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*)	See API definitions.
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*)	See API definitions.
3. The way in which a base output URI is established is implementation-defined (See *2.3.6 Post-processing the Raw Result*)	See API definitions.
4. The mechanisms for creating new extension instructions and extension functions are implementation-defined. (See *2.8 Extensibility*)	See http://www.saxonica.com/documentation/#!extensibility
5. It is implementation-defined whether type errors are signaled statically. (See *2.11 Error Handling*)	Type errors are signalled statically where possible if the inferred static type of the supplied value is disjoint with the required type.
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*)	TBA. We have not completed design work in this area.
7. Mechanisms to locate the source or executable code of a package are implementation-defined. (See *3.6.2 Dependencies between Packages*)	TBA. We have not completed design work in this area.
8. When XQuery functions and variables are used from XSLT, it is implementation-defined how any differences between XSLT and XQuery	Saxon treats imported XQuery functions and variables as closely as possible to XSLT functions and global variables. Node identity is

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*)	preserved, and execution is within the same evaluation scope. Calling functions with side-effects is permitted, but the effects are not entirely predictable because the optimizer does not treat them specially.
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*)	See API definitions.
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*)	Saxon defines one namespace for Saxon extensions, and allows users to register other namespaces via its API.
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*)	Current Saxon releases attach no semantics to any user-defined data element.
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*)	Current behaviour (9.6) allows any version number. This may change.
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*)	XSLT 1.0 backwards compatible behaviour is supported.
14. The way in which the URI reference appearing in an xsl:include or xsl:import declaration is used to locate a representation of	This can be configured by supplying a user-written URIResolver. The behaviour of the default URIResolver is described

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*)	(not very well) in the product documentation.
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?
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*)	There are no such mechanisms.
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*)	This is configurable.
18. Limits on the value space of primitive datatypes, where not fixed by [XML Schema Part 2], are implementation-defined. (See *4.7 Limits*)	For xs:decimal: defined by Java BigDecimal class. For dates and times: the year is a 32-bit int; precision is in microseconds. For durations: the number months and the number of microseconds are integers. For strings, the length is a 32-bit int. For sequences, the size is a 32-bit int.
19. The set of statically known documents*XP30* is implementation-defined. (See *5.4.1 Initializing the Static Context*)	The empty set.
20. The set of statically known collections*XP30* is implementation-defined. (See *5.4.1 Initializing the Static Context*)	The empty set.
21. The statically known default collection type*XP30* is implementation-defined. (See *5.4.1 Initializing the Static*	None (i.e. node()*)

Context)	
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*)	Such mechanisms are provided by configuration options; in the case of collections, also by a query parameter on the collection URI.
23. The implicit timezone for a transformation is implementation-defined. (See *5.4.3.2 Other Components of the XPath Dynamic Context*)	Derived from the system clock; overridable using the product API.
24. The default collection*XP30* is implementation-defined. (See *5.4.3.2 Other Components of the XPath Dynamic Context*)	See API definitions.
25. The availability of dynamic context information within extension functions is implementation-defined. (See *5.4.4 Additional Dynamic Context Components used by XSLT*)	Context information is made available.
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*)	The defaults are no/yes respectively, but can be configured.
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*)	Warnings are notified to a user-supplied ErrorListener.
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*)	See API definitions.
29. The mechanism by which the caller supplies a value for a stylesheet parameter is implementation-defined. (See *9.5 Global Variables and Parameters*)	See API definitions.
30. The set of extension functions	See API definitions. All 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*)	available for static calls are also available for use within xsl:evaluate.
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*)	Attribute value normalization is applied.
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*)	Varies by product edition; additional numbering sequences can be supplied by users or third parties. Documentation not readily available.
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*)	Documentation not readily 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*)	Varies by product edition. In Saxon-EE, the set of languages supported is the set supported by the ICU-J library.
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*)	Documentation not readily available.
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*)	Any other value is 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*)	See API definitions. Collations can be implemented by users and given arbitrary URIs.

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 lang and case-order attributes are used to select a Java locale, and the default collation for that Java locale is used.
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 Unicode codepoint collation.
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*)	Need to check.
41. The posture and sweep of an extension instruction are implementation-defined. (See *19.8.4.2 Streamability of extension instructions*)	Extension instructions are roaming and free-ranging.
42. The posture and sweep of a call to an extension function are implementation-defined. (See *19.8.7.13 Streamability of Function Calls*)	Extension functions are roaming and free-ranging.
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*)	Extension functions are roaming and free-ranging.
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*)	Fragment identifiers are resolved by the (user-supplied) URIResolver. The default URIResolver interprets fragments as XML ID values, regardless of media type.
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*)	See http://www.saxonica.com/documentation/#!functions/fn/system-property (which could be improved)

46. The destination and formatting of messages written using the xsl:message instruction are implementation-defined. (See *22.1 Messages*)	See API definitions.
47. The detail of any external mechanism allowing a processor to disable checking of assertions is implementation-defined. (See *22.2 Assertions*)	No such mechanism is provided.
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*)	See http://www.saxonica.com/documentation/#!extensibility
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*)	In general, the returned string is not checked. Invalid characters may or may not cause a problem in downstream processing.
50. The way in which external objects are represented in the type system is implementation-defined. (See *23.1.3 External Objects*)	External objects are represented as an additional kind of Item, on the same level as atomic values, nodes, or functions.
51. The way in which the results of the transformation are delivered to an application is implementation-defined. (See *24 Transformation Results*)	See API definitions.
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*)	In the absence of a user-supplied OutputURIResolver, the only URIs accepted are file system URIs.
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*)	See API definitions.
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	See API definitions, in particular the OutputURIResolver interface.

	URI references used to reference one tree from another remain valid. (See *25 Serialization*)	
55.	The default value of the encoding attribute of the xsl:output element is implementation-defined. (See *25 Serialization*)	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 or 1.1, HTML 4.0 or 5, XHTML 1.0.
57.	The default value of the byte-order-mark serialization parameter is implementation-defined in the case of UTF-8 encoding. (See *25 Serialization*)	Default is "no".
58.	It is implementation-defined whether, and under what circumstances, disabling output escaping is supported. (See *25.2 Disabling Output Escaping*)	The attribute is supported provided the transformation result is being sent to a Saxon serializer.
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*)	TBA.
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*)	TBA.