr>
<tr>
<td>ReturnVerificationTimeInfo</td>
<td>returnVerificationTime</td>
</tr>
<tr>
<td>AdditionalKeyInfo</td>
<td>addKeyInfo</td>
</tr>
<tr>
<td>ReturnProcessingDetails</td>
<td>returnProcDetails</td>
</tr>
<tr>
<td>ReturnSigningTimeInfo</td>
<td>returnSigningTime</td>
</tr>
<tr>
<td>ReturnSignerIdentity</td>
<td>returnSigner</td>
</tr>
<tr>
<td>ReturnUpdatedSignature</td>
<td>returnUpdated</td>
</tr>
<tr>
<td>ReturnTransformedDocument</td>
<td>returnTransformed</td>
</tr>
<tr>
<td>ReturnTimestampedSignature</td>
<td>returnTimestamped</td>
</tr>
<tr>
<td>VerifyManifests</td>
<td>verifyManifests</td>
</tr>
</tbody>
</table>

### 3.1.26 Component OptionalOutputsBase

#### 3.1.26.1 Semantics

The OptionalOutputsBase contains a common set of additional outputs associated with the processing of the request. The client MAY request the server to respond with certain optional outputs by sending certain optional inputs. The server MAY also respond with outputs the client didn’t request, depending on the
server’s profile and policy. If a server doesn’t recognize or can’t handle any optional input, it MUST reject
the request with a ResultMajor code of RequesterError and a ResultMinor code of NotSupported.
Below follows a list of the sub-components that MAY be present within this component:

- The optional AppliedPolicy element MAY occur zero or more times containing a URI. This element lists the set of DSS policies used by the server.

- The optional TransformedDocument element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.54. The TransformedDocument element contains a document corresponding to the specified <ds:Reference>, after all the transforms in the reference have been applied. In other words, the hash value of the returned document should equal the <ds:Reference> element’s <ds:DigestValue>.

- The optional Schemas element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.30. The Schemas element is typically used as an optional input in a VerifyRequest. However, there are situations where it may be used as an optional output. For example, a service that makes use of the ReturnUpdatedSignature mechanism may, after verifying a signature over an input document, generate a signature over a document of a different schema than the input document. In this case the Schemas element MAY be used to communicate the XML schemas required for validating a returned XML document.

- The optional DocumentWithSignature element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.40. DocumentWithSignature element contains the input document with the signature inserted.

- The optional Other element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.37. Other may contain arbitrary content that may be specified in a profile and can also be used to extend the Protocol.

3.1.26.2 XML Syntax

The XML type OptionalOutputsBaseType SHALL implement the requirements defined in the OptionalOutputsBase component.

The OptionalOutputsBaseType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType abstract="true" name="OptionalOutputsBaseType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="AppliedPolicy" type="xs:anyURI"/>
    <xs:element maxOccurs="1" minOccurs="0" name="TransformedDocument" type="dss:TransformedDocumentType"/>
    <xs:element maxOccurs="1" minOccurs="0" name="Schemas" type="dss:SchemasType"/>
    <xs:element maxOccurs="1" minOccurs="0" name="DocumentWithSignature" type="dss:DocumentWithSignatureType"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="Other" type="dss:PropertyType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of OptionalOutputsBaseType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.
3.1.26.3 JSON Syntax

The component OptionalOutputsBase is abstract and therefore has no JSON definition.

3.1.27 Component OptionalOutputsSign

3.1.27.1 Semantics

The OptionalOutputsSignType component defines a set of additional outputs associated with the processing of a signing request. This document does not define any additional outputs but profiles may extend the set of additional outputs.

Below follows a list of the sub-components that MAY be present within this component:

A set of sub-components is inherited from component 3.1.26 and is not repeated here.

3.1.27.2 XML Syntax

The XML type OptionalOutputsSignType SHALL implement the requirements defined in the OptionalOutputsSign component.

The OptionalOutputsSignType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="OptionalOutputsSignType">
  <xs:complexContent>
    <xs:extension base="dss:OptionalOutputsBaseType"/>
  </xs:complexContent>
</xs:complexType>
```

Each child element of OptionalOutputsSignType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.27.3 JSON Syntax

The OptionalOutputsSignType JSON object SHALL implement in JSON syntax the requirements defined in the OptionalOutputsSign component.

The OptionalOutputsSignType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-OptionalOutputsSignType": {
  "type": "object",
  "properties": {
    "policy": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  },
  "transformed": {
    "type": "object",
    "$ref": "/#definitions/dss-TransformedDocumentType"
  },
  "schemas": {
   ...
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of OptionalOutputsSign component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppliedPolicy</td>
<td>policy</td>
</tr>
<tr>
<td>TransformedDocument</td>
<td>transformed</td>
</tr>
<tr>
<td>Schemas</td>
<td>schemas</td>
</tr>
<tr>
<td>DocumentWithSignature</td>
<td>docWithSignature</td>
</tr>
<tr>
<td>Other</td>
<td>other</td>
</tr>
</tbody>
</table>

### 3.1.28 Component OptionalOutputsVerify

#### 3.1.28.1 Semantics

The OptionalOutputsVerify component defines a set of additional outputs associated with the processing of a verification request.

Below follows a list of the sub-components that MAY be present within this component:
The optional `VerifyManifestResults` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.43.

The optional `SigningTimeInfo` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.51. The `SigningTimeInfo` element returns the signature’s creation date and time. When there’s no way for the server to determine the signing time, the server MUST omit this element.

The optional `VerificationTimeInfo` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.47. In addition to the verification time, the server MAY include in the `VerificationTimeInfo` element any other relevant time instants that may have been used when determining the verification time or that may be useful for its qualification.

The optional `ProcessingDetails` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.49. The `ProcessingDetails` element elaborates on what signature verification steps succeeded or failed.

The optional `SignerIdentity` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.2.3. The `SignerIdentity` element contains an indication of who performed the signature.

The optional `UpdatedSignature` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.52. The `UpdatedSignature` element contains the returned signature.

The optional `TimestampedSignature` element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.52. The `TimestampedSignature` element contains the returned timestamped signature.

A set of sub-components is inherited from component 3.1.26 and is not repeated here.

### 3.1.28.2 XML Syntax

The XML type `OptionalOutputsVerifyType` SHALL implement the requirements defined in the `OptionalOutputsVerify` component.

The `OptionalOutputsVerifyType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="OptionalOutputsVerifyType">
  <xs:complexContent>
    <xs:extension base="dss:OptionalOutputsBaseType">
      <xs:sequence>
        <xs:choice>
          <xs:element maxOccurs="1" minOccurs="0" name="VerifyManifestResults" type="dss:VerifyManifestResultsType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SigningTimeInfo" type="dss:SigningTimeInfoType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="VerificationTimeInfo" type="dss:VerificationTimeInfoType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="ProcessingDetails" type="dss:ProcessingDetailsType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="SignerIdentity" type="saml2:NameIDType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="UpdatedSignature" type="dss:UpdatedSignatureType"/>
          <xs:element maxOccurs="1" minOccurs="0" name="TimestampedSignature" type="dss:UpdatedSignatureType"/>
        </xs:choice>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
Each child element of `OptionalOutputsVerifyType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.28.3 JSON Syntax

The `OptionalOutputsVerifyType` JSON object SHALL implement in JSON syntax the requirements defined in the `OptionalOutputsVerify` component.

The `OptionalOutputsVerifyType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-OptionalOutputsVerifyType": {
  "type": "object",
  "properties": {
    "policy": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "transformed": {
      "type": "object",
      "$ref": "#/definitions/dss-TransformedDocumentType"
    },
    "schemas": {
      "type": "object",
      "$ref": "#/definitions/dss-SchemasType"
    },
    "docWithSignature": {
      "type": "object",
      "$ref": "#/definitions/dss-DocumentWithSignatureType"
    },
    "other": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-PropertyType"
      }
    },
    "result": {
      "type": "object",
      "$ref": "#/definitions/dss-VerifyManifestResultsType"
    },
    "signingTimeInfo": {
      "type": "object",
      "$ref": "#/definitions/dss-SigningTimeInfoType"
    },
    "verificationTimeInfo": {
      "type": "object",
      "$ref": "#/definitions/dss-VerificationTimeInfoType"
    },
    "procDetails": {
```
Properties in the JSON schema above SHALL implement sub-component of OptionalOutputsVerify component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppliedPolicy</td>
<td>policy</td>
</tr>
<tr>
<td>TransformedDocument</td>
<td>transformed</td>
</tr>
<tr>
<td>Schemas</td>
<td>schemas</td>
</tr>
<tr>
<td>DocumentWithSignature</td>
<td>docWithSignature</td>
</tr>
<tr>
<td>Other</td>
<td>other</td>
</tr>
<tr>
<td>VerifyManifestResults</td>
<td>result</td>
</tr>
<tr>
<td>SigningTimeInfo</td>
<td>signingTimeInfo</td>
</tr>
<tr>
<td>VerificationTimeInfo</td>
<td>verificationTimeInfo</td>
</tr>
<tr>
<td>ProcessingDetails</td>
<td>procDetails</td>
</tr>
<tr>
<td>SignerIdentity</td>
<td>signerIdentity</td>
</tr>
<tr>
<td>UpdatedSignature</td>
<td>updSignature</td>
</tr>
<tr>
<td>TimestampedSignature</td>
<td>timestampedSignature</td>
</tr>
</tbody>
</table>

### 3.1.29 Component ClaimedIdentity

#### 3.1.29.1 Semantics

This element indicates the identity of the client who is making a request. The server may use this to parameterize any aspect of its processing. Profiles that make use of this element MUST define its semantics.
Below follows a list of the sub-components that MAY be present within this component:

- The Name element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in section 3.2.3. The claimed identity may be authenticated using the security binding, according to section 6, or using authentication data provided in the SupportingInfo element. The server MUST check that the asserted Name is authenticated before relying upon the Name.

- The optional SupportingInfo element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.1.2. The SupportingInfo element can be used by profiles to carry information related to the claimed identity. One possible use of SupportingInfo is to carry authentication data that authenticates the request as originating from the claimed identity (examples of authentication data include a password or SAML Assertion, a signature or MAC calculated over the request using a client key).

### 3.1.29.2 XML Syntax

The XML type ClaimedIdentityType SHALL implement the requirements defined in the ClaimedIdentity component.

The ClaimedIdentityType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="ClaimedIdentityType">
  <xs:sequence>
    <xs:element name="Name" type="saml2:NameIDType"/>
    <xs:element minOccurs="0" name="SupportingInfo" type="dsb:AnyType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of ClaimedIdentityType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.29.3 JSON Syntax

The ClaimedIdentityType JSON object SHALL implement in JSON syntax the requirements defined in the ClaimedIdentity component.

The ClaimedIdentityType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-ClaimedIdentityType": {
  "type": "object",
  "properties": {
    "name": {
      "type": "object",
      "$ref": "/definitions/saml2-NameIDType"
    },
    "suppInfo": {
      "type": "object",
      "$ref": "/definitions/dsb-AnyType"
    }
  },
  "required": ["name"]
}
```
Properties in the JSON schema above SHALL implement sub-component of `ClaimedIdentity` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>name</td>
</tr>
<tr>
<td>SupportingInfo</td>
<td>suppInfo</td>
</tr>
</tbody>
</table>

### 3.1.30 Component Schemas

#### 3.1.30.1 Semantics

The Schemas component provides an in band mechanism for communicating XML schemas required for validating an XML document.

Below follows a list of the sub-components that MAY be present within this component:

- The Schema element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in the core specification in section 3.1.14.

**Non-normative Comment:**

Note: It is recommended to use `xml:id` as defined in [xml:id] as id in the payload being referenced by a `<ds:Reference>`, because the schema then does not have to be supplied for identifying the ID attributes.

#### 3.1.30.2 XML Syntax

The XML type `SchemasType` SHALL implement the requirements defined in the Schemas component.

The `SchemasType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SchemasType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="Schema" type="dss:DocumentType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of `SchemasType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.30.3 JSON Syntax

The `SchemasType` JSON object SHALL implement in JSON syntax the requirements defined in the Schemas component.

The `SchemasType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SchemasType": {
  "type": "object",
  "properties": {
    "schema": {
      "type": "array",
      "items": {
        "type": "string",
        "format": "uri"}
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of Schemas component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>schema</td>
</tr>
</tbody>
</table>

### 3.1.31 Component UpdateSignatureInstruction

#### 3.1.31.1 Semantics

The UpdateSignatureInstruction component can be used an optional input for both signing and verification requests and defines the type timestamp that should to be applied.

Below follows a list of the sub-components that MAY be present within this component:

- The optional Type element MUST contain one instance of a URI. The Type element indicates what type of timestamp to apply. Profiles that use this optional input MUST define the allowed values, and the default value, for the Type element (unless only a single type of timestamp is supported, in which case the Type attribute can be omitted).

#### 3.1.31.2 XML Syntax

The XML type UpdateSignatureInstructionType SHALL implement the requirements defined in the UpdateSignatureInstruction component.

The UpdateSignatureInstructionType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="UpdateSignatureInstructionType">
  <xs:attribute name="Type" type="xs:anyURI" use="optional"/>
</xs:complexType>
```

Each child element of UpdateSignatureInstructionType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.31.3 JSON Syntax

The UpdateSignatureInstructionType JSON object SHALL implement in JSON syntax the requirements defined in the UpdateSignatureInstruction component.

The UpdateSignatureInstructionType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.
Properties in the JSON schema above SHALL implement sub-component of
UpdateSignatureInstruction component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>type</td>
</tr>
</tbody>
</table>

3.1.32 Component IntendedAudience

3.1.32.1 Semantics

The IntendedAudience element tells the server who the target audience of this signature is. The server MAY use this to parameterize any aspect of its processing (for example, the server MAY choose to sign with a key that it knows a particular recipient trusts).

Below follows a list of the sub-components that MAY be present within this component:

- The Recipient element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in section 3.2.3.

3.1.32.2 XML Syntax

The XML type IntendedAudienceType SHALL implement the requirements defined in the IntendedAudience component.

The IntendedAudienceType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="IntendedAudienceType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="Recipient" type="saml2:NameIDType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of IntendedAudienceType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.32.3 JSON Syntax

The IntendedAudienceType JSON object SHALL implement in JSON syntax the requirements defined in the IntendedAudience component.
The `IntendedAudienceType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-IntendedAudienceType": {
  "type": "object",
  "properties": {
    "recipient": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/saml2-NameIDType"
      }
    }
  },
  "required": ["recipient"]
}
```

Properties in the JSON schema above SHALL implement sub-component of `IntendedAudience` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipient</td>
<td>recipient</td>
</tr>
</tbody>
</table>

### 3.1.33 Component KeySelector

#### 3.1.33.1 Semantics

The `KeySelector` component holds data that selects a specific key or certificate or group of certificates. Only one of its sub-components MUST be present. But a `KeySelector` component can occur multiple times as a sub-component in the `OptionalInputsSign` component.

Below follows a list of the sub-components that MAY be present within this component:

- The `X509Digest` element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in the core specification in section 3.1.34.
- The `X509SubjectName` element MUST contain one instance of a string. The `X509SubjectName` element contains an X.509 subject distinguished name that SHOULD be represented as a string that complies with section 3 of RFC4514 [LDAP-DN].
- The `X509SKI` element MUST contain one instance of base64 encoded binary data. The `X509SKI` element contains the base64 encoded plain (i.e. non-DER-encoded) value of a X509 V.3 SubjectKeyIdentifier extension.
- The `X509Certificate` element MUST contain one instance of base64 encoded binary data. The `X509Certificate` element contains a base64-encoded [X509V3] certificate.
- The `KeyName` element MUST contain one instance of a string. It selects a key to be used for signing in a generic way. Usually the client knows about the valid values for `KeyName`.

#### 3.1.33.2 XML Syntax

The XML type `KeySelectorType` SHALL implement the requirements defined in the KeySelector component.
The `KeySelectorType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="KeySelectorType">
  <xs:choice>
    <xs:element name="X509Digest" type="dss:X509DigestType"/>
    <xs:element name="X509SubjectName" type="xs:string"/>
    <xs:element name="X509SKI" type="xs:base64Binary"/>
    <xs:element name="X509Certificate" type="xs:base64Binary"/>
    <xs:element name="KeyName" type="xs:string"/>
  </xs:choice>
</xs:complexType>
```

Each child element of `KeySelectorType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.33.3 JSON Syntax

The `KeySelectorType` JSON object SHALL implement in JSON syntax the requirements defined in the `KeySelector` component.

The `KeySelectorType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-KeySelectorType": {
  "type": "object",
  "properties": {
    "x509Digest": {
      "type": "object",
      "$ref": "/definitions/dss-X509DigestType"
    },
    "subject": {
      "type": "string"
    },
    "ski": {
      "type": "string"
    },
    "cert": {
      "type": "string"
    },
    "name": {
      "type": "string"
    }
  },
  "minProperties": 1,
  "maxProperties": 1
}
```

Properties in the JSON schema above SHALL implement sub-component of `KeySelector` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>X509Digest</td>
<td>x509Digest</td>
</tr>
<tr>
<td>X509SubjectName</td>
<td>subject</td>
</tr>
</tbody>
</table>
3.1.34 Component X509Digest

3.1.34.1 Semantics

The X509Digest component contains a base64-encoded digest of a certificate. The digest algorithm URI is identified with a required Algorithm element. The input to the digest MUST be the raw octets that would be base64-encoded of a X509Certificate.

Below follows a list of the sub-components that MAY be present within this component:

- The value element MUST contain one instance of base64 encoded binary data.
- The Algorithm element MUST contain one instance of a string. The string describes the digest algorithm in an appropriate way for the server side processing. Depending on the signature format this may be an OID (e.g. ‘2.16.840.1.101.3.4.2.1’), an URI (e.g. ‘http://www.w3.org/2001/04/xmlenc#sha256’) or a descriptive string (‘SHA-256’).

3.1.34.2 XML Syntax

The XML type X509DigestType SHALL implement the requirements defined in the X509Digest component.

The X509DigestType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="X509DigestType">
  <xs:simpleContent>
    <xs:extension base="xs:base64Binary">
      <xs:attribute name="Algorithm" type="xs:string" use="required"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

Each child element of X509DigestType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name. The element 'value' holding the base64 content is represented by the component's XML tag text content.

3.1.34.3 JSON Syntax

The X509DigestType JSON object SHALL implement in JSON syntax the requirements defined in the X509Digest component.

The X509DigestType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-X509DigestType": {
  "type": "object",
  "properties": {
    "value": {
    
```

<table>
<thead>
<tr>
<th>X509SKI</th>
<th>ski</th>
</tr>
</thead>
<tbody>
<tr>
<td>X509Certificate</td>
<td>cert</td>
</tr>
<tr>
<td>KeyName</td>
<td>name</td>
</tr>
</tbody>
</table>
Properties in the JSON schema above SHALL implement sub-component of X509Digest component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Algorithm</td>
<td>algo</td>
</tr>
</tbody>
</table>

### 3.135 Component PropertiesHolder

#### 3.135.1 Semantics

The PropertiesHolder component is used to request that the server add certain signed or unsigned properties (aka "signature attributes") into the signature. The client can send the server a particular value to use for each property, or leave the value up to the server to determine. The server can add additional properties, even if these aren’t requested by the client.

Below follows a list of the sub-components that MAY be present within this component:

- The optional SignedProperties element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.36. These properties will be covered by the signature.
- The optional UnsignedProperties element MUST contain a sub-component. A given element MUST satisfy the requirements specified in the core specification in section 3.1.36. These properties will **not** be covered by the signature.

#### 3.135.2 XML Syntax

The XML type PropertiesHolderType SHALL implement the requirements defined in the PropertiesHolder component.

The PropertiesHolderType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="PropertiesHolderType">
  <xs:sequence>
    <xs:element minOccurs="0" name="SignedProperties" type="dss:PropertiesType"/>
    <xs:element minOccurs="0" name="UnsignedProperties" type="dss:PropertiesType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of PropertiesHolderType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.
### 3.1.35.3 JSON Syntax

The `PropertiesHolderType` JSON object SHALL implement in JSON syntax the requirements defined in the `PropertiesHolder` component.

The `PropertiesHolderType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-PropertiesHolderType": {  
  "type": "object",  
  "properties": {  
    "signedProps": {  
      "type": "object",  
      "$ref": "/definitions/dss-PropertiesType"  
    },  
    "unsignedProps": {  
      "type": "object",  
      "$ref": "/definitions/dss-PropertiesType"  
    }  
  }  
}
```

Properties in the JSON schema above SHALL implement sub-component of `PropertiesHolder` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignedProperties</td>
<td>signedProps</td>
</tr>
<tr>
<td>UnsignedProperties</td>
<td>unsignedProps</td>
</tr>
</tbody>
</table>

### 3.1.36 Component Properties

#### 3.1.36.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:

- The `Property` element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in the core specification in section 3.1.37.

#### 3.1.36.2 XML Syntax

The XML type `PropertiesType` SHALL implement the requirements defined in the `Properties` component.

The `PropertiesType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="PropertiesType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="Property" type="dss:PropertyType"/>
  </xs:sequence>
</xs:complexType>
```
Each child element of PropertiesType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.36.3 JSON Syntax

The PropertiesType JSON object SHALL implement in JSON syntax the requirements defined in the Properties component.

The PropertiesType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-PropertiesType": {
    "type": "object",
    "properties": {
        "prop": {
            "type": "array",
            "items": {
                "type": "object",
                "$ref": "#/definitions/dss-PropertyType"
            }
        }
    },
    "required": ["prop"]
}
```

Properties in the JSON schema above SHALL implement sub-component of Properties component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property</td>
<td>prop</td>
</tr>
</tbody>
</table>

### 3.1.37 Component Property

#### 3.1.37.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:

- The Identifier element MUST contain one instance of a string.
- The optional Value element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.1.2. The Value element contains arbitrary content wrapped in an Fehler! Verweisquelle konnte nicht gefunden werden.

#### 3.1.37.2 XML Syntax

The XML type PropertyType SHALL implement the requirements defined in the Property component. The PropertyType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.
Each child element of `PropertyType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name. Therefore it occurs in the XML schema, too.

### 3.1.37.3 JSON Syntax

The `PropertyType` JSON object SHALL implement in JSON syntax the requirements defined in the `Property` component.

The `PropertyType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-PropertyType": {  
  "type": "object",  
  "properties": {  
    "id": {  
      "type": "string"  
    },  
    "value": {  
      "type": "object",  
      "$ref": "/definitions/dsb-AnyType"  
    }  
  },  
  "required": ["id"]
}
```

Properties in the JSON schema above SHALL implement sub-component of `Property` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td><code>id</code></td>
</tr>
<tr>
<td>Value</td>
<td><code>value</code></td>
</tr>
</tbody>
</table>

### 3.1.38 Component IncludeObject

#### 3.1.38.1 Semantics

The `IncludeObject` component is used to request the creation of an XMLSig enveloping signature. Multiple occurrences of this optional input can be present in a single `SignRequest` message. Each occurrence will cause the inclusion of an object inside the signature being created.

Below follows a list of the sub-components that MAY be present within this component:
- The optional `WhichDocument` element MUST contain one instance of a unique identifier reference. This element identifies the input document which will be inserted into the returned signature.

- The optional `hasObjectTagsAndAttributesSet` element MUST contain one instance of a boolean. Its default value is 'false'. If 'true' this element indicates that the Document contains a `<ds:Object>` element which has been prepared ready for direct inclusion in the `<ds:Signature>`.

- The optional `ObjId` element MUST contain one instance of a string. It sets the `Id` attribute on the returned `<ds:Object>`.

- The optional `createReference` element MUST contain one instance of a boolean. Its default value is 'true'. If the `createReference` element is set to false inhibits the creation of the `<ds:Reference>` associated to the RefURI element of the input document referred by the `WhichDocument` element, effectively allowing clients to include `<ds:Object>` elements not covered/protected by the signature being created.

### 3.1.38.2 XML Syntax

The XML type `IncludeObjectType` SHALL implement the requirements defined in the `IncludeObject` component.

The `IncludeObjectType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="IncludeObjectType">
  <xs:attribute name="WhichDocument" type="xs:IDREF"/>
  <xs:attribute default="false" name="hasObjectTagsAndAttributesSet" type="xs:boolean"/>
  <xs:attribute name="ObjId" type="xs:string" use="optional"/>
  <xs:attribute default="true" name="createReference" type="xs:boolean" use="optional"/>
</xs:complexType>
```

Each child element of `IncludeObjectType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.38.3 JSON Syntax

The `IncludeObjectType` JSON object SHALL implement in JSON syntax the requirements defined in the `IncludeObject` component.

The `IncludeObjectType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-IncludeObjectType": {
  "type": "object",
  "properties": {
    "whichDoc": {
      "type": "object",
      "$ref": "/definitions/dss-DocumentBaseType"
    },
    "hasObjectTagsAndAttributesSet": {
      "type": "boolean",
      "default": "false"
    },
    "objId": {
```
"type": "string"
},
"createRef": {
  "type": "boolean",
  "default": "true"
}
}
}

Properties in the JSON schema above SHALL implement sub-component of IncludeObject component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WhichDocument</td>
<td>whichDoc</td>
</tr>
<tr>
<td>hasObjectTagsAndAttributesSet</td>
<td>hasObjectTagsAndAttributesSet</td>
</tr>
<tr>
<td>ObjId</td>
<td>objId</td>
</tr>
<tr>
<td>createReference</td>
<td>createRef</td>
</tr>
</tbody>
</table>

3.1.39 Component SignaturePlacement

3.1.39.1 Semantics

The SignaturePlacement component is used to request the creation of an XMLSig enveloped signature placed within an input document. The resulting document with the enveloped signature is placed in the optional output DocumentWithSignature element. The server places the signature in the document identified using the WhichDocument attribute. In the case of a non-XML input document then the server will return an error unless alternative procedures are defined by a profile or in the server policy for handling such a situation.

Below follows a list of the sub-components that MAY be present within this component:
• **The XPathAfter** element MUST contain one instance of a string. This element holds an XPath expression which identifies an element, inside the XML input document, after which the signature will be inserted.

• **The XPathFirstChildOf** element MUST contain one instance of a string. This element holds an XPath expression which identifies an element, in the XML input document, which the signature will be inserted as the first child of.

• **The optional NsPrefixMapping** element MAY occur zero or more times containing a sub-component. If present, each instance MUST satisfy the requirements specified in section 3.1.1.

• **The optional WhichDocument** element MUST contain one instance of a unique identifier reference. The WhichDocument element identifies the input document which the signature will be inserted into.

• **The optional CreateEnvelopedSignature** element MUST contain one instance of a boolean. Its default value is 'true'. If the CreateEnvelopedSignature element is set to true a reference having an enveloped signature transform is created.

### 3.1.39.2 XML Syntax

The XML type `SignaturePlacementType` SHALL implement the requirements defined in the `SignaturePlacement` component.

The `SignaturePlacementType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SignaturePlacementType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="XPathAfter" type="xs:string"/>
      <xs:element name="XPathFirstChildOf" type="xs:string"/>
    </xs:choice>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="NsPrefixMapping" type="dsb:NsPrefixMappingType"/>
  </xs:sequence>
  <xs:attribute name="WhichDocument" type="xs:IDREF"/>
  <xs:attribute default="true" name="CreateEnvelopedSignature" type="xs:boolean"/>
</xs:complexType>
```

Each child element of `SignaturePlacementType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.39.3 JSON Syntax

The `SignaturePlacementType` JSON object SHALL implement in JSON syntax the requirements defined in the `SignaturePlacement` component.

The `SignaturePlacementType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignaturePlacementType": {
  "type": "object",
  "properties": {
    "XPathAfter": {
      "type": "string"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of SignaturePlacement component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>XPathAfter</td>
<td>XPathAfter</td>
</tr>
<tr>
<td>XPathFirstChildOf</td>
<td>XPathFirstChildOf</td>
</tr>
<tr>
<td>NsPrefixMapping</td>
<td>nsDecl</td>
</tr>
<tr>
<td>WhichDocument</td>
<td>whichDoc</td>
</tr>
<tr>
<td>CreateEnvelopedSignature</td>
<td>createEnvelopedSignature</td>
</tr>
</tbody>
</table>

### 3.1.40 Component DocumentWithSignature

#### 3.1.40.1 Semantics

The DocumentWithSignature component contains a 3.1.14 with the signature inserted as requested with the SignaturePlacement component.

Below follows a list of the sub-components that MAY be present within this component:

- The Document element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in the core specification in section 3.1.14. This contains the input document with a signature inserted in some fashion.

#### 3.1.40.2 XML Syntax

The XML type DocumentWithSignatureType SHALL implement the requirements defined in the DocumentWithSignature component.
The DocumentWithSignatureType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DocumentWithSignatureType">
  <xs:sequence>
    <xs:element name="Document" type="dss:DocumentType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of DocumentWithSignatureType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.40.3 JSON Syntax

The DocumentWithSignatureType JSON object SHALL implement in JSON syntax the requirements defined in the DocumentWithSignature component.

The DocumentWithSignatureType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-DocumentWithSignatureType": {
  "type": "object",
  "properties": {
    "doc": {
      "type": "object",
      "$ref": "#/definitions/dss-DocumentType"
    },
    "required": ["doc"]
  }
}
```

Properties in the JSON schema above SHALL implement sub-component of DocumentWithSignature component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>doc</td>
</tr>
</tbody>
</table>

### 3.1.41 Component SignedReferences

#### 3.1.41.1 Semantics

The SignedReferences component gives the client greater control over how the `<ds:Reference>` elements are formed.

Below follows a list of the sub-components that MAY be present within this component:
• The `SignedReference` element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in the core specification in section 3.1.42.

### 3.1.41.2 XML Syntax

The XML type `SignedReferencesType` SHALL implement the requirements defined in the `SignedReferences` component.

The `SignedReferencesType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SignedReferencesType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" name="SignedReference" type="dss:SignedReferenceType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of `SignedReferencesType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.41.3 JSON Syntax

The `SignedReferencesType` JSON object SHALL implement in JSON syntax the requirements defined in the `SignedReferences` component.

The `SignedReferencesType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SignedReferencesType": {
  "type": "object",
  "properties": {
    "signedRef": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-SignedReferenceType"
      }
    }
  }
},
"required": ["signedRef"]
```

Properties in the JSON schema above SHALL implement sub-component of `SignedReferences` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignedReference</td>
<td>signedRef</td>
</tr>
</tbody>
</table>
3.1.42 Component SignedReference

3.1.42.1 Semantics

Each SignedReference component refers to an input document and allows multiple <ds:Reference> elements to be based on a single input document. Furthermore, the client can request additional transforms to be applied to each <ds:Reference>, and can set each <ds:Reference> element's Id or URI attribute. These aspects of the <ds:Reference> can only be set through the SignedReference component; they cannot be set through the input documents, since they are aspects of the reference to the input document, not the input document itself.

Below follows a list of the sub-components that MAY be present within this component:

- The optional Transforms element MUST contain a sub-component. A given element MUST satisfy the requirements specified in section 3.2.1. The Transforms element requests the server to perform additional transforms on this reference.
- The WhichDocument element MUST contain one instance of a unique identifier reference. This defines which input document this reference refers to.
- The optional RefURI element MUST contain one instance of a URI. If this element is present, the corresponding <ds:Reference> element's URI attribute is set to its value. If it is not present, the URI attribute is omitted in the corresponding <ds:Reference>.
- The optional RefId element MUST contain one instance of a string. This element sets the Id attribute of the corresponding <ds:Reference>.

3.1.42.2 XML Syntax

The XML type SignedReferenceType SHALL implement the requirements defined in the SignedReference component.

The SignedReferenceType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SignedReferenceType">
  <xs:sequence>
    <xs:element minOccurs="0" name="Transforms" type="ds:TransformsType"/>
  </xs:sequence>
  <xs:attribute name="WhichDocument" type="xs:IDREF" use="required"/>
  <xs:attribute name="RefURI" type="xs:anyURI" use="optional"/>
  <xs:attribute name="RefId" type="xs:string" use="optional"/>
</xs:complexType>
```

Each child element of SignedReferenceType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.42.3 JSON Syntax

The SignedReferenceType JSON object SHALL implement in JSON syntax the requirements defined in the SignedReference component.

The SignedReferenceType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
  "dss-SignedReferenceType": { 
    "type": "object",
    "properties": { 
```
Properties in the JSON schema above SHALL implement sub-component of SignedReference component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transforms</td>
<td>transforms</td>
</tr>
<tr>
<td>WhichDocument</td>
<td>whichDoc</td>
</tr>
<tr>
<td>RefURI</td>
<td>refURI</td>
</tr>
<tr>
<td>RefId</td>
<td>refId</td>
</tr>
</tbody>
</table>

### 3.1.43 Component VerifyManifestResults

#### 3.1.43.1 Semantics

The results of verifying individual `<ds:Reference>`'s within a `<ds:Manifest>` are returned in the VerifyManifestResults component.

Below follows a list of the sub-components that MAY be present within this component:

- The `ManifestResult` element MUST occur 1 or more times containing a sub-component. Each instance MUST satisfy the requirements specified in the core specification in section 3.1.44.

#### 3.1.43.2 XML Syntax

The XML type `VerifyManifestResultsType` SHALL implement the requirements defined in the VerifyManifestResults component.

The `VerifyManifestResultsType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="VerifyManifestResultsType">
  <xs:sequence>
  </xs:sequence>
</xs:complexType>
```
Each child element of `VerifyManifestResultsType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.43.3 JSON Syntax

The `VerifyManifestResultsType` JSON object SHALL implement in JSON syntax the requirements defined in the `VerifyManifestResults` component.

The `VerifyManifestResultsType` JSON object SHALL be defined as in JSON Schema file `[JSON SCHEMA FILE NAME]` whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-VerifyManifestResultsType": {  
  "type": "object",  
  "properties": {  
    "signedRef": {  
      "type": "array",  
      "items": {  
        "type": "object",  
        "$ref": "#/definitions/dss-ManifestResultType"  
      }  
    }  
  },  
  "required": ["signedRef"]  
}
```

Properties in the JSON schema above SHALL implement sub-component of `VerifyManifestResults` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManifestResult</td>
<td>signedRef</td>
</tr>
</tbody>
</table>

### 3.1.44 Component ManifestResult

#### 3.1.44.1 Semantics

The `VerifyManifestResults` component is comprised of one or more `ManifestResult` elements. Below follows a list of the sub-components that MAY be present within this component:
• The ReferenceXpath element MUST contain one instance of a string. This element identifies the manifest reference, in the XML signature, to which this result pertains.

• The Status element MUST contain one instance of a URI. Its value is limited to an item of the following set:
  urn:oasis:names:tc:dss:1.0:manifeststatus:Valid
  urn:oasis:names:tc:dss:1.0:manifeststatus:Invalid
This element indicates the manifest validation outcome.

• The optional NsPrefixMapping element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in section 3.1.1.

3.1.44.2 XML Syntax

The XML type ManifestResultType SHALL implement the requirements defined in the ManifestResult component.

The ManifestResultType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="ManifestResultType">
  <xs:sequence>
    <xs:element name="ReferenceXpath" type="xs:string"/>
    <xs:element name="Status">
      <xs:simpleType>
        <xs:restriction base="xs:anyURI">
          <xs:enumeration value="urn:oasis:names:tc:dss:1.0:manifeststatus:Valid"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="NsPrefixMapping" type="dsb:NsPrefixMappingType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of ManifestResultType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.44.3 JSON Syntax

The ManifestResultType JSON object SHALL implement in JSON syntax the requirements defined in the ManifestResult component.

The ManifestResultType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-ManifestResultType": {
  "type": "object",
  "properties": {
    "xPath": {
      "type": "string"
    },
    "status": {
      "type": "string"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of ManifestResult component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReferenceXpath</td>
<td>xPath</td>
</tr>
<tr>
<td>Status</td>
<td>status</td>
</tr>
<tr>
<td>NsPrefixMapping</td>
<td>nsDecl</td>
</tr>
</tbody>
</table>

3.1.45 Component UseVerificationTime

3.1.45.1 Semantics

This UseVerificationTime component instructs the server to attempt to determine the signature’s validity at the specified time, instead of a time determined by the server policy.

Below follows a list of the sub-components that MAY be present within this component:

- The CurrentTime element MUST contain one instance of a boolean. Its default value is ‘false’. This element instructs the server to use its current time (normally the time associated with the server-side request processing).

- The SpecificTime element MUST contain one instance of a date/time value. The SpecificTime element allows the client to manage manually the time instant used in the verification process. It SHOULD be expressed as UTC time (Coordinated Universal Time) to reduce confusion with the local time zone use.

- The optional Base64Content element MUST contain base64 encoded binary data. The Base64Content element allows the provision of additional date/time data.

3.1.45.2 XML Syntax

The XML type UseVerificationTimeType SHALL implement the requirements defined in the UseVerificationTime component.

The UseVerificationTimeType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="UseVerificationTimeType">
    <xs:choice>
        <!-- XML schema definitions here -->
    </xs:choice>
</xs:complexType>"
Each child element of `UseVerificationTimeType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.45.3 JSON Syntax

The `UseVerificationTimeType` JSON object SHALL implement in JSON syntax the requirements defined in the `UseVerificationTime` component.

The `UseVerificationTimeType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-UseVerificationTimeType": {
  "type": "object",
  "properties": {
    "currTime": {
      "type": "boolean",
      "default": "false"
    },
    "specTime": {
      "type": "integer",
      "format": "utc-millisecond"
    },
    "b64Content": {
      "type": "string"
    }
  },
  "minProperties": 1,
  "maxProperties": 1
}
```

Properties in the JSON schema above SHALL implement sub-component of `UseVerificationTime` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CurrentTime</td>
<td>currTime</td>
</tr>
<tr>
<td>SpecificTime</td>
<td>specTime</td>
</tr>
<tr>
<td>Base64Content</td>
<td>b64Content</td>
</tr>
</tbody>
</table>
3.1.46 Component AdditionalTimeInfo

3.1.46.1 Semantics

The AdditionalTimeInfo component contains other time instant(s) relevant in the context of the verification time determination.

Below follows a list of the sub-components that MAY be present within this component:

- The **value** element MUST contain one instance of a date/time value.
- The **Type** element MUST contain one instance of a URI. Its value is limited to an item of the following set:
  - `urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signatureTimestamp`
  - `urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signatureTimemark`
  - `urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signedObjectTimestamp`
  - `urn:oasis:names:tc:dss:1.0:additionaltimeinfo:claimedSigningTime`

The **Type** attribute qualifies the kind of time information included in the response. This specification defines the listed types, whose values MUST satisfy the format defined as `xs:dateTime` and SHOULD be expressed as UTC time (Coordinated Universal Time). Profiles MAY include and define new values for the **Type** attribute.

- The optional **Ref** element MUST contain one instance of a string. It allows to establish references to the source of the time information, and SHOULD be used when there is a need to disambiguate several AdditionalTimeInfo components with the same **Type** attribute.

3.1.46.2 XML Syntax

The XML type `AdditionalTimeInfoType` SHALL implement the requirements defined in the AdditionalTimeInfo component.

The `AdditionalTimeInfoType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="AdditionalTimeInfoType">
  <xs:simpleContent>
    <xs:extension base="xs:dateTime">
      <xs:attribute name="Type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:anyURI">
            <xs:enumeration value="urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signatureTimestamp"/>
            <xs:enumeration value="urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signatureTimemark"/>
            <xs:enumeration value="urn:oasis:names:tc:dss:1.0:additionaltimeinfo:signedObjectTimestamp"/>
            <xs:enumeration value="urn:oasis:names:tc:dss:1.0:additionaltimeinfo:claimedSigningTime"/>
          </xs:restriction>
        </xs:simpleType>
        <xs:attribute name="Ref" type="xs:string" use="optional"/>
      </xs:extension>
    </xs:attribute>
  </xs:simpleContent>
</xs:complexType>
```

Each child element of `AdditionalTimeInfoType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name. The element 'value' holding a date/time is represented by the component's XML tag text content.
3.1.46.3 JSON Syntax

The AdditionalTimeInfoType JSON object SHALL implement in JSON syntax the requirements defined in the AdditionalTimeInfo component.

The AdditionalTimeInfoType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-AdditionalTimeInfoType": {
  "type": "object",
  "properties": {
    "value": {
      "type": "integer",
      "format": "utc-millisecond"
    },
    "type": {
      "type": "string",
      "format": "uri"
    },
    "ref": {
      "type": "string"
    }
  },
  "required": ["type"]
}
```

Properties in the JSON schema above SHALL implement sub-component of AdditionalTimeInfo component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Type</td>
<td>type</td>
</tr>
<tr>
<td>Ref</td>
<td>ref</td>
</tr>
</tbody>
</table>

3.1.47 Component VerificationTimeInfo

3.1.47.1 Semantics

The VerificationTimeInfo component allows the client to obtain the time instant used by the server to validate the signature.

Below follows a list of the sub-components that MAY be present within this component:
• The VerificationTime element MUST contain one instance of a date/time value. This time instant used by the server when verifying the signature. It SHOULD be expressed as UTC time (Coordinated Universal Time) to reduce confusion with the local time zone use.

• The optional AdditionalTimeInfo element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.46. The AdditionalTimeInfo element can contain any other time instant(s) relevant in the context of the verification time determination.

3.1.47.2 XML Syntax

The XML type VerificationTimeInfoType SHALL implement the requirements defined in the VerificationTimeInfo component.

The VerificationTimeInfoType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="VerificationTimeInfoType">
  <xs:sequence>
    <xs:element name="VerificationTime" type="xs:dateTime"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="AdditionalTimeInfo" type="dss:AdditionalTimeInfoType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of VerificationTimeInfoType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.1.47.3 JSON Syntax

The VerificationTimeInfoType JSON object SHALL implement in JSON syntax the requirements defined in the VerificationTimeInfo component.

The VerificationTimeInfoType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-VerificationTimeInfoType": {
  "type": "object",
  "properties": {
    "verificationTime": {
      "type": "integer",
      "format": "utc-millisec"
    },
    "additionalTimeInfo": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-AdditionalTimeInfoType"
      }
    }
  }
},
"required": ["verificationTime"]
```

Properties in the JSON schema above SHALL implement sub-component of VerificationTimeInfo component mapped by names as shown in the table below.
Element | Implementing JSON member name
--- | ---
VerificationTime | verificationTime
AdditionalTimeInfo | additionalTimeInfo

### 3.1.48 Component AdditionalKeyInfo

#### 3.1.48.1 Semantics

The `AdditionalKeyInfo` component provides the server with additional data (such as certificates and CRLs) which it can use to validate the signature.

Below follows a list of the sub-components that MAY be present within this component:

- The `X509CRL` element MUST contain one instance of base64 encoded binary data. In addition to the elements included in component 3.1.33 the `X509CRL` element holds a CRL.

A set of sub-components is inherited from component 3.1.33 and is not repeated here.

#### 3.1.48.2 XML Syntax

The XML type `AdditionalKeyInfoType` SHALL implement the requirements defined in the `AdditionalKeyInfo` component.

The `AdditionalKeyInfoType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="AdditionalKeyInfoType">
  <xs:complexContent>
    <xs:extension base="dss:KeySelectorType">
      <xs:choice>
        <xs:element name="X509CRL" type="xs:base64Binary"/>
      </xs:choice>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Each child element of `AdditionalKeyInfoType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.48.3 JSON Syntax

The `AdditionalKeyInfoType` JSON object SHALL implement in JSON syntax the requirements defined in the `AdditionalKeyInfo` component.

The `AdditionalKeyInfoType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-AdditionalKeyInfoType": {
  "type": "object",
  "properties": {
    "x509Digest": {
      "type": "object",
```

---

Standards Track Draft
Copyright © OASIS Open 2017. All Rights Reserved.
Properties in the JSON schema above SHALL implement sub-component of AdditionalKeyInfo component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>X509Digest</td>
<td>x509Digest</td>
</tr>
<tr>
<td>X509SubjectName</td>
<td>subject</td>
</tr>
<tr>
<td>X509SKI</td>
<td>ski</td>
</tr>
<tr>
<td>X509Certificate</td>
<td>cert</td>
</tr>
<tr>
<td>KeyName</td>
<td>name</td>
</tr>
<tr>
<td>X509CRL</td>
<td>X509CRL</td>
</tr>
</tbody>
</table>

### 3.1.49 Component ProcessingDetails

#### 3.1.49.1 Semantics

The ProcessingDetails component elaborates on what signature verification steps succeeded or failed.

Below follows a list of the sub-components that MAY be present within this component:
The optional `ValidDetail` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.50. The `ValidDetail` element holds verification details that were evaluated and found to be valid.

The optional `IndeterminateDetail` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.50. The `IndeterminateDetail` element holds verification details that could not be evaluated or were evaluated and returned an indeterminate result.

The optional `InvalidDetail` element MAY occur zero or more times containing a sub-component. If present each instance MUST satisfy the requirements specified in the core specification in section 3.1.50. The optional `InvalidDetail` element holds verification details that were evaluated and found to be invalid.

### 3.1.49.2 XML Syntax

The XML type `ProcessingDetailsType` SHALL implement the requirements defined in the `ProcessingDetails` component.

The `ProcessingDetailsType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="ProcessingDetailsType">
  <xs:sequence>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="ValidDetail" type="dss:DetailType"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="IndeterminateDetail" type="dss:DetailType"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="InvalidDetail" type="dss:DetailType"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of `ProcessingDetailsType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.49.3 JSON Syntax

The `ProcessingDetailsType` JSON object SHALL implement in JSON syntax the requirements defined in the `ProcessingDetails` component.

The `ProcessingDetailsType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-ProcessingDetailsType": {
  "type": "object",
  "properties": {
    "valid": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "#/definitions/dss-DetailType"
      }
    },
    "indeterminate": {
      "type": "array",
```
Properties in the JSON schema above SHALL implement sub-component of ProcessingDetails component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValidDetail</td>
<td>valid</td>
</tr>
<tr>
<td>IndeterminateDetail</td>
<td>indeterminate</td>
</tr>
<tr>
<td>InvalidDetail</td>
<td>invalid</td>
</tr>
</tbody>
</table>

### 3.1.50 Component Detail

#### 3.1.50.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:
- The optional **Code** element **MUST** contain a URI. This URI which more precisely specifies why this detail is valid, invalid, or indeterminate. It must be a value defined by some other specification, since this specification defines no values for this element.

- The optional **Message** element **MUST** contain a sub-component. A given element **MUST** satisfy the requirements specified in section 3.1.3. This is a human-readable message which MAY be logged, used for debugging, etc.

- The optional **Base64Content** element **MUST** contain base64 encoded binary data.

- The **Type** element **MUST** contain one instance of a URI. The **Type** URI identifies the detail. It may be a value defined by this specification, or a value defined by some other specification. Multiple detail elements of the same **Type** may appear in a single **ProcessingDetails** component. For example, when a signature contains a certificate chain that certifies the signing key, there may be details of the same **Type** present for each certificate in the chain, describing how each certificate was processed.

**Non-normative Comment:**

Multiple detail elements of the same **Type** may appear in a single **ProcessingDetails**. For example, when a signature contains a certificate chain that certifies the signing key, there may be details of the same **Type** present for each certificate in the chain, describing how each certificate was processed.

### 3.1.50.2 XML Syntax

The XML type **DetailType** **SHALL** implement the requirements defined in the **Detail** component. The **DetailType** XML element **SHALL** be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="DetailType">
  <xs:sequence>
    <xs:element minOccurs="0" name="Code" type="xs:anyURI"/>
    <xs:element minOccurs="0" name="Message" type="dsb:InternationalStringType"/>
    <xs:element maxOccurs="1" minOccurs="0" name="Base64Content" type="xs:base64Binary"/>
  </xs:sequence>
  <xs:attribute name="Type" type="xs:anyURI" use="required"/>
</xs:complexType>
```

Each child element of **DetailType** XML element **SHALL** implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.50.3 JSON Syntax

The **DetailType** JSON object **SHALL** implement in JSON syntax the requirements defined in the **Detail** component. The **DetailType** JSON object **SHALL** be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-DetailType": {
  "type": "object",
  "properties": {
    "code": {
      "type": "string"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of Detail component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code</td>
<td>code</td>
</tr>
<tr>
<td>Message</td>
<td>msg</td>
</tr>
<tr>
<td>Base64Content</td>
<td>b64Content</td>
</tr>
<tr>
<td>Type</td>
<td>type</td>
</tr>
</tbody>
</table>

3.1.51 Component SigningTimeInfo

3.1.51.1 Semantics

This SigningTimeInfo component allows the client to obtain the time instant associated to the signature creation.

Below follows a list of the sub-components that MAY be present within this component:

- The SigningTime element MUST contain one instance of a date/time value. This element returns the time value considered by the server to be the signature creation time.
- The optional SigningTimeBoundaries element MUST contain sub-components. This element returns the trusted time values considered as lower and upper limits for the signing time.
- The optional LowerBoundary element MUST contain a date/time value. The SigningTimeBoundaries element MUST contain at least one of the LowerBoundary or UpperBoundary elements.
- The optional UpperBoundary element MUST contain a date/time value. The SigningTimeBoundaries element MUST contain at least one of the LowerBoundary or UpperBoundary elements.

3.1.51.2 XML Syntax

The XML type SigningTimeInfoType SHALL implement the requirements defined in the SigningTimeInfo component.
The `SigningTimeInfoType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="SigningTimeInfoType">
  <xs:sequence>
    <xs:element name="SigningTime" type="xs:dateTime"/>
    <xs:element minOccurs="0" name="SigningTimeBoundaries">
      <xs:complexType>
        <xs:sequence>
          <xs:element minOccurs="0" name="LowerBoundary" type="xs:dateTime"/>
          <xs:element minOccurs="0" name="UpperBoundary" type="xs:dateTime"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:sequence>
</xs:complexType>
```

Each child element of `SigningTimeInfoType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.51.3 JSON Syntax

The `SigningTimeInfoType` JSON object SHALL implement in JSON syntax the requirements defined in the `SigningTimeInfo component`.

The `SigningTimeInfoType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dss-SigningTimeInfoType": {
  "type": "object",
  "properties": {
    "signingTime": {
      "type": "integer",
      "format": "utc-millisec"
    },
    "signingTimeBounds": {
      "type": "object",
      "$ref": "/definitions/dss-SigningTimeInfoType:SigningTimeBoundaries"
    }
  },
  "required": ["signingTime"]
},
"dss-SigningTimeInfoType:SigningTimeBoundaries": {
  "type": "object",
  "properties": {
    "lowerBound": {
      "type": "integer",
      "format": "utc-millisec"
    },
    "upperBound": {
      "type": "integer",
      "format": "utc-millisec"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of `SigningTimeInfo` component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SigningTime</td>
<td><code>signingTime</code></td>
</tr>
<tr>
<td>SigningTimeBoundaries</td>
<td><code>signingTimeBounds</code></td>
</tr>
<tr>
<td>LowerBoundary</td>
<td><code>lowerBound</code></td>
</tr>
<tr>
<td>UpperBoundary</td>
<td><code>upperBound</code></td>
</tr>
</tbody>
</table>

### 3.1.52 Component UpdatedSignature

#### 3.1.52.1 Semantics

The `UpdatedSignature` component contains the resulting updated signature or timestamp or, in the case of a signature being enveloped in an output document, a pointer to the signature.

Below follows a list of the sub-components that MAY be present within this component:

- The `SignatureObject` element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in the core specification in section 3.1.19. This element contains an updated signature or timestamp.
- The optional `Type` element MUST contain one instance of a URI. The URI defines what “update” was applied to the signature.

#### 3.1.52.2 XML Syntax

The XML type `UpdatedSignatureType` SHALL implement the requirements defined in the `UpdatedSignature` component.

The `UpdatedSignatureType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="UpdatedSignatureType">
  <xs:sequence>
    <xs:element name="SignatureObject" type="dss:SignatureObjectType"/>
  </xs:sequence>
  <xs:attribute name="Type" type="xs:anyURI" use="optional"/>
</xs:complexType>
```

Each child element of `UpdatedSignatureType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

#### 3.1.52.3 JSON Syntax

The `UpdatedSignatureType` JSON object SHALL implement in JSON syntax the requirements defined in the `UpdatedSignature` component.
The UpdatedSignatureType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-UpdatedSignatureType": {
  "type": "object",
  "properties": {
    "sigObj": {
      "type": "object",
      "$ref": "#/definitions/dss-SignatureObjectType"
    },
    "type": {
      "type": "string",
      "format": "uri"
    }
  },
  "required": ["sigObj"
}
```

Properties in the JSON schema above SHALL implement sub-component of UpdatedSignature component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignatureObject</td>
<td>sigObj</td>
</tr>
<tr>
<td>Type</td>
<td>type</td>
</tr>
</tbody>
</table>

### 3.1.53 Component ReturnTransformedDocument

#### 3.1.53.1 Semantics

The ReturnTransformedDocument component instructs the server to return an input document to which the XML signature transforms specified by a particular `<ds:Reference>` have been applied. The `<ds:Reference>` is indicated by the zero-based WhichReference attribute (0 means the first `<ds:Reference>` in the signature, 1 means the second, and so on). Multiple occurrences of this optional input can be present in a single verify request message. Each occurrence will generate a corresponding optional output.

Below follows a list of the sub-components that MAY be present within this component:

- The WhichReference element MUST contain one instance of an integer. To match outputs to inputs, each TransformedDocument will contain a WhichReference attribute which matches the corresponding optional input.

#### 3.1.53.2 XML Syntax

The XML type ReturnTransformedDocumentType SHALL implement the requirements defined in the ReturnTransformedDocument component.

The ReturnTransformedDocumentType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="ReturnTransformedDocumentType">
  <xs:attribute name="WhichReference" type="xs:integer" use="required"/>
</xs:complexType>
```
Each child element of ReturnTransformedDocumentType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.53.3 JSON Syntax

The ReturnTransformedDocumentType JSON object SHALL implement in JSON syntax the requirements defined in the ReturnTransformedDocument component.

The ReturnTransformedDocumentType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-ReturnTransformedDocumentType": {
  "type": "object",
  "properties": {
    "whichRef": {
      "type": "integer"
    }
  },
  "required": ["whichRef"]
}
```

Properties in the JSON schema above SHALL implement sub-component of ReturnTransformedDocument component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>WhichReference</td>
<td>whichRef</td>
</tr>
</tbody>
</table>

### 3.1.54 Component TransformedDocument

#### 3.1.54.1 Semantics

The TransformedDocument component contains a document corresponding to the specified `<ds:Reference>`, after all the transforms in the reference have been applied.

Below follows a list of the sub-components that MAY be present within this component:

- The Document element MUST contain one instance of a sub-component. This element MUST satisfy the requirements specified in the core specification in section 3.1.14. This element contains the transformed document.
- The WhichReference element MUST contain one instance of an integer. To match outputs to inputs, each TransformedDocument will contain a WhichReference element which matches the corresponding optional input.

#### 3.1.54.2 XML Syntax

The XML type TransformedDocumentType SHALL implement the requirements defined in the TransformedDocument component.
The TransformedDocumentType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```
<xs:complexType name="TransformedDocumentType">
  <xs:sequence>
    <xs:element name="Document" type="dss:DocumentType"/>
  </xs:sequence>
  <xs:attribute name="WhichReference" type="xs:integer" use="required"/>
</xs:complexType>
```

Each child element of TransformedDocumentType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.1.54.3 JSON Syntax

The TransformedDocumentType JSON object SHALL implement in JSON syntax the requirements defined in the TransformedDocument component.

The TransformedDocumentType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```
"dss-TransformedDocumentType": {
  "type": "object",
  "properties": {
    "doc": {
      "type": "object",
      "$ref": "#/definitions/dss-DocumentType"
    },
    "whichRef": {
      "type": "integer"
    }
  },
  "required": ["doc", "whichRef"]
}
```

Properties in the JSON schema above SHALL implement sub-component of TransformedDocument component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Document</td>
<td>doc</td>
</tr>
<tr>
<td>WhichReference</td>
<td>whichRef</td>
</tr>
</tbody>
</table>

### 3.2 Referenced Structure Models from other documents

#### 3.2.1 Component Transforms

#### 3.2.1.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:
The Transform element MUST occur 1 or more times containing sub-component. Each instance MUST satisfy the requirements specified in section 3.2.2.

### 3.2.1.2 XML Syntax

The XML element is defined in the XML namespace http://www.w3.org/2000/09/xmldsig#. The XML type TransformsType SHALL implement the requirements defined in the Transforms component.

The TransformsType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<complexType name="TransformsType">
  <sequence>
    <xs:element maxOccurs="unbounded" ref="ds:Transform"/>
  </sequence>
</complexType>
```

Each child element of TransformsType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

### 3.2.1.3 JSON Syntax

The TransformsType JSON object SHALL implement in JSON syntax the requirements defined in the Transforms component.

The TransformsType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dsig-TransformsType": {
  "type": "object",
  "properties": {
    "transform": {
      "type": "array",
      "items": {
        "type": "object",
        "$ref": "/definitions/dsig-TransformType"
      }
    }
  },
  "required": ["transform"]
}
```

Properties in the JSON schema above SHALL implement sub-component of Transforms component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform</td>
<td>transform</td>
</tr>
</tbody>
</table>
3.2.2 Component Transform

3.2.2.1 Semantics

Below follows a list of the sub-components that MAY be present within this component:

- The optional value element MUST contain a string.
- The optional Base64Content element MUST contain base64 encoded binary data.
- The optional XPath element MAY occur zero or more times containing a string.
- The optional NsPrefixMapping element MAY occur zero or more times containing sub-component. If present each instance MUST satisfy the requirements specified in section 3.1.1.
- The Algorithm element MUST contain one instance of a URI.

3.2.2.2 XML Syntax

The XML element is defined in the XML namespace http://www.w3.org/2000/09/xmldsig#. The original definition of this element uses the 'mixed' content attribute. To support non-XML syntax using a common object model the attribute is dropped and a 'value' component is introduced. The XML type TransformType SHALL implement the requirements defined in the Transform component.

The TransformType XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
<xs:complexType name="TransformType">
  <xs:sequence>
    <xs:element maxOccurs="1" minOccurs="0" name="value" type="string"/>
    <xs:element maxOccurs="1" minOccurs="0" name="Base64Content" type="xs:base64Binary"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="XPath" type="string"/>
    <xs:element maxOccurs="unbounded" minOccurs="0" name="NsPrefixMapping" type="dsb:NsPrefixMappingType"/>
    <xs:attribute name="Algorithm" type="xs:anyURI" use="required"/>
  </xs:sequence>
</xs:complexType>
```

Each child element of TransformType XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name.

3.2.2.3 JSON Syntax

The TransformType JSON object SHALL implement in JSON syntax the requirements defined in the Transform component.

The TransformType JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"dsig-TransformType": { 
  "type": "object", 
  "properties": { 
    "value": { 
      "type": "string"
    },
    "b64Content": { 
      "type": "string"
    }
  }
}
```
Properties in the JSON schema above SHALL implement sub-component of Transform component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Base64Content</td>
<td>b64Content</td>
</tr>
<tr>
<td>XPath</td>
<td>xpath</td>
</tr>
<tr>
<td>NsPrefixMapping</td>
<td>nsDecl</td>
</tr>
<tr>
<td>Algorithm</td>
<td>algo</td>
</tr>
</tbody>
</table>

### 3.2.3 Component NameID

#### 3.2.3.1 Semantics

The NameID component is used when an element serves to represent an entity by a string-valued name. Below follows a list of the sub-components that MAY be present within this component:
The `value` element MUST contain one instance of a string. In non-XML representations the `value` element contains the actual identifier.

The optional `Format` element MUST contain one instance of a URI. The `Format` element represents the classification of string-based identifier information.

The optional `SPProvidedID` element MUST contain one instance of a string. The `SPProvidedID` element defines the alternative identifier of the principal most recently set by the service provider or affiliation, if any.

The optional `NameQualifier` element MUST contain one instance of a string. The `NameQualifier` element contains the security or administrative domain that qualifies the name. This attribute provides a means to federate names from disparate user stores without collision.

The optional `SPNameQualifier` element MUST contain one instance of a string. The `SPNameQualifier` element further qualifies a name with the name of a service provider or affiliation of providers. This attribute provides an additional means to federate names on the basis of the relying party or parties.

### 3.2.3.2 XML Syntax

The XML element is defined in the XML namespace `urn:oasis:names:tc:SAML:2.0:assertion`. The `NameIDType` XML element SHALL implement the requirements defined in the `NameID` component.

The `NameIDType` XML element SHALL be defined as in XML Schema file [FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE XSD], and is copied below for information.

```xml
c<complexType name="NameIDType">
  <simpleContent>
    <extension base="string">
      <attributeGroup ref="saml2:IDNameQualifiers"/>
      <xs:attribute name="Format" type="anyURI" use="optional"/>
      <xs:attribute name="SPProvidedID" type="string" use="optional"/>
    </extension>
  </simpleContent>
</complexType>
```

Each child element of `NameIDType` XML element SHALL implement in XML syntax the sub-component that has a name equal to its local name. The element 'value' is represented by the component's XML tag text content.

### 3.2.3.3 JSON Syntax

The `NameIDType` JSON object SHALL implement in JSON syntax the requirements defined in the `NameID` component.

The `NameIDType` JSON object SHALL be defined as in JSON Schema file [JSON SCHEMA FILE NAME] whose location is detailed in clause [CLAUSE FOR LINK TO THE JSON SCHEMA FILE], and is copied below for information.

```json
"saml2-NameIDType": {  
  "type": "object",  
  "properties": {  
    "value": {  
      "type": "string"  
    },  
    "format": {  
      "type": "string"  
    }  
  }  
}
```
Properties in the JSON schema above SHALL implement sub-component of NameID component mapped by names as shown in the table below.

<table>
<thead>
<tr>
<th>Element</th>
<th>Implementing JSON member name</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>value</td>
</tr>
<tr>
<td>Format</td>
<td>format</td>
</tr>
<tr>
<td>SPProvidedID</td>
<td>provId</td>
</tr>
<tr>
<td>NameQualifier</td>
<td>nameQual</td>
</tr>
<tr>
<td>SPNameQualifier</td>
<td>spNameQual</td>
</tr>
</tbody>
</table>

The Base64Data component is a generic holder for arbitrary data. In addition to the data itself it also contains additional elements to qualify the MimeType of the data. It also offers an Id / Reference pair to implement a deduplication strategy, useful especially for bigger data blobs. The content is contained inside the mutually exclusive elements Value or AttRefURI. The Value element or the XML tag’s content MAY be empty. If it is empty, the AttRefURI element MUST point to the components content transferred as an attachment.

There are different standards defined for handling and referencing an attachment. Maybe there will be more to come. Therefore the attachment reference mechanism is somehow generic here.

Note: If MIME is used as encapsulation mechanism, the MIME content-type is available via a MIME header. However, the MIME headers may not be available to implementations and the SOAP 1.2 attachment feature is not restricted to MIME. Further the MIME header is not secured by the AttachmentReference's DigestInfo, which is calculated over the binary attachment data (not including the MIME headers).
4 The Processing Model For Signing

The following process diagram illustrates the major buildings blocks of the processing of a signing request. The sub processes are described in the next chapters.

![Diagram](image)

Figure 2

The workflow splits into the sections for XMLDSig and CMS signature processing. The signature will be selected by the server considering a given SignatureType element of OptionalInputsSign (see section 3.1.24) and its configuration and policies. Profiles may introduce additional signature types and MUST define the adequate processing steps.

If the element AddTimestamp of OptionalInputsSign is set to 'true' the sub-process 'add Timestamp' adds a timestamp to the signature.

The task of building the SignResponse (see section 3.1.18) is shared between all signature formats.

4.1 Processing for XML Signatures

The first sub-process 'process references' of the XML signature creation is the processing of the references. The second sub-process handles the creation of the XML signature. These two sub-processes are described in detail below.

If the element CreateEnvelopedSignature of SignaturePlacement (see section 3.1.39) is set to true the signature will be inserted into the document and location selected by SignaturePlacement.

4.1.1 Sub process ‘process references’

The following process diagram illustrates the processing steps for the assembly of references.

![Diagram](image)

Figure 3

The input documents are read from the Base64Data element of the referred Document component into an octet stream. This data MUST be a well formed XML Document as defined in [XML] section 2.1.
If the optional input `SignedReferences` is present (see section 3.1.41) each `SignedReference` element (see section 3.1.42) controls the creation of a corresponding `<ds:Reference>`. The task 'collect references' handles the `SignedReferences`.

Otherwise there will be a `<ds:Reference>` element for each given input document. The set of transforms and their parameter will be selected by the server. The task 'use default transforms' select this set of `<ds:Reference>`.

Note: Transforms can be applied as a server implementation MAY choose to increase robustness of the Signatures created. These Transforms may reflect idiosyncrasies of different parsers or solve encoding issues or the like. Servers MAY choose not to apply transforms in basic processing and extract the binary data for direct hashing or canonicalize the data directly if certain optional inputs are not to be implemented.

If the element `CreateEnvelopedSignature` of `SignaturePlacement` (see section 3.1.39) is set to `true` the list of transforms will be prepended with an `EnvelopedSignatureTransform` entry. The task 'add EnvelopedSignatureTransform' processes the corresponding `<ds:Reference>`.

The `RefURI` attribute of `<ds:Reference>` element MUST be set to include a "same-document" URI which references either:

- The whole Document containing the signature (by using a `RefURI`="")
- The relevant parts of the Document to be covered/protected by the signature (by using a "same-document" `RefURI` attribute having a value starting with ",", like `RefURI="#some-id"`, `RefURI="#xpointer(/)"`, `RefURI="#xpointer(/DocumentElement/ToBeSignedElement)"` or the like).

If the result of evaluating the expression included in the `RefURI` attribute doesn’t fit in any of the options described above, the server MUST reject the request using a `ResultMajor RequesterError` which MAY be qualified by a `ResultMinor urn:oasis:names:tc:dss:1.0:resultminor:InvalidRefURI`.

This alignment will be performed by the task 'align same-doc references'.

### 4.1.2 Sub process ‘create XML signature’

![Diagram](image)

Figure 4

The first task (‘calculate remaining transforms’) of this section applies the given set of transforms. If a `TransformedData` (see section 3.1.15) element is provided by the client these calculations MUST be respected and just the remaining set of transforms must be processed by the server. The case of a `Document` (see section 3.1.14) as base for a reference processing all transform steps MUST be applied.

Note: As required in [XMLDSIG] if the end result is an XML node set, the server MUST attempt to convert the node set back into an octet stream using Canonical XML [XML-C14N].

The ‘calculate / use given hash’ task computes the digest upon the transformation output. If a `DocumentHash` (see section 3.1.16) element is provided by the client the hash values are used as input for the following steps. The `DocumentHash` MAY contain digests of different algorithms. The server selects the appropriate hash algorithm.
Performing the task ‘build XMLDSig’ the server forms a set of `<ds:Reference>` with the elements and attributes set as follows:

- If the Document has a RefURI attribute, the `<ds:Reference>` element’s URI attribute is set to the value of the RefURI attribute, else this attribute is omitted. A signature MUST NOT be created if more than one RefURI is omitted in the set of input documents and the server MUST report a RequesterError by setting ResultMajor RequesterError qualified by a ResultMinor.
- If the Document has a RefType attribute, the `<ds:Reference>` element’s Type attribute is set to the value of the RefType attribute, else this attribute is omitted.
- The `<ds:DigestMethod>` element is set to the hash method used.
- The `<ds:DigestValue>` element is set to the hash value that is to be calculated as per [XMLSIG].
- The `<ds:Transforms>` element is set to the sequence of transforms applied by the server in step b. This sequence MUST describe the effective transform as a reproducible procedure from parsing until hash.
- References resulting from processing of optional inputs MUST be included. In doing so, the server MAY reflect the ordering of the Document elements.

The server creates an XML signature using these `<ds:Reference>` elements according to the processing rules in [XMLSIG].

The last task ‘insert ds:Object’ handles the creation of an enveloping signature. If one or more optional input elements IncludeObject (see section 3.1.38) are present they will cause the inclusion of an object inside the signature being created.

### 4.1.2.1 XML Signatures Variant Optional Input IncludeObject

An enveloping signature is a signature having `<ds:Object>`s which are referenced by `<ds:Reference>`s having a same-document URI.

For each `<IncludeObject>` the server creates a new `<ds:Object>` containing the document, as identified using the `WhichDocument` element, as its child. This object is carried within the enveloping signature. The ordering of the `<IncludeObject>` optional inputs MAY be ignored by the server.

This `<Document>` MUST include a “same-document” RefURI attribute (having a value starting with “#”) which references either:

- The whole newly-created `<ds:Object>`.
- The relevant parts of the newly-created `<ds:Object>`’s contents to be covered/protected by the signature.

If the result of evaluating the expression included in the RefURI element doesn’t fit in any of the options described above, the server MUST reject the request using a ResultMajor RequesterError which MAY be qualified by a ResultMinor urn:oasis:names:tc:dss:1.0:resultminor:InvalidRefURI

Note: If the server does not support the ordering of `<ds:Object>`, it is recommended either to use ID-based referencing to the `<ds:Object>` (using the client-generated ID included in the ObjId attribute) or to rely on expressions based on `<ds:Object>`’s contents that allow to unambiguously refer to the included object or their relevant parts.

The URI in the RefURI element of this `<Document>` should at least reference the relevant parts of the Object to be included in the calculation for the corresponding reference. Clients MUST generate requests
For each IncludeObject the server MUST carry out the following steps before performing Basic Processing:

1. The server identifies the Document that is to be placed into a <ds:Object> as indicated by the WhichDocument element.
2. The data to be carried in the enveloping signature is extracted and decoded.
3. If the hasObjectTagsAndAttributesSet element is false or not present the server builds the <ds:Object> as follows:
   a. The server generates the new <ds:Object> and sets its Id attribute to the value indicated in ObjId element of the optional input if present.
   b. In the case of the Document pointed at by WhichDocument having Base64Data, <ds:Object>('s) MIME Type is to be set to the value of Base64Data('s) MIME Type value and the Encoding is to be set to http://www.w3.org/TR/xmlschema-2/#base64Binary
4. The server splices the to-be-enveloped documents as <ds:Object>(s) into the <ds:Signature>, which is to be returned.
5. If CreateReference is set to true generate a ds:Reference element referencing the spliced <ds:Object> and exclude this <Document> from the set of <Document>s ready for further processing. Otherwise just exclude this <Document> from the set of <Document>s ready for further processing.

4.2 Processing for CMS Signatures

4.2.1 Sub process ‘process digest’

The following process diagram illustrates the processing steps required to calculate the digest for a CMS signature.

Figure 5

The SignRequest component MUST contain either a single Document (not having RefURI or RefType set) or a single DocumentHash component not having RefURI, RefType, Transforms.

If the InputDocuments (see section 3.1.12) component contains a Document element, the server hashes the octet stream represented by the Document. This is performed by the task ‘calculate digest’. If the InputDocuments (see section 3.1.12) component contains a DocumentHash element (see section 3.1.16), the server uses the hash values as an input for the following steps. The DocumentHash MAY contain digests of different algorithms. The server selects the appropriate hash algorithm.

4.2.2 Sub process ‘create CMS signature’

The following process diagram illustrates the processing steps to create a CMS signature.
If the InputDocuments (see section 3.1.12) component contains a Document element and the IncludeEContent element of the OptionalInputsSign component (see section 3.1.27) is set to true then the task ‘include content’ creates a CMS structure with the document enveloped within the signature. For CMS details in this context please refer to [RFC 3852] sections 5.1 “SignedData Type” and 5.2 “EncapsulatedContentInfo Type”.

Otherwise the resulting signature MUST be detached (aka. external or “without eContent”).

The following task ‘build CMS signature’ builds a SignedData structure containing the SignerInfo computed as follows:

The server forms a SignerInfo structure based on the input document. The components of the SignerInfo are set as follows:

- The digestAlgorithm field is set to the OID value for the hash method that was used in the previous processing step.
- The signedAttributes field’s message-digest attribute contains the hash value that was calculated / provided in previous processing step. Other signedAttributes MAY be added by the server, according to its profile or policy, or according to the Properties optional input (see section 3.1.35).
- The remaining fields (sid, signatureAlgorithm, and signature) are filled in as per a normal CMS signature.

### 4.3 General processing

#### 4.3.1 Sub process ‘add Timestamp’

The following process diagram illustrates the processing steps to insert a timestamp.

The AddTimestamp element (see section 3.1.52) of OptionalInputsSign (see section 3.1.24) indicates that the client wishes the server to embed a timestamp token as a property or attribute of the resultant or the supplied signature. The timestamp token will be applied to the signature value in the case of CMS/PKCS7 signatures or the <ds:SignatureValue> element in the case of XML signatures.
Note: Procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS AdES profile [DSS-AdES-P] defines procedures for generating timestamps over the content which is about to be signed (sometimes called content timestamps), and the DSS Timestamp profile [DSS-TS-P] defines procedures for handling standalone timestamps.

The Type element, if present, indicates what type of timestamp to apply. Profiles that use this optional input MUST define the allowed values, and the default value, for the Type attribute (unless only a single type of timestamp is supported, in which case the Type attribute can be omitted).

Two scenarios for the timestamping of both CMS and XML signatures are supported by this Optional Input. They are as follows:

- Create and embed a timestamp token into the signature being created as part of this SignRequest.
- Create and embed a timestamp token into an existing signature, without verification, which is passed in the InputDocuments element of this SignRequest.

The following subsections specify the use of RFC 3161 timestamps with CMS signatures and the use of XML Timestamps or RFC 3161 timestamps with XML Signature. These subsections address both scenarios.

Note: The server SHOULD not verify the incoming signature before adding the timestamp. If a client wishes that its signatures be verified as a condition of time stamping, the client SHOULD use the AddTimestamp optional input of the Verify protocol.

### 4.3.1.1 Processing for CMS signatures time-stamping

If the MimeType element of the Base64Data component (see section Fehler! Verweisquelle konnte nicht gefunden werden.) is set to 'application/pkcs7-signature' a timestamp token is created and embedded into the existing signature, without verification, which is passed in the InputDocuments component of this SignRequest. Otherwise a timestamp token is created and embedded into the signature being created as part of the processing of this SignRequest.

In both scenarios, the timestamp token created by the server SHALL be created according to [RFC 3161]. The MessageImprint field within the TstInfo structure of the timestamp token will be derived from the signature value of the just-created or incoming signature depending on the scenario. The timestamp SHALL be embedded in the CMS signature as an unsigned attribute with the object identifier (see Appendix A of [RFC 3161]):

```plaintext
{ iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) id-aa(2) 14}
```

The signature and its embedded timestamp is returned in the SignatureObject element of the SignResponse component (see section 3.1.18).

### 4.3.1.2 Processing for XML Timestamps on XML signatures

If the type attribute in the optional input AddTimestamp (see section 3.1.52) is urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken and signature being timestamped is an XML signature, then the XML signature MUST contain <dss:timestamp> as defined in DSS 1.0, section 5.1, placed in a <xades:XMLTimestamp> within a <xades:SignatureTimeStamp> as defined in [XAdES].

The <dss:timestamp> MUST contain <ds:Signature> with at least two <ds:Reference> elements:
• One with the Type attribute set to
  urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken
  and referencing a <ds:Object> element whose content is a <TSTInfo> element.
• The other referencing the <ds:SignatureValue> being timestamped.

The present specification defines a format for XML timestamp tokens. In addition, XAdES defines a
mechanism for incorporating signature timestamps in XML signatures. The present document mandates
that signature timestamps in XML format MUST follow the syntax defined in DSS 1.0, section 5.1 of this
document. These time-stamp tokens MUST be added to XML signatures as specified by XAdES.
The signature and its embedded timestamp SHALL be returned in the <SignatureObject> of the
<SignResponse>.

4.3.1.3 Processing for RFC 3161 Timestamps on XML signatures

If the type attribute in this optional input AddTimestamp (see section 3.1.52) is
urn:ietf:rfc:3161
and signature being timestamped is an XML signature then the XML signature MUST contain an RFC
3161, placed in a <xades:EncapsulatedTimeStamp> within a <xades:SignatureTimeStamp> as
defined in [XAdES].
5 The Processing Model For Verification

A DSS server that verifies XML signatures SHOULD perform the following steps, upon receiving a VerifyRequest. These steps may be changed or overridden by the optional inputs, or by the profile or policy the server is operating under. For more details on multi-signature verification, see section Fehler! Verweisquelle konnte nicht gefunden werden.

The following process diagram illustrates the major buildings blocks of the processing of a verification request. The sub processes are described in the following sub-chapters.

Figure 8

The workflow splits into the sections for XMLDSig and CMS signature processing. The processing path will be selected by the server considering a given SignatureType element of OptionalInputsVerify (see section 3.1.25) and its configuration and policies. Profiles may introduce additional signature types and MUST define the adequate processing steps.

- If the element ReturnTimestampedSignature of OptionalInputsVerify is present, the sub-process 'timestamp Signature' adds a timestamp to the signature.
- If the element ReturnUpdatedSignature of OptionalInputsVerify is 'true' the sub-process 'update Signature' inserts the updated signature into the OptionalOutputsVerify.
5.1 Processing for XML Signatures

5.1.1 Sub process ‘retrieve XML signature’

The server retrieves one or more \(<\text{ds:Signature}\rangle\) objects as follows:

- If the \(\text{SignatureObject}\) is present, the server retrieves either the \(<\text{ds:Signature}\rangle\) that is a child element of the \(\text{SignatureObject}\) (see: Note at the end of this section), or those \(<\text{ds:Signature}\rangle\) objects which are pointed to by the \(\text{SignaturePtr}\) in the \(\text{SignatureObject}\).

- If the \(\text{SignaturePtr}\) points to an input document but not a specific element in that document, the pointed-to input document must be a \(\text{Document}\) element containing XML.

- If the \(\text{SignatureObject}\) is omitted, there MUST be only a single \(\text{Document}\) element. This case is handled as if a \(\text{SignaturePtr}\) pointing to the single \(\text{Document}\) was present: the server will search and find every \(<\text{ds:Signature}\rangle\) element in this input document and verify each \(<\text{ds:Signature}\rangle\) according to the steps below.

5.1.1.1 Multi-Signature Verification

If a client requests verification of an entire input document, either using a \(\text{SignaturePtr}\) without an \(\text{XPath}\) or a missing \(\text{SignaturePtr}\) (see section 5.1 step 1), then the server MUST determine whether the input document contains zero, one, or more than one \(<\text{ds:Signature}\rangle\) elements. If zero, the server should return a ResultMajor code of RequesterError.

If more than one \(<\text{ds:Signature}\rangle\) elements are present, the server MUST either reject the request with a ResultMajor code of RequesterError and a ResultMinor code of NotSupported, or accept the request and try to verify all of the signatures.

If the server accepts the request in the multi-signature case (or if only a single signature is present) and one of the signatures fails to verify, the server should return one of the error codes in section 5.2, reflecting the first error encountered.

If all of the signatures verify correctly, the server should return the Success ResultMajor code and the following ResultMinor code:

\text{urn:oasis:names:tc:dss:1.0:resultminor:ValidMultiSignatures}

Note: These procedures only define procedures for handling of multiple signatures on one input document. The procedures for handling multiple signatures on multiple documents are not defined in this core specification, but however such procedures, along
with any optional elements that may be required, may be defined in profiles of this specification.

Only certain optional inputs and outputs are allowed when performing multi-signature verification. See section 5.3 for details.

5.1.2 Sub process ‘recalculate references’

For each `<ds:Reference>` in the `<ds:Signature>`, the server finds the input document with matching RefURI and RefType values (omitted attributes match omitted attributes).

- If the `<ds:Reference>` uses a same-document URI, the XPointer should be evaluated against the input document the `<ds:Signature>` is contained within, or against the `<ds:Signature>` itself if it is contained within the SignatureObject element.
- The SchemaRef element or optional input Schema of the input document or SignatureObject will be used, if present, to identify ID attributes when evaluating the XPointer expression.
- If the `<ds:Reference>` uses an external URI and the corresponding input document is not present, the server will skip the `<ds:Reference>`, and later return a result code such as ReferencedDocumentNotPresent to indicate this. The RefURI MAY be omitted in at most one of the set of Input documents.
- If the input document is a Document, the server extracts and decodes as described in 4.3.1 Step 1.Fehler! Verweisquelle konnte nicht gefunden werden. (or equivalent step in variants of the basic process as defined in Fehler! Verweisquelle konnte nicht gefunden werden. onwards depending of the form of the input document).
- If the input document is a TransformedData, the server MAY check that the `<ds:Transforms>` (if supplied) match between the TransformedData and the `<ds:Reference>` and then hashes the resultant data object according to `<ds:DigestMethod>`, and MUST check that the result matches `<ds:DigestValue>`.
- If the input document is a DocumentHash, the server MAY check that the `<ds:Transforms>`, `<ds:DigestMethod>` (if supplied) and `<ds:DigestValue>` elements match between the DocumentHash and the `<ds:Reference>`.
- If the combination of RefURI and RefType matches more than one input document all of them MUST be either a TransformedData or a DocumentHash otherwise a RequesterError is issued qualified by result minor of ReferencedDocumentNotPresent. Only one of them is allowed to have a WhichReference value that matches the order of the
<ds:Reference> within the <ds:SignedInfo> in question otherwise a RequesterError is issued qualified by result minor of ReferencedDocumentNotPresent.

### 5.1.3 Sub process ‘verify XML signature’

Figure 11

If one or more timestamps are present on the given signature this / these timestamps MUST be verified. The ‘time of existence’ asserted by the timestamp MAY be used to decide the verification time. For details see section 5.1.3.1 and 5.1.3.2.

The server verifies the validity of the signature at a particular time (i.e. current time, assumed signing time or other time), depending on the server policy. This behavior MAY be altered by using the optional input UseVerificationTime (see section 3.1.47).

If the VerifyManifests element of OptionalInputsVerify (see section 3.1.25) is set to ‘true’ the server validates the manifests in an XML signature. In accordance with [XMLDSIG] section 5.1, DSS Manifest validation does not affect a signature’s core validation. The results of verifying individual <ds:Reference>’s within a <ds:Manifest> are returned in the VerifyManifestResults (see section 3.1.43) within the OptionalOutputsVerify (see section 3.1.28). If the optional input VerifyManifests is set to ‘true’ and the XMLSig core validation succeeds, then the returned ResultMinor is urn:oasis:names:tc:dss:1.0:resultminor:valid:hasManifestResults

In case of a negative XMLSig core validation no attempt is made to verify manifests.

If the signature validates correctly, the server returns one of the first three ResultMinor codes listed in section 5.2, depending on the relationship of the signature to the input documents (not including the relationship of the signature to those XML elements that were resolved through XPointer evaluation; the client will have to inspect those relationships manually). If the signature fails to validate correctly, the server returns some other code; either one defined in section 5.2 of this specification, or one defined by some profile of this specification.

#### 5.1.3.1 Processing for RFC 3161 timestamp tokens on XML Signatures

The present section describes the processing rules for verifying an RFC 3161 timestamp token embedded within an XML signature as an unsigned property. This XML signature may be passed in on a Verify call within the SignatureObject or embedded within a Document’s child.

The server shall verify the timestamp token performing the steps detailed below. If any one of them results in failure, then the timestamp token SHOULD be rejected.
1. Extract the timestamp token embedded in the incoming signature as defined in Fehler! Verweisquelle konnte nicht gefunden werden.

2. Verify that the token's public verification certificate is authorized for time stamping by examining the Extended Key Usage field for the presence of the time stamping OID "1.3.6.1.5.5.7.3.8".

3. Process the signature timestamp as defined in [XAdES] Annex G.2.2.16.1.3.

4. Verify that the public verification certificate conforms to all relevant aspects of the relying-party's policy including algorithm usage, policy OIDs, time accuracy tolerances, and the Nonce value.

5. Set the Result element as appropriate.

   *Minor Error* urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:InvalidSignatureTimestamp MAY be used to indicate that the signature is valid but the timestamp against that signature is invalid.

### 5.1.3.2 Processing for XML timestamp tokens on XML signatures

The present section describes the processing rules for verifying and XML Signature timestamp token embedded within an XML signature using the incorporation mechanisms specified in XAdES (i.e., in the <xades:XMLTimeStamp> <xades:SignatureTimeStamp> element's child). This XML signature may be passed in on a Verify call within the SignatureObject or embedded within a Document's child.

The server shall verify the timestamp token performing the steps detailed below. If any one of them results in failure, then the timestamp token SHOULD be rejected.

1. Extract the timestamp token embedded in the incoming signature as defined in Fehler! Verweisquelle konnte nicht gefunden werden.

2. Verify that the verification key and algorithms used conforms to all relevant aspects of the applicable policy. Should this key come within a public certificate, verify that the certificate conforms to all relevant aspects of the applicable policy including algorithm usage, policy OIDs, and time accuracy tolerances.

3. Verify that the aforementioned verification key is consistent with the ds:SignedInfo/SignatureMethod/@Algorithm attribute value.

4. Verify the timestamp token signature in accordance with the rules defined in [XMLDSIG].

5. Verify that the <ds:SignedInfo> element contains at least two <ds:Reference> elements.

6. Verify that one of the <ds:Reference> elements has its Type attribute set to "urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken". Take this one and proceed as indicated below:

   a. Retrieve the referenced data object. Verify that it references a <ds:Object> element, which in turn envelopes a <TSTInfo> element.

   b. Verify that the <TSTInfo> element has a valid layout as per the present specification.

   c. Extract the digest value and associated algorithm from its <ds:DigestValue> and <ds:DigestMethod> elements respectively.

   d. Recalculate the digest of the retrieved data object as specified by [XMLDSIG] with the digest algorithm indicated in <ds:DigestMethod>, and compare this result with the contents of <ds:DigestValue>.

7. Take each of the other <ds:Reference> elements and for each validate the hash as specified in [XMLDSIG].

8. Check that for one of the <ds:Reference> elements the retrieved data object is actually the <ds:SignatureValue> element and that it contains its digest after canonicalization.

9. Set the Result element as appropriate. Minor Error

   *Minor Error* urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:InvalidSignatureTimestamp MAY be used to indicate that the signature is valid but the timestamp against that signature is invalid.
5.2 Basic Processing for CMS Signatures

A DSS server that verifies CMS signatures SHOULDS perform the following steps, upon receiving a VerifyRequest. These steps may be changed or overridden by the optional inputs, or by the profile or policy the server is operating under.

5.2.1 Sub process ‘retrieve CMS signature’

1. The server retrieves the CMS signature by decoding the Base64Signature child of SignatureObject (see section 3.1.19).
2. The server retrieves the input data. If the CMS signature is detached, there must be a single input document: i.e. a single Document or DocumentHash element. Otherwise, if the CMS signature is enveloping, it contains its own input data and there MUST NOT be any input documents present.
3. The CMS signature and input data are verified in the conventional way (see [RFC 3852] for details).
4. If the signature validates correctly, the server returns the first ResultMinor code listed in section 5.2. If the signature fails to validate correctly, the server returns some other code; either one defined in section 5.2 of this specification, or one defined by some profile of this specification.

5.2.2 Sub process ‘verify CMS signature’

Figure 12

If one or more timestamps are present on the given signature this / these timestamps MUST be verified. The ‘time of existence’ asserted by the timestamp MAY be used to decide the verification time. For details see section 5.2.2.1.

The server verifies the validity of the signature at a particular time (i.e. current time, assumed signing time or other time), depending on the server policy. This behavior MAY be altered by using the optional input UseVerificationTime (see section 3.1.47).

If the signature validates correctly, the server returns one of the first three ResultMinor codes listed in section 5.2. If the signature fails to validate correctly, the server returns some other code; either one defined in section 5.2 of this specification, or one defined by some profile of this specification.
5.2.2.1 Processing for RFC 3161 Timestamp tokens on CMS Signatures.

The present section describes the processing rules for verifying a CMS RFC3161 timestamp token passed in on a Verify call within the SignatureObject of the VerifyRequest element. In the CMS case, since the "signature timestamp" is embedded in the signature as an unsigned attribute, only the time stamped signature is required for verification processing. As such, no additional input is required.

The processing by the server is broken down into the following steps:

1. The signature timestamp is embedded in the incoming signature as an unsigned attribute whose object identifier is 1.2.840.11359.1.9.16.2.14. Extract and verify the timestamp token.
2. Verify that the token's public verification certificate is authorized for time stamping by examining the Extended Key Usage field for the presence of the time stamping OID "1.3.6.1.5.5.7.3.8".
3. Validate that the TstInfo structure has a valid layout as defined in [RFC 3161].
4. Extract the MessageImprint hash value and associated algorithm from the TstInfo structure which will be compared against the hash value derived in the next step.
5. Recalculate the hash of the signature value field of the signature in which the timestamp is embedded.
6. Compare the hash values from the two previous steps, and if they are equivalent, then this timestamp is valid for the signature that was time stamped.
7. Verify that the public verification certificate conforms to all relevant aspects of the relying-party's policy including algorithm usage, policy OIDs, time accuracy tolerances, and the Nonce value.
8. Set the Result element as defined in this specification. Minor Error urn:oasis:names:tc:dss:1.0:resultminor:valid:signature:InvalidSignatureTimestamp MAY be used to indicate that the signature is valid but the timestamp against that signature is invalid.

5.3 General processing

The following steps are shared between all signature types.

5.3.1 Sub process ‘update Signature’

The presence of the ReturnUpdatedSignature element (see section 3.1.31) of OptionalInputsVerify (see section 3.1.25) instructs the server to return an UpdatedSignature (see section 3.1.52) output, containing a new or updated signature.

Figure 13

The Type element of ReturnUpdatedSignature defines the process of “updating” a signature. For example, the updated signature may be the original signature with some additional unsigned signature properties added to it (such as timestamps, counter-signatures, or additional information for use in verification), or the updated signature could be an entirely new signature calculated on the same input documents as the input signature. Profiles that use this optional input MUST define the allowed values and their semantics, and the default value of ReturnUpdatedSignature (unless only a single type of updated signature is supported, in which case the element can be omitted).
Multiple occurrences of this optional input can be present in a single verify request message. If multiple occurrences are present, each occurrence MUST have a different value. Each occurrence will generate a corresponding `UpdatedSignature` optional output. These optional outputs SHALL be distinguishable based on their `Type` element, which will match each output with an input.

A DSS server SHOULD perform the following steps to return the updated signature appropriately. These steps may be changed or overridden by a profile or policy the server is operating under. (e.g. for PDF documents enveloping CMS signatures).

![Diagram](image)

**Figure 14**

- If the detached or enveloping signature to be verified and updated appears within a `Base64Signature` then the `UpdatedSignature` optional output MUST contain the modified `SignatureObject` with the updated signature.
- If the signature to be verified and updated is enveloped, and if the `VerifyRequest` contains a `SignatureObject` with a `SignaturePtr` pointing to an `InputDocument` enveloping the signature then the server MUST produce the following TWO optional outputs, first a `DocumentWithSignature` optional output containing the document that envelopes the updated signature, second an `UpdatedSignature` optional output containing a `SignatureObject` having a `SignaturePtr` element that MUST point to the former `DocumentWithSignature`.
- If there is no `SignatureObject` included in the request then the server MUST produce a `DocumentWithSignature` optional output containing the document with the updated signature, only. No `UpdatedSignature` element will be generated.

If created the `DocumentWithSignature` optional output (for the schema refer to section 3.1.30) contains the input document with the given signature inserted. The server places the signature in the document identified using the `SignatureObject`/`SignaturePtr`/`WhichDocument` element. This `Document` MUST include a same-document `RefURI` element which references the data updated (e.g. of the form `RefURI`).

### 5.3.2 Sub process ‘timestamp Signature’

If the `ReturnTimestampedSignature` element (see section 3.1.31) of `OptionalInputsVerify` (see section 3.1.25) is present the server updates the signature after its verification by embedding a signature timestamp token as an unauthenticated attribute (see "unauthAttrs" in section 9.1 [RFC 3852]) or *unsigned* property (see section 6.2.5 "The UnsignedSignatureProperties element" and section 7.3 "The SignatureTimeStamp element" [XAdES]) of the supplied signature.

The timestamp token will be on the signature value in the case of CMS/PKCS7 signatures or the `<ds:SignatureValue>` element in the case of XML signatures.
The Type element of ReturnTimestampedSignature, if present, indicates what type of timestamp to build. This document defines two values for it, namely:

- urn:ietf:rfc:3161 for generating a RFC 3161 timestamp token on the signature
- urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken, for generating a XML timestamp token as defined in section 5 of this document.

Profiles that use this optional input MUST define the allowed values and the default value for the Type element (unless only a single type of timestamp is supported, in which case Type can be omitted).

The sub process of returning the updated signatures is the same as described in the sub process ‘update Signature’ (see section 5.3.1).

Note: Procedures for handling other forms of timestamp may be defined in profiles of the Core. In particular, the DSS XAdES profile [DSS-XAdES-P] defines procedures for handling timestamps against the document being signed, and the DSS Timestamp profile [DSS-TS-P] defines procedures for handling standalone timestamps.

### 5.3.3 Task ‘build VerifyResponse’

The task of building the VerifyResponse (see section 3.1.22) is shared between all signature formats. The OptionalInputsVerify (see section 3.1.25) element, server configuration and applied policies may affect the set of elements included in the OptionalOutputsVerify (see section 3.1.28).

If the ReturnVerificationTimeInfo element of OptionalInputsVerify (see section 3.1.25) is set to ‘true’ the server returns the VerificationTimeInfo (see section 3.1.47) within the OptionalOutputsVerify (see section 3.1.28). It contains the verification time and optionally other relevant time instants that may have been used when determining the verification time or that may be useful for its qualification.

If the ReturnSigningTimeInfo element of OptionalInputsVerify (see section 3.1.25) is set to ‘true’ the server returns the SigningTimeInfo (see section 3.1.51) within the OptionalOutputsVerify (see section 3.1.28). It allows the client to obtain the time instant associated to the signature creation. Depending on the applicable server policy, this signing time needs to be qualified, in order to avoid unacceptable measurement errors or false claims, using time boundaries associated to trustworthy time values (based on timestamps or time-marks created using trusted time sources). In this case, the server MAY include these values in the LowerBoundary and UpperBoundary elements, respectively.

Criteria for determining when a time instant can be considered trustworthy and for determining the maximum acceptable delays between the signing time and their boundaries (if any) is outside the scope of this specification.
When there's no way for the server to determine the signing time, the server MUST omit the SigningTimeInfo output.

If the ReturnSignerIdentity element of OptionalInputsVerify (see section 3.1.25) is set to 'true' the server returns the SignerIdentity (see section 3.2.3) element within the OptionalOutputsVerify (see section 3.1.28). The SignerIdentity optional output contains an indication of who performed the signature. This option is not allowed in multi-signature verification.

If the ReturnTransformedDocument element (see section 3.1.53) of OptionalInputsVerify (see section 3.1.25) is present the server returns an input document to which the XML signature transforms specified by a particular <ds:Reference> have been applied. The <ds:Reference> is indicated by the zero-based WhichReference element (0 means the first <ds:Reference> in the signature, 1 means the second, and so on). Multiple occurrences of this optional input can be present in a single verify request message. Each occurrence will generate a corresponding optional output. These options are not allowed in multi-signature verification.

The TransformedDocument (see section 3.1.54) element within the OptionalOutputsVerify (see section 3.1.28) contains a document corresponding to the specified <ds:Reference> after all the transforms in the reference have been applied. In other words, the hash value of the returned document should equal the <ds:Reference> element's <ds:DigestValue>. To match outputs to inputs, each TransformedDocument component will contain a WhichReference element which matches the corresponding ReturnTransformedDocument optional input element.

If the ReturnProcessingDetails element of OptionalInputsVerify (see section 3.1.25) is set to 'true' the server returns the ProcessingDetails (see section 3.1.49) element within the OptionalOutputsVerify (see section 3.1.28). The ProcessingDetails element elaborates on what signature verification steps succeeded or failed. This option is not allowed in multi-signature verification.
6 DSS Core Bindings

Mappings from DSS messages into standard communications protocols are called DSS bindings. *Transport bindings* specify how DSS messages are encoded and carried over some lower-level transport protocol. *Security bindings* specify how confidentiality, authentication, and integrity can be achieved for DSS messages in the context of some transport binding.

Below we specify an initial set of bindings for DSS. Future bindings may be introduced by the OASIS DSS TC or by other parties.

6.1 HTTP POST Transport Binding

In this binding, the DSS request/response exchange occurs within an HTTP POST exchange [RFC 2616]. The following rules apply to the HTTP request:

The client may send an HTTP/1.0 or HTTP/1.1 request.

The Request URI may be used to indicate a particular service endpoint.

The Content-Type header MUST be set to “application/xml” or “application/json”.

The Content-Length header MUST be present and correct.

The DSS request message MUST be sent in the body of the HTTP Request.

The following rules apply to the HTTP Response:

The Content-Type header MUST be set to “text/xml” or “application/json”.

The Content-Length header MUST be present and correct.

The DSS response message MUST be sent in the body of the HTTP Response.

The HTTP status code MUST be set to 200 if a DSS response message is returned. Otherwise, the status code can be set to 3xx to indicate a redirection, 4xx to indicate a low-level client error (such as a malformed request), or 5xx to indicate a low-level server error.

6.2 SOAP 1.2 Transport Binding

In this binding, the DSS request/response exchange occurs using the SOAP 1.2 message protocol [SOAP]. The following rules apply to the SOAP request:

A single DSS SignRequest or VerifyRequest element will be transmitted within the body of the SOAP message.

The client MUST NOT include any additional XML elements in the SOAP body.

The UTF-8 character encoding must be used for the SOAP message.

Arbitrary SOAP headers may be present.

The following rules apply to the SOAP response:

The server MUST return either a single DSS SignResponse or VerifyResponse element within the body of the SOAP message, or a SOAP fault code.

The server MUST NOT include any additional XML elements in the SOAP body.

If a DSS server cannot parse a DSS request, or there is some error with the SOAP envelope, the server MUST return a SOAP fault code. Otherwise, a DSS result code should be used to signal errors.

The UTF-8 character encoding must be used for the SOAP message.

Arbitrary SOAP headers may be present.

On receiving a DSS response in a SOAP message, the client MUST NOT send a fault code to the DSS server.
6.3 Security Bindings

It is good practice to use a security binding (e.g. TLS) to provide confidentiality, authentication and integrity. Details regarding protocols and cipher suites are out of scope of this document.
7  JSON Format

Here we place the JSON extended world view on DSS AND_REMOVE__WHEN_FINISHED.

JSON, as described in [RFC7159], defines a text format for serializing structured data. Objects are serialized as an unordered collection of name/value pairs.

JSON does not define any semantics around the name/value pairs that make up an object, nor does it define an extensibility mechanism for adding control information to a payload.

DSS’s JSON format extends JSON by defining general conventions for name/value pairs that annotate a JSON object, property or array. DSS defines a set of canonical annotations for control information such as ids, types, and links, and custom annotations MAY be used to add domain-specific information to the payload.

Annotations are used in JSON to capture control information that cannot be predicted as well as a mechanism to provide values where a computed value would be wrong.
8 XML Format
Here we place the XML world view on DSS AND_REMOVE_THIS_SENTENCE_WHEN_FINISHED.
9  DSS-Defined Identifiers

The following sections define various URI-based identifiers. Where possible an existing URN is used to specify a protocol. In the case of IETF protocols the URN of the most current RFC that specifies the protocol is used (see [RFC 2648]). URI references created specifically for DSS have the following stem: urn:oasis:names:tc:dss:1.0:

9.1 Signature Type Identifiers

The following identifiers MAY be used as the content of the <SignatureType> optional input (see section 3.5.1).

9.1.1 XML Signature

- **URI**: urn:ietf:rfc:3275
  - This refers to an XML signature per [XMLDSIG].

9.1.2 XML TimeStampToken

- **URI**: urn:oasis:names:tc:dss:1.0:core:schema:XMLTimeStampToken
  - This refers to an XML timestamp containing an XML signature, per section 5.1.

9.1.3 RFC 3161 TimeStampToken

- **URI**: urn:ietf:rfc:3161
  - This refers to an XML timestamp containing an ASN.1 TimeStampToken, per [RFC 3161].

9.1.4 CMS Signature

- **URI**: urn:ietf:rfc:3369
  - This refers to a CMS signature per [RFC 3852] or prior versions of CMS.

9.1.5 PGP Signature

- **URI**: urn:ietf:rfc:2440
  - This refers to a PGP signature per [RFC 2440].
10. Conformance

10.1 Conformance as a DSS version 2.0 document

To ease communication and subsequent resolution of any specific partial conformance violation, the preceding chapters already provide minimal requirements, that a specific instance component must fulfill, to permit conformance of the complete DSS version 2.0 document.

10.1.1 Conformance for XML format

The following clause offers a simple three step process, to either prove or disprove the conformance of a complete XML document (formulated in terms specific to that implementation language) to this version of DSS:

\( \n\) An XML document instance conforms to this specification as a DSS document if it meets all of the following three conditions:

1. Is well-formed XML.
2. Consists of a single dss:whatever element instance as defined in the namespace http://docs.oasis-open.org/dss-x/ns/dss-core/v2.0/dss.
3. Is valid XML.

\( \Delta [DSS-5.1.1-1] \)

10.1.2 Conformance for JSON format

The following clause offers a simple COUNT_ME step process, to either prove or disprove the conformance of a complete JSON document (formulated in terms specific to that implementation language) to this version of DSS:

\( \n\) A JSON document instance conforms to this specification as a DSS document if it meets all of the following COUNT_ME conditions:

1. Is valid JSON
2. Other COUNT_ME minus 1 criteria …
3. 

\( \Delta [DSS-5.1.2-1] \)
Appendix A. Acknowledgments

The following individuals were members of the OASIS DSS-X Technical Committee during the creation of this specification and their contributions are gratefully acknowledged:

Andreas Kuehne, Individual
Andreas Reiter, A-SIT, Zentrum fuer sichere Informationstechnologie Austria
Chet Ensign, OASIS
Detlef Huehnlein, Individual
Ernst Jan van Nigtevecht, Sonnenglanz Consulting
Ezer Farhi, DocuSign, Inc.
Herbert Leitold, A-SIT, Zentrum fuer sichere Informationstechnologie Austria
Juan Cruellas, Departamento de Arquitectura de Computadores, Univ Politecnica de Cataluna
Mark Klamerus, Individual
Michael Yatsko, DocuSign, Inc.
Pim van der Eijk, Sonnenglanz Consulting
Robin Cover, OASIS
Stefan Hagen, Individual
Appendix B. Table of Types, Elements and Attributes

Es wurden keine Einträge für das Inhaltsverzeichnis gefunden.
## Appendix C. List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>15</td>
</tr>
<tr>
<td>Figure 2</td>
<td>113</td>
</tr>
<tr>
<td>Figure 3</td>
<td>113</td>
</tr>
<tr>
<td>Figure 4</td>
<td>114</td>
</tr>
<tr>
<td>Figure 5</td>
<td>116</td>
</tr>
<tr>
<td>Figure 6</td>
<td>117</td>
</tr>
<tr>
<td>Figure 7</td>
<td>117</td>
</tr>
<tr>
<td>Figure 8</td>
<td>120</td>
</tr>
<tr>
<td>Figure 9</td>
<td>121</td>
</tr>
<tr>
<td>Figure 10</td>
<td>122</td>
</tr>
<tr>
<td>Figure 11</td>
<td>123</td>
</tr>
<tr>
<td>Figure 12</td>
<td>125</td>
</tr>
<tr>
<td>Figure 13</td>
<td>126</td>
</tr>
<tr>
<td>Figure 14</td>
<td>127</td>
</tr>
<tr>
<td>Figure 15</td>
<td>128</td>
</tr>
</tbody>
</table>
Appendix D. Index

DateTime  13
Appendix E. JSON Helpers

Here we may offer guidance on helping to make the DSS world look even more JSONesque.
## Appendix F. Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Editor</th>
<th>Changes Made</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Rev number]</td>
<td>[Rev Date]</td>
<td>Andreas Kuehne and Stefan Hagen</td>
<td>Initial Draft version with feedback from the TC</td>
</tr>
</tbody>
</table>