Report Exselt features of implementation defined features from the spec

Implementation defined feature (appendix F)	Support level
1. If the initialization of any global variables or parameter depends on the context item, a dynamic error can occur if the context item is absent. It is implementation-defined whether this error occurs during priming of the stylesheet or subsequently when the variable is referenced; and it is implementation-defined whether the error occurs at all if the variable or parameter is never referenced. (See *2.3.2 Priming a Stylesheet*)	Error occurs when variable is referenced and used, except for some special cases (as when a param value is provided as an expression)
2. The way in which an XSLT processor is invoked, and the way in which values are supplied for the source document, starting node, stylesheet parameters, and base output URI, are implementation-defined. (See *2.3.2 Priming a Stylesheet*)	Available through API, commandline args and configuration file
3. The way in which a base output URI is established is implementation-defined (See *2.3.6 Post-processing the Raw Result*)	Through API
4. The mechanisms for creating new extension instructions and extension functions are implementation-defined. (See *2.8 Extensibility*)	Through API
5. It is implementation-defined whether type errors are signaled statically. (See *2.11 Error Handling*)	Some are raised statically, some not, depending on whether we can find the static type statically
6. It is implementation-defined how a package is located given its name and version, and which version of a package is chosen if several are available. (See *3.6.2 Dependencies between Packages*)	Through API, and using a package-catalog XML file
7. Mechanisms to locate the source or executable code of a package are implementation-defined. (See *3.6.2 Dependencies between Packages*)	Through API and using a package-catalog XML file

8. When XQuery functions and variables are used from XSLT, it is implementation-defined how any differences between XSLT and XQuery semantics are handled; it is implementation-defined whether XQuery code is evaluated within the same execution scope[FO30] as XSLT code; and it is implementation-defined whether node identity is preserved across the interface. The effect of calling XQuery functions that are updateing or nondeterministic is also implementation-defined. (See *3.6.7 Using an XQuery Library Package*)	XQuery is not supported
9. In the absence of an [xsl:]default-collation attribute, the default collation may be set by the calling application in an implementation-defined way. (See *3.8.1 The default-collation Attribute*)	Not yet supported, but will be made available through API
10. The set of namespaces that are specially recognized by the implementation (for example, for user-defined data elements, and extension attributes) is implementation-defined. (See *3.8.3 User-defined Data Elements*)	The Exselt extension namespace is automatically recognized for Exselt extension functions
11. The effect of user-defined data elements whose name is in a namespace recognized by the implementation is implementation-defined. (See *3.8.3 User-defined Data Elements*)	User defined data elements are not given any special semantics
12. If the effective version of any element in the stylesheet is not 1.0 or 2.0 but is less than 3.0, the recommended action is to report a static error; however, processors may recognize such values and process the element in an implementation-defined way. (See *3.10 Backwards Compatible Processing*)	We currently don't treat it as a static error, but this will likely change for versions like 0.9, 2.1 etc.
13. It is implementation-defined whether an XSLT 3.0 processor supports backwards compatible behavior for any XSLT version earlier than XSLT 3.0. (See 3.10 Backwards Compatible Processing)	We support 1.0 backwards compatibility behavior. Also 2.0, but the spec defines no differences
14. The way in which the URI reference appearing in an xsl:include or xsl:import declaration is used to locate a representation of a stylesheet module, and the way in which the stylesheet module is constructed from that representation, are implementation-defined. In particular, it is implementation-defined which URI schemes are supported, whether fragment identifiers are supported, and what media types are supported. (See *3.12.1 Locating Stylesheet Modules*)	Through API. Fragment identifiers are supported and media types are recognized.

15. It is implementation-defined what forms of URI reference are acceptable in the href attribute of the xsl:include and xsl:import elements, for example, the URI schemes that may be used, the forms of fragment identifier that may be used, and the media types that are supported. (See *3.12.1 Locating Stylesheet Modules*)	Duplicate, see above
16. An implementation may define mechanisms, above and beyond xsl:import-schema that allow schema components such as type definitions to be made available within a stylesheet. (See *3.15 Built-in Types*)	Not supported explicitly, but potentially available through static type context API.
17. It is implementation-defined which versions and editions of XML and XML Namespaces (1.0 and/or 1.1) are supported. (See 4.1 XML Versions)	We support 1.1 for serialization only, unsupported for reading, but planned.
18. Limits on the value space of primitive datatypes, where not fixed by [XML Schema Part 2], are implementation-defined. (See *4.7 Limits*)	Dates: negative and high positive years supported (limited by Int32) for most operations. Precision is in the range of xs:decimal fractional part. Integer is 64 bit. Decimal is 128 bits, strings are 2^32 max.
19. The set of statically known documents[XP30] is implementation-defined. (See *5.4.1 Initializing the Static Context*)	Defaults to empty set, or configurable.
20. The set of statically known collections[XP30] is implementation-defined. (See *5.4.1 Initializing the Static Context*)	Defaults to empty set, or configurable.
21. The statically known default collection type[XP30] is implementation-defined. (See *5.4.1 Initializing the Static Context*)	We do not allow overriding this type, it defaults to node()*
22. Implementations may provide user options that relax the requirement for the doc[FO30] and collection[FO30] functions (and therefore, by implication, the document function) to return stable results. The manner in which such user options are provided, if at all, is implementation-defined. (See *5.4.3 Initializing the Dynamic Context*)	Defaults to stable. We do not allow this to be configurable (but this may change).

23. The implicit timezone for a transformation is implementation-defined. (See 5.4.3.2 Other Components of the XPath Dynamic Context)	Through API
24. The default collection[XP30] is implementation-defined. (See *5.4.3.2 Other Components of the XPath Dynamic Context*)	The spec uses the term **default node collection**. Defaults to empty set. Can be set through API
25. The availability of dynamic context information within extension functions is implementation-defined. (See *5.4.4 Additional Dynamic Context Components used by XSLT*)	The dynamic context is available when writing extension functions.
26. The default values for the warning-on-no-match and warning-on-multiple-match attributes of xsl:mode are implementation-defined. (See *6.6.1 Declaring Modes*)	Defaults to "no", cannot be overridden
27. The form of any warnings output when there is no matching template rule, or when there are multiple matching template rules, is implementation-defined. (See *6.6.1 Declaring Modes*)	Can be configured through ErrorListener or with the result of the transform (from commandline, to stderr)
28. Streamed processing may be initiated by invoking the transformation with an initial mode declared as streamable, while supplying the initial match selection (in an implementation-defined way) as a streamed document. (See *6.6.4 Streamable Templates*)	Through API, commandline, or automatically, if the initial mode is streamable
29. The mechanism by which the caller supplies a value for a stylesheet parameter is implementation-defined. (See *9.5 Global Variables and Parameters*)	Through API, config file, or commandline, can also be an expression
30. The set of extension functions available in the static context for the target expression of xsl:evaluate is implementation-defined. (See *10.4.1 Static context for the target expression*)	Through API, but not user-configurable yet. By default, ext functions are allowed
31. If an xml:id attribute that has not been subjected to attribute value normalization is copied from a source tree to a result tree, it is implementation-defined whether attribute value normalization will be applied during the copy process. (See 11.9.1 Shallow Copy)	The attribute is normalized, this is not overridable.

32. The numbering sequences supported by the xsl:number instructions, beyond those defined in this specification, are implementation-defined. (See *12.4 Number to String Conversion Attributes*)	Most unicode number sequences are supported. Configurable through API
33. There may be implementation-defined upper bounds on the numbers that can be formatted by xsl:number using any particular numbering sequence. (See *12.4 Number to String Conversion Attributes*)	Limited to size of the used datatype. No override available.
34. The set of languages for which numbering is supported by xsl:number, and the method of choosing a default language, are implementation-defined. (See *12.4 Number to String Conversion Attributes*)	International numbering is supported for some languages.
35. With xsl:number, it is implementation-defined what combinations of values of the format token, the language, and the ordinal attribute are supported. (See *12.4 Number to String Conversion Attributes*)	Limits can be interdependent, not easy to summarize.
36. If the data-type attribute of the xsl:sort element has a value other than text or number, the effect is implementation-defined. (See *13.1.2 Comparing Sort Key Values*)	We try to atomize, if that fails, we raise an error
37. The facilities for defining collations and allocating URIs to identify them are largely implementation-defined. (See 13.1.3 Sorting Using Collations)	Through API. Parameters on the URI can be used to set up collation.
38. The algorithm used by xsl:sort to locate a collation, given the values of the lang and case-order attributes, is implementation-defined. (See *13.1.3 Sorting Using Collations*)	The relevant collation is found by a lookup mechanism
39. If none of the collation, lang, or case-order attributes is present (on xsl:sort), the collation is chosen in an implementation-defined way. (See *13.1.3 Sorting Using Collations*)	The default is the Unicode collation. It will use the default-collation attribute. This is also configurable by the API
40. When using the family of URIs that invoke the Unicode Collation Algorithm, the effect of supplying a query keyword or value not defined in this specification is implementation-defined. The defaults for query keywords are also implementation-defined unless otherwise stated. (See *13.4 The Unicode Collation Algorithm*)	The collations in this section are not yet fully supported, so we don't know the defaults yet either

41. The posture and sweep of an extension instruction are implementation-defined. (See *19.8.4.2 Streamability of extension instructions*)	We plan to allow extension instructions to be streamable, but this is not yet implemented
42. The posture and sweep of a call to an extension function are implementation-defined. (See *19.8.7.13 Streamability of Function Calls*)	We plan to allow extension functions to be streamable by the new categories, but this is not yet implemented
43. The posture and sweep of a NamedFunctionRef referring to an extension function are implementation-defined. (See *19.8.7.14 Streamability of Named Function References*)	Depends on resolution of previous point
44. The set of media types recognized by the processor, for the purpose of interpreting fragment identifiers in URI references passed to the document function, is implementation-defined. (See *20.1 fn:document*)	By default this is limited by built-in .NET functionality of the UriResolver used by XmlReader, this resolver can be overridden.
45. The values returned by the system-property function, and the names of the additional properties that are recognized, are implementation-defined. (See *20.3.4 fn:system-property*)	We do not yet have other properties, but this will likely change.
46. The destination and formatting of messages written using the xsl:message instruction are implementation-defined. (See *22.1 Messages*)	Are treated as events, which can be rerouted by MessageEventContainer (API)
47. The detail of any external mechanism allowing a processor to disable checking of assertions is implementation-defined. (See *22.2 Assertions*)	We allow assertions to be switched on/off from settings or commandline or config file
48. This specification does not define any mechanism for creating or binding implementations of extension instructions or extension functions, and it is not required that implementations support any such mechanism. Such mechanisms, if they exist, are implementation-defined. (See *23 Extensibility and Fallback*)	Unknown at this point

49. The effect of an extension function returning a string containing characters that are not permitted in XML is implementation-defined. (See 23.1.2 Calling Extension Functions)	(currently) non-XML characters are not allowed
50. The way in which external objects are represented in the type system is implementation-defined. (See 23.1.3 External Objects)	(currently) unsupported types to be returned is not allowed
51. The way in which the results of the transformation are delivered to an application is implementation-defined. (See 24 Transformation Results)	Several ways are available through API, commandline and config file, including streaming output (in development).
52. There may be implementation-defined restrictions on the form of absolute URI that may be used in the href attribute of the xsl:result-document instruction. (See *24.1 Creating Secondary Results*)	It is limited to the UriResolver and this can be overridden by the user.
53. Implementations may provide additional mechanisms allowing users to define the way in which final result trees are processed. (See *24.1 Creating Secondary Results*)	Similar to item 51
54. If serialization is supported, then the location to which a final result tree is serialized is implementation-defined, subject to the constraint that relative URI references used to reference one tree from another remain valid. (See *25 Serialization*)	Similar to item 51
55. The default value of the encoding attribute of the xsl:output element is implementation-defined. (See *25 Serialization*)	Defaults to UTF-8
56. It is implementation-defined which versions of XML, HTML, and XHTML are supported in the version attribute of the xsl:output declaration. (See *25 Serialization*)	XML 1.0/1.1, HTML 4.0 and 5.0, XHTML 1.0 and 1.1 (but xhtml 1.1 is limited)
57. The default value of the byte-order-mark serialization parameter is implementation-defined in the case of UTF-8 encoding. (See 25 Serialization)	Defaults to YES
58. It is implementation-defined whether, and under what circumstances, disabling output escaping is supported. (See 25.2 Disabling Output Escaping)	Not supported, and no plans to support it

59. It is implementation-defined whether (and if so how) an XSLT 3.0 processor is able to work with versions of XPath later than XPath 3.0. (See *26 Conformance*)	Some functions are supported, but not yet implemented
60. It is implementation-defined whether (and if so how) an XSLT 3.0 processor is able to work with versions of [XSLT and XQuery Serialization] later than 3.0. (See *26.3 Serialization Feature*)	Not supported yet.