
XACML-Based Web Services Policy Constraint Language (WS-PolicyConstraints)

Working Draft 06, 24 October 2005

Authors:

Anne Anderson (anne.anderson@sun.com)
Balasubramanian Devaraj (bala.devaraj@sun.com)

Abstract:

This specification describes a generic language – *WS-PolicyConstraints* - for expressing one component of a web services policy: the policy constraints or assertions on individual policy variables, or “vocabulary items”. Examples of policy constraints include specifying an acceptable value for price or encryption mechanism, a range of acceptable values for bandwidth or time of day, or a set of acceptable values for IP address or security token types. *WS-PolicyConstraints* is suitable for standardization, as it can be used by any domain vocabulary. The vocabulary items themselves may be arbitrary XML schema elements, SAML Attributes, or XACML Attributes, and are defined by other standards groups, consortia, enterprises, etc. outside the scope of this specification.

In addition to specifying the language for expressing constraints, this specification also describes how to use *WS-PolicyConstraints* to verify that a specific set of vocabulary items conforms to a set of constraints, and to compute the intersection of constraints placed on the same vocabulary items in two different policies. An intersection might be needed to determine whether a consumer and a provider, each with their own policies, have any mutually compatible sets of policy constraints.

WS-PolicyConstraints is designed to be used with another policy framework language that specifies acceptable combinations of constrained vocabulary items. By using *WS-PolicyConstraints* together with a policy framework language, policy processors are provided with all the information required to verify a set of actual values against any policy, or to match or intersect any two policies, without needing any vocabulary-specific code modules or configuration information. This means policy processors do not need to be modified in order to handle policies that use new vocabulary schemas or updates to existing schemas, even if those vocabulary schemas are themselves proprietary.

Status:

This version of the specification is a working draft.

34 Table of Contents

35	1 Introduction (non-normative).....	3
36	1.1 Policy architecture.....	4
37	1.2 Policies.....	4
38	1.3 Policy example.....	5
39	1.4 Policy vocabularies.....	5
40	1.5 Policy statements.....	6
41	1.6 Specification of policy constraints.....	7
42	1.7 Intersection of policy constraints and policies.....	9
43	1.8 Policy merging and overwriting.....	10
44	1.9 Translation of vocabulary specifications.....	10
45	1.10 Use models.....	10
46	1.11 Benefits of standardizing policy constraints.....	12
47	1.12 Related work.....	13
48	1.13 Terminology.....	14
49	1.14 Notation.....	15
50	2 Specification of constraints (normative).....	16
51	2.1 Vocabulary item references.....	16
52	2.2 Literal values.....	17
53	2.3 Constraints.....	17
54	2.4 Constraint examples (non-normative).....	18
55	3 Validation of vocabulary instances (normative).....	20
56	3.1 Verification of a constraint.....	20
57	4 Intersection of policies (normative).....	21
58	4.1 Incompatible constraints.....	21
59	4.2 Intersection of constraints.....	21
60	5 New constraint functions (normative).....	22
61	5.1 Presence/absence functions.....	22
62	5.2 [namespace:]limit-scope.....	22
63	5.3 [namespace:]ipAddress-match.....	23
64	5.4 [namespace:]string-getKrb5SName.....	23
65	5.5 [namespace:]hexBinary-getCertExtensionValue.....	23
66	5.6 [namespace:]string-to-uri.....	23
67	6 Supported constraint functions and their intersections (normative).....	25
68	6.1 Intersection with semantic hierarchies.....	26
69	7 Preferences (normative).....	28
70	8 References.....	29
71	8.1 Normative references.....	29
72	8.2 Non-normative references.....	29
73	Example of policy normalization (non-normative).....	31
74	Acknowledgments.....	34
75	Revision History.....	35
76	Notices.....	36

1 Introduction (*non-normative*)

This specification describes the syntax and semantics of a language – *WS-PolicyConstraints* - for specifying, verifying, intersecting, combining, and applying **constraints** on the discrete elements, or **vocabulary items**, included in a **policy**. *WS-PolicyConstraints* is designed particularly for web services, but is also suitable for other types of **policies**. *WS-PolicyConstraints* is based on the OASIS Standard *eXtensible Access Control Markup Language* [XACML] and draws particularly from the *XACML profile for Web-services* [WSPL].

WS-PolicyConstraints allows **constraints** or “Assertions” to be expressed over domain-dependent **vocabulary items** using a domain-independent language.

A small example will help illustrate the functions of *WS-PolicyConstraints*. Consider a web service that uses a particular schema such as *WS-Security* [WSS] to specify message security headers. One element in this header might be a specification of the length of the encryption key that is used. This “encryption key length” is one **vocabulary item** in the **vocabulary** (the schema) used in the security header to describe message security characteristics. The service may require that encryption key lengths must be greater than or equal to some minimum length considered secure, but less than or equal to some maximum length determined by limits on lengths the service's encryption software is designed to support. *WS-PolicyConstraints* can be used by the service to express the range of key length values that the service will accept. It can be used by a potential client to express one key length that the client wants to use, or a range of key lengths that the client is able to use. Given the service's expression of the range of values that the service will accept and a particular message security header, *WS-PolicyConstraints* can be used to verify that the key length specified in the header satisfies the requirements of the service. Given the service's expression of the key lengths that it will accept, and a potential client's expression of the key lengths it is able or willing to use, *WS-PolicyConstraints* can be used by a service broker to determine that the service and the client have a mutually acceptable range of key length values. A service deployer can use *WS-PolicyConstraints* to describe the range of key length values that the service is to support in a particular deployment (out of a possibly larger range that the service software is capable of supporting).

Specifying **constraints** on **vocabulary items** is only one component of a web service **policy**. Web services **policy** can be described using four functional layers:

1. **Vocabulary specification layer:** this layer specifies the syntax and semantics of discrete elements, or **vocabulary items**, to be used in some particular type of **policy**. Examples of such specifications are *WS-Security*[WSS] and *WS-Reliability*[WSRM]. Some **vocabulary items** will be instances of particular elements in the domain language itself, but in many cases, additional **vocabulary items** will need to be defined to express the **policies** needed by that domain. The *Web Ontology Language* [OWL] attempts to provide a generic language for expressing semantics at this level suitable for use with any **vocabulary**.
2. **Policy constraints specification layer:** this layer specifies sets of values, or constraints, for individual **vocabulary items** in a **policy**. Examples of such specifications are *WS-PolicyConstraints* and, in theory, the *Web Ontology Language* [OWL].
3. **Policy framework layer:** this layer specifies which combinations of **constrained vocabulary items** are acceptable for a **policy**; i.e. It expresses various interdependent combinations of **constrained vocabulary items** that are acceptable. Examples of such specifications are *WS-Policy* [WSPOLICY] and a proposed extension called *Compositors* [COMPOSITORS] to the Features and Properties section of the *Web Services Description Language (WSDL) Version 2.0* [WSDL].
4. **Bindings specification layer:** this layer specifies which policies are bound to which aspects of a service. Examples of such specifications are *WS-PolicyAttachments* [WSPA] and the Features and Properties section of WSDL 2.0 [WSDL].

Satisfactory solutions are already in use for the **vocabulary** layer, and new ones appear regularly. Several solutions have been proposed for the **policy framework** and **bindings** layers. To date,

127 however, **policy** designers have either left the expression and interpretation of **policy constraints** to
 128 custom software for supporting each type of **vocabulary**, or have proposed use of a semantic web
 129 language such as OWL [OWL]. No specific OWL functions have been developed for expressing the
 130 types of **constraints** required for web service **policies**, however, and **policy** implementers have not
 131 chosen to use semantic web approaches to specifying **policies**. *WS-PolicyConstraints* is a proposed
 132 alternative that specifies a rich, but policy-targeted set of functions that can be used with existing
 133 **vocabulary** specifications or can be used to express new **vocabularies** in a representation-independent
 134 way.

135 The concepts and use model for *WS-PolicyConstraints* are described in detail in the following sections of
 136 this Introduction.

137 1.1 Policy architecture

138 The **policy** layers mentioned above, with the addition of **vocabulary translation** and **service metadata**
 139 layers, are diagrammed in Table 1 . The **vocabulary translation** layer is needed in the case where the
 140 **policy constraints** do not accept the native syntax used by the **vocabulary**. The **service metadata**
 141 layer is where **policy** information is made available to other clients or services.

Layer Specifications				Layer Descriptions
WS-Security	WS-Reliability	Vocabulary Specification: Defines vocabulary items used to characterize a service or aspect of a service. These are the policy variables.
XSLT	XPath	[pass-through]		Vocabulary Translation: Translates existing vocabulary syntax to the syntax used by the Policy Constraints Layer if necessary
OWL		[part of vocabulary specification]	Vocabulary Semantics: Defines domain-specific semantics of the vocabulary	
		WS-PolicyConstraints	Policy Constraints: Defines constraints (generic semantics) on individual vocabulary items and how to compute the intersection of such constraints	
WSDL Compositors	WS-Policy			Policy Framework: Defines policy sets , which are acceptable combinations or sets of policy constraints and how to compute intersection and composition of such combinations; constraints themselves are opaque to this layer.
WSDL F&P	WS-PolicyAttachments			Policy Bindings: Binds policies to service metadata descriptions
WSDL	ebXML Reg/Rep	SOAP	WS-MetadataExchange	Service Metadata: Makes policies available to clients or other services

Table 1 Policy architecture diagram

142 1.2 Policies

143 A **policy** is a description of behavior, particularly of desired or intended behavior. We are familiar with
 144 **policies** that govern corporate behavior, such as equal employment **policies** or privacy **policies**, but
 145 computer applications also have behaviors, and these can also be described using **policies**. One
 146 example familiar to most of us is an “access control **policy**”: a description of how an application should
 147 behave with respect to allowing access to resources. But this is only one example from a broad range of
 148 computer application behaviors that could be described using a **policy**. Such **policies** describe the
 149 **constraints** that determine the application's behavior. Some application **constraints** are determined by

150 inherent capabilities; for example, an application either has the code required to support a given
151 message encryption mechanism or not. Other application **constraints** are determined by decisions that
152 constrain how the inherent capabilities are to be used; for example, the deployer of a service application
153 may have enabled only one of two built-in encryption mechanisms. Any **policy** that describes the
154 behavior of a computer application must take into account both the decisions and the inherent
155 capabilities that constrain the application's behavior.

156 In the web services arena, **policies** may be associated with a entire set of web services, with a single
157 web service as a whole, or with particular parts of a service such as ports or endpoints, bindings, port
158 types or interfaces, operations, messages, or even with fractions of messages. A **policy** may be
159 associated with a generic type of service or with a specific instance of a service or one of its sub-parts. It
160 is the job of the **Policy Bindings** Layer to associate **policies** with services or parts of services. WSDL
161 [WSDL] (using Features and Properties) and *WS-PolicyAttachments* [WSPA] provide ways to associate
162 **policies** with services. WSDL [WSDL] itself, UDDI [UDDI], SOAP [SOAP], and *WS-MetadataExchange*
163 [WSMD] all provide ways in which **policies** and other service metadata are made available to service
164 clients or to other services.

165 **1.3 Policy example**

166 An example of a web services **policy** for message security headers follows. This example will be used
167 in the subsequent sections.

168 The security token type must always be either a KerberosV5TGT or an X509v3 certificate. If the
169 message part being transmitted is "AccountNumber", then the 3DES-CBC encryption algorithm
170 must be used; otherwise encryption is optional but must be 3DES-CBC if used.

171 **1.4 Policy vocabularies**

172 A **policy** uses one or more particular **vocabularies** to specify behaviors. In the world of corporate
173 **policies**, the **vocabulary** for privacy **policies** might include **items** such as "data category", "user
174 category", "third party", "consent to disclose", "mailing list", "record retention period", etc. In the world of
175 web services, as used by this specification, a **vocabulary** is an independent set of technical features and
176 parameters, called **vocabulary items**, that are associated with some aspect of using a web service. For
177 example, the **vocabulary** in the policy in Section 1.3 includes **vocabulary items** such as "security token
178 type", "message part", and "encryption algorithm". "KerberosV5TGT" and "X509v3" are possible values
179 for the "security token type" **vocabulary item**, "AccountNumber" is a value for the "message part"
180 **vocabulary item**, and "3DES-CBC" is a value for the "encryption algorithm" **vocabulary item**.

181 As further examples, a **vocabulary** associated with reliable messaging might include **items** such as
182 "time to live", "number of retries" and "interval between retries". A **vocabulary** associated with services
183 that offer computational resources might include **items** such as "gigaflops", "number of processors
184 available", and "platform architecture". Items that might occur in many **vocabularies** include "offer
185 price", "minimum bandwidth", "date of service", "time of service request", "client security token type", and
186 "client IP address".

187 **Vocabularies** for web services are defined in written specifications, of which there are several
188 categories. Some specifications describe capabilities associated with almost any web service.
189 Examples of this type of specification include *WS-Reliability*[WSRM] and *WS-Security*[WSS]. Such
190 specifications are usually created by cross-industry standards organizations. Other specifications
191 describe capabilities common to a particular class of services, such as services offering computational
192 resources or services offering music downloads. These types of specifications are usually created by
193 standards groups within a specific industry. Yet other specifications describe capabilities associated with
194 a particular service. These types of specifications may be created by an individual vendor or other
195 enterprise, and may be proprietary or internal to the enterprise.

196 **Vocabulary items** may be expressed in various forms. Web service specifications are usually
197 associated with an XML Schema, and messages will contain instances of nodes defined in that schema.
198 If the **policy** needs to place constraints on the value of such a node, the node itself can be used as a

199 **vocabulary item** by specifying an XPath expression that selects that node. For example, if the web
200 service application schema includes a `<Price>` element, then constraints may be placed on the text
201 node associated with that element by specifying the **vocabulary item** as an XPath expression of the
202 form `“../Price/text()”`. With such a **vocabulary item**, the **policy constraint** may be used to
203 directly verify that a message containing a `<Price>` element, for example, conforms to the **policy**.
204 **Vocabulary items** may also be expressed as XACML [XACML] or SAML [SAML] Attributes, or may be
205 converted to one of these from some other format. Such Attributes can be used for **vocabulary items**
206 that are not reflected in the content of messages, or **items** that are too difficult to select using an XPath
207 expression. Each domain should specify the **vocabulary items** to be used in **policies** for that domain.

208 A **policy** may use **vocabulary items** from multiple **vocabularies**. As an example, the **policy** may
209 specify various reliable messaging parameters using a *WS-Reliability vocabulary*, but may qualify these
210 by using environmental **vocabulary items** such as time of day specified using an XACML `“current-
211 time”` Attribute.

212 The semantics, or meanings, of **vocabulary items** are usually expressed in written form as part of the
213 **vocabulary** specification. Semantic web languages such as OWL [OWL] and RDF [RDF] attempt to
214 provide a standard computer-readable description of the semantics of a **vocabulary**, but currently most
215 semantic descriptions are captured only in custom code modules.

216 Note that, conceptually, **vocabulary items** need not be independent. For example, **item** “age” may be
217 related to another **item** “birthdate”, “ship date” may be related to another item “order date”. Requiring
218 **vocabulary items** to be independent, however, eliminates the possibility of various types of errors in
219 **policies**, such as inconsistent circular dependencies (A > B AND B > C AND C > A). More research is
220 needed to identify the types of dependencies that can be supported without compromising the ability to
221 compute **policy** intersections.

222 1.5 Policy statements

223 A **policy statement** is a computer-readable expression used to convey a **policy** between entities. An
224 application deployer, for example, may use a **policy statement** to convey to the application itself the
225 **constraints** that are to be placed on the application's inherent capabilities. In another example, a
226 service provider application may use a **policy statement** to convey the **constraints** of the application to
227 potential service consumers. In yet another example, a service consumer client application may use a
228 **policy statement** to convey the **constraints** that its user requires of a service to a service broker or to a
229 service itself.

230 **Policies** are not always in the form of explicit **policy statements**. An application's **policies** may be
231 hard-coded, or may be set via some application- or platform-specific set of deployment parameters. It
232 should be possible, however, to capture this information in the form of a **policy statement** if needed.

233 In the past, web service specifications have had to define their own **policy statement** syntax. Greater
234 functionality and interoperability, as well as smaller footprints and easier management of services, is
235 possible if the syntax for **policy statements** is specified in a standard way. Standard **policy statements**
236 can be useful even if not all web service components use them; for example, a client application can be
237 written to understand standard **policy statements** exported by services the client will use even if the
238 client's own **policies** are stored in an internal format – having one format to deal with makes the client
239 easier to design, code, and maintain.

240 One layer that can be defined to help express standard **policy statements** is a way of describing the
241 combinations of predicates, or **constraints**, on **vocabulary items** that are acceptable for a given **policy**.
242 There are various ways of expressing this aspect of the example **policy** from Section 1.3, but they are all
243 equivalent to the following set of sets, which are expressed in **disjunctive normal form** (an OR of
244 ANDed simple predicates):

```
245     { { “SecurityTokenType” EQUALS “KerberosV5TGT”,  
246         “MessagePart” EQUALS “AccountNumber”,  
247         “EncryptionAlgorithm” EQUALS “3DES-CBC” },
```

```

248
249     { "SecurityTokenType" EQUALS X509v3,
250       "MessagePart" EQUALS "AccountNumber",
251       "EncryptionAlgorithm" EQUALS "3DES-CBC" },
252
253     { "SecurityTokenType" EQUALS "KerberosV5TGT",
254       "MessagePart" NOT EQUAL TO "AccountNumber",
255       "EncryptionAlgorithm" EQUALS "3DES-CBC"},
256
257     { "SecurityTokenType" EQUALS "X509v3",
258       "MessagePart" NOT EQUAL TO "AccountNumber",
259       "EncryptionAlgorithm" EQUALS "3DES-CBC"},
260
261     { "SecurityTokenType" EQUALS "KerberosV5TGT",
262       "MessagePart" NOT EQUAL TO "AccountNumber",
263       "EncryptionAlgorithm" NOT PRESENT },
264
265     { "SecurityTokenType" EQUALS "X509v3",
266       "MessagePart" NOT EQUAL TO "AccountNumber",
267       "EncryptionAlgorithm" NOT PRESENT } }

```

268 The **policy** is satisfied if at least one of these sets of **constraints** is satisfied, so this collection is called
269 the **policy sets**. The layer used to express **policy sets** is called the **policy framework**. In addition to
270 specifying the **policy sets**, the **policy framework** layer should also provide a way to express relative
271 preferences among the various sets. If the language allows **policies** to be expressed in a form other
272 than **disjunctive normal form**, the language should provide an algorithm for obtaining the **policy sets** in
273 normal form, since this is the best form for automatic matching of **policies**.

274 At least two proposals have been made for languages that address the **policy framework** layer: *Web*
275 *Services Description Language 2.0* [WSDL] Features and Properties and *WS-Policy* [WSPOLICY] Both
276 provide operators to create expressions equivalent to the **policy sets** above, but neither proposal
277 currently includes the specification of preferences.

278 Neither proposal provides ways to represent the EQUALS operator, nor do they provide ways to
279 represent other possible predicate operators such as GREATER THAN, or ELEMENT OF <subset>. So
280 these proposals, and the **policy framework** layer, can be characterized as describing the various sets of
281 **constraints** that are acceptable for a **policy**, but as not describing the **constraints** themselves.

282 In order to make the management of **policies** tractable, a **policy** usually will consist of multiple sets of
283 **policy sets**, each associated with a different aspect of communication or with a different service
284 interface. For example, there may be one **policy set** for security parameters, another for reliable
285 messaging parameters, and another for service-specific options such as price and quantity. The **policy**
286 **framework** layer must provide some mechanism for determining which aspect each **policy** applies to;
287 this might be specified, for example, by **constraints** that indicate the **vocabularies** used by a particular
288 collection of **policy sets**. The **policy bindings** layer will provide some mechanism for determine which
289 interface each **policy** applies to.

290 1.6 Specification of policy constraints

291 The authors of both proposed **policy framework** languages assume that the language for expressing
292 the individual **constraints** in a **policy** will be specified as part of each **vocabulary** specification, and

293 therefore should not be standardized as part of the specification of **policy** itself. There are strong
294 advantages to having a standard language for expressing **constraints**, however; and these advantages
295 will be described in Section 1.11.

296 But first, let's describe what such a **policy constraints** language would look like. Each **constraint** in a
297 **policy** specifies an allowable value, range of values, or set of values for a **vocabulary item** by
298 describing a **vocabulary item**, an operator, and another **vocabulary item** or a literal value. For
299 example, using the **policy statement** in the previous section, the individual **constraints** might be
300 expressed as:

301 "SecurityTokenType" IS A MEMBER OF {"KerberosV5TGT", "X509v3"}

302 "MessagePart" EQUALS "AccountNumber"

303 "MessagePart" NOT PRESENT

304 "EncryptionAlgorithm" EQUALS "3DES-CBC"

305 "EncryptionAlgorithm" NOT PRESENT

306 A language at the **policy constraints** layer provide ways to express these **constraints**. It should
307 provide a rich set of operators for specifying values, ranges of values, and sets of values of various
308 types. The **policy constraints** language should also provide a way to specify which end of a range of
309 acceptable values, or which elements in a set of acceptable values are preferred.

310 The **policy framework** layer will use the **constraints** specified by the **policy constraints** layer to
311 express acceptable sets of **constraints**:

312 { "SecurityTokenType" IS A MEMBER OF {"KerberosV5TGT", "X509v3"},
313 "MessagePart" EQUALS "AccountNumber",
314 "EncryptionAlgorithm" EQUALS "3DES-CBC" },

315
316 { "SecurityTokenType" IS A MEMBER OF {"KerberosV5TGT", "X509v3"},
317 "MessagePart" NOT EQUAL TO "AccountNumber",
318 "EncryptionAlgorithm" EQUALS "3DES-CBC"},

319
320 { "SecurityTokenType" IS A MEMBER OF {"KerberosV5TGT", "X509v3"},
321 "MessagePart" NOT EQUAL TO "AccountNumber",
322 "EncryptionAlgorithm" NOT PRESENT },

323 Notice that, by supporting operators such as "IS A MEMBER OF <set>", the number of **constraints**
324 required to express the **policy sets** can be reduced: rather than one **constraint** to say
325 "SecurityTokenType" EQUALS "KerberosV5TGT" and another **constraint** to say "SecurityTokenType"
326 EQUALS "X509v3", there can be a single **constraint** saying "SecurityTokenType" IS A MEMBER OF
327 {KerberosV5TGT, X509v3}. Using appropriate **constraint** operators is even more important in the case
328 of **vocabulary items** that may have a large range of values, such as time periods, IP addresses, or
329 prices: it would be prohibitive (or impossible, depending on the granularity needed) to try to express
330 every possible time between 9am and 5pm for example.

331 **Vocabulary items** that must not be present could logically be handled either by the **policy constraints**
332 layer or by the **policy framework** layer. In the **policy constraints** layer, this can be treated as a
333 constraint on the **vocabulary item** itself. In the **policy framework** layer, this can be treated as a
334 characteristic of the set. One further **constraint** type that could logically be handled at either layer is a
335 **vocabulary item** that must be present, but may have any value (i.e. is unconstrained).

336 A **policy constraint** language should provide predicate operators such as EQUALS <value>, EQUALS
337 <value set>, IS GREATER THAN <value>, IS GREATER THAN OR EQUAL TO <value>, IS LESS
338 THAN <value>, IS LESS THAN OR EQUAL TO <value>, IS AN ELEMENT OF <value set>, and IS A
339 SUBSET OF <value set>. Depending on the **policy framework** language used, the **policy constraint**
340 language may also need to provide operators for MUST BE PRESENT BUT MAY HAVE ANY VALUE,

341 and MUST NOT BE PRESENT. Since the implementation of the comparison and set operators depends
342 on the data type of the values involved, the **policy constraint** language must provide a way to determine
343 the data type of each **vocabulary item** or **item** value, and should be able to handle a rich set of data
344 types.

345 The **policy constraint** language should also provide a way of specifying preferred **vocabulary item**
346 values where a **constraint** allows a **vocabulary item** to have a range or set of values.

347 1.7 Intersection of policy constraints and policies

348 Use and analysis of **policies** usually requires the ability to compute the intersections of **constraints** and
349 of **policies**. The intersection of two **policies** is a single policy that will accept every set of **vocabulary**
350 **items** that would be accepted by both of the original **policies**. The intersection of two **policy**
351 **constraints** is a single constraint that will accept every **vocabulary item** value that would be accepted
352 by both of the original **constraints**. Even if no physical intersection document is constructed, users of
353 **policies** need to have the types of information required to perform an intersection, so precise
354 specification of these semantics is useful.

355 As an example of the need for **constraint** intersections, a **policy set** may contain two **constraints** that
356 are inconsistent – they can never both be satisfied by any **vocabulary item** value. More formally, two
357 **constraints** are inconsistent if they place constraints on the same **vocabulary item**, but the intersection
358 of the sets of values accepted by the two **constraints** is empty. A **policy set** that contains two or more
359 inconsistent **constraints** is itself inconsistent, and can be eliminated from the set of **policy sets** because
360 that **policy set** can never be satisfied (this elimination does not change the meaning of the rest of the
361 **policy** since **policy sets** are logically connected by OR operators). If, after all inconsistent **policy sets**
362 have been eliminated, there are no remaining **policy sets**, then the **policy** itself can never be satisfied.
363 For any **policy** that will be evaluated multiple times, it is more efficient to remove inconsistent **policy**
364 **sets** before doing any evaluation, rather than waste effort evaluating impossible-to-satisfy conditions
365 over and over.

366 As an example of the need for **policy** intersections, take the case of two web service entities that wish to
367 communicate. In order to do so, they must use a set of **vocabulary item** values that satisfies the
368 **policies** of both entities; i.e. they must use a set of **vocabulary item** values that satisfies the intersection
369 of both **policies**. When **vocabulary items** are independent, the acceptable **vocabulary items** will be
370 those that satisfy all the **constraints** in one **policy set** from the first entity's **policy** and all the
371 **constraints** in one **policy set** from the second entity's **policy**. These combinations of **policy sets** that
372 must be satisfied are represented by the cross-product, or intersection, of the two sets of **policy sets**.
373 Many of the **policy sets** in this intersection are likely to be inconsistent. In order to know whether the
374 two entities have any mutually acceptable **policy**, the inconsistent **policy sets** must be eliminated. If at
375 least one **policy set** remains, then the two **policies** are consistent, and any set of **vocabulary items**
376 that satisfies at least one of the **policy sets** left will be acceptable. Again, eliminating the inconsistent
377 **policy sets** requires being able to determine whether any **constraints** are inconsistent, which is the
378 ability to determine the intersection of **constraints** on the same **vocabulary items**.

379 A **policy constraint** language should provide an algorithm for determining the intersection of any
380 **constraints**. From that, the intersection of any two **policy sets** and the intersection of any two **policies**
381 can be determined.

382 Some **policy** engines need to determine not only the **policy** intersection, but also to select from the
383 intersection a specific preferred **policy set** containing preferred values for **vocabulary items**. This
384 selection might be done by various entities: the initiator of a communication, the consumer, the provider,
385 a third party service broker, etc. Since the preferences of a consumer may conflict with the preferences
386 of a provider, the entity that selects the actual **vocabulary item** values to use must have some algorithm
387 for deciding how to resolve conflicts. When the selection is done by the consumer or by the provider,
388 those entities will presumably use their own preferred value. When the selection is done by a third party,
389 some “fair” resolution algorithm might be implemented.

390 1.8 Policy merging and overwriting

391 In some cases, **policies** need to be merged or replaced. For example, a **policy bindings** layer
392 language may allow one **policy** to be specified for a service as a whole, and another **policy** to be
393 specified for some particular interface of the service. The **policy bindings** layer language is responsible
394 for defining whether the specific interface **policy** is supposed to replace, strengthen, or weaken the
395 overall service **policy**. Applying a replacement **policy** is straightforward, but computing strengthened or
396 weakened **policies** is somewhat more complex. If one **policy** strengthens another, then only a set of
397 **vocabulary items** that satisfies both **policies** is acceptable. The effective joint **policy** in this case is the
398 intersection of the two **policies**, as described in Section 1.7. If one **policy** weakens another **policy**, then
399 any set of **vocabulary items** that satisfies either **policy** is acceptable. The effective **policy** in this case
400 is a **policy** consisting of the union of the **policy sets** of the two original **policies**.

401 A **policy** union may contain redundant **policy sets**. That is, one **policy set** may accept a subset of the
402 **vocabulary items** accepted by another **policy set**. For efficient application or for analysis of the
403 effective **policy**, it is useful to eliminate redundant **policy sets**. Formally, a **policy set** is redundant with
404 respect to a second **policy set** if the intersection of the two **policy sets** is equal to the first **policy set**.

405 So, once again, a **policy constraint** language should provide an algorithm for determining the
406 intersection of any two **constraints**. From that, the intersection of any two **policy sets** can be
407 determined, and the union of any two **policy sets** can be minimized.

408 1.9 Translation of vocabulary specifications

409 Sometimes a **vocabulary** specification will not be available to a **policy** processing engine. This might
410 be the case, for example, where the **vocabulary** specification uses a proprietary XML schema.

411 In these cases, someone with access to the **vocabulary** schema can write a set of XSLT transforms to
412 convert instances of the **vocabulary** specification to a non-proprietary format, such as another XML
413 format or to a set of XACML `<Attribute>`s. The resulting format can be published as an alternative
414 **vocabulary** specification. **Policies** can then be written against this alternative, non-proprietary
415 **vocabulary** specification. Any instance of the proprietary **vocabulary** specification can be processed by
416 the XSLT transforms prior to use with **policy** processing entities.

417 For example, a service using a proprietary **vocabulary** specification could perform the XSLT transforms,
418 and then use the resulting non-proprietary form of the **vocabulary** specification in its WSDL description.
419 **Policies** for that service can then be written against the non-proprietary form of the **vocabulary**
420 specification. Standard **policy** processing engines can process any such **policies**.

421 1.10 Use models

422 So how are these web service **policies** going to be used? This section describes several use models.
423 This is not intended to be an exhaustive list.

424 Grid service broker

425 In a Grid environment, service descriptions written in WSDL may be registered with a Grid service broker
426 for each instance of a given service. Each service description contains **policies** associated by that
427 service instance with the service as a whole or with particular interfaces. A potential client sends the
428 service broker a skeleton WSDL instance, identifying one or more service interfaces the client wants to
429 use, along with the client's **policies**. Both the service **policies** and the client **policies** are written in a
430 **policy framework** language that uses *WS-PolicyConstraints* to express the individual **policy**
431 **constraints**. The service broker computes the intersection of the client's **policies** with each candidate
432 service's **policies**. If the intersection is not empty, then the client's requirements are consistent with
433 those of the service instance, so service broker sends the address of the service instance back to the
434 client, along with the **policy** representing the intersection. The client constructs messages to the service
435 using **vocabulary item** values that satisfy the client's preferred **policy set** in this intersected **policy**.

436 The client sends the set of **vocabulary item** values it intends to use to the service as part of session
 437 establishment. The service uses this set in responding to the client's messages.
 438 A possible scenario for this use case is diagrammed below.

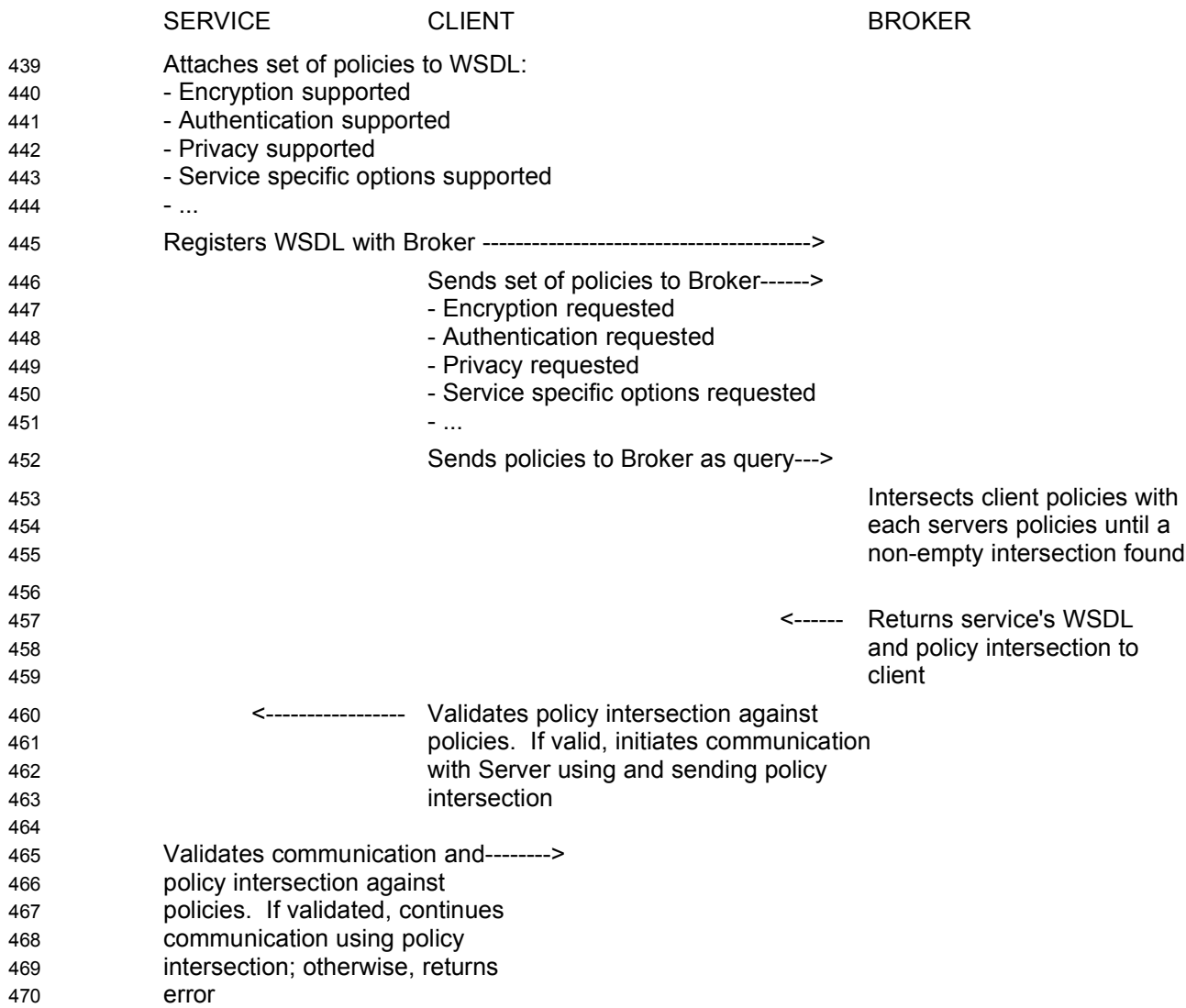


Figure x: Grid policy usage scenario

471 **WSDL publication**

472 An application server publishes the **policies** for a service instance it supports in a publicly accessible
 473 repository as part of the service instance's WSDL description. A client application retrieves the WSDL
 474 description from the repository, and extracts the **policy** for an interface the client plans to invoke. The
 475 client software may contain generic code modules it uses to match its internal configuration against the
 476 **policy constraints** in various **policy sets**. It uses the results to select a preferred **policy set** and
 477 preferred values for the **vocabulary items** as constrained by this **policy set**. In this scenario, the client
 478 never constructs its **policies** physically; it implements **policy constraint** matching functions internally in
 479 a way consistent with the semantics of *WS-PolicyConstraints*. The client's **policy** in this case is virtual,
 480 while the service instance's **policy** is a physical instantiation.

481 Service deployment configuration

482 A service developer describes the inherent capabilities of a service application in the form of **policies**
483 associated with various service interfaces. A service deployment tool reads these basic **policies** and
484 presents their options (**policy sets**) using the **policy framework** language to a site deployment
485 engineer. The deployment engineer chooses **vocabulary item** values consistent with the preferred
486 **policy sets** for each **policy**. The deployment tool then configures the service instance to use the
487 specified **policies**. As in the case of WSDL publication above, the service instance may not have its
488 **policies** expressed in a physical **policy** schema instance, but might process the **policy constraints**
489 using specific code modules.

490 Message verification

491 A service instance verifies the **vocabulary items** used in a message received from a client against the
492 instance's **policy** for the message interface. If the **items** do not comply with the **policy**, the service
493 rejects the message, possibly sending an error describing the failed **constraint** intersections back to the
494 client. A client similarly may verify that the **vocabulary items** used in a response from a service are
495 consistent with the client's **policies** or with a negotiated **policy** that the client and service have agreed to
496 use.

497 1.11 Benefits of standardizing policy constraints

498 A web service and its clients must understand the full semantics of a **vocabulary item** in order to
499 implement the associated functionality. For example, the encryption components of a web service and
500 its client must know how to implement an encryption algorithm that is specified as necessary in a **policy**.

501 But a number of important components that use or process web service **policies** do not need to
502 implement the semantics of **vocabulary items**, but do need to implement the semantics of **constraints**
503 on those items. Some examples of components that need to understand only the **constraints** include:

- 504 1. **Policy** verifiers, that must verify that a given service communication conforms to at least one
505 acceptable set of **policy constraints**. **Policy** verifiers may also need to find the intersection of
506 separately specified **policies**, such as the **policy** for a particular service interface that inherits
507 **constraints** from the **policy** for the entire service, but augments or overrides them with
508 **constraints** of its own.
- 509 2. Deployment tools that set the options for a web service to the values specified in a deployment
510 **policy**.
- 511 3. Service brokers that match clients having given capabilities and requirements against services
512 with **policies** that indicate they support compatible capabilities and requirements. Such brokers
513 need to compute the intersections of **policies**.
- 514 4. Service compositors that combine several services to create a single virtual service; compositors
515 must combine the **policies** of the component services into a single set of **policies** for presentation
516 to potential clients of the virtual service. Compositors can simplify the **policies** presented to
517 potential clients if they are able to eliminate subset **policies**. Compositors may also need to
518 determine that the **policies** of services acting as consumers of other services acting as providers
519 are compatible: this involves computing the intersection of two **policies**.
- 520 5. **Policy** query interfaces that can handle various types of requests for information about a **policy**.
521 These interfaces may need to find the intersection of service-wide and specific interface **policies**,
522 determine whether a specific set of **vocabulary items** provided by the querying entity satisfy a
523 **policy**, or return the **policy sets** that meet certain criteria provided by the querying entity.
- 524 6. **Policy set** appliers that choose one **policy set** from a **policy's policy sets** and applies
525 **vocabulary item** values that satisfy that **policy set** to a particular service communication.
- 526 7. **Policy** authoring and analysis tools.

527 All these functions can be performed by standard, generic code modules if **policy constraints** are
528 specified using a standard **policy constraints** language. These modules can handle any **policy**, even if
529 the **vocabularies** involved are new, changed, or proprietary. This is clearly a big advantage in reducing
530 footprint size, increasing interoperability, improving correctness, reliability, and usability of web service
531 **policies**. With a standard language, it is feasible to invest in good tools for dealing with **policies**, since
532 those tools will be used for multiple functions and with multiple types of **policies**.

533 1.12 Related work

534 Various semantic web languages such as OWL [OWL] and its derivatives have been proposed [WSCC]
535 as appropriate for expressing **policy constraints**, since they can theoretically express the semantics of
536 both the **vocabulary items** and of the **constraints** on those **items**. Two problems recommend against
537 this approach. First, none of the current semantic web languages support arithmetic comparison
538 operators. Since this is a major component of **policy constraints**, this is a serious deficiency. But, one
539 could argue, let's add such operators to an existing semantic web language rather than invent a new
540 one. This brings us to the second problem, which is that semantic web languages require specification
541 of far more, and more complex semantics, than a web service **policy** requires. Developing the
542 appropriate taxonomies and ontologies requires someone skilled in semantic web concepts and
543 languages, and such skills are currently not readily available to web services developers. In the future, if
544 an appropriate semantic web profile is developed for expressing web service **policies**, it should be easy
545 to translate between any standard **policy constraints** language and such a profile, since they both are
546 expressing generic **constraints**. This *WS-PolicyConstraints* specification does suggest linking to an
547 *Resource Description Framework* [RDF] specification where it is appropriate to consider hierarchical
548 categories or category equivalences in a **policy**. This allows a **policy** processor to match **vocabulary**
549 **items** from **vocabularies** that may express the same **item** at different levels of generalization, or simply
550 by using different names. Using an appropriate RDF taxonomy, for example, it is possible to determine
551 that a "James Bond Movie" **item** satisfies a request for an "Action Movie" **item**, and matches and
552 intersections can then be computed.

553 Besides semantic web approaches, a second potential candidate for handling the functions of a **policy**
554 **constraints** language is the *Object Constraint Language* (OCL) [OCL]. This language has many of the
555 types of predicate functions needed in a **policy constraint** language. The arguments against using
556 OCL, however, are that it is not an XML-based language, which is important for web services, and that it
557 is not designed to deal with intersection of predicate functions. In addition, OCL has not been approved
558 by any standards organization and has achieved little commercial acceptance to date.

559 A third candidate is the *XACML profile for Web-services*, or *Web Services Policy Language* [WSPL].
560 WSPL handles the tasks of both the **policy framework** layer and of the **policy constraints** layer, and is
561 specifically designed for web services. WSPL, while it was developed in the OASIS XACML TC
562 standards group, was not accepted for progression toward a standard by that group due to charter
563 conflicts. WSPL also conflicts with both of the dominant candidates for a **policy framework** layer
564 language, *WS-Policy* [WSPOLICY] and *WSDL with Compositors* [COMPOSITORS], making it
565 politically difficult to progress. This *WS-PolicyConstraints* specification is an attempt to take the
566 predicate language used by WSPL and separate it from the **policy framework** components so that it can
567 be used with other **policy framework** languages. The cost of this separation is that the resulting
568 language is no longer a subset of the OASIS Standard *eXtensible Access Control Markup Language*
569 [XACML], and thus is not as easily used as a vehicle for XACML policies. This cost may be alleviated by
570 translators between XACML and whatever language becomes the approved **policy framework**
571 language. Such translators should be simple to develop, since the proposed **policy framework**
572 languages are basically Boolean operator specifications that map easily to XACML Boolean operators.
573 Not all XACML policies can be expressed in *WS-PolicyConstraints* since XACML supports operators and
574 predicates for which the intersection can not readily be computed. This is also true of WSPL, so there is
575 no additional loss there.

576 A fourth candidate is the *ebXML Collaboration-Protocol Profile and Agreement Specification* [CPP/A].
577 This specification includes one schema (*CPP*) for describing policies regarding a wide range of
578 "collaboration protocols", including data-transport, reliable messaging, signaling, security, packaging, and

579 specific message exchanges, and another schema (*CPA*) that describes an agreed-upon set of
580 collaboration protocol parameters. There is a non-normative appendix that suggests ways in which two
581 *CPPs* could be intersected to produce a *CPA*. *CPP/A* has no general **policy set** or **policy constraint**
582 mechanisms: all of the functionality uses **vocabulary items** that have implicit semantics. The
583 intersection process must implement these implicit semantics, and even then *CPP/A* suggests that many
584 aspects of *CPA* generation may require manual negotiation. Thus *CPP/A* is unable to express
585 **constraints** on arbitrary new **vocabularies**, and depends on users accepting the *CPP/A* **vocabulary**. It
586 does not provide preference, “must not be present”, or explicit choice mechanisms. This is an example
587 of an ad hoc **policy** language that has neither generality nor built-in intersection functionality.

588 To summarize, it does not appear that there is any existing language that satisfies the requirements of a
589 **policy constraints** layer. This specification attempts to satisfy those requirements, while building as
590 much as possible on existing standards such as XACML.

591 1.13 Terminology

592 **Assertion** – a statement about the acceptable values for a particular **policy vocabulary item**. This
593 specification uses the term **policy constraint** rather than **assertion**.

594 **Attribute** – The identity, data type, and value for a single **vocabulary item**.

595 **Bag** – An unordered collection of values that MAY contain duplicates.

596 **Coincident constraints** – Two **constraints** are **coincident** if they contain references to at least one
597 common **vocabulary item**.

598 **Constraint** – See **Policy constraint**.

599 **Disjunctive normal form** – A standard way of organizing a logical expression as sets of predicates,
600 where the predicates in a set are related by AND, while the sets themselves are related by OR; such as
601 “{a AND b AND c} OR {b AND d}”. **disjunctive normal form** is useful in determining whether an
602 expression is satisfied by a particular set of terms because each set can be evaluated separately; if any
603 set is satisfied, then the entire expression is satisfied. Any Boolean combination of independent[?]
604 predicates can be expressed in **disjunctive normal form**.

605 **Equivalent policies**: Two **policies** are equivalent if the set of **vocabulary instances** validated by the
606 first **policy** is equal to the set of **policies** validated by the second **policy**.

607 **Policy** – A set of rules that describe some aspect of the behavior of an entity. More concretely, and as
608 used in this specification, it is an expression describing all the acceptable sets of **constraints** on items in
609 a given **vocabulary**.

610 **Policy constraint** - A reference to one or more **vocabulary items** along with a specification of particular
611 values allowed for those **items**. It is a statement of an individual constraint and capability in the **policy**
612 These are also called **policy assertions** or **policy predicates**.

613 **Policy framework** – The language used to express **policy sets**.

614 **Policy set** – A set of simple **constraints** (predicates) on **vocabulary items** such that any set of
615 **vocabulary items** that satisfies this set also satisfies the entire **policy**. If a **policy** is expressed in
616 **disjunctive normal form**, then each ANDed set of simple predicates is a **policy set**. A **policy** is the
617 union of all its **policy sets**. The **policy framework** language is used to express **policy sets**.

618 **Policy statement** – A concrete representation of a policy.

619 **Valid**: A **vocabulary instance** is **valid** against, or with respect to, a given **policy** if at least one **policy**
620 **set** of the **policy** is satisfied by the **vocabulary instance**.

621 **Vocabulary** – The identities, formats, and semantics of an independent set of technical features and
622 parameters associated with use of a web service. A **vocabulary** may be defined as a set of **Attributes**
623 or as an XML schema. One might think of the **vocabulary** as the set of variables used in stating a
624 **policy**.

625 **Vocabulary instance** – A specific instance of a **vocabulary**. A set of **Attributes** with their values or a

626 schema instance that provides values for various **vocabulary items** in a **vocabulary**.
627 **Vocabulary item** - One independent technical feature or parameter in a **vocabulary**. One might think of
628 this as a variable that can be used in a policy.

629 **1.14 Notation**

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

633 "they MUST only be used where it is actually required for interoperation or to limit behavior which
634 has potential for causing harm (e.g., limiting retransmissions)"

635 These keywords are thus capitalized when used to unambiguously specify requirements over protocol
636 and application features and behavior that affect the interoperability and security of implementations.
637 When these words are not capitalized, they are meant in their natural-language sense.

638 Conventional XML namespace prefixes are used throughout the listings in this specification to stand for
639 their respective namespaces as follows, whether or not a namespace declaration is present in the
640 example:

641 The prefix `xacml`: stands for the XACML policy namespace.

642 The prefix `xs`: stands for the W3C XML Schema namespace **[XS]**.

643 The prefix `xf`: stands for the *XQuery 1.0 and XPath 2.0 Function and Operators* specification
644 namespace **[XF]**.

645 In examples, the following XML Internal Entities are assumed to have been defined in the policy
646 specification:

```
647 <!ENTITY data "http://www.w3.org/2001/XMLSchema#">  
648 <!ENTITY function "urn:oasis:names:tc:xacml:1.0:function:">
```

2 Specification of constraints (normative)

649

650 This section specifies how to express **constraints** on **vocabulary items**.

651 A **constraint** SHALL be expressed as an XACML[XACML] predicate (<Apply> element) over one or
652 more **vocabulary items** or literal values contained in a **vocabulary instance**. The XACML predicate
653 SHALL conform to the restrictions specified in the following sections. A non-normative copy of each
654 relevant section of the XACML Policy Schema is reproduced for the convenience of the reader. The
655 normative syntax and semantics of these XACML schema elements SHALL be that contained in the
656 *XACML 2.0 Core Specification*[XACML].

2.1 Vocabulary item references

657

658 A **vocabulary item** used in a **constraint** SHALL be referenced using either an XACML
659 <AttributeSelector> or an XACML <AttributeDesignator> instance.

```
660 <xs:element name="Expression" type="xacml:ExpressionType" abstract="true"/>
661
662 <xs:complexType name="ExpressionType" abstract="true"/>
663
664 <xs:element name="AttributeSelector" type="xacml:AttributeSelectorType"
665 substitutionGroup="xacml:Expression"/>
666 <xs:complexType name="AttributeSelectorType">
667   <xs:complexContent>
668     <xs:extension base="xacml:ExpressionType">
669       <xs:attribute name="RequestContextPath" type="xs:string"
670 use="required"/>
671       <xs:attribute name="DataType" type="xs:anyURI"
672 use="required"/>
673       <xs:attribute name="MustBePresent" type="xs:boolean"
674 use="optional" default="false"/>
675     </xs:extension>
676   </xs:complexContent>
677 </xs:complexType>
678
679 <xs:element name="ResourceAttributeDesignator" type="xacml:AttributeDesignatorType"
680 substitutionGroup="xacml:Expression"/>
681
682 <xs:element name="ActionAttributeDesignator" type="xacml:AttributeDesignatorType"
683 substitutionGroup="xacml:Expression"/>
684
685 <xs:element name="EnvironmentAttributeDesignator"
686 type="xacml:AttributeDesignatorType" substitutionGroup="xacml:Expression"/>
687
688 <xs:complexType name="AttributeDesignatorType">
689   <xs:complexContent>
690     <xs:extension base="xacml:ExpressionType">
691       <xs:attribute name="AttributeId" type="xs:anyURI"
692 use="required"/>
693       <xs:attribute name="DataType" type="xs:anyURI"
694 use="required"/>
695       <xs:attribute name="Issuer" type="xs:string"
696 use="optional"/>
697       <xs:attribute name="MustBePresent" type="xs:boolean"
698 use="optional" default="false"/>
699     </xs:extension>
700   </xs:complexContent>
701 </xs:complexType>
702
703 <xs:element name="SubjectAttributeDesignator"
704 type="xacml:SubjectAttributeDesignatorType" substitutionGroup="xacml:Expression"/>
705
706 <xs:complexType name="SubjectAttributeDesignatorType">
707   <xs:complexContent>
708     <xs:extension base="xacml:AttributeDesignatorType">
709       <xs:attribute name="SubjectCategory" type="xs:anyURI"
710 use="optional" default="urn:oasis:names:tc:xacml:1.0:subject-category:access-
711 subject"/>
712     </xs:extension>
713   </xs:complexContent>
714 </xs:complexType>
```

715 **Vocabulary item reference examples (non-normative):**

716 1. An XACML <AttributeSelector> instance:

```

717 <AttributeSelector
718   RequestContextPath="//Signature/SignatureMethod@Algorithm/fn:base-
719   uri()"
720   DataType="&type;anyURI"/>

```

721 This <AttributeSelector> references a **vocabulary item** named `Algorithm` that is an
722 XML attribute in an *XML Digital Signature* schema instance. The `RequestContextPath` in the
723 <AttributeSelector> is an XML attribute that contains an XPath expression that selects the
724 `Algorithm` value as a URI. Since there could conceivably be more than one instance of any
725 given **vocabulary item** in a **vocabulary instance**, the set of all values that are selected by the
726 `RequestContextPath` are returned as a **bag**, which is an unordered collection of values that
727 MAY contain duplicates. If the **vocabulary instance** contains no **items** that match the XPath
728 selection criteria, then the **bag** returned will be empty.

729 2. An XACML <ResourceAttributeDesignator> instance:

```

730 <ResourceAttributeDesignator
731   AttributeId="data-rate"
732   DataType="&type;integer"/>

```

733 This <ResourceAttributeDesignator> references a **vocabulary item** named `data-rate`
734 that is expressed as an XACML Resource **Attribute** in a **vocabulary instance**. The identifier for
735 the **Attribute** is “`data-rate`” (such identifiers are actually URIs or URLs with specific
736 namespaces), and the data type is “`integer`”. Again, since there could conceivably be more
737 than one instance of any given **vocabulary item** in a **vocabulary instance**, the set of all values
738 in all Resource **Attributes** that match this identity and data type will be returned as a **bag** of
739 values. If the **vocabulary instance** contains no **Attributes** that match both of the
740 <AttributeDesignator> XML attributes, then the **bag** returned will be empty.

741 Any XACML **Attribute** MAY have an “`Issuer`” XML attribute that identifies the issuer of the
742 **Attribute**; there is no default value for “`Issuer`”. XACML Subject **Attributes** MAY also have a
743 “`SubjectCategory`” XML attribute that explicitly identifies the type of subject (user, machine,
744 application, receiver, etc.); the default “`SubjectCategory`” value is “`&category;access-`
745 `user`”, meaning the human user on whose behalf the access is being made.

746 2.2 Literal values

747 A literal value used in a **constraint** SHALL be expressed using an XACML <AttributeValue>
748 instance.

```

749 <xs:element name="AttributeValue" type="xacml:AttributeValueType"
750   substitutionGroup="xacml:Expression"/>
751
752 <xs:complexType name="AttributeValueType" mixed="true">
753   <xs:complexContent mixed="true">
754     <xs:extension base="xacml:ExpressionType">
755       <xs:sequence>
756         <xs:any namespace="##any" processContents="lax"
757   minOccurs="0" maxOccurs="unbounded"/>
758       </xs:sequence>
759       <xs:attribute name="DataType" type="xs:anyURI"
760   use="required"/>
761       <xs:anyAttribute namespace="##any" processContents="lax"/>
762     </xs:extension>
763   </xs:complexContent>
764 </xs:complexType>

```

765 2.3 Constraints

766 A **policy constraint** SHALL be expressed using an XACML <Apply> instance, with the addition of an
767 optional `Preference` XML attribute.

```

768 <xs:element name="Apply" type="xacml:ApplyType"
769   substitutionGroup="xacml:Expression"/>
770
771 <xs:complexType name="ApplyType">
772   <xs:complexContent>
773     <xs:extension base="xacml:ExpressionType">
774       <xs:sequence>

```

```

775                                     <xs:element ref="xacml:Expression" minOccurs="0"
776 maxOccurs="unbounded"/>
777                                     </xs:sequence>
778                                     <xs:attribute name="FunctionId" type="xs:anyURI"
779 use="required"/>
780                                     <xs:attribute name="Preference" type="xs:string"
781 use="optional"/>
782                                     </xs:extension>
783                                     </xs:complexContent>
784 </xs:complexType>

```

785 Exactly one **vocabulary item** SHALL be referenced in a single **constraint**.

786 TBD: this can probably be relaxed under certain restrictions. Problems occur with circular
787 references (A > B, B > C, C > A), and with intersection of references where one vocabulary item
788 is constrained by one or more literal values (A > B, B = 10).

789 The values for the "Preference" attribute are "greater" or "lesser". A value of "greater" means
790 that greater or later (for time or date types) values are preferred. A value of "lesser" means that lesser
791 or earlier values are preferred. This attribute is required when the "FunctionId" attribute specifies a
792 function that matches a range of values. This attribute is ignored otherwise. See Section 7 for more
793 information about the use of the "Preference" attribute.

794 <Apply> instances SHALL NOT be nested except for the purpose of converting a bag to a single value
795 where needed by the **constraint** function or for purposes of limiting the scope of a set of **constraints**.
796 That is to say, an Expression used in a **constraint** predicate SHALL be one of the following:

- 797 1. An <AttributeValue>,
- 798 2. An <AttributeSelector>,
- 799 3. One of the <AttributeDesignator> types, or
- 800 4. An <Apply> element having a FunctionId value corresponding to one of the XACML <type>-
801 one-and-only functions and a single Expression element that is an <AttributeSelector> or
802 one of the <AttributeDesignator> types.
- 803 5. An <Apply> element with a FunctionId value equal to "[namespace:]limit-scope".
804 These elements may be arbitrarily nested. All **constraints** specified under a "[namespace:]
805 limit-scope" function SHALL be considered as if they were individual **constraints** under a single
806 AND operator at the **policy framework** level.

807 Except as specified in the preceding list, the FunctionId attribute in an <Apply> element SHALL be
808 one of those listed in Table 2 of Section 6.

809 2.4 Constraint examples (non-normative)

810 The constraint

811 "There must be a single data-rate **vocabulary item** that must be greater than or equal to
812 64000"

813 (where data-rate has been defined in the **vocabulary** as an XACML Resource Attribute of type
814 integer), is expressed in *WS-PolicyConstraints* as follows:

```

815 <Apply FunctionId="&function;integer-greater-than-or-equal">
816   <Apply FunctionId="&function;integer-one-and-only">
817     <ResourceAttributeDesignator
818       AttributeId="data-rate"
819       DataType="&type;integer"/>
820   </Apply>
821   <AttributeValue DataType="&type;integer">64000</AttributeValue>
822 </Apply>

```

823 The constraint

824 "The one XML Digital Signature present must use the RSA-SHA1 signature method."

825 (where the Digital Signature vocabulary is that defined in *XML Digital Signature*[XMLDSig]), is expressed
826 in *WS-PolicyConstraints* as follows:

```

827 <Apply FunctionId="function:anyURI-equal">
828   <Apply FunctionId="&function:anyURI-one-and-only">

```

```
829         <AttributeSelector
830             RequestContextPath=
831             "//Signature/SignatureMethod@Algorithm/fn:base-uri()"
832             DataType="&type;anyURI"/>
833     </Apply>
834     <AttributeValue
835     DataType="&type;anyURI">http://www.w3.org/2000/09/xmldsig#rsa-sha1</AttributeValue>
836 </Apply>
```

837 **The constraint**

838 "At least one of the credit card types accepted must be VISA."

839 (where "credit-card-type" is defined in the **vocabulary** as an XACML Resource Attribute with type
840 "string"), is expressed in *WS-PolicyConstraints* as follows:

```
841 <Apply FunctionId="&function;string-is-in">
842     <AttributeValue DataType="&type;string">VISA</AttributeValue>
843     <ResourceAttributeDesignator
844         AttributeId="credit-card-type"
845         DataType="&type;string"/>
846 </Apply>
```

847

3 Validation of vocabulary instances (normative)

848 This section specifies how to **validate** a given **vocabulary instance** against a given **policy** that is in
849 Disjunctive Normal Form as **policy sets**. It is the task of the **Policy Framework Layer** to specify how to
850 normalize a given **policy**. [A non-normative example of **policy** normalization is included in Appendix A.]

851 If all **constraints** in at least one **policy set** in a **policy** have been successfully verified against a given
852 **vocabulary instance**, and if any additional requirements from the **policy framework** layer have been
853 satisfied, then that **vocabulary instance** has been “**validated**” against that **policy**. If the **policy**
854 **framework** layer handles the specification of **vocabulary items** that are unconstrained or that must not
855 be present, then those requirements must be satisfied as part of the validation of a **vocabulary instance**
856 against a **policy set**.

3.1 Verification of a constraint

858 A specific **constraint** is verified by dereferencing all **vocabulary item** references in the **constraint**
859 against the given **vocabulary instance**, and evaluating the **constraint** function(s) using the semantics
860 specified in the XACML 2.0 Core Specification[XACML]. If the result is Boolean “**True**”, then the
861 **constraint** has been successfully verified against the **vocabulary instance**. If the result is anything
862 other than Boolean “**True**” (e.g. A result of “**False**” or a function error or inability to dereference the
863 **item** reference), then the **constraint** has failed to verify against the **vocabulary instance** and the **policy**
864 **set** containing the **constraint** has failed to validate.

865 4 Intersection of policies (normative)

866 This section describes how to determine the intersection of two **policies**.

867 The intersection of two **policies** is a single **policy** that will validate exactly the set of **vocabulary**
868 **instances** that is the intersection of the sets of **vocabulary instances** validated by the two **policies**.

869 This intersection is computed by taking the cross product of the **policy sets** in the two **policies**.

870 For example, if the **policy sets** in Policy 1 are:

871 {A, B, C}, {E, F}

872 And the **policy sets** in Policy 2 are:

873 {G, H, I}, {J, K}

874 Then the cross product is:

875 {A, B, C} + {G, H, I}, {A, B, C} + {J, K}, {E, F} + {G, H, I}, {E, F} + {J, K}

876 or

877 {A, B, C, G, H, I}, {A, B, C, J, K}, {E, F, G, H, I}, {E, F, J, K}

878 This cross product MAY be simplified by eliminating **policy sets** containing incompatible **constraints**.

879 4.1 Incompatible constraints

880 Two **constraints** that reference the same **vocabulary item** are **coincident**. If the intersection of two
881 **coincident constraints** is the empty set, then the **constraints** are incompatible. Any **policy set** that
882 contains incompatible **constraints** MAY be eliminated. The resulting **policy** is **equivalent** to the **policy**
883 prior to the elimination of incompatible **constraints** since the incompatible **constraints** could never
884 return a result of Boolean "True" against any **vocabulary instance**.

885 4.2 Intersection of constraints

886 The intersection of two **coincident constraints** is either the empty set, the two **constraints**, or a single
887 **constraint**. The result of taking the intersection of two **constraints**, and the `FunctionId` and
888 `<AttributeValue>` of any single resulting **constraint** are specified in Table 2 in Section 6. If the
889 intersection of the two **constraints** is the empty set, then the **constraints** are incompatible.

890 5 New constraint functions (normative)

891 This section defines several new functions that MAY be used in specifying **policy constraints**. Any
892 compliant *WS-PolicyConstraints* processor SHALL implement these functions.

893 5.1 Presence/absence functions

894 Where a **policy** requires that certain **vocabulary items** must not be present, or must be present but may
895 have any value, and if the **policy framework** language lacks a mechanism for indicating these types of
896 requirements, then a **policy set** MAY be augmented with an instance of one of the following functions as
897 appropriate for purposes of computing intersections.

898 [namespace:]must-be-present

899 This function SHALL take one `<AttributeDescriptor>` or an `<AttributeValue>` with data type
900 “&xml:string” as its argument and SHALL return a Boolean result. The `<AttributeValue>` string
901 SHALL be interpreted as an XPath expression. The result SHALL be “true” if the bag containing the
902 values of the specified `<AttributeDescriptor>` or the nodeset representing the XPath expression
903 contains at least one value; the result SHALL be “false” otherwise.

904 This function is equivalent to applying the “bag-size” function to the `<AttributeDescriptor>` and
905 comparing the result using the “&function;integer-greater-than-or-equal” function to the
906 integer value “1”.

907 [namespace:]must-not-be-present

908 This function SHALL take one `<AttributeDescriptor>` or an `<AttributeValue>` with data type
909 “&xml:string” as its argument and SHALL return a Boolean result. The `<AttributeValue>` string
910 SHALL be interpreted as an XPath expression. The result SHALL be “true” if the bag containing the
911 values of the specified `<AttributeDescriptor>` or the nodeset representing the XPath expression
912 contains no values (i.e. is an empty bag or set); the result SHALL be “false” otherwise.

913 This function is equivalent to applying the “bag-size” function to the `<AttributeDescriptor>` and
914 comparing the result using the “function:integer-equal” function to the integer value “0”.

915 5.2 [namespace:]limit-scope

916 This function is used to group **constraints** that must all be satisfied by a single element in an XML
917 schema instance.

918 This function SHALL take two or more parameters, where the first parameter is and
919 `<AttributeValue>` with data type “&datatype:string”, and the remaining parameters are **policy**
920 **constraints**. It SHALL return a Boolean result. The `<AttributeValue>` used as the first parameters
921 SHALL be interpreted as an XPath expression. The result SHALL be “true” if the **constraints** are all
922 “true” when applied to at least one single **node** in the nodeset selected by the `<AttributeValue>`'s
923 XPath expression. That is, all **constraints** must be satisfied by the same node, although there may be
924 multiple such **nodes** where all **constraints** are satisfied.

925 Instances of the “limit-scope” function SHALL be treated for purposes of **policy** intersection as if all the
926 **constraints** were individual **constraints** specified separately without the “limit-scope” function. This
927 function may be nested arbitrarily deeply, but all **constraints** are treated as if they were specified at the
928 top-most **policy set constraint** level.

929 Non-normative example:

```
930 <Apply FunctionId="[namespace:]limit-scope">  
931 <AttributeValue DataType="&xml:string"/>security/key-  
932 info</AttributeValue>  
933 <Apply FunctionId="&function;integer-equal">
```

```

934         <AttributeSelector ElementId="/key-length"
935         DataType="&xml;integer">
936             <AttributeValue DataType="&xml;integer">96</AttributeValue>
937         </Apply>
938         <Apply FunctionId="&function;string-equal">
939             <AttributeSelector ElementId="/algorithm"
940             DataType="&xml;string">
941                 <AttributeValue DataType="&xml;string">DES-CBC</AttributeValue>
942             </Apply>
943         </Apply>
944

```

945 This function returns “true” only if there is at least one “/security/key-info” element that
946 has a “key-length” child with value 96 and an “algorithm” child with value “DES-CBC”.
947 [XPath syntax might treat parent “limit-scope” as new root, or we might require evaluator to
948 follow only paths that lead into same element.]

949 This function specifies two **policy constraints**. For intersection purposes, these **constraints**
950 are considered as if they were enclosed with an AND operator at the **policy framework** level.

951 5.3 [namespace:]ipAddress-match

952 This function is used to specify a range of IP address values, optionally qualified by port-ranges.

953 This function SHALL take two parameters, where the first parameter that is an <AttributeValue> of type
954 “&xml;string” and the second parameter is an <AttributeSelector> or <AttributeDescriptor” with datatype
955 “&datatype;ipAddress”.

956 This function SHALL return “True” if the IP addresses and ports (if included) specified by the
957 <AttributeSelector> or <AttributeDescriptor> are a subset of those specified by the <AttributeValue>.

958 [Note: this function is used rather than the XACML ipAddress-regexp-match function because it
959 is very difficult to specify ranges of IP addresses and port using regular expressions, and
960 because taking the intersection of two such regular expressions may not be well-defined.]

961 5.4 [namespace:]string-getKrb5SName

962 This function is used to obtain the service's “sname” from a Kerberos ticket.

963 This function SHALL take one parameter, which SHALL be an <AttributeSelector> or
964 <AttributeDescriptor> with datatype “&xml;base64Binary”.

965 This function SHALL return a value with datatype “&xml;string”. This value SHALL be the “sname” of the
966 service in the Kerberos ticket specified by the <AttributeSelector> or <AttributeDescriptor>. If this value
967 does not contain a valid Kerberos ticket, then the result SHALL be “Indeterminate” (error).

968 5.5 [namespace:]hexBinary-getCertExtensionValue

969 This function is used to obtain the value of an extension in an X.509 certificate.

970 This function SHALL take three parameters, where the first parameter is an <AttributeValue> of type
971 “&xml;string”, the second parameter is another <AttributeValue> of type “&xml;string”, and the third
972 parameter is an <AttributeSelector> or <AttributeDescriptor> of type “&xml;base64Binary”.

973 This function SHALL return the value contained in the X.509 certificate extension from the X.509
974 certificate specified by the third parameter having the OID specified in the first parameter and the
975 criticality specified in the second parameter. The acceptable values of the second parameter are
976 “Critical”, “NotCritical”, and “CriticalOrNot”. The third parameter must contain an ASN.1 encoded X.509
977 certificate. If any of the parameters is invalid, then the result SHALL be “Indeterminate” (error).

978 5.6 [namespace:]string-to-uri

979 This function takes as input a value of type “&xml;string”, and produces as output a value of type
980 “&xml:anyURI”. The input parameter SHALL be a string that can be interpreted as a URI. If the input

981 value is not a valid URI, the result of this function SHALL be “Indeterminate” (error).

6 Supported constraint functions and their intersections (normative)

The following table lists the XACML functions that are supported for use in **policy constraints**, and describes how to compute the intersection of any two **constraints**.

If two **constraints** in the same **policy set** constrain the same **vocabulary item**, then they are said to be **coincident**. If two **constraints** in the same **policy set** are not **coincident**, then their intersection is the two original **constraints**.

If two **coincident constraints** are not compatible according to the “Compatibility test” column of Table 2 below, then the two **constraints** are incompatible and the **policy set** containing them **MUST** be discarded.

If two **coincident constraints** are compatible according to the “Compatibility test” column, then their intersection is the **constraint** specified in the “Replacement constraint” column of Table 2.

Table 2 is to be interpreted according to the following key.

Columns one, two and four contain shorthand versions of an XACML `<Apply>` element. The portion before the open parenthesis (e.g. “type-equal” in the first row) represents the `<Apply>` element’s `FunctionId` attribute value. The “type-” portion represents any of the type-specific parts of the standard XACML function identifiers.

Alphabetic symbols (e.g. “a” in the first row) represent XACML `<AttributeDesignator>`, `<AttributeSelector>` or `<AttributeValue>` elements. Where a **constraint** `FunctionId` takes a single value rather than a bag for an argument where an `<AttributeDesignator>` or `<AttributeSelector>` is used, the `<AttributeDesignator>` or `<AttributeSelector>` **SHALL** be enclosed in an inner `<Apply>` element having as its `FunctionId` the appropriate “type-one-and-only” function.

Where “Keep both constraints” appears in the “Replacement constraint” column, there is no single replacement `<Apply>` element: the **predicates** are compatible, but not combinable. In these cases, the original `<Apply>` elements **MUST NOT** be modified by this step in the procedure.

any-constraint(a) is “true” if there is any **constraint** in the **policy set** that applies to **vocabulary item** a. no-constant(a) is “true” if there is no **constraint** in the **policy set** that applies to **vocabulary item** a.

\cap means set intersection.

\subseteq means “is a subset of”.

	First constraint	Second constraint	Compatibility test	Replacement constraint
1	type-equal(a,b)	type-equal(a,c)	$b == c$	type-equal(a,b)
2	type-equal(a,b)	type-greater-than(a,c)	$b > c$	type-equal(a,b)
3	type-equal(a,b)	type-greater-than-or-equal(a,c)	$b \geq c$	type-equal(a,b)
4	type-equal(a,b)	type-less-than(a,c)	$b < c$	type-equal(a,b)
5	type-equal(a,b)	type-less-than-or-equal(a,c)	$b \leq c$	type-equal(a,b)
6	type-greater-than(a,b)	type-greater-than(a,c)		type-greater-than(a,max(b,c))

7	type-greater-than (a,b)	type-greater-than-or- equal(a,c)		Where $b \geq c$	type-greater-than (a,b)
8				Where $b < c$	type-greater- than-or-equal (a,c)
9	type-greater-than- or-equal(a,b)	type-greater-than-or- equal(a,c)		type-greater-than-or-equal (a,max(b,c))	
10	type-less-than (a,b)	type-less-than(a,c)		type-less-than(a,min(b,c))	
11	type-less-than (a,b)	type-less-than-or-equal (a,c)		Where $b > c$	type-less-than-or- equal(a,c)
12				Where $b \leq c$	type-less-than (a,b)
13	type-less-than-or- equal(a,b)	type-less-than-or-equal (a,c)		type-less-than-or-equal (a,min(b,c))	
14	type-greater-than (a,b)	type-less-than(a,c)	$b < c$	Keep both constraints	
15	type-greater-than (a,b)	type-less-than-or-equal (a,c)	$b < c$	Keep both constraints	
16	type-greater-than- or-equal(a,b)	type-less-than(a,c)	$b < c$	Keep both constraints	
17	type-greater-than- or-equal(a,b)	type--less-than-or-equal (a,c)	$b < c$	Keep both constraints	
18	set-equals(a,b)	set-equals(a,c)	$b == c$	set-equals(a,b)	
19	set-equals(a,b)	subset(a,c)	$b \subseteq c$	set-equals(a,b)	
20	subset(a,b)	subset(a,c)	$\cap (b,c) \neq 0$	subset (a, $\cap (b,c)$)	
21	must-be-present (a)	Any constraint other than must-not-be-present(a)		The second constraint	
22	must-not-be- present(a)	must-not-be-present(a)		must-not-be-present(a)	
23	any-constraint(a)	no constraint(a)		any-constraint(a)	
24	time-in-range (a,b,c)	time-equal(a,d)	$b \leq d \leq c$	time-equal(a, d)	
25	time-in-range (a,b,c)	time-greater-than-or- equal(a,d)	$b \leq d \leq c$	time-in-range(a, d, c)	
26	time-in-range (a,b,c)	time-less-than-or-equal (a,d)	$b \leq d \leq c$	time-in-range(a, b, d)	
27	type-regexp- match (a, b)	type-equal(c, b)	$\cap (a,c) \neq 0$	type-equal(c, b)	
28	type-regexp- match (a, b)	Type-regexp-match(c, b)	$\cap (a,c) \neq 0$	type-regexp-match($\cap (a,c)$, b)	

1015 **Table 2 - Intersection of coincident constraints**

1016 **6.1 Intersection with semantic hierarchies**

1017 The values used in a **constraint** may be specified to be associated with a semantic hierarchy, either by
1018 the **policy framework** layer or by the **vocabulary specification** layer.

1019 The intersection of two **coincident** equality **constraints** using values that are not equal, but are in the
1020 same semantic hierarchy SHALL be an equality **constraint** in which the **vocabulary item** is the same
1021 and the literal value is the value that has the union of the semantics of the two input values (i.e. the
1022 descendant, or most specific, value). If the two values are specified as having equivalent semantics,
1023 then the intersection SHALL be either one of the two input **constraints**. If the values are not in the same
1024 semantic hierarchy, then the two **constraints** are not compatible.

1025 Non-normative example: if the first **constraint** is “string-equal(Movie type, “Action movie”)” and

1026 the second **constraint** is 'string-equal(Movie type, "James Bond movie")', and "James Bond
1027 movie" inherits all the semantics of "Action movie", then the intersection is the **constraint** 'string-
1028 equal(Movie type, "James Bond movie")'.

7 Preferences (normative)

1029

1030 Unless specified otherwise in the **policy framework layer**, a **policy** issuer's preferences SHALL be
1031 considered to be first in order of **policy set** preference, and then in order of value preference for each
1032 **vocabulary item** in a **policy set**.

1033 Unless **policy set preferences** are specified using another mechanism in the **policy framework** layer,
1034 then **policy sets** SHALL be considered preferred in the order specified when the **policy** is in Disjunctive
1035 Normal Form. If the **policy** is not specified in Disjunctive Normal Form, then a DNF generation algorithm
1036 preserving preference SHALL generate sets using nodes to the left under any operator having "OR"
1037 semantics before using nodes to the right.

1038 Non-normative note: If **policy sets** are generated using an algorithm that produces results
1039 equivalent to that in Appendix A, then those **policy sets** will be generated in order of preference.

1040 Non-normative example: If the **policy** is equivalent to the following (where A, B, C, and D are
1041 individual **constraints** on **vocabulary items**):

1042 AND(OR(A,B), OR(C,D))

1043 then the **policy sets** in preference order are:

1044 {A, C}, {B, C}, {A, D}, {B, D}

1045 Unless **constraint** preferences are specified using another mechanism in the **policy framework** layer,
1046 **constraints** SHALL be considered preferred in the order that they are specified in a **policy set** that has
1047 been generated in **policy set** preference order. That is, if two **constraints** reference the same
1048 **vocabulary item**, then the **constraint** listed earlier in the **policy set** is preferred over a **constraint** listed
1049 later in the same **policy set**.

1050 Some **constraints** express a set or range of **vocabulary item** values. In such a case, unless
1051 **vocabulary item** value preferences are specified using another mechanism in the **policy framework** or
1052 **vocabulary item** definition layers, these preferences SHALL be specified as follows. For a set, values
1053 SHALL be considered preferred in the order they are specified in the set of values. For **constraints** that
1054 specify a range of **vocabulary item** values, the "Preference" XML attribute in the "Apply" element
1055 SHALL be used to indicate whether greater or lesser values in the range are preferred.

1056 Non-normative examples:

1057 In the **constraint**

```
1058 <Apply FunctionID="time-in-range" Preference="greater">  
1059   <AttributeSelector RequestContextPath=".../transaction-  
1060   time/text()">  
1061     <AttributeValue>9am</AttributeValue>  
1062     <AttributeValue>5pm</AttributeValue>  
1063   </Apply>
```

1064 the preferred values are the ones closer to 5pm, since those time values (in 24-hour time) are
1065 greater than values closer to 9am.

1066 In the **constraint**

```
1067 <Apply FunctionId="time-is-in">  
1068   <AttributeSelector RequestContextPath=".../transaction-  
1069   time/text()">  
1070     <Apply FunctionId="time-bag">  
1071       <AttributeValue>5pm</AttributeValue>  
1072       <AttributeValue>8pm</AttributeValue>  
1073       <AttributeValue>7am</AttributeValue>  
1074     </Apply>  
1075   </Apply>
```

1076 the most preferred value is 5pm, followed by 8pm, followed by 7am, since the values are
1077 specified in this order in the **constraint**.

1078

8 References

1079

8.1 Normative references

- 1080 **[RFC2119]** S. Bradner, Key words for use in RFCs to Indicate Requirement Levels, IETF
1081 RFC 2119, March 1997, <http://www.ietf.org/rfc/rfc2119.txt>.
- 1082 **[XACML]** T. Moses, ed., eXtensible Access Control Markup Language (XACML) Version
1083 2.0, OASIS Access Control (XACML) TC, OASIS Standard, [http://docs.oasis-](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf)
1084 [open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf](http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf)
1085

1086

8.2 Non-normative references

- 1087 **[Barth04]** A. Barth, et al., Conflict and combination in privacy policy languages, Workshop
1088 on Privacy in the Electronic Society, 8 November 2004,
1089 <http://portal.acm.org/citation.cfm?id=1029195&coll=portal&dl=ACM>
- 1090 **[COMPOSITORS]** U. Yalcinalp, Proposal for adding Compositors to WSDL 2.0, 26 January 2004,
1091 <http://lists.w3.org/Archives/Public/www-ws-desc/2004Jan/0153.html>.
- 1092 **[CPP/A]** Collaboration-Protocol Profile and Agreement Specification Version 2.0, OASIS
1093 ebXML Collaboration Protocol Profile and Agreement Technical Committee, 23
1094 September 2002, [http://www.oasis-open.org/committees/ebxml-](http://www.oasis-open.org/committees/ebxml-cppa/documents/ebCPP-2.0.pdf)
1095 [cppa/documents/ebCPP-2.0.pdf](http://www.oasis-open.org/committees/ebxml-cppa/documents/ebCPP-2.0.pdf)
- 1096 **[OCL]** Response to the UML 2.0 OCL RfP (ad/2000-09-03), Revised Submission,
1097 Version 1.6, 6 January 2003, OMG Document ad/2003-01-07,
1098 <http://www.omg.org/docs/ad/03-01-07.pdf>.
- 1099 **[OWL]** W3C, Web Ontology Language,
1100 **[RDF]** W3C, Resource Description Framework (RDF),
1101 **[SAML]** S. Cantor, et al., eds., Assertions and Protocols for the OASIS Security
1102 Assertion Markup Language (SAML) V2.0, OASIS Security Services TC, OASIS
1103 Committee Draft 02, 24 September 2004, [http://www.oasis-](http://www.oasis-open.org/committees/download.php/9455/sstc-saml-core-2.0-cd-02.pdf)
1104 [open.org/committees/download.php/9455/sstc-saml-core-2.0-cd-02.pdf](http://www.oasis-open.org/committees/download.php/9455/sstc-saml-core-2.0-cd-02.pdf).
- 1105 **[SOAP]** M. Gudgin, et al., SOAP Version 1.2 Part 1: Messaging Framework,
1106 **[UDDI]** UDDI Version 2 Specifications, OASIS UDDI Specification TC, OASIS
1107 Standard, July 2002, [http://www.oasis-open.org/committees/uddi-](http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2)
1108 [spec/doc/tcspecs.htm#uddiv2](http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2).
- 1109 **[WSCC]** W3C, W3C Workshop on Constraints and Capabilities for Web Services, 12-13
1110 October 2004, <http://www.w3.org/2004/09/ws-cc-program.html>.
- 1111 **[WSDL]** E. Christensen, et al., Web Services Description Language (WSDL) 1.1, W3C
1112 Note, 15 March 2001, <http://www.w3.org/TR/wsdl>.
- 1113 **[WSMD]** WS-MetadataExchange, [http://www-](http://www-106.ibm.com/developerworks/library/specification/ws-mex/)
1114 [106.ibm.com/developerworks/library/specification/ws-mex/](http://www-106.ibm.com/developerworks/library/specification/ws-mex/)
- 1115 **[WSRM]** K. Iwasa, et al., eds., WS-Reliability v1.1, OASIS Web Service Reliable
1116 Messaging TC, OASIS Standard, 15 November 2004, [http://www.oasis-](http://www.oasis-open.org/committees/download.php/9330/WS-Reliability-CD1.086.zip)
1117 [open.org/committees/download.php/9330/WS-Reliability-CD1.086.zip](http://www.oasis-open.org/committees/download.php/9330/WS-Reliability-CD1.086.zip).
- 1118 **[WSPOLICY]** S. Bajaj, et al., Web Services Policy Framework (WS-Policy), September 2004,
1119 <http://www-106.ibm.com/developerworks/library/specification/ws-polfram/>
- 1120 **[WSPA]** WS-PolicyAttachments
1121 **[WSPL]** T. Moses, ed., XACML profile for Web-services, OASIS Access Control
1122 (XACML) TC, Working Draft 04, 29 September 2004, [http://www.oasis-](http://www.oasis-open.org/committees/download.php/3661/draft-xacml-wspl-04.pdf)
1123 [open.org/committees/download.php/3661/draft-xacml-wspl-04.pdf](http://www.oasis-open.org/committees/download.php/3661/draft-xacml-wspl-04.pdf).

1124 **[WSS]** T. Nadalin, et al., eds., Web Services Security: SOAP Message Security 1.0
1125 (WS-Security 2004), OASIS Web Services Security TC, OASIS Standard
1126 200401, March 2004, [http://docs.oasis-open.org/wss/2004/01/oasis-200401-
1127 wss-soap-message-security-1.0.pdf](http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf).
1128 **[XMLDSig]** W3C, XML Digital Signature,
1129

1130

A. Example of policy normalization (non-normative)

1131 *Note: this section is half-baked and needs work. Reference to a standard algorithm should be included.*

1132 *The algorithm used in “The Simple Digital Library Interoperability Protocol (SDLIP-Core)” [http://www-](http://www-diglib.stanford.edu/~testbed/doc2/SDLIP/javadoc2/sdlip/helpers/QueryUtil.html)*

1133 *diglib.stanford.edu/~testbed/doc2/SDLIP/javadoc2/sdlip/helpers/QueryUtil.html* might be used.

1134 *javax.wbem.query.QueryExp class, described at*

1135 *[http://wbemservices.sourceforge.net/](http://wbemservices.sourceforge.net/WBEMSDKDG_html/p25.html)*

1136 *WBEMSDKDG_html/p25.html* contains a method, “*canonicalizeDOC*”, that could be used to convert an

1137 *arbitrary tree containing Boolean predicates into Disjunctive Normal Form.*

1138 Any **policy** that is expressed in the form of a Boolean combination of independent[?] predicates can be

1139 converted to **disjunctive normal form (DNF)** via standard algorithms. For many **policy** operations that

1140 require matching or intersecting **policy sets**, it is not necessary to convert the entire **policy** to **DNF**:

1141 **policy sets** can be constructed and tested incrementally, terminating the process when a suitable **policy**

1142 **set** is found.

1143 For convenience, an example of such an algorithm is provided here, where the **policy** is assumed to be

1144 in the form of a tree with root node <root>, where each node in the tree is <AND>, <OR>, <NOT>, or a

1145 simple predicate. A <NOT> node may have only a single child node.

1146 In this example, required presence or absence of vocabulary items in a particular **policy set** is handled

1147 via [???].

1148 1. Initialize:

1149 PolicySets={}

1150 Combinations={<root>}

1151 2. Select first element node C in Combinations.

1152 3. If C contains no non-leaf nodes, then C is a valid **policy set**. Move C from Combinations to

1153 PolicySets.

1154 4. If Combinations is empty, terminate. PolicySets contains all **policy sets** in the original **policy**.

1155 5. Choose first non-leaf child N of C.

1156 6. If N is <AND>, replace C with a copy of C where N has been replaced by all children of N. If N is

1157 <OR>, make a copy of C for each child of N. In each copy, replace C with one of the children of N. If

1158 doing a breadth-first construction, replace C with the copy containing the first child of C, and append

1159 remaining copies to the end of Combinations. If doing a depth-first construction, replace C with all

1160 copies. If N is <NOT>, if the child of N is a simple predicate, replace N with the negation of the child

1161 of N. If the child of N is <AND>, then for each child c[i] of AND, replace the child with <NOT> c[i].

1162 Replace N with these modified children. If the child of N is <OR>, then...

1163 7. Go to step 2.

1164

B. Future work

1165 This section describes further work that might be considered for inclusion in this specification.

1166 1. THEORY:

1167 1. Dependent vocabulary items: definition currently says “independent”. Under what conditions can
1168 we handle $\text{AttrA} \geq \text{AttrB}$ types of constraints? E.g. $A \geq B \cap A \geq 3$ is both constraints. $A \geq B \cap$
1169 $A = B$ is $A = B$. Have to worry about $A > B$, $B > C$, $C > A$ types of circular (and inconsistent)
1170 dependencies.

1171 2. Formalize intersection of constraints: previous work on “constraint narrowing” is relevant here.

1172 2. FUNCTIONS/OPERATORS:

1173 1. Intersectable XPath expressions: although it is easy to determine, when applied to a specific
1174 schema instance, whether two different XPath expressions select the same set of nodes, it is not
1175 possible to determine this in general for any schema instance. A subset of XPath is needed for
1176 which intersections of two XPath expressions can be determined with respect to any valid schema
1177 instance. Proposal: prohibit XPath query operators and element order specifiers [x]. All
1178 expressions must be absolute rather than relative (in the case of the limit-scope function, they can
1179 be relative to the node(s) selected by the limit-scope function.

1180 2. Add more XACML standard functions to the table? How about high-order bag functions?

1181 3. Add reference to an algorithm for computing intersection of two regular expression (intersection of
1182 two FSAs): Janusz A. Brzozowski, “Derivatives of Regular Expressions”, Journal of the ACM, V.
1183 11, Issue 4, Oct. 1964, pp. 481-494, <http://portal.acm.org/citation.cfm?id=321249>. Algorithms in
1184 Java at: http://dmoz.org/Computers/Programming/Languages/Regular_Expressions/Java/

1185 4. Add support for arithmetic operators? e.g. Birthdate + 21 years \leq CurrentDate?

1186 5. Day of Week match function? NO. This can be handled via sets of days fairly easily.

1187 6. Specify support for additional functions: these may be helpful if trying to support constraints over
1188 URI references or over legacy encoding formats. The alternative is to express the **vocabulary**
1189 **items** for these constraints as independent policy **vocabulary items**, and not try to directly verify
1190 them.

1191 a) `string-to-uri`: convert a value of type `string` (that can be interpreted as a valid URI) to a
1192 value of type `anyURI`. XACML has a `url-string-concatenate` function, but this is not
1193 capable of composing a URI from fragments that are themselves not valid URIs (e.g. `#body`).
1194 More flexible composition of URIs may be needed for matching references in the body of the
1195 SOAP message to the corresponding elements in the SOAP header.

1196 b) `hexBinary-getCertExtensionValue`: takes as input a string value interpreted as the OID
1197 of the desired extension, a string value indicating required criticality (with acceptable values of
1198 “Critical”, “NotCritical”, and “CriticalOrNot”), and a reference to an ASN-encoded
1199 X.509 certificate, encoded as `&xsd;hexBinary`. A corresponding function that takes a
1200 certificate encoded in `&xsd;base64Binary` may also be needed. Functions that return the
1201 extension value as `&xsd:string` or `&xsd;integer` may also be useful.

1202 c) `string-getKrb5SName`: takes as input a reference to a Kerberos ticket in a
1203 `<wsse:BinarySecurityToken>` that is encoded in `&xsd;base64Binary` or
1204 `&xsd;hexBinary`. It returns the service's sname as a string.

1205 3. INTERSECTION ISSUES

1206 1. Intersection where one policy constrains an item, but another policy being intersected with it does
1207 not: WS-Policy says “If the vocabulary of one policy includes an assertion type that is not in the
1208 vocabulary of another policy, then the behavior associated with that assertion type is prohibited in
1209 the intersection of those policies. Is this adequate? I would think “item is not constrained” is a
1210 more logical interpretation of missing assertion types, and explicit use of “must not be present”
1211 function represents “assertion type is prohibited”. For example, if one policy says something must

1212 be encrypted, and another policy does not constrain encryption, then certainly the intersection is
1213 not to prohibit encryption: either the policies are incompatible, or the combined policy must require
1214 encryption. Needs to be worked out together with “must be present”/“must not be present”
1215 semantics.

1216 2. Way to indicate that intersection should include all instances of some element that occurs in both
1217 policies. Example from CPP/A is that all comments from both client and server CPPs should be
1218 included in the CPA.

1219

C. Acknowledgments

1220

The authors would like to acknowledge the contributions of the OASIS Access Control (XACML)

1221

Technical Committee, and of the editor and contributors to the XACML profile for Web-services[WSPL]

1222

on which this specification is based.

D. Revision History

Rev	Date	By Whom	What
01	15 Dec 2004	Anne Anderson	Initial version based on scaling back Web services profile of XACML.
02	21 Dec 2004	Anne Anderson	Intro improved; "must be present"/not added
03	7 Jan 2005	Anne Anderson	Add preferences, semantic hierarchies, more supported functions, more complete TBD section.
04	20 Jan 2005	Anne Anderson	Corrected semantic hierarchy handling, added full non-normative WS-Security example in an Appendix. Added diagram for Grid usage scenario. Changed syntax of must-be-present and must-not-be-present functions to take an XPath string rather than an <AttributeSelector> (since <AttributeSelector> returning non-leaf nodeset is undefined). Added limit-scope function for applying multiple constraints to a single XML node and its children. Added ipAddress-match function for matching IP addresses, subnets, and port ranges.
05	27 Jun 2005	Anne Anderson	Minor edits; changed legal notice. Added functions string-getKrb5SName, hexBinary-getCertExtensionValue, and string-to-uri. Removed WS-Security example, since that is now a separate document.
06	24 Oct 2005	Anne Anderson	Minor edits.

1225

E. Notices

1226 Copyright © 2004-2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054,
1227 U.S.A. All rights reserved.

1228 Sun, Sun Microsystems, the Sun logo and Java are trademarks or registered trademarks of Sun
1229 Microsystems, Inc. in the U.S. and other countries.

1230 THIS DOCUMENT IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS,
1231 REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF
1232 MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE
1233 DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY
1234 INVALID.