Using capability classes to classify and match CC/PP And UAProf profiles

Mark H. Butler
Client and Media Systems Laboratory
HP Laboratories Bristol
HPL-2002-89
April 16th , 2002*

E-mail: mark-h_butler@hp.com

device independence, composite capabilities / preferences profile (CC/PP), resource description framework (RDF), wireless access protocol (WAP), user agent profile (UAProf), capability classes

In order for a web server to provide optimised content to different client devices it requires a description of the capabilities of the client known as the delivery context. In previous work we demonstrated DELI, an open-source library that allows Java servlets to resolve HTTP requests containing delivery context information in CC/PP or UAProf formats. Subsequently DELI has been incorporated into the Apache Cocoon XML publishing framework in order to demonstrate how delivery context information can be used in conjunction with content transformation via XSLT. During this work, it was found that it is cumbersome to match this information using constraints written in XPath. Furthermore there is no easy method of abstraction so that common sets of constraints may be reused in multiple stylesheets. This report describes an alternative mechanism for delivery context matching called *capability classes* (patent pending). This report outlines how to implement capability classes and how they may be applied to various content specialization techniques such as content transformation, negotiation or generation. It also compares and contrasts capability classes with *device classes* and *media queries.*

Using capability classes to classify and match CC/PP and UAProf profiles

Mark H. Butler
mark-h_butler@hp.com
18 / 02 / 2002

Abstract
In order for a web server to provide optimised content to different client devices it requires a description of the capabilities of the client known as the delivery context. In previous work we demonstrated DELI, an open-source library that allows Java servlets to resolve HTTP requests containing delivery context information in CC/PP or UAProf formats. Subsequently DELI has been incorporated into the Apache Cocoon XML publishing framework in order to demonstrate how delivery context information can be used in conjunction with content transformation via XSLT. During this work, it was found that it is cumbersome to match this information using constraints written in XPath. Furthermore there is no easy method of abstraction so that common sets of constraints may be reused in multiple stylesheets. This report describes an alternative mechanism for delivery context matching called *capability classes* (patent pending). This report outlines how to implement capability classes and how they may be applied to various content specialisation techniques such as content transformation, negotiation or generation. It also compares and contrasts capability classes with *device classes* and *media queries*.

Keywords
Device Independence, Composite Capabilities / Preferences Profile (CC/PP), Resource Description Framework (RDF), Wireless Access Protocol (WAP), User Agent Profile (UAProf), Capability Classes

1 Introduction

Different web-enabled devices have different input, output, hardware, software, network and browser capabilities. In order for a web server to provide optimised content to different clients it requires a description of the client capabilities. Recently two new compatible standards have been created for describing delivery context based on the Resource Description Framework (RDF)[1]: Composite Capabilities / Preferences Profile (CC/PP) [2] created by the W3C and User Agent Profile (UAProf) created by the WAP Forum[3].

Previous work[4] described DELI, an open-source library developed at HP Labs that allows Java servlets to resolve HTTP requests containing CC/PP or UAProf information. DELI has subsequently been incorporated[5] in the Apache Cocoon XML publishing framework[6] in order to demonstrate how CC/PP or UAProf information can used in conjunction with content specialisation methods such as content transformation via XSLT[7],[8]. Typically this requires creating conditionals in XSLT that query the profile using a related standard called XPath[9]. During this work, it was found that specifying constraints for matching device profiles in XPath is complicated and cumbersome. Furthermore there is no easy method of abstraction so that common sets of constraints may be reused in multiple stylesheets. This report describes an alternative mechanism for profile matching called *capability classes*. This works as follows: a number of capability classes are defined where each class is associated with a set of constraints. When a server receives a profile, it evaluates each set of constraints to determine if the target device belongs to one or more of the capability classes. Once it has determined which capability classes are supported by the device, this information is passed to the stylesheet to guide transformation. The same mechanism may also be used for other types of content specialisation such as selecting stylesheets, performing content negotiation or content generation.

This report is structured as follows: first it outlines an existing method of matching devices to transformations called *device classes*. This method is compared and contrasted with the new capability class solution. Next it explains how to specify conditionals based on device profiles using XPath within XSLT stylesheets. Then it describes a method of implementing capability classes that does not require complex constraints in XPath. The advantages of this approach are outlined and there is an explanation of how it may be used with various methods of content specialisation. Finally capability classes are compared and contrasted with *media queries*, a new mechanism for adapting web content for specific target devices within cascading stylesheets. The report suggests that ideally media queries and capability classes should be combined to provide a single approach to adapting and styling content based on device capabilities at both the client or the server. It is also proposed this work should leverage proposals for a modular set of capability vocabularies within the W3C Device Independence activity.

2 Device Classes

Many current approaches to device independence[10] that provide content for different devices e.g. PCs, phones, PDAs etc use an abstraction called *device classes*. Device classes are often used to map a specific device onto a transform that adapts content for the target device. This works as follows: each device or browser is associated with an identifier called a *user-agent string* that is unique to that make, model or version. When the device makes a HTTP request to the server, it includes the user-agent string in the HTTP request headers e.g.:

```
User-agent: Mozilla/4.04 (X11; I; SunOS 5.4 sun4m)
```

Unfortunately the user-agent string may not be unique as several devices use *cloaking*. This is when a device or browser, e.g. Microsoft Internet Explorer, claims to be another browser, e.g. Mozilla, in order to ensure web servers will send it the correct content. Therefore when we use the user-agent string in this context we need to disambiguate any cloaking that may be occurring. When the server has received the request, it looks up the user-agent string in a database. Typically disambiguation is done by searching for devices in a specific order so that impersonating devices are identified prior to the devices they are trying to impersonate. This enables the database to map user-agent strings onto device classes. In an XML publishing framework content, represented as XML[11], is then transformed using an XSLT stylesheet that has been identified using the device class of the requesting device and the item of content requested.

There are a number of problems with this approach. Firstly it does not scale well for a large number of capabilities. In *content negotiation*[12], a specific type of content specialisation, if a capability is critical to content specialisation and cannot be inferred from another capability then it is commonly referred to as an *axis of negotiation*. With device classes, the number of classes increases factorially with the number of axes of negotiation. Therefore if we use a one to one mapping between device classes and XSLT transforms, a small number of axes of negotiation will require a large number of device classes and hence a large number of XSLT stylesheets. For example consider a situation where we need to consider two axes of negotiation: screen size and keyboard type. For simplicity we assume that screen size can take three values (*small*, *medium* and *large)* whereas keyboard can take two values (*QWERTY* or *keypad)*. To represent all the possibilities we need six device classes i.e. *small QWERTY*, *medium QWERTY*, *large QWERTY*, *small keypad*, *medium keypad*, *large keypad*[i]. This means we need to define six stylesheets, one for each class. If we use different schemes for marking up different pages of content, we may need six stylesheets for each page of XML content.

By contrast, the capability class approach scales much better for large numbers of capabilities. Here instead of using monolithic descriptions of web pages, we break those web pages down into component parts. Typically we try to break the page into resources where each resource uses a single modality. This means we only need to consider a small subset of the available capability classes for any specific component resource as it only uses a single modality. For example in the above scenario in order for the server to decide what size of graphic to include in a web page we would need to specify alternates or transforms for small, medium and large devices in advance. Alternatively for the server to decide what kind of input method to use we would need to specify alternates or transforms for QWERTY input and keypad input. In the first situation we need to specify three alternates rather than six whereas in the second situation we need to specify two alternates rather than six.

One solution previously proposed to reduce the complexity of device classes is to use device class hierarchies in order to reduce the complexity of specifying content specialisation methods for multiple devices. For example we might use a class hierarchy as shown in Figure 1. This can be used with stylesheets as follows: we use one stylesheet to convert content to HTML and one stylesheet that converts it into WML. Once content is in HTML, it can receive additional content specialisation depending on whether the target browser is Netscape or Microsoft IE. This approach reduces replicated code in the stylesheets. However we still require a large number of stylesheets to support a few devices - in this case we need six to support four devices. In addition determining what styling is common to all HTML devices whereas what styling is common to a specific HTML device can sometimes be trial and error. Capability classes, by contrast, can be thought of as providing a multiple inheritance mechanism as shown in Figure 2.

A second problem with device classes is that the device capability information is implicit in the mapping from devices to device classes to transforms. For an example of implicit information, consider a web author creating some content. Typically when authors do this they are creating the content for a specific device, i.e. the device they are using, which uses a specific browser, has a specific screensize, color capability

[i] Often device class solutions choose to ignore some possible combinations to reduce the number of device classes. However those assumptions may not be justified - for example an Internet TV might well qualify as a large keypad device, whereas a mobile phone like the Ericsson R380s might qualify as a medium keypad device.

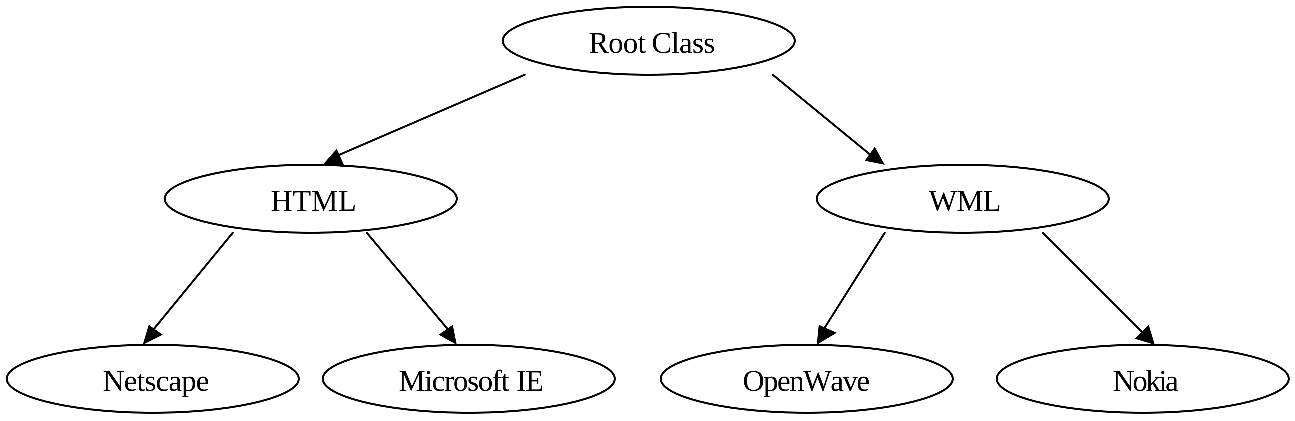

Figure 1 - Device class hierarchies

etc. However generally this information is not noted anywhere so can be regarded as implicit. In the same way in most device class aware websites, the fact that a device has a screen of a certain capability is never recorded explicitly; just the design descisions that are made as a result of that capability.

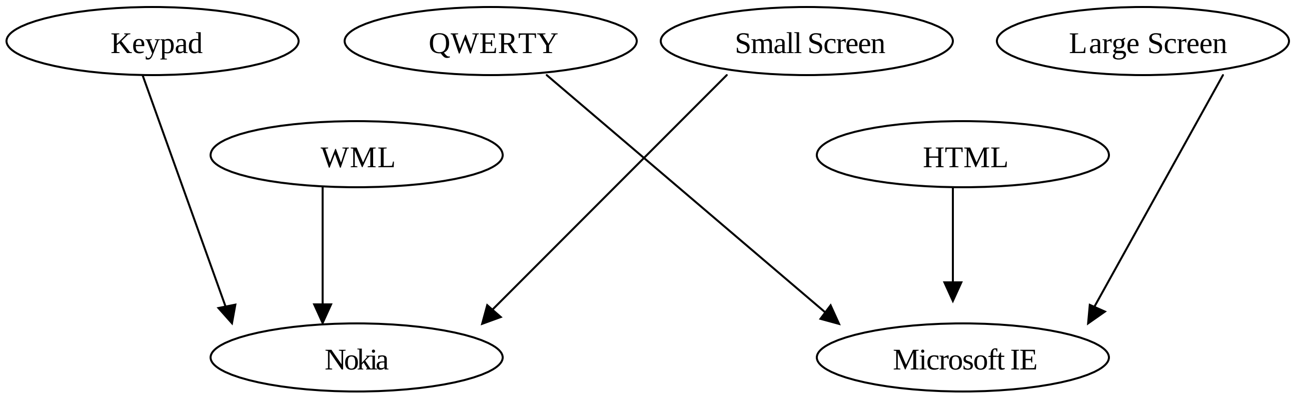

Figure 2 - Capability class hierarchies

A third problem with device classes is that devices have to be included in the device identifier database in advance. If the server encounters a device with a user-agent string it has never encountered before then it cannot classify the device. By contrast in the capability class approach, the device capabilities are made explicit via the device profile and the capability class constraints are made explicit via the capability class constraints definition. Therefore capability classes can cope with new devices they have never encountered as long as the device has a suitable device profile.

3 Using CC/PP profiles in XSLT

As previously noted DELI has been incorporated into Apache Cocoon, an open-source XML publishing framework, to demonstrate how CC/PP and UAProf information may be used in conjunction with XML and XSLT. One problem with manipulating CC/PP or UAProf profiles in XSLT is that these profiles are represented using RDF. Although RDF models can be represented in an XML serialisation, it is difficult to manipulate this serialisation in XSLT as it can represent the same model in many different ways. Models may vary depending on whether they use elements or attributes to indicate properties. Furthermore typically the XPath expression necessary to query a certain property value may not be representative of the underlying RDF structure of the profile. In order to avoid these problems DELI creates a “flattened” version of the UAProf or CC/PP profile available to XSLT stylesheets via a parameter called `deli-capabilities`. The profile is “flattened” because it is just a list of profile properties as XML elements without any component definitions or resource typing. The only obvious remnant of RDF is the way individual attribute values for complex attributes are separated using `<li>` elements. This heavily simplified profile form has the additional advantage of making the XPath expression correspond to the profile structure. For example the following profile demonstrates the flattened form:

```
<browser>
  <ScreenSize>90x120</ScreenSize>
  <IsColorCapable>Yes</IsColorCapable>
  <CcppAccept>
    <li>text/html</li>
    <li>text/plain</li>
    <li>image/jpeg</li>
  </CcppAccept>
<browser>
```

Flattening profiles in this way is not ideal because it means stylesheet authors encounter profiles in non-standard form. However it does solve the problems associated with processing profiles in XSLT. The following stylesheet demonstrates how we can use XPath conditional to query profiles within XSLT. For example the following stylesheet only generates a WML page if the device is WML capable, colour capable and has a screen size 90x120 pixels.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:param name="deli-capabilities"/>
  <xsl:template match="/">
     <xsl:if test="contains($deli-capabilities/browser/CcppAccept,’wml’) and
       contains($deli-capabilites/browser/ScreenSize,'90x120') and
       contains($deli-capabilities/browser/IsColorCapable,’Yes’)">
      <wml>
        <card id="init" newcontext="true">
          <p>Color device with 90x120 screen</p>
        </card>
      </wml>
     </xsl:if>
  </xsl:template>
</xsl:stylesheet>
```

In addition to the contains() function we can also use the >, >=, <, <=, =, and != expressions in conditionals. However UAProf, one important variant of CC/PP, uses various data types that are difficult to process using these conditionals. Firstly UAProf has a data type called dimension that consists of two numbers separated by an x e.g. 90x120. It is not possible to apply numerical expressions to this data type, so only the contains() function may be used. Secondly numbers in UAProf are integers, so instead of representing version numbers as numbers they are represented as string literals. Again it is desirable to have some conditionals specifically for this data type such as isBackwardsCompatible.

In addition, typically these conditionals must be duplicated many times in XSLT files as there is no easy way of abstracting the conditionals apart from using generating stylesheets using stylesheets[13]. As tweaking these conditionals may considerably impact how well the site works with certain target devices, it is highly desirable to be able to manipulate them using a level of abstraction.

4 Implementing Capability Classes

Capability classes overcome a number of problems described above. They avoid the need to "flatten" profiles to simplify their use in XSLT. They can incorporate new conditionals not supported in XPath that can manipulate new data types used in profiles. They also provide a means of abstraction so that constraints only need to be defined once as opposed to every XSLT file. Finally they may be used with other methods of content specialisation apart from content transformation.

Capability classes work as follows: a capability class definition file specifies each class name along with a set of constraints for that class. For example consider the file shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<classes>
  <class name="smallScreen">
    <or>
     <lessthan value="160x160">ScreenSize</lessthan>
     <lessthan value="20x20">ScreenSizeChar</lessthan>
    </or>
  </class>
  <class name="largeScreen">
    <or>
     <greaterthan value="320x240">ScreenSize</greaterthan>
     <greaterthan value="80x40">ScreenSizeChar</greaterthan>
    </or>
  </class>
  <class name="jpegcapable">
    <contains value="image/jpeg">CcppAccept</contains>
  </class>
  <class name="color">
    <true>ColorCapable</true>
  </class>
  <class name="blackandwhite">
    <not>
     <true>ColorCapable</true>
    </not>
  </class>
  <class name=”colorphone”>
    <and>
     <lessthan value=”90x120”>ScreenSize</lessthan>
     <contains value=”wml”>CcppAccept</contains>
     <true>IsColorCapable</true>
    </and>
  </class>
</classes>
```

This file defines four capability classes: *smallScreen*, *largeScreen*, *jpegcapable* and *color*. In the case of *smallScreen*, the constraints are that the device has a screen smaller than 160 wide and 160 pixels high or if it has a screen that is smaller than 20 characters wide and smaller than 20 characters high. Alternatively a device meets the *jpegcapable* capability class criteria if it can display the MIME type image/jpeg.

Capability class files can contain three Boolean expressions for aggregating constraints: *and*, *or* and *not*. It provides a number of conditionals: *lessthan*, *lessthanequals*, *greaterthan*, *greaterthanequals*, *equals*, *contains* and *true*. Each conditional is only applicable to specific attribute types as shown in the following table. For Dimensions, lessthan means the attribute will fit inside the value it is being compared to i.e. both axes are smaller whereas morethan means the attribute will encompass the value it is being compared to i.e. both axes are bigger.

Conditional	**Compatible UAProf data types**
`lessthan`	number, dimension
`lessthanequals`	number, dimension
`greaterthan`	number, dimension
`greaterthenequals`	number, dimension
`equals`	number, dimension, single literal
`contains`	set of literals, sequence of literals
`true`	boolean

The CC/P P (or UAProf) processor uses the capability class description file as follows: it parses the file and constructs a postfix description[14] of each set of constraints. It stores this postfix description in a vector for evaluation later. For example the colorphone class in the previous capability class definition file is represented as:

expression type	**expression**	**profile attribute**	**value**	**children**
operator	lessthan	ScreenSize	90x120	
operator	contains	CcppAccept	wml	
operator	true	IsColorCapable		
operand	and			3

The processor evalutes the postfix description of a set of constraints by retrieving each operator or operand in turn from the vector, evaluating it and then writing the result back to a results stack. In the case of the colorphone class it examines *ScreenSize* and determines if it is less than 90x120. If both values are less than 90x120, then it pushes True on to the results stack otherwise it pushes False. The processor then determines if *CcppAccept* contains the value WML and writes the result to the results stack. Then it determines if *IsColorCapable* is True and again writing the result to the results stack. Then it pops the previous three values from the results stack, applies the AND operand and writes the result back to the result stack. This indicates if the device is a member of the colorphone capability class. The processor repeats this process for the postfix description of each capability class and returns a vector containing the names of any capability classes where the device meets the associated constraints.

We may wish to make decisions within stylesheets based on a device's capability classes. In XSLT stylesheets this is done using a mechanism called *modes*. Modes are rules that are only executed when called directly. For example consider the stylesheet below.

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
        <xsl:param name="capabilities"/>
        <xsl:template match="/">
                <xsl:if test="contains($capabilities,'wmlDevice')">
```

```
                        <xsl:call-template name="wmldevice"/>
                </xsl:if>
                <xsl:if test="contains($capabilities,'pdaDevce')">
                        <xsl:call-template name="pdadevice"/>
                </xsl:if>
                <xsl:if test="contains($capabilities,'voice')">
                        <xsl:call-template name="voice"/>
                </xsl:if>
        </xsl:template>
        <xsl:template name="wmldevice">
                <!-- styling for wmldevice here -->
        </xsl:template>
        <xsl:template name="pdadevice">
                <!-- styling for pdadevice here -->
        </xsl:template>
        <xsl:template name="voice">
                <!-- styling for voice here -->
        </xsl:template>
</xsl:stylesheet>
```

Here the capability class list for the device is made available via the *xsl:param* statement. The stylesheet consists of four rules indicated by *xsl:template*. One rule, the default rule, indicated by *match="/"*, is executed for all documents. The other three rules, *wmldevice, pdadevice* and *voice*, are modes so have to be called within the default rule. Which rules are called depends on the capability classes possessed by the device. In the example stylesheet these rules just contain comments without any actual code. If we assume that the *wmlDevice* and *pdaDevice* capability classes are mutually exclusive i.e. a device can belong to one or the other but not both, then this means the stylesheet supports four different device classes i.e. combinations of capability classes e.g. *wmlDevice* with *voice*, *wmlDevice* without *voice*, *pdaDevice* with *voice* and *pdaDevice* without *voice*. In this way stylesheets can support multiple capability classes and capability classes can be combined.

In the future it is our intention to add support for capability class to other parts of Cocoon. Capability classes could be used in conditionals in the sitemap, a configuration file that defines how particular resources are generated, so that resources are generated in a specific way for a particular capability class e.g.

```
<map:match pattern="deli.wml">
  <map:capabilityclass type="colorphone"/>
  <map:generate src="docs/samples/hello-page.xml"/>
  <map:transform src="stylesheets/deli_test.xsl" type="xslt"/>
</map:match>
```

In the example sitemap fragment above a mechanism is defined for creating a resource called *deli.wml* if it is requested via a device that belongs to the colorphone capability class. This resource is generated from *hello-page.xml* by applying the *deli_test.xsl* XSLT stylesheet. Using capability classes rather than user agent strings reduces the number of mappings required due to the way capability classes can be used to generalise across devices. Alternatively capability classes can be used to transcode resources e.g. an image could be transcoded to change the target format and resolution. The transcoder could be configured as follows:

```
<generateImage>
        <urlmatch>managers/keegan</urlmatch>
        <content>managers/picture/KevinKeegan.jpg</content>
        <transcodings>
                <transcode>
                        <capabilityClass>wmlDevice</capabilityClass>
                        <targetFormat>wbmp<targetFormat>
                        <targetSizeX>60</targetSizeX>
                        <targetSizeY>60</targetSizeY>
                        <targetBitDepth>1</targetBitDepth>
                        <colorType>monochrome</colorType>
                </transcode>
                <transcode>
                        <capabilityClass>pdaDevice</capabilityClass>
                        <targetFormat>gif<targetFormat>
                        <targetSizeX>100</targetSizeX>
                        <targetSizeY>100</targetSizeY>
                        <targetBitDepth>4</targetBitDepth>
                        <colorType>greyscale</colorType>
                </transcode>
        </transcodings>
</generateImage>
```

This says when the server receives a request for the resource *managers/keegan*, it is to be created by transcoding the resource *managers/pictures/KevinKeegan.jpg*. If the target device belongs to the *wmlDevice* capability class, then the image is transcoded to produce an image in the wbmp format which is 60 pixels wide by 60 pixels high and contains one bit per pixel i.e. is a monochrome image. Alternatively if the target device belongs to the *pdaDevice* capability class then the image is transcoded to produce an image in the GIF image format which is 100 pixels wide by 100 pixels high and contains 4 bits per pixel i.e. is a greyscale image.

5 Using capability classes with different vocabularies

The capability class examples above use the data types available in UAProf. There are a couple of problems with this. Firstly, as already noted, UAProf has a Dimension data type so it is necessary to overload existing conditionals so they a slightly different meaning when applied to this data type. Ideally this data type should be avoided when creating new vocabularies so that overloading is not necessary. For example instead of using *ScreenSize* use *ScreenSizeHeight* and *ScreenSizeWidth*. Secondly UAProf uses the literal data type to store version numbers. If capability classes are to be used with UAProf ideally UAProf should create a new data type for version numbers and then a new conditional could be created for this data type e.g. isBackwardsCompatible. Thirdly there is a potential problem with applying capability classes to different vocabularies as each vocabulary may define its own set of data types. One way to resolve this would be for CC/PP to define a fixed set of data types that are available to vocabularies. Restricting CC/PP in this way is necessary for other reasons e.g. ensuring CC/PP vocabularies are compatible with all CC/PP processors. Capability classes could then offer a set of conditionals that operate on these data types that could be guaranteed to work across all vocabularies.

6 Relationship between capability classes and media queries

Within the work on cascading stylesheets[15], media queries[16] have been proposed as a mechanism to enable presentations to be tailored to a specific range of output devices without changing the content itself. A *media query* consists of a media type and one or more expressions involving media features e.g.

```
<link rel=”styleshet” media=”screen and (color)”
 href=”http://style.com/color” />
```

```
<link rel="stylesheet" media="aural and (min-device-width: 800px)"
  href="http://style.com/aural" />
```

There are many similarities between media queries and capability classes. Firstly a media query, like a capability class, is a logical expression that is evaluated to be either true or false. However they are only evaluated at the client whereas capability classes are evaluated at the server or proxy so are reliant on mechanisms such as CC/PP to send the capability information from the client to the server. Media queries also use the operands AND, OR and NOT just like capability classes but unlike capability classes they combine conditionals and attributes together to form an expression that can be directly evaluated e.g. min-height, max-height, min-device-width, max-device-width etc. In media queries the conditional is known as a *pretext* ; this approach has been taken to avoid the use of the "<" and ">" characters which conflict with HTML and XML. One potential problem is that media queries only offer three conditionals: min, max and the implicit equals. As we have already noted we may need a richer set of conditionals if we need to use a variety of data types in device profiles. Another difference is that we propose capability classes may be defined separately from style or content information. This has been done to provide a level of abstraction so that capability class definitions are reusable and so that capability classes may be easily integrated with existing device independence techniques. It is not clear if media queries can be used in the same way.

Ideally, in the author's opinion, work on media queries and capability classes should be combined to create a transparent mechanism of querying device capabilities that can be used at both the client and the server. There are two issues that need to be resolved in order to achieve this goal: firstly CSS media queries are based on a non-XML syntax; for example you need to parse expressions to isolate the pretext from the attribute name using a text parser. This is in contrast to capability classes which have deliberately adopted an XML syntax so they are processable with an XML parser. It is not clear which approach is better. Secondly capability classes do not define a common vocabulary whereas media queries do define a common vocabulary. In the authors opinion although it is desirable to have a common vocabulary, it is not desirable for it to be created within the cascading stylesheets activity. Furthermore the author would like to suggest rather than creating a single monolithic vocabulary, namespacing should be use to create a modular vocabulary. This common vocabulary should have a number of properties: firstly it would be possible to use it in conjunction with CC/PP profiles. Secondly if a modular approach was adopted, each module could correspond to a particular input or output modality used by devices. As new modalities become available on devices, these modalities can be encompassed by extending the modules available. Furthermore neither capability classes nor media queries currently consider namespacing. However namespacing is an essential concept in CC/PP and UAProf so it is desirable that the technique is namespace aware.

7 Conclusions

Capability classes make it easy to classify the capabilities of different devices and use these classifications with XSLT or with other content specialisation mechanisms such as configuration files that describe how resources are selected or generated via transformation and transcoding. They have a number of advantages over the previous device class methods of classifying devices. A sample implementation of capability classes for evaluating this technique is currently available in the DELI library and will shortly be made available within the Cocoon framework via DELI.

[1] *Resource Description Framework*, World Wide Web Consortium, http://www.w3.org/RDF/
[2] *Composite Capabilities / Preferences Profile*, World Wide Web Consortium http://www.w3.org/Mobile/CCPP/
[3] *Wireless Application Forum*, http://www.wapforum.org/
[4] *HPL-2001-260 DELI: A Delivery Context Library for CC/PP and UAProf*, Mark H. Butler, http://www-uk.hpl.hp.com/people/marbut/DeliUserGuideWEB.htm
[5] *Documentation describing Cocoon / DELI integration*, Mark H. Butler, http:// xml.apache.org/cocoon/developing/deli.html
[6] *Apache Cocoon*, Apache Software Foundation http://xml.apache.org/cocoon/
[7] *What is XSL*, World Wide Web Consortium http://www.w3.org/Style/XSL/WhatIsXSL.html
[8] *XSLT*, World Wide Web Consortium http://www.w3.org/Style/XSL/
[9] *XPath*, World Wide Web Consortium http://www.w3.org/TR/xpath
[10] *Current Technologies For Device Independence*, Mark H. Butler, http://www.hpl.hp.com/techreports/2001/HPL-2001-83.html
[11] *XML*, World Wide Web Consortium, http://www.w3.org/XML/
[12] *RFC 2616: HTTP 1.1 Content Negotiation*, page 70-73, The Internet Society, ftp://ftp.isi.edu/in-notes/rfc2616.txt
[13] *Improve your XSLT coding in five ways*, Benoit Marchal, (See Tip 5), http://www-106.ibm.com/developerworks/library/x-xslt5.html
[14] *Postfix notation*, Bob Brown, Southern Polytechnic State University http://www.spsu.edu/cs/faculty/bbrown/web_lectures/postfix/
[15] Cascading stylesheets, http://www.w3.org/Style/CSS/
[16] Media Queries, http://www.w3.org/TR/css3-mediaqueries/