ETSI TS 102 523 V1.1.1 (2006-09)

Technical Specification

Digital Video Broadcasting (DVB); Portable Content Format (PCF) specification 1.0

Reference
DTS/JTC-DVB-173

Keywords
broadcasting, digital, DVB, TV, video

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (http://webapp.etsi.org/IPR/home.asp).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by Joint Technical Committee (JTC) Broadcast of the European Broadcasting Union (EBU), Comité Européen de Normalisation ELECtrotechnique (CENELEC) and the European Telecommunications Standards Institute (ETSI).

NOTE: The EBU/ETSI JTC Broadcast was established in 1990 to co-ordinate the drafting of standards in the specific field of broadcasting and related fields. Since 1995 the JTC Broadcast became a tripartite body by including in the Memorandum of Understanding also CENELEC, which is responsible for the standardization of radio and television receivers. The EBU is a professional association of broadcasting organizations whose work includes the co-ordination of its members' activities in the technical, legal, programme-making and programme-exchange domains. The EBU has active members in about 60 countries in the European broadcasting area; its headquarters is in Geneva.

European Broadcasting Union
CH-1218 GRAND SACONNEX (Geneva)
Switzerland
Tel: +41 22 717 21 11
Fax: +41 22 717 24 81

Founded in September 1993, the DVB Project is a market-led consortium of public and private sector organizations in the television industry. Its aim is to establish the framework for the introduction of MPEG-2 based digital television services. Now comprising over 200 organizations from more than 25 countries around the world, DVB fosters market-led systems, which meet the real needs, and economic circumstances, of the consumer electronics and the broadcast industry.

Introduction

The DVB Portable Content Format (PCF) is a standard means to describe an interactive digital television (iTV) service. It provides the industry with a platform-independent format for the business-to-business interchange of interactive content, and consequently a means to increase the interoperability of authoring tools, head-end systems and broadcast networks.

The PCF allows an iTV service description to be authored independently of specific target platforms. This is achieved by capturing the intended viewer experience rather than how it shall be implemented, allowing the greatest possible portability.

A description captured using the PCF is not intended for actual transmission in a digital television network and the form of the PCF reflects this. Rather, it is an intermediate form that needs to be converted to some platform-specific representation prior to transmission.

The PCF will allow a broad range of interactive services to be deployed on multiple target platforms, including MHP and legacy platforms, with a minimum of re-authoring. A description captured using the PCF will be practical to convert by machine, providing application developers with a degree of independence from target platforms.

The PCF is a data format for the description of an interactive service. It is not an authoring tool, though authoring tools may be developed that write output data according to the PCF. It is not the specification of a technology to be deployed in a digital television receiver; DVB already has such a technology, the Multimedia Home Platform (MHP).

The development of a proposition to be delivered to multiple target platforms can be challenging because of the variation between platforms. This can be due to:

- Physical constraints, such as a platform not having a return path.
- Commercial constraints, such as applications from a particular provider not having permission to access certain platform resources (e.g. persistent storage).
- Operational constraints, such as part of the screen being reserved by a platform operator for special navigation features or branding.
- Inherent differences between platforms, such as the available fonts and text rendering rules.

The consequence is that for some interactive services it is currently difficult to deliver an identical experience to the viewer independent of platform, no matter how authored. However, it is generally possible to offer, as determined by the interactive service provider, a sufficiently similar experience to the viewer independent of platform. The PCF provides a means to allow application authors to describe the intended experience, along with additional facilities to assist in dealing with platform-to-platform variations.

The PCF provides the industry with a tool to facilitate co-existence with and migration to MHP by enabling the exchange of interactive television service descriptions across multiple platforms. This will also minimize the total end-to-end cost of deployment of interactive digital television services across multiple platforms, increasing the reach of an interactive digital television service authored to the format.

1 Scope

The present document specifies the DVB Portable Content Format (PCF) which provides an interactive television (iTV) service description format. This has been designed in response to the commercial requirements given in DVB CM-MHP-0651 v2.0.

2 References

The following documents contain provisions which, through reference in this text, constitute provisions of the present document.

- References are either specific (identified by date of publication and/or edition number or version number) or non-specific.
- For a specific reference, subsequent revisions do not apply.
- For a non-specific reference, the latest version applies.

Referenced documents which are not found to be publicly available in the expected location might be found at http://docbox.etsi.org/Reference.

[1] IETF RFC 2045: "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies".

[2] IETF RFC 2046: "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types".

[3] IETF RFC 2048: "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures".

[4] W3C XML Schema 2: "XML Schema Part 2: Datatypes second edition".

NOTE: http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[5] W3C XML 1.0:"Extensible Markup Language (XML) 1.0 (third edition)". .

NOTE: http://www.w3c.org/TR/2004/REC-xml-20040204/

[6] ISO 10646-1: "Information technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane".

[7] ETSI TS 102 812: "Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.1".

[8] W3C HTML 4.01: "HTML 4.01 Specification".

NOTE: http://www.w3.org/TR/1999/REC-html401-19991224

[9] IETF RFC 2396: "Uniform Resource Identifiers (URI): Generic Syntax".

[10] IETF RFC 2141: "URN Syntax".

[11] GIF Compuserve Incorporated Version 89a: "Graphics Interchange Format"; Columbus, Ohio.

[12] W3C PNG: "Portable Network Graphics (PNG) Specification (Second Edition) Information technology - Computer graphics and image processing - Portable Network Graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E)".

NOTE: http://www.w3.org/TR/PNG/

[13] ISO/IEC 10918-1: "Information technology -- Digital compression and coding of continuous-tone still images: Requirements and guidelines".

NOTE: http://www.w3.org/Graphics/JPEG/itu-t81.pdf

[14] IETF RFC 3003: "The audio/mpeg Media Type".

[15] ISO/IEC 13818-2: Information technology - Generic coding of moving pictures and associated audio information: Video (MPEG-2 Video)".

[16] IETF RFC 3339: "Date and time on the Internet: Timestamps".

[17] EBU TS N12-1999: "Time-and-control codes for television recording".

[18] ANSI/SMPTE 12M-1999: "Television, Audio and Film - Time and Control Code".

[19] W3C SOAP: "SOAP Version 1.2 Part 1: Messaging Framework".

NOTE: http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[20] ETSI ES 201 812: "Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.0.3".

[21] IETF RFC 2616: "Hypertext Transfer Protocol - HTTP/1.1".

[22] W3C CSS2: "Cascading Style Sheets, level 2: CSS2 Specification".

NOTE: http://www.w3.org/TR/1998/REC-CSS2-19980512/

[23] W3C Canonical XML: "Canonical XML: Version 1.0". .

NOTE: http://www.w3.org/TR/2001/REC-xml-c14n-20010315

[24] Openwave WMRF-13-002 (Openwave Systems Inc. October 2001): "WML 1.3 Language Reference".

NOTE: http://developer.openwave.com/docs/51/wml_ref.pdf

[25] W3C XSL 1.0: "Extensible Stylesheet Language (XSL) Version 1.0".

NOTE: http://www.w3c.org/TR/2001/REC-xsl-20011015/

[26] IETF RFC 1505: "Encoding Header Field for Internet Messages".

[27] ETSI EN 300 468: "Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems".

[28] W3C XML: "Extensible Markup Language(XML) 1.1", clause 6.

NOTE: http://www.w3.org/TR/xml11

[29] IETF RFC 1591: "Domain Name System Structure and Delegation".

NOTE: ftp://ftp.rfc-editor.org/in-notes/rfc1591.txt

3 Definitions and abbreviations

3.1 Definitions

For the purposes of the present document, the following terms and definitions apply:

announcement: invitation displayed to the end user to initiate the PCF service

broadcast stream event: event delivered synchronously with the broadcast stream

coherence: condition whereby all direct and indirect references within the loaded assets of a PCF service description can be resolved

component: functional unit within a service description that can be uniquely referred to, and whose state may be accessed and in some cases modified

NOTE: Example components may include:

- high-level structural units such as scenes and tables;
- lower-level functional units such as menus, images and text boxes;
- custom units, typically with platform-dependent behaviour.

constant: value immutable for the life of the enclosing PCF service

container component: component that can contain other components

content: data available for presentation

data type: set of values from which a variable, constant or some other expression may take its value

evaluation: resolution of an expression into a value

event: occurrence during the lifetime of a service that any number of interested components may respond to

NOTE: Events may have associated parameters. Example events include user key press, change of state of return path connection, broadcast stream event.

information set; infoset: set of definitions for use in other specifications, including other parts of the PCF specification itself, that need to refer to information in the PCF description of an interactive service

NOTE: An infoset consists of information items which have information item properties.

interaction; interactivity; interactive: activity by which a viewer can influence the data and processing of an application

interactive engine: technology that is present in the receiver that supports the rendering of the interactive service, e.g. presentation engines, execution engines, middlewares etc.

interactive service; interactive digital television service: collection of audio-visual perceivable units, typically bundled together into a coherent entity, that can be rendered by an interactive digital television platform

NOTE: The content to be rendered may change over time in response to user interaction and other stimuli. The interactive service may form an essential or optional aspect of a DVB Service, may span a number of DVB Services or may be a DVB Service in its own right.

intrinsic: not directly under the control of the service author

layout: visual arrangement of screen elements within a scene

NOTE: Layout can be absolutely defined, where the service author defines pixel positioning for visible elements, or automatic, where the transcoder or user agent will determine layout at run-time, according to rules and behaviours.

loaded asset: PCF asset that has been successfully passed to a service registration context on a PCF transcoder via a service transaction

master asset: PCF source document within which the root component of the PCF service description is found

Octet data: packaging unit for non-XML PCF content

online update: periodic retrieval of a PCF asset by the transcoder during the lifetime of that asset

operator: mathematical operation such as +, -, ×, / etc. used in expressions to combine values

PCF asset: union of PCF source document and octet data

PCF entity: PCF item or octet data

PCF item: value of any PCF data type

PCF source document: packaging unit for PCF items

perceivable unit: set of material which, when rendered, may be perceived by a viewer and with which interaction may be possible

NOTE: Most perceivable units provide both presentation and the means for interaction. However, on some types of device, such as printers, perceivable units might contain only presentation.

platform; interactive digital television platform: combination of receiver and network infrastructure capabilities that enable the rendering of interactive services

profile: defined sub-set of the PCF specification that provides an easy way to describe the minimum capabilities, i.e. the minimum profile, that a platform must provide in order to achieve the authorial intent captured in the PCF description of an interactive service

NOTE: This allows an author to have confidence that their service description will work on a particular target platform, or more precisely a particular target device, without having to meticulously match up every feature they have used with the features provided by that target.

property: quantifiable characteristic of a component that may be referenced, and that may potentially be read and/or modified

NOTE: Example properties may include:

- The size and position of a visible component.
- The value of an integer variable.
- The text within a TextBox component.

push update: delivery of a PCF asset with a transaction message to a transcoder

pull update: retrieval of a PCF asset by the transcoder subsequent to a transaction message

receiver; interactive digital television receiver: device deployed within a particular interactive digital television platform capable of rendering an interactive service

NOTE: Depending on the platform, the device may have access to a unidirectional, e.g. broadcast, and/or a bi-directional, e.g. return path connection.

rendering: act of converting perceivable units into physical effects that can be perceived by a viewer and which may be interacted with

resource: can be anything that has an identity

resource resolver: function, typically residing within a PCF transcoder, that identifies and maps the portable resource references of a PCF-authored interactive service to the specific resources available on the target platform

NOTE: For example, "video_stream_2" in generic PCF may be transcoded by a resource resolver in a transcoder to "Transport_Stream_ID; Service_ID; Event; VideoElementary_Stream; Component_Tag_2" for a particular target platform.

return path: communications mechanism that provides a connection between a receiver and a remote server

run-time: duration of time between instantiation and termination of the user session with a PCF service

scene: components, content and layout that appear and remain on screen for a "discernable period of calm"

NOTE: A scene will generally not consist of a random collection of visible screen furniture, but rather will display qualities of graphic and visual internal consistency. A scene is not necessarily static. It is possible that some elements may change in a screen update without a scene transition occurring. For example, activation of a select box may be represented by a new popup object appearing on screen, but this would not be considered to represent a scene transition.

scene transition; transition: act of moving between scenes

NOTE: A scene transition will normally involve a screen refresh, where the screen will blank, then redraw to display the new scene, although it is also possible that some new piece of screen furniture could appear, or a visible piece could disappear, without a full screen refresh.

screen: physical viewing area of a television set

NOTE: It is possible that a screen may be physically smaller or larger (i.e. contain fewer or more pixels) than the scenes that comprise an interactive service.

script: sequence of combined actions and control statements including expression evaluations that is executed to change the state of a service, including any associated change to visual components displayed as part of the current scene

NOTE: A script may be triggered by an event or be executed as the result of another action.

service; interactive service: An overall user experience that display internally consistency from the user's point of view

NOTE: A single user experience may be built from a variety of sub services, which may provide discrete functional components.

service component; pcf service component: PCF component containing an internally consistent collection of PCF scenes, describing the author's intent

service provider; interactive digital television service provider: party responsible for the provision of an interactive service including any dynamic aspects such as real-time assets, e.g. video, sports scores, and personalized information, e.g. regional weather, bank statement.

session: duration of time in which a viewer interacts with a service

NOTE: Within that duration, the service would normally maintain a record of the state of the interaction to ensure presentation of appropriate content and scenes at the appropriate moments for each user's interaction.

table: tabular layout for content

NOTE: Two kinds of tables are supported in PCF - the PCF Table component, and tables within marked-up text.

target; target device; target platform: platform or device intended to deliver the interactive service described by PCF

NOTE: It is intended that PCF service descriptions should be transcoded into a native format for a target device.

transcoder: target platform specific functionality that is capable of ingesting a PCF service description, and automatically outputting a version of that service suitable for delivery to a specific viewing apparatus, such as digital TV set-top-box

Uniform Resource Identifier (URI): string that identifies a resource

user agent: client within an interactive digital television receiver that performs rendering

NOTE: Browsers are examples of user agents.

variable: quantity that can assume any of a set of values according to its data type and can be uniquely identified

NOTE: A variable can be instantiated at some point during an interactive service and is likely to vary in response to actions. A variable forms part of the state of a service.

visual component: visible screen object that delivers defined functionality

NOTE: Standard visual components include TextBox, Image and Video.

volatile asset: PCF asset that changes aperiodically over the service registration lifetime

x-height: measure of height of a typeface, based on the height of the lower case "x" character

NOTE: As defined in CSS2 http://www.w3.org/TR/REC-CSS2.

3.2 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Programming Interface
DVB	Digital Video Broadcasting
iTV	interactive TeleVision
MHP	Multimedia Home Platform
PCF	Portable Content Format
STB	Set-Top Box
TV	TeleVision
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
XML	eXtensible Markup Language
ELC	Explicit Layout Container
TFC	Truncate Flow Container
MIME	Multipurpose Internet Mail Extensions
OSD	On Screen Display
RPTF	Return Path Transaction Format

4 Conventions

All example XML service description fragments in the present document are assumed to have the following namespace declarations unless otherwise stated:

- xmlns="http://www.dvb.org/pcf/pcf" (pcf)
- xmlns:mut="http://www.dvb.org/pcf/x-dvb-pcf" (marked up text)

All example XML service digest fragments in the present document are assumed to have to following namespace declaration:

- xmlns="http://www.dvb.org/pcf/servicedigest" (service digest)

All components, property group, event and error specification XML fragments in the present document are assumed to have to following namespace declarations:

- xmlns="http://www.dvb.org/pcf/components" (component specification)
- xmlns:pcf="http://www.dvb.org/pcf/pcf" (pcf)

Grammar production syntax is specified in the present document using Extended Backus-Naur Form (EBNF) notation [28].

5 Service author guide (informative)

5.1 Introduction

This clause of the specification is intended to provide a high-level overview of the various elements available for use within the PCF, and how these are combined to form a complete interactive TV service description.

The PCF is an interactive television (iTV) service description format designed to be platform independent. A service described using the PCF can be automatically converted into a format suitable for any target platform.

The PCF accomplishes this by allowing the iTV service to be described using a high-level declarative syntax. The format describes the intended viewer experience without prescribing exactly how it should be rendered on a target platform. This means that any automatic conversion process has sufficient freedom to transform the description into the execution model required by the target platform. To enable a service to be described most efficiently, and in a way that best conveys the author's intent, the format uses a referencing mechanism. This allows the service description to be flexibly partitioned and gives the transcoder considerable freedom to decide the best way of packaging the service optimally to suit the target platform.

The form of the PCF reflects the fact that it is not intended as an actual transmission format, but rather as an intermediate format that will need conversion to a platform-specific representation prior to transmission. It is primarily intended for business-to-business interchange, and as such is meant to facilitate interoperability between authoring tools, head-end systems and broadcast networks.

To assist adoption of the PCF itself, the PCF specification describes standardized (but non-mandatory) mechanisms for profiling the PCF into sub-sets of features and for the interchange of service descriptions.

In addition to the rest of this clause a number of example PCF service description are provided in annex R.

5.2 An overview of a PCF service description

The following is an example of a simple PCF service:

```
<PCF xmlns="http://www.dvb.org/pcf/pcf">
    <Service name="hello_world">
        <String name="pcfSpecVersion" value="1.0"/>
        <URI name="firstScene" value="#hello_scene"/>
        <Scene name="hello_scene">
            <Component class="TextBox" name="hello_text">
                <Size name="size" value="100 40"/>
                <String name="content" value="Hello, World!"/>
            </Component>
            <OnEvent name="exit">
                <Trigger eventtype="KeyEvent">
                    <UserKey name="key" value="VK_PREV"/>
                </Trigger>
                <ExitAction/>
            </OnEvent>
        </Scene>
    </Service>
</PCF>
```

Whilst this service is very small, it uses many of the key PCF structures. The **Service** element provides the entry point into the service, and the rest of the service description is either contained, or referenced from, within this element.

Immediately within the Service element is a single **Scene** element. All PCF services are sub-divided into one or more Scenes, and each Scene represents a destination that may be navigated to within the service. The visual appearance of a Scene is defined by the set of **Components** the Scene contains. In this case there is only a single component, a **TextBox**, which is used to display the message "Hello, World!". PCF defines a toolkit of standard components that may be included within a Scene description to describe the desired appearance and behaviour.

All component types have a number of **Properties**. For the TextBox component, only its **size**, and its **content** are being explicitly set. A TextBox has a number of other properties that have not been explicitly set, and in this case their default values will be used.

The last element within the Scene describes an event handler, **OnEvent**. For this Scene the event handler responds to the viewer pressing the virtual key "VK_PREV" that will be mapped to a platform specific remote control key, e.g. the "Backup", "Back", "Previous", "Cancel". The event handler contains a single action, **ExitAction**, that instructs the service to exit.

Whilst this is a very simple service it does illustrate core aspects of a PCF service description:

- *Components* - the building blocks of a PCF service description.
- *Content* - managed and presented using PCF components.
- *Behaviour* - the response to events generated at run-time.

5.3 Components

Components are the building blocks of a PCF service description. Many components are visual "widgets" used to describe the visual appearance of the service at a particular point, for example TextBox, Video and Menu. There are also non-visual components, which can be used for the presentation of non-visual content or providing other aspects of service functionality, for example Audio, Stream and Random.

All component types may be declared using a generic *component item*. This has a *class* property to specify the type of component and a *provider* property to identify the body that has specified the component. For standard DVB components specified within the present document the provider is "dvb.org". However, for such components the provider property may be omitted from their declaration, since "dvb.org" is the default value for this property.

Each class of component has a set of associated named properties, and setting these properties defines how a particular instance of a component will look and behave. The following is an example of a standard DVB TextBox component declaration (where the provider attribute is omitted and so by default takes the value "dvb.org"):

```
<Component class="TextBox" name="MainText">
    <Size name="size" value="100 40"/>
    <Color name="fillcolor" value="#0000AA"/>
    <Color name="textcolor" value="#FFFFFF"/>
    <String name="content" value="Hello, World!"/>
</Component>
```

It should be noted that the *provider* property has not been explicitly stated in the declaration, since "dvb.org" is the default value for this property.

In this case four of the TextBox's properties are being explicitly set: its *size*, its *fillcolor* (background colour), its *textcolor* and the *content* string. As well as being defined during instantiation, many component properties can be dynamically changed at run-time. In the above example, the TextBox's *content* property can be updated to display a different block of text, or one of its *colour* properties can be updated to change that aspect of the component's presentation.

Using the generic *Component* provides a number of advantages regarding the extensibility of the PCF format, and the ability to declare custom components using the same syntax as standard PCF components. However, a number of standard components exist that may also be declared in a class specific way. These components are known as *schema components*, and always have the "dvb.org" provider. For example, the following example describes a TextBox that is equivalent to the Component of the TextBox class:

```
<TextBox name="MainText">
    <Size name="size" value="100 40"/>
    <Color name="fillcolor" value="#0000AA"/>
    <Color name="textcolor" value="#FFFFFF"/>
    <String name="content" value="Hello, World"/>
</TextBox>
```

Component types can be specified to have intrinsic behaviour that will cause them to change in response to external events, such as key presses, broadcast stream events, or a change in state of the return path connection. For example, a Menu component will react to specified key presses by changing the position of its menu item highlight.

Components can also be specified to generate events to signal some internal change. This provides a mechanism to allow the service description to contain some custom behaviour beyond that defined in the component specification. For example, if a visual component receives the focus it will generate an "OnFocus" event, that could be used to drive some custom "roll-over" behaviour, so that contextual information is presented in a TextBox (somewhere else in the Scene).

5.3.1 The Service component

The *service* component is a form of *explicit layout container* (see clause 5.3.3.1) within which *scene* components and other components that are required to persist between different Scenes may be described, for example, an IntegerVar component used to store a high-score. In addition to this, the Service component provides the entry point into the service by specifying the initial Scene to be presented.

5.3.2 The Scene component

The *scene* component is also a form of *explicit layout container* (see clause 5.3.3.1) within which other components may be described.

Normally, the lifetime of all components within a scene is the same as the lifetime of the scene itself. A transition from a scene will cause all child components and accompanying state to disappear. For this reason, any components that need to persist between scenes (for example, a background image or a high-score variable) must be included in the service component.

5.3.3 Layout components

5.3.3.1 Explicit layout

Explicit layout is described using the *ELC* (Explicit Layout Container) component. This component has no visual appearance of its own. It is instead a container for other components that need to be positioned at an exact location.

An ELC component has an *origin* property, but no size. Components declared within an ELC component have their positions specified relative to this origin. Consequently, moving the parent ELC component's origin will cause all the child components to move by the same relative amount. This is illustrated in figure 1. Diagram A shows three components which are contained within an ELC, the origin of which is also shown. In diagram B, the origin of the ELC has been moved, which has resulted in all its child components moving as well.

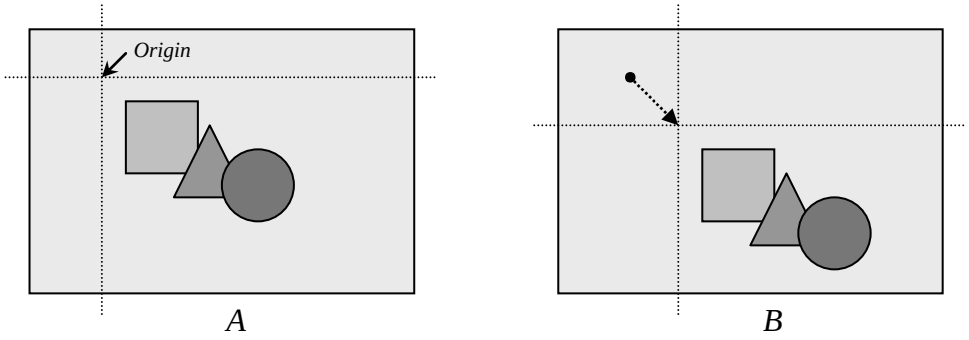

Figure 1: Moving an Explicit Layout Container

The z-order of child components contained within an ELC component is implicit from the order in which the child components are declared. This order can be modified at run-time.

When the visible property of an ELC is "true", all of its child components are visible. Similarly, when the visible property of an ELC is "false", none of its child components are visible.

There is a variant of the ELC component called the *StaticELC* component that does not support run-time modification to the position or z-order of child components. Whilst the StaticELC component is less capable than the ELC component it is more portable.

5.3.3.2 Flow layout

Flow layout is described using a combination of *Flow* components and *flow layout container* components.

Flow components are used to manage the content to be presented, which may be a mixture of text and other PCF components.

Flow container components are used to position the content described by a Flow component. This is achieved using "flow" rules, so that the content described by a Flow component is presented without requiring the service author to specify exact positions. This mechanism is very similar to the way the content within an HTML page is flowed by an HTML browser.

There are three type of flow container component:

- A **SFC** (*Scroll Flow Container*) component presents flowed content in a rectangular area of fixed width, height and origin. Flowed content that exceeds the limits of the SFC can be viewed by scrolling the content up and down.

- A **PFC** (*Page Flow Container*) component presents flowed content in a rectangular area of fixed width, height and origin. Flowed content that exceeds the limits of the PFC can be viewed by "paging" forward and back through the content.

- A **TFC** (*Truncate Flow Container*) component presents flowed content in a rectangular area of fixed width and origin, but variable height. The TFC may stretch, within author-specified limits, only in a downward direction as required to accommodate flowed content. Flowed content that exceeds the maximum size of the stretch container shall be truncated.

The content to be presented in any of the flow container components is described using one of the following flow components:

- A **Flow** component can contain a mixture of text and other PCF visual components.

- A **Table** component is a special instance of Flow where the flowed content is organized into a tabular structure.

- A **TextFlow** component is a restricted form of Flow that is limited to containing simple marked-up text.

5.3.4 Return path components

There are many instances when a service must send some data back to the head-end or content provider. This is accomplished, where possible, using some form of return path. These broadly fall into two categories:

- Always On Return Path (e.g. as seen in many xDSL / Cable Modem systems).

- Intermittent / On Request / Dial Up (e.g. as seen in many Satellite systems).

Both of these are catered for by the PCF.

Clearly there will be platforms where no return path exists and in such instances there are alternative methods by which a limited set of return path functionality can be achieved. For instance, the viewer using a particular service could be asked to phone a number, or send a text message using their mobile phone, which could be used at the head end to gather data from the interactive service. An example of this may be an interactive voting service. The ideal desired behaviour would be that the service returns the user's vote transparently, via an always on or dial up return path. However, on platforms where such return paths do not exist, or are not available, the user could be prompted to send an SMS text message stating "A", "B" or "C" to a given number using their mobile phone. The PCF, however, does not consider this to be return path functionality as it could equally be applied to all platforms, including those with always on and dial up return paths.

The standard return path implementation is built around three items: a ReturnPath component, a Transaction component and a *transfer collection*. A fourth component, the Indicate component, is a cut down version of the ReturnPath component, whose purpose is to a enable very simple return path functionality whereby a connection is made but no transaction occurs, such as is used in a very simple voting application. A fifth component, the *SecureReturnPath* component allows for the secure transfer of data to take place.

The data transfer collection defines a sequence of information to be transferred over the return path. Only serialized data can be transferred through the return path. Serialized data is a restricted set of data types, appropriate for transfer through the return path, for example integers, strings, variables, dates but **not** Rectangles, Buttons, Menus etc. The PCF specification defines whether or not a component type is serializable. The transfer of graphics images through the return path is not required in the initial version of the PCF and therefore images are not defined as serializable.

The ReturnPath component embodies the return path itself and manages the actual data transfer exchange process. This component has no visual representation and implements return path functionality for both always-on and dial-up return path platforms. Its properties incorporate information to define the target application server and the current status of the return path connection (i.e. "closed", "open", "opening", "closing"). At run-time the ReturnPath component is passed a reference to a data Transaction component that contains the source and/or destination data transfer collections. The Transaction component embodies the status of an actual data transfer exchange process. (i.e. "idle", "busy") and upon completion it generates an outcome event which defines the success or failure of the transaction.

PCF Profiles and levels, as described in clause 5.7.2 and clause 11 can be used to identify a platform's return path capabilities.

5.3.5 Custom components

The set of component classes standardized by DVB can be augmented by the creation of *custom components*.

A custom component can be created for a target platform to exploit platform capabilities that are not available through the use of standard PCF components alone. When using a custom component, the provider property must always be set explicitly.

NOTE: As custom components use the same component declaration as standard components they can be integrated seamlessly into the PCF service declaration. The responsibility then lies with the publication process to implement the component on the target platform correctly.

PCF service descriptions containing custom components will only work on target platforms that implement these components, and so are likely to be less portable than PCF services that use only standard components.

5.4 Content

Content is managed within a PCF service description using a set of standard PCF data types as follows:

- **Primitive types** - basic data types such as integer and Boolean.
- **Core types** - core data types such as Size, Colour or MarkedUpText, that incorporate one or more primitive data values, but which are core to the PCF.
- **Octet data** - this is encoded binary data. For example, octet data may be used to handle image data, stream data and character data in various encodings.
- **Compound Types** - These types contain a number of other values, ordered and accessed in a particular way. For example, key-value pairs, whereby the value may be looked up using the key.

The PCF provides means for items to be grouped together and referenced in the form of structures and arrays. Structures, such as the Collection data type, provide a mechanism for related data items to be accessed as a single logical unit. Arrays allow multiple items of the same data type to be grouped so they can be easily indexed and iterated over.

5.5 Behaviour

All run-time behaviour that contributes to the viewer experience of a PCF service is event-driven.

When an event occurs it may be consumed by a component resulting in the execution of behaviour defined as part of that component class specification. For example, the Menu component is specified such that it will react to the UP/DOWN user keys by moving its highlight up and down over its menuitems. The service author need not describe this behaviour.

Alternatively, the event may be consumed by a custom handler declared within the service description. This might be used to create custom behaviour that is not an inherent part of an available component's functionality, or for linking the behaviour of one component to that of another. For example, when a menuitem in the same Menu component receives focus it will generate an "OnFocus" event, that could be used to drive some custom "roll-over" behaviour, so that contextual information is presented in a TextBox (somewhere else in the Scene).

Many of the features available for defining behaviour within the PCF, whether as part of a component specification or declaring custom behaviour, are derived from the statemachine-oriented features available in UML statecharts. A set of states can be defined alongside a set of rules to define the conditions for making transitions between these states. A set of actions may then be defined for execution in response to specific events, or in response to a specific state transition.

5.5.1 Events

There are three categories of events:

- **System events** - occur without the viewer's intervention, for examplebroadacst stream events, error conditions etc.
- **Component events** - generated by a component to indicate a change to its internal state.
- **User input events** - generated in response to the viewer doing something, e.g. pressing a key on the remote control.

Depending upon the type of event and the state of the service, a generated event will be made available for consumption to certain components within the service description according to defined set of event propagation rules.

The PCF supports the concept of **focus.** Focus effectively represents where the viewer's attention is currently focused, and so user input events are initially directed at the focused component. If the focused component is not interested then the event is propagated up to its parent, and if not consumed there then to that component's parent and so on until it is consumed.

5.5.2 Action language

The PCF includes an action language to specify sequences of actions to be executed either in direct response to an event occurring, or as a result of a state transition caused by an event. The PCF Action Language consists of *actions*, *variables*, *conditions* and *controls*. These may be combined to form simple scripts.

Actions are essentially commands directed at specific components, including potentially the current scene and service components. For example, an action may query a component's property or set it to a specific value. In addition to reading and modifying component properties, the PCF action language supports actions for making scene *transitions*, performing simple mathematical and logical operations, manipulating strings and for executing component-specific commands.

Variables are used to store and retrieve data that is required to persist during the execution of a sequence of actions, or between successive sequences.

Conditions are used to make the execution of a particular sequence dependent upon some Boolean condition being met, allowing a choice of execution paths to be specified.

Controls are used in conjunction with conditions to further affect the flow of an action sequence's execution by specifying loop and jump behaviour. For example, a transition to the "Game Over" scene that occurs only once the value of the IntegerVar component containing the number of remaining lives equals zero.

By default, an action is executed within the scope of its parent component and can only "see" (i.e. access and modify) components that are also within the same scope. There are some situations where it is necessary to be able to access components that are outside this normal scope. For example, an action may need to access an integerVar component defined in the service component to update a game score that must persist between scenes. The PCF provides a mechanism, in the form of *parameter items*, to achieve this.

The following shows example use of some simple action language:

```
<Scene name="scene">
    <Collection name="variables">
        <IntegerVar name="counter">
            <Integer name="value" value="1"/>
        </IntegerVar>
    </Collection>
    <OnEvent>
        <Trigger eventtype="KeyEvent">
                <UserKey name="key" value="VK_ENTER"/>
        </Trigger>
        <ActionLanguage name="jumpCounter">
            variables.counter.value += 10;
        </ActionLanguage>
    </OnEvent>
</Scene>
```

Alongside the notation for the full PCF Action Language syntax, as used in the example above, there is a "shortcuts" notation that provides an alternative way of describing certain commonly used actions. For example, the following are equivalent:

```
<ActionLanguage name="jumpCounter">
     SceneNavigate(<uri>#../../../next_scene</uri>, <enum>forget</enum>, nil);
</ActionLanguage>

<SceneNavigate>
     <URI name="target" value="#../../../next_scene"/>
     <String name="type" value="forget"/>
</SceneNavigate>
```

5.6 Structuring a PCF service description

For a simple service it may be appropriate that all aspects of the service description are in one PCF source document, and all relationships between elements are explicitly described through the nesting of elements within the description hierarchy.

For larger services, however, it may not be feasible to locate the entire service description in a single PCF source document, for reasons of efficient description of authorial intent and/or the modularity required in some business-to-business interchange situations.

The PCF provides flexible referencing mechanisms to accommodate these extremes of service description structure.

5.6.1 The href

In order to provide the flexibility to partition a service description in the most appropriate way, the PCF uses a referencing scheme to allow an element to remotely include other PCF items, or groups of PCF items. Such a reference can be described using the *href* property. The value of this property describes a path to the item to be referenced, using the values of the *name* property within the structural hierarchy of the service. In the following example, the TextBox does not include the content text directly, but instead contains a reference. This reference then points to the actual content text, which is declared separately:

```
<Component type="TextBox" name="hello_text">
    <String name="content" href="#../mycontent/msg"/>
</Component>

<Collection name=mycontent">
    <String name="msg" value="Hello, World!"/>
</Collection>
```

The benefits of such referencing include:

- The same set of components can be populated with multiple sets of content. A set of components can be used as a presentation "template", which can be combined with different sets of content in different scenes.
- A set of components declared once can be included within multiple Scenes. This allows "boilerplate" scene furniture to be reused multiple times, and also allows new complex components to be created, by allowing a set of simple components and some additional behaviour to be declared in one place, and then included wherever it is required.

5.6.2 Copy

A second method of referencing uses the *copy item*. This enables the referencing of items contained within some other item. The copy item is effectively replaced with the items contained within the item it refers to.

For example, the collection "MyContent" below contains two *string items*. The Scene declaration includes a Copy in order to "pull in" these *string items* so that they are now described within the scene. The TextBox components contain *String items* that use the *href* functionality to identify the individual piece of text to use as its content.

```
<Collection name="MyContent">
    <String name="Item1" value="Mozart"/>
    <String name="Item2" value="Beethoven"/>
</Collection>
```

```
<Scene name="Composers">
    <!-- Include referenced content within collection -->
    <Copy href="#../MyContent"/>

    <TextBox name="Box1">
        <!-- Refer to included content item -->
        <String name="content"  href="#../Item1"/>
        <Size name="size" value="100 40"/>
    </TextBox>

    <TextBox name="Box2">
        <!-- Refer to included content item -->
        <String name="content"  href="#../Item2"/>
        <Size name="size" value="100 40"/>
    </TextBox>
</Scene>
```

5.7 Managing differences between target platforms

The PCF provides a rich toolkit of functionality to allow interactive television services to be described in a platform independent manner. The PCF removes the requirement for a service author to have a detailed knowledge of the programming languages embodied in the various interactive middlewares deployed. This does not mean that the inherent differences between platforms simply disappear: if a particular platform does not have, say, an integrated return path, there is no way that the PCF, or indeed any other approach to authoring, can enable secure bi-directional communication. To give the service author a way of dealing with this, the PCF embodies two concepts, *degrees of freedom* and *profiles*.

5.7.1 Degrees of freedom

Despite the very different nature of various deployed interactive technologies it is often possible to deliver the same viewer experience on a number of platforms, to the point where it is indistinguishable to all but the expert. This is mainly due to the fact that underneath the different interactive middlewares there are very similar hardware architectures, in some cases implemented using the same chipsets. In general the problem is not instantiating a particular component, such as a TextBox, on a number of platforms. Rather, the challenge for the PCF is to create component definitions that strike a balance between:

- capturing authorial intent of sufficient precision to deliver a consistent viewer experience;
- ensuring the portability of each component to a wide range of targets;
- minimizing component specialization as a means of maximizing the exploitation of each platform, to avoid a proliferation of very similar components.

The PCF addresses this by allowing a *degree of freedom* in the definition of each PCF component. Wherever possible the degree of freedom is minimized, ideally to zero. However, where a degree of freedom is defined it may indicate if there is an ideal and a minimum acceptable implementation or if any implementation within the degree of freedom is as good as any other.

As an example, consider a TextBox component. This might be defined such that a line of text content will always be completely rendered within the visible area of the component. However, it might be defined with a degree of freedom such that the exact placement of each rendered character does not matter as long as the rendered line of text is completely visible. In this case no particular implementation can be considered more "ideal" than any other. Instead, this degree of freedom is essential simply because there is no standard for text rendering across interactive middlewares, i.e. a fundamental problem.

5.7.2 Profiles

To ensure that the PCF is relevant to as many platforms as possible it is deliberately designed to be unconstrained by the abilities of the least powerful target device being considered, i.e. the lowest common denominator. This characteristic of the PCF is essential if the increasing capabilities and diversity of target devices, which can be expected as platforms evolve over time, are to be accommodated. However, this inevitably means that not all the functionality that may be expressed using the PCF can be transcoded to work on all platforms, or even all devices within a particular platform.

So that an author can have confidence that their service description will work on a particular target platform, or more precisely a particular target device, without having to meticulously match up every feature they have used with the features provided by that target, the concept of PCF *profiles* has been defined.

Typically, each profile will embody a set of PCF features such that it maps well to a particular application area, very much as profiles have been used within MHP, e.g. enhanced broadcast, return path enabled, PVR enabled etc. In addition, to increase adoptability of the PCF, profiles have been designed to reflect steps in complexity of the PCF transcoder that is required.

In this way profiles provide a more manageable level of granularity for determining the feature support of a particular target and hence whether a particular service can be successfully deployed so as to deliver the intended viewer experience.

5.8 Transport and packaging

The PCF is intended primarily for business-to-business interchange, so it is important that it can be exchanged easily between different tools, content systems and networks. Although not mandatory, the PCF specification provides a business interface model that describes the encapsulation and workflow for this interchange. This means that systems implementing the model can, in addition to understanding a PCF service description itself, exchange the service's assets easily, in a way that guarantees service integrity.

To allow multiple PCF assets (PCF source documents and other content assets referenced by this source) within a service to be updated atomically, they can be grouped together into *transactions* for submission across the business interface. Each transaction carries a *priority* and this is to assist the receiving process in allocating resources to the task of processing the contents of the transaction. For example, if parts of a PCF service contain rapidly updating live sport scores, whilst another part of the service contains slowly updating, but sizeable, news stories, it is desirable to specify that the sport score content should be processed more quickly, and in preference to, the less latency-sensitive news content.

Each transaction must result in a coherent PCF service at the receiving end. Assets within a transaction may reference other assets defined within the scope of that transaction or assets that are known about already from previous transactions within the same service. All references within a transaction must be resolvable, and this implies that the state of the entire service persists at the receiving end for the entire duration of the interchange. Should a transaction contain unresolved references or syntax errors, that transaction will fail and all its contained assets will be discarded. In such a case a suitable error report is returned to the process at the sending end.

Transactions may be initiated either by the sending process (the push model) or the receiving process (the pull model). Generally, the push model is more efficient, as the sender will know when aspects of a PCF service have changed, rather than the receiving process needing to poll the sender. However, there are situations where pull operation is required, for instance when a transaction is initiated by a return path transfer.

PCF Services within transactions are identified by a service identifier and an organization identifier that has been previously agreed between the sender and receiver. This is to ensure that the receiving process is able to correctly identify which service a transaction relates to, as it may be processing a number of services concurrently. This mechanism may be augmented with additional security measures, for example SSL certificates, but this is not specified by the PCF.

Transactions may contain *hints* to assist the transcoder operation. Some hints, such as the priority of a particular scene transition, potentially apply to all target platforms. Other hints are target platform specific, in which case they are known as *directives*. The PCF specifies a syntax for exchanging hints and a set of portable hint types that a transcoder should support.

Some PCF services may require references to entities that only exist in the domain of the network operator, for example AV components, or entities whose details are not under the control of the PCF author, for example the return path configuration or links to other PCF service entry points. These must be specified by a PCF service as a list of *external references*. These references are abstract URNs that can be resolved during transcoding into actual assets or data. These URNs must also be portable, so the PCF specifies a syntax and recommends that a registered naming authority process be put in place for managing their allocation.

6 Architecture

The architecture of a format is the framework and structure that the format provides to an author so as to enable his description of a portable interactive service. Included in the format's architecture are: *data types*, *data structures*, *item identification* and a *referencing model*. The architecture describes foundational elements of PCF from which other areas of the PCF are constructed. The format's architecture is not a deployment system architecture, which describes the deployment environment for a PCF transcoder.

The specification of the architecture for PCF starts in clause 6.1 with an introduction to the architectural concepts that are the foundation of PCF. The technical specification of each architectural feature follows in clauses 6.2 to 6.6.

6.1 Introduction

The aim of specifying PCF architecture independently from all other aspects of PCF is to ensure that a consistent and interoperable framework description exists for all parts of a PCF service. The PCF architecture:

- describes complete interactive services;
- enables aspects of a service description to be created and modified independently;
- has an extensible syntax to support the creation of new features;
- expresses features at different levels of granularity;
- is practical to convert by machine;
- is an unconstrained service description that exploits features of all systems;
- encapsulates descriptions of additional capabilities;
- supports efficient validation and verification of a service.

Each of these affects all aspects of a PCF service description and so the underlying architecture of the format provides centralized and consistent solutions. Common architectural structures simplify the authoring process and provide a framework for building PCF extensions.

NOTE 1: A consistent underlying PCF architecture will simplify the implementation of authoring tools and transcoders. This is because these tools will be able to make use of a common and shared library of code for all aspects of PCF. For example, this library could contain common functions for parsing lexical values into internal representations according to rules for a given data type.

The PCF needs to be easy to adopt by the industry and, where possible, based on existing published standards. The PCF architecture is based on a set of standards in common use and from respected organizations such as DVB and W3C. These standards have been selected as appropriate for the description of interactive TV services. In this way, an author can use a description format that is consistent across both PCF service descriptions and other content formats.

NOTE 2: All PCF tool implementations may be able to incorporate or reuse both existing tools and code libraries that implement the standards that form part of the PCF architecture.

6.1.1 Strong typing

The PCF architecture supports a strong typing model to enable portability and extensibility. This provides a PCF-compliant platform-specific transcoder the ability to convert data to the most platform-appropriate type for rendering, either at a head-end system or on a receiver. It also provides extensibility, enabling the creation of new features within the framework of a pre-built set of data types.

An author shall explicitly state the data type of every value, thus enabling a transcoder the choice of mechanism that is used to convert a value to a platform-appropriate type. The data type for a value shall not be determined implicitly, for example the location where a value is defined in a service description. Validation of the type of a value by a transcoder is essential, as an author cannot assume anything about whether a particular value will be converted to its realizable form on the head-end or receiver.

EXAMPLE: Consider what would happen in a weakly typed environment where a string representing a colour value contains an error. For example, the value "#7F7F7" is missing a necessary digit. Without strong typing, how would a transcoder: recognize the error; realize that it needed conversion to a palette index on certain platforms; prevent its broadcast for interpretation at a receiver, which may cause a run-time error.

As a consequence of the strongly typed nature of PCF, the specification of a large number of data types is necessary and these are presented in clause 6.2. These are categorized as follows:

- **primitive types** - The atoms of the data type system, including integer and string.
- **core types** - Values with a fixed structure constructed from a mixture of types. For example, a position coordinate is constructed from two integer values.
- **octet data items** - Wrappers around a block of octet data, with parameters that are used by a transcoder to decode and interpret that data. The model is based on the Multipurpose Internet Mail Extensions (MIME) standards [1], [2] and [3]. These wrappers are used for data such as images, sound clips and references to video streams.
- **compound types** - Structures of data established by collecting together values of any data type in a service description. Compound types have constraints stating how items of data within a collection are identified and referred to. For example, in an array compound type, items are identified and referred to by their index.

One of the core types is a data type for representing marked up text content. This content contains style and structure information embedded within the flow of the text. This is described in detail in clause 6.6. Included with the present document is an XML Schema [4] for this mark up and all PCF tools can implement a parser for this format.

NOTE: The text mark up schema is defined independently from other PCF schemas in annex I.

All values within PCF shall be identifiable as a member of one of the PCF data types. A data type's specification includes constraining facets that characterize and constrain the values of that data type, such as the maximum value or allowable lexical patterns for the representation of values.

6.1.2 Static and active description

A service description written using PCF can be used for business-to-business interchange of interactive services that, by their very nature, describe a user-experience including interactivity. Therefore, items of a PCF service description shall comprise:

- **static description** - *scene items* and their content described as they are first presented to a user;
- **active description** - the portable description of activity that may occur during the lifetime of a service, in terms of states, state transitions and action language.

The two kinds of description shall be compatible so that run-time modifications can be made to a defined subset of features of the static description. To achieve this, the PCF action language shall be strongly typed so that a value of a particular data type in the static description can be manipulated at run-time from the active description.

EXAMPLE: The fill colour of a rectangle in a PCF static description will be specified according to the PCF colour data type. The value's data type will be explicitly stated in the static description, as this enables appropriate conversion by a transcoder to map to a target platform's native colour representation.

It will be possible to change the fill colour of the rectangle, in response to a user key press say, in the active description. The new colour is provided within some action language and this value needs to be converted to its platform-specific form. The action language must explicitly state the data type of the colour value, rather than implicitly represent the value as a string, to enable the transcoder to carry out the same value mapping process for the static and interactive descriptions.

6.1.3 Service representation

The high-level anatomy of a PCF service description is consistent with the information necessary to represent an interactive service. PCF enables an author to express their intended user experience in terms of an interactive service, its constituent high-level component parts and the scenes that provide a unit of presentation and interactivity to a user.

The architecture is specified as a PCF **information set**, which is independent of the format's representation. For example, it is intended that a service description in the format is independent of representation as:

- a file or files (text or binary);
- instantiated objects to a PCF object model;
- tables in a relational database.

However, where values are represented in a PCF service description in a lexical form, the format of these values is specified by the PCF architecture. For data exchange between PCF-compliant systems, a standardized way of representing PCF description data as an XML document [5] is provided. Version 1.0 of the schemas accompany the present document and are contained in archive ts_102523v010101p0.zip.

PCF service descriptions consist of *information items*, many of which are represented as PCF components as described in clause 7. The root item of a PCF description is a *service item*. Each *service item* shall contain one or more *scene items* and may specify which one of these scene items is first displayed to a user.

A PCF service description shall provide a clearly bounded set of all the information, whether directly or by resolvable reference, required to create a complete service. No more than one scene item shall be presented to a user at any point during the lifetime of an active service item on a single receiver. The currently presented scene can be changed with a scene navigation system action. All scene items available for rendering within a service item shall be contained within the root service item or one of its sub-containers. It shall not be possible to carry out an internal scene transition to a scene item that is not contained within the currently active service item. In this way, it shall always be possible to determine if an internal scene transition is valid.

NOTE 1: Containment of a scene item within a service item does not imply that they have to exist within the same file. See clause 6.1.4.

NOTE 2: Transcoders can permit the dynamic update of services. One possible on-the-fly modification is to add a new scene item to a service item prior to creating an internal transition that targets the scene item.

Service items and *scene items* are all derived from the PCF *map* compound type, as described in clause 6.2.5.2. Each of these elements can contain a grouping of other PCF items, with the constraint that each item shall have a unique name. *Collection items* also derive from the map type and contain author-defined groupings of other PCF items. Therefore, a PCF description shall be hierarchical structure of PCF items.

6.1.4 Referencing model

The referencing model within PCF enables links to be made, in absolute or relative specification, between any:

- identifiable item within a PCF service description;
- PCF item defined in any local file or equivalent data container;
- PCF item defined in a location given by a URI [9];
- resolvable resource associated with the service description.

NOTE: PCF *item references*, *map reference items* and *navigation reference items* provide details of a connection that exists between *service items*. These links can be resolved in a platform-dependent way, for example on a head-end system or at a receiver, as long as this resolution is in accordance with the rules in the present document.

6.1.5 Data partitioning and reuse

Many external factors will affect the way in which the items of a PCF service description are partitioned into assets. These include:

- the way in which a service may be delivered, e.g. with dynamic updates or as a one-off static description;
- the different roles of those describing the service, e.g. a graphic designer or a journalist;
- the systems deployed to produce a service, e.g. cooperating content management systems.

To enable those responsible for writing and deploying PCF systems the freedom to describe a service in the most appropriate way to their business, the architecture of the format is specifically designed to support flexible partitioning of descriptions. This enables the reuse of items by reference and the arbitrary distribution of items into files. The minimum unit that defines the granularity of this partitioning shall be a PCF *information item*.

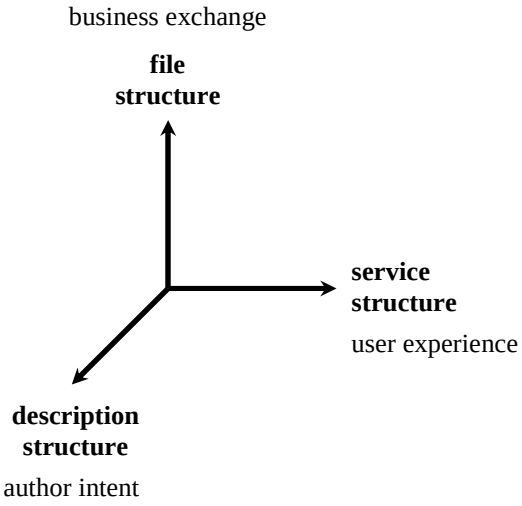

Figure 2: Axes of PCF description

Figure 2 shows the three degrees of freedom for expression of a service description that the PCF architecture provides. Each degree of freedom allows an author to express a service in a different way and the most effective description is likely to be a combination of all three. The degrees of freedom are characterized as:

- **service structure** - Service is described entirely consistently with the hierarchy of components that defines the rendering of the service. This allows an author explicit control of every aspect and parameter of the definition of every part of a service as rendered and experienced by a user. A service expressed using only service structure is:
  - a complete and verbose-everywhere description of the user-experience of a service;
  - represented in one file only;
  - makes no use of PCF *reference items*.
- **description structure** - Service is expressed in a modular way, enabling items to be defined once and reused by reference. This allows an author to indicate their intended service framework, such as to state that the same presentation template should be defined once in one place and repeated on several scenes. A service expressed using mainly description structure is:
  - defined by stating all non-structural items once at a high-level, with the hierarchy of components that make the service defined separately and predominantly by intra-file reference;
  - represented in one file only;
  - makes use of PCF *reference items* for specifying intra-file connections.

- **file structure** - Service description is partitioned into several files. Files are useful collections of PCF information items that suit the transfer of PCF data for business-to-business interchange or transcoder deployment. This can be used to minimize the impact of service updates and allows an author to distribute data into the most appropriate collections for management of data on a particular system. A service expressed entirely as separate files is:
  - described with one *information item* per file with inter-file references used to determine the component hierarchy of the service;
  - represented in many files;
  - makes use of PCF *reference items* for specifying inter-file connections.

NOTE: The use of the term file in the examples above is intended in an abstract sense to imply a logical collection of data according to an available storage unit on a system implementing a PCF interpreter. For example, this could be a file on disk, a PCF object in memory, a database table or a serialized data structure for transfer over a network socket.

6.2 Data types

6.2.1 Data type description

6.2.1.1 Data type model

The PCF data type model is a meta-model that describes the sets of values for instances of PCF items. The high-level classification of PCF data types and the PCF items that are members of those data types are shown in figure 3.

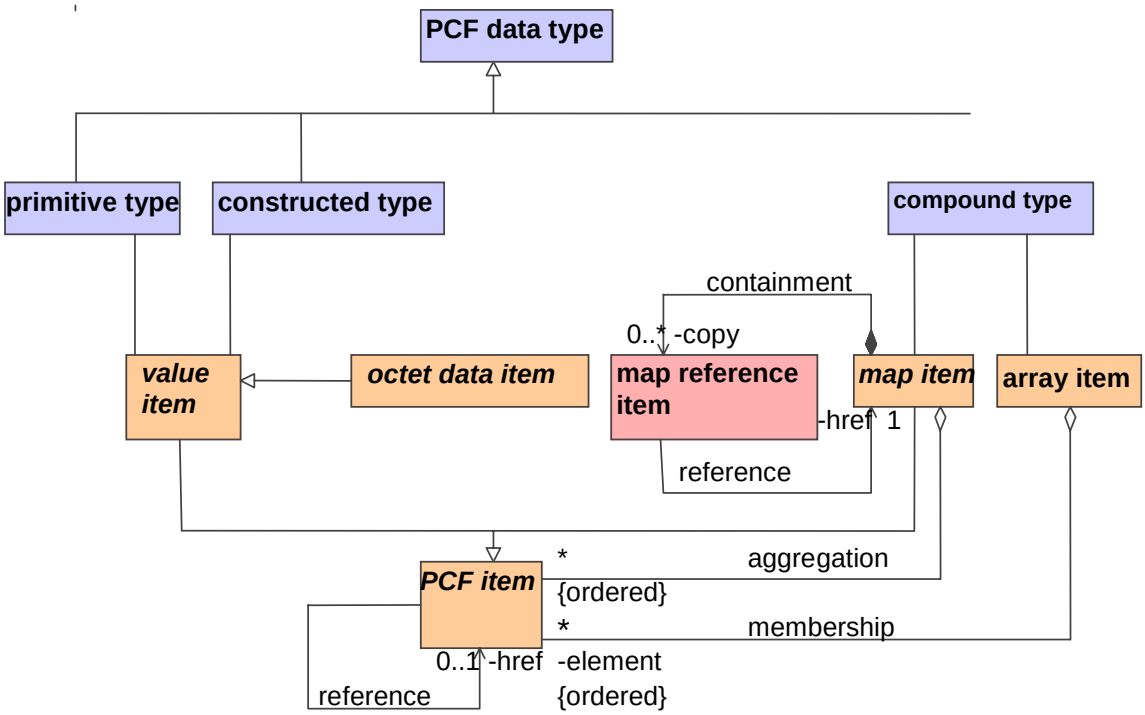

Figure 3: Data type model

A PCF item shall represent a value of any PCF data type. A PCF item may have the following three properties:

- **name** - An optional name for the value that can be used to identify it within a PCF service description. The name shall be of the "NCName" data type defined in clause 3.3.7 of XML Schema Part 2 [4], which is also the PCF name data type described in clause 6.2.3.8.

- **href** - A path to define that the value of the PCF item is defined by the resolution of a reference, as described in clause 6.4.1. Where the href property is non-empty, the value of the PCF item shall be established by resolving a reference and replacing the item, otherwise the value shall be defined as part of the PCF item.

- **context -** Value is an enumeration either shall be set to "original" or "derived", defining the context is which a non-empty href reference associated with the item shall be resolved. The default value is "original". This property is described further in clause 6.4.3.

Each of the data types is introduced in clause 6.1.1 and further described in clauses 6.2.2 to 6.2.5. A PCF item is either a value that is a member of one of the sets defined by a PCF data type or a reference to a value. A PCF service description shall consist of four different specializations of a PCF item, as listed below:

- **value item** - A single value of any PCF primitive data type, or a single value of any PCF core data type. Value items are defined in clause 6.2.1.3.

- **octet data item** - An item defined by a block of octet data that specifies either its content or location, its type and encoding. *Octet data items* are defined in clause 6.2.4.

- **map item** - An item containing a collection of other PCF items, where each item within the *map item* shall be identified by its unique **name** within the collection. *Map items* are defined in clause 6.2.5.2.

- **array item** - An item containing an ordered collection of other PCF items, where each item within the sequence shall be identified by its **index** within the collection. *Array items* are defined in clause 6.2.5.3.

The *map reference item* and its *href* attribute, a means for providing reuse and common templates , are defined in clause 6.4.4.

6.2.1.2 Description space

Each data type within PCF is described by a *value space*, a *lexical space* and a *set of constraining facets*, in a similar way to the data types of an XML schema [4]. These spaces and sets are defined as:

- **value space** - The set of possible values for a given data type. Each value is denoted by one or more literals in its lexical space. A PCF-compliant tool, such as an authoring tool or transcoder, shall be capable of representing all values of any PCF data type.

- **lexical space** - The set of valid literals for a data type. All literals in the lexical space shall have a corresponding value in the value space. Data types also have a *canonical lexical representation* that is a subset of the lexical space. Every value in the value space shall have exactly one corresponding representation in the canonical lexical space.

NOTE: Canonical representations are used whenever it is necessary to represent the value of a PCF item in a textual form in a way that is unambiguous. A tool that supports PCF should print out internal stored values using their canonical lexical representation.

- **set of constraining facets** - A facet is a single defining aspect of the value space. Generally speaking, each facet characterizes a value space along an independent axis or dimension. Constraining facets include the minimum and maximum values of a countable data type, acceptable patterns for an enumeration. A PCF-compliant tool shall only represent values that satisfy a data type within the bounds of the stated constraining facets. A validating PCF interpreter shall not accept a value that is inconsistent with the constraining facets of its data type.

Each normative data type is described according these terms. In the present document, this description is represented in a tabular form, as illustrated and described in table 1.

Table 1: Example data type description table

Name	data type name
Value space	allowable values of the data type
Lexical space	<regular expression for literal> and further description
Canonical lexical rep.	<regular expression for canonical literal> and further description
Constraining facets	Facet 1 Facet 2 Facet 3, etc.

NOTE 1: In the case of a data type such as string, the value space is defined by its literal representation, a sequence of allowable characters.

Regular expressions are used to specify the sequence of characters that make up the allowable lexical representation and canonical lexical representation of a value of a data type. These regular expressions use the same format as specified for regular expressions in appendix F of XML Schema part 2 [4]. All regular expressions are shown enclosed within left and right angles brackets, "<" and ">".

NOTE 2: To make regular expressions easier to read, white space and new line characters appear in the printed regular expressions shown here. These should be ignored.

6.2.1.3 Value items

Every PCF primitive and core data type has a corresponding *value item* that enables an author to represent a value of a data type within a PCF structure. For each data type, each *value item* shall consist of the following properties in addition to those specified for a PCF item in clause 6.2.1.1:

- **type** - The name of the PCF data type represented by the value item.
- **value** - The value represented according to the data type.
- **exact** - A Boolean value, set to true to indicate that a user must be presented with an exact representation of the value and false if a platform is allowed to present an approximation of the value. The default value is false.
- **nil** - Indicates that a PCF item has no value. Where a value is provided and nil is true, the value shall be ignored. The default value for this parameter is false.

NOTE 1: When written in service descriptions according to the PCF XML schema, value items are represented as elements with element names matching the data type name they correspond to. The XML elements do not have a type attribute. This allows a system that is validating a PCF XML service description check value attributes are consistent with their data type.

EXAMPLE 1: Using the PCF core types XML Schema, a value of ten according to the integer data type is represented as a value item using the integer element as follows:

```
<Integer name="number_ten" value="10"/>
```

When the *exact* property is set to true, a user should be presented with a rendering that is not an approximation of the *value* property of the value item. In the case where an exact representation of the value cannot be achieved on a target platform, the platform shall report an error. When the *exact* property is set to false, no error shall be reported.

NOTE 2: The mechanism by which an error is reported is dependent on the transport and packaging mechanism used to communicate with a PCF tool or translator. PCF defines an optional transport and packaging mechanism in clause 13.

EXAMPLE 2: Any component that includes the following fill colour specification must, when transcoded, be rendered with a 24-bit representation. Any platform that cannot achieve an exact presentation should report an error.

```
<Color name="fillColor" value="#F9EDC5" exact="true"/>
```

6.2.2 Primitive types

The primitive data types of the PCF are the building blocks for all other types of the format. Values are atomic and not expressible in terms of other constructs within the format.

NOTE: The PCF does not have a floating point or decimal numeric type.

6.2.2.1 Boolean

The *Boolean* data type shall be used for values that can be in one of two states, either true or false. This supports the mathematical concept of binary-valued logic. The Boolean data type is defined according to table 2.

Table 2: Boolean data type

Name	Boolean
Value space	{ true, false }
Lexical space	< true | false | 0 | 1 >
Canonical lexical rep.	< true | false >
Constraining facets	No white space permitted. Pattern must match lexical space.

The lexical representations "1" and "true" shall be interpreted as the value true. The lexical representations "0" and "false" shall be interpreted as the value false.

6.2.2.2 Integer

The *integer* data type is defined in table 3 and is based on the "int" data type as described in clause 3.3.17 of XML Schema [4].

Table 3: Integer data type

Name	integer
Value space	{-2147483648, …, -1, 0, 1, …, 2147483647}
Lexical space	< (\+|-)? [0-9]+ >
Canonical lexical rep.	< 0 | (-? [1-9] [0-9]*) >
Constraining facets	No white space permitted. Pattern must match lexical space. Minimum value inclusive is -2147483648. Maximum value inclusive is 2147483647.

NOTE: The integer data type is based on a signed 32-bit representation of an integer. Some platforms may exist that are unable to support this representation directly. Such platforms may substitute integer representations with a smaller numeric range.

When the *exact* property is set to true for an *integer value item*, a full 32-bit integer representation must be provided by the platform. The minimum acceptable word size for a platform implementation shall be 16 bits.

6.2.2.3 Enumeration

The *enumeration* data type provides an abstract mechanism for properties to be defined using a semantically appropriate set of labels. In this way, it is possible to define a set of data types that can be extended in future versions of PCF. A bijective mapping containing x enumeration items shall define an enumeration called e. Each enumeration item consists of a map from an integer in the range 0 to (x - 1) and a literal from the sequence l_0 to $l_{(x - 1)}$. Given this mapping, the enumeration data type is defined according to table 4.

Table 4: Enumeration data types

Name	enumeration_e
Value space	{ 0, … , (x - 1) }
Lexical space	{ l_0, …, $l_{(x-1)}$ }. A sequence of "NCName" values according to XML schema [4].
Canonical lexical rep.	Same as the lexical space.
Constraining facets	No white space permitted in labels. Pattern must match a literal in the lexical space.

NOTE: The literals that form part of the literal sequence are case sensitive.

The mechanism used to specify enumerations is described as part of the component meta-model in clause 7.2.2.3.

EXAMPLE: A text component has a property for representing horizontal alignment called "h-align". The following bijective mapping is defined for the enumeration that is the data type of this property: { 0 ↔ "left", 1 ↔ "center", 2 ↔ "right", 3 ↔ "justify" }. Only literals represented in this set can be used to specify values for the property.

6.2.2.4 String

The *string* data type shall represent a sequence of Unicode [6] characters. The string data type is defined in table 5.

Table 5: String data type

Name	string
Value space	A sequence of any Unicode characters.
Lexical space	< (. | \n | \r)* >
Canonical lexical rep.	< (. | \n | \r)* >
Constraining facets	Length defined by number of characters.

The PCF does not define its own escape sequence. In PCF service description written according to the PCF XML Schemas provided with the present document, standard XML character markup shall be used, as specified in clause 2.4 of XML [5].

NOTE 1: It is intended that PCF services can be represented as XML documents in the UTF-8, UTF-16 and UTF-32 encodings, as specified in Unicode [6].

NOTE 2: Other forms of service descriptions, for example those based on Java (see bibliography), should use escape sequences native to their representation.

NOTE 3: Where a specified Unicode character is not supported on a particular platform, a transcoder is responsible for mapping each character to either:

- the closest character or appropriate string substitution in the character sets available on the platform;
- a single white-space character.

When the exact property of a string data type is set to true, every character in the string shall have a representation on a platform according to the Unicode specification. If no representation exists on a platform, an error shall be reported.

A *string value item* shall also be a kind of *octet data item*, as described in clause 6.2.4.4.1.

EXAMPLE: Using the PCF XML schema included with the present document, a *string value item* can be represented in two ways. The following two string value items are equivalent.

```
<String name="eg" value="Example string value."/>

<String name="eg">
    <PlainTextData>Example string value.</PlainTextData>
</String>
```

6.2.3 Core types

6.2.3.1 Colour

The *colour* data type provides a means for an author to describe a colour in a platform-independent way. The lexical representation is based on acceptable colour values from HTML [8], extended to support transparency values as available within the MHP Java class "org.dvb.ui.DVBColor" [7]. The colour data type is defined in table 6 in terms of red, green, and blue components of the colour, with an additional value representing the transparency level for the colour.

The minimum value for the red, green or blue component of a colour is 0, which shall represent the case where none of the component colour is present. This value rises on a linear scale to a maximum value 255 that shall represent the maximum intensity for the colour component. For the transparency of a colour, 0 shall represent fully transparent rising on a linear scale to 255 that shall indicate the colour is fully opaque.

In the lexical form, all values for colour components and transparency shall be represented as two digit hexadecimal numbers. For each two digit hexadecimal number, a value of less than 16 shall be expressed with a leading "0", with the value 0 represented by "00".

Table 6: Colour data type

Name	color
Value space	{ A sequence of 4 integer values between 0 and 255, representing red, green, blue components and the transparency of a colour respectively. }
Lexical space	< # [0-9A-Fa-f]{6} ([0-9A-Fa-f]{2})? >.
Canonical lexical rep.	< # [0-9A-F]{8} > All hexadecimal digits expressed in uppercase and transparency value must be shown.
Constraining facets	Minimum inclusive value of "#00000000". Maximum inclusive value of "#FFFFFFFF". No white space allowed within colour values.

EXAMPLE: Full intensity opaque white is "#FFFFFFFF" and opaque black is "#000000FF".

NOTE 1: All colours with a transparency value of 0 ("00") appear fully transparent and so, effectively, the values for the red, green and blue components can be ignored in this case.

Where a specified colour value is not available and the *exact* property is set to false, the colour may be approximated.

NOTE 2: It is anticipated that many platforms will not be able to represent every colour value expressible using the PCF colour data type. In this case, a transcoder should implement a consistent mapping strategy from the PCF specified colour to the closest colour available on a platform. The advantage of including a colour data type in PCF is that a transcoder can easily distinguish colour values from strings and apply appropriate transformations.

6.2.3.2 Currency

The *currency* data type represents numerical values, such as amounts in money, that are normally expressed using a decimal point and two decimal digits. The PCF does not have a floating-point data type. The currency data type providing a means to represent commonly required figures in many currencies that contain a decimal point. The currency data type is defined in table 7.

Table 7: Currency data type

Name	currency
Value space	{ -21474836.48, …, -0.01, 0.00, 0.01, …, 21474836.47 }
Lexical space	< (\+|-)? [0-9]+ \. [0-9] [0-9] >
Canonical lexical rep.	< (0 | -? [1-9] [0-9]*) \. [0-9] [0-9] >
Constraining facets	Minimum inclusive value is -21474836.48. Maximum inclusive value is 21474836.47. No white space allowed within currency values. Precision is 2 decimal places at all times.

EXAMPLE: Zero is represented as "0.00", ten as "10.00".

NOTE: The currency data type is designed for representation within networks as a 32-bit integer value, which is then displayed to a user of a service as if a point has been added and the value has been shifted two places to the right. Transcoders can make the most appropriate decision as to how to map currency values to appropriate platform specific variations and, within a network, whether this occurs at the head-end or receiver.

Where a full 32-bit representation of a currency value cannot be achieved on a particular platform and the *exact* property is false, a representation that uses fewer bits may be substituted. The minimum number of bits that shall be used for a currency value is 16 bits.

6.2.3.3 Date

The *date* data type represents a top-open interval of exactly one day in length. This shall represent a value according to the "date" data type defined in clause 3.2.9 of XML Schema Part 2 [4], restricted to the current era as according to [16]. These are both based on ISO 8601 (see bibliography). The date data type is described in table 8.

Table 8: Date data type

Name	date
Value space	All valid dates according to the Gregorian calendar between 1st January 0000 up to 31st December 9999 within a specified timezone. The timezone can range from +12:00 to -11:59.
Lexical space	< [0-9]{4} - [0-1] [0-9] - [0-3] [0-9] (((\+|-) [0-1] [0-9] : [0-5] [0-9]) | Z)? > Pattern represents: year, month and day in the month are each separated by hyphens; followed by an optional timezone. Where the timezone is specified, "Z" shall represent UTC or any numerical value can be specified as an offset from UTC in the range +14:00 to -14:00.
Canonical lexical rep.	As lexical representation, except that the timezone shall be shown in the normalized form, ranging from +12:00 to -11:59. The canonical form of the zero-length timezone +00:00, -00:00 or "Z" shall be "Z".
Constraining facets	No white space is allowable within a date. The year zero ("0000") is valid. Month and day representations must be valid Gregorian dates. The minimum inclusive date is "0000-01-01+12:00". The maximum inclusive data is "9999-12-31-11:59".

NOTE: Current versions of the XML Schema specification [4] do not include the year "0000". However, as the year "0000" is included in the latest second edition of ISO 8601 (see bibliography), the XML specification may be updated to include "0000" in the near future. PCF tools may support the year "0000" where it is available, otherwise the minimum inclusive date is "0001-01-01".

EXAMPLE: The first day of the current millennium in London England is represented as "2000-01-01Z", where as in Sydney Australia this date was "2000-01-01+11:00" and in New York USA this was "2000-01-01-05:00".

No part of a date value can be approximated and so the *exact* property of a date value shall be ignored.

6.2.3.4 Date and time

The *date and time* data type represents an instant in time by combining a date representation with a time during the day on that date in terms of hours, minutes, seconds and milliseconds from the start of that date. The date and time data type is defined in table 9 and is based on the "dateTime" data type defined in clause 3.2.7 of XML Schema Part 2 [4]. The representation has been restricted to the Internet profile of IS0 8601 that is defined in [16].

Table 9: Date and time data type

Name	dateTime
Value space	All valid combined dates and times in the Gregorian calendar between 1st January 0000 and 31st December 9999 within a specified timezone. The timezone can range from +12:00 to -11:59.
Lexical space	< [0-9]{4} - [0-1] [0-9] - [0-3] [0-9] T [0-2] [0-9] : [0-5] [0-9] : [0-5] [0-9] (. [0-9] [0-9]? [0-9]?)? (((\+|-) [0-1] [0-9] ":" [0-5] [0-9]) | Z)? > Values are written as: year, month and day separated by hyphens ("-"); followed by "T"; followed by the hours, minutes and seconds in the 24 hour clock separated by colons (":"); followed by an optional decimal fraction of a second, with the default value when not present of ".0"; followed by an optional timezone that can be expressed as "Z" for UTC or a value between +14:00 to -14:00 for an offset from UTC.
Canonical lexical rep.	As lexical representation, with the following restrictions: 1. if the decimal fraction of a second is ".000", then it shall not be shown, otherwise the "." and all three digits shall be shown. 2. the timezone shall be shown in the normalized form, ranging from +12:00 to -11:59. The canonical form of the zero-length timezone +00:00, -00:00 or "Z" shall be "Z".
Constraining facets	No white space is allowable within a date. The year zero "0000" is valid. Month and day representations must be valid Gregorian dates. The minimum inclusive date and time is "0000-01-01T00:00:00+12:00". The maximum inclusive date and time is "9999-12-31T23:59:59.999-11:59".

NOTE: See the note in clause 6.2.3.3 about the inclusion of year "0000".

If the *exact* property of a *date and time value item* is set to false, the decimal fraction of a second may be approximated.

6.2.3.5 Font family

The *font family* data type represents font family descriptions that are compatible with the "font-family" specifications described in clause 15.2.2 of CSS2 [22]. A font family *value item* shall specify a prioritized list of explicit font family names and/or generic font family names, where the generic font family names shall be as specified in clause 15.2.6 of CSS2 [22]. The font family data type is defined in table 10.

Table 10: Font family data type

Name	fontFamily
Value space	List of one or more font families identified by explicit or generic font family names.
Lexical space	Comma separated list of one or more explicit or generic font family names.
Canonical lexical rep.	As lexical representation with normalized white space.
Constraining facets	Font names shall not contain whitespace. Explicit font families shall not have the same names as the generic font family names.

NOTE: Generic font family names defined in the CSS2 specification are:

- **serif** e.g. Times
- **sans-serif** e.g. Helvetica
- **cursive** e.g. Zapf-Chancery
- **fantasy** e.g. Western
- **monospace** e.g. Courier

EXAMPLE: The following *value item* shows a font family description represented according to PCF XML schema.

```
<font-family value="Tiresias, Helvetica, sans-serif"/>
```

If the *exact* property of a *font family value item* is set to true, an error should be reported if a platform cannot render an item using any of the choices provided in the font-family description.

6.2.3.6 Font size

The **font size** data type represents font size descriptions that are compatible with the "font-size" specifications described in clause 15.2.4 of CSS2 [22]. A font size *value item* shall specify an absolute size, relative size, length or percentage, as specified in clause 15.2.4 of CSS2 [22]. The font size data type is defined in table 11.

Table 11: Font size data type

Name	fontSize
Value space	See clause 15.2.4 of [22]
Lexical space	See clause 15.2.4 of [22]
Canonical lexical rep.	As lexical representation
Constraining facets	No negative integers

NOTE 1: Absolute font sizes defined in the CSS2 specification are:

- **xx-small**
- **x-small**
- **small**
- **medium**

- **large**
- **x-large**
- **xx-large**

NOTE 2: Relative font sizes defined in the CSS2 definition are:

- **Larger**
- **Smaller**

NOTE 3: Font sizes that measure length use integer values representing pixels relative to a reference screen size, as described by the PCF layout in clause 8.7. The PCF architecture does not currently define a separate data type for representing font sizes.

If the *exact* property of a *font* size *value item* is set to true, an error should be reported if a platform cannot render an item the specified size.

6.2.3.7 Marked up text

The *marked up text* data type contains values that are XML documents [5] according to the marked up text representation defined in clause 6.6. The marked up text format enables strings of text content to be structured into paragraphs and tables, with an option for an author to apply styling parameters. The marked up text type is described further in table 12.

Table 12: Marked up text data type

Name	markedUpText
Value space	Instances of XML document objects that are valid according to XML Schema "x-dvb-pcf.xsd".
Lexical space	Well formed XML document according to XML Schema "x-dvb-pcf.xsd".
Canonical lexical rep.	Marked up text shall be represented in its Canonical XML form, as defined in [23].
Constraining facets	Consistent with standards for XML [5] and XML Schema [4].

If the *exact* property is set to true, every character in a *marked up text value item* shall have a representation on a platform according to the Unicode specification [6]. If no representation exists on a platform, an error shall be reported.

NOTE: All styling within marked up text content is optional and does not have to be rendered by a transcoder. Therefore, the exact property of a marked up text value does not affect whether the rendering of the item reports an error due to approximation of styling properties.

A *value item* representing a MarkedUpText shall also be an *octet data item* as defined in clause 6.2.4.4.2.

EXAMPLE: Using the PCF schema associated with the present document, a MarkedUpText *value item* example is shown below:

```
<MarkedUpText name="markedUp">
  <body xmlns="http://www.dvb.org/pcf/x-dvb-pcf">
    <p>Example text with <em>emphasis</em>.</p>
  </body>
</MarkedUpText>
```

6.2.3.8 Name

The PCF *name* data type defines values for all possible names that can be used within PCF. These are equivalent to those defined as the "NCName" data type in clause 3.3.7 of XML Schema Part 2 [4] and described in table 13.

Table 13: Name data type

Name	name
Value space	As lexical space.
Lexical space	< [\p{L}_] [\p{L}\p{N}_\.\-]* >
Canonical lexical rep.	As lexical space.
Constraining facets	No white space allowed in names.

EXAMPLE: The following names are valid values of the name data type: "_loop3", "_39.3-2". The following names are not valid: "white space", "3degrees", "namespace:like".

Name values cannot be approximated and so the *exact* property shall be ignored.

6.2.3.9 Position

The *position* data type shall be used to represent values that locate a visual component on the reference screen, as defined in clause 8.7. Values are constructed from a pair of integer values as defined in table 14. For visual components, a position value should represent the top-most and left-most inclusive pixel of the reference screen. In the table, "integer" corresponds to the integer value space defined in clause 6.2.2.2.

Table 14: Position data type

Name	position
Value space	{ (integer × integer) }
Lexical space	< (\+|-)? [0-9]+ \p{Zs} (\+|-)? [0-9]+ >
Canonical lexical rep.	< (0 | -? [1-9] [0-9]*) \p{Zs} (0 | -? [1-9] [0-9]*) >
Constraining facets	Integer sequence of length 2. No white space allowed within integer values.

The first value of the pair shall be a pixel coordinate along the horizontal axis, measured from left to right. The second value of the pair shall be a pixel coordinate along the vertical access, measured from top to bottom. The value "0 0" shall be the top-left-hand origin of the reference screen area and refers to the first pixel inside the reference screen at the top-left-hand corner.

NOTE: Negative positions and large integer values greater than the dimensions of the reference screen may be defined. Values of the position data type may locate visual components partly or completely outside the reference screen area.

EXAMPLE: The pixel at the bottom-right-hand corner of the default reference screen area is at position "719 575".

The precision of representation and the *exact* property for position values is defined to be: each *integer value item* that defines the position should be represented according to the precision and exact property for integer values, as defined in clause 6.2.2.2.

6.2.3.10 Proportion

The *proportion* data type represents scaling factors. These allow one value to be expressed as a proportion of another value. The proportion data type is defined in table 15.

Table 15: Proportion data type

Name	proportion
Value space	{ (integer × integer) } All proportions expressible within the range of the integer values.
Lexical space	< (\+|-)? [0-9]+ \p{Zs} (\+|-)? [0-9]+ >
Canonical lexical rep.	< (0 | -? [1-9] [0-9]*) \p{Zs} [0-9]+ > All fractions are in irreducible form.
Constraining facets	Integer sequence of length 2. No white space allowed within integer values. Second integer must not be equal to 0.

NOTE: The proportion data type is expressed as two integer values rather than a proportional value as the PCF does not support floating-point values. A proportional representation allows a proportion of one third ("1 3") to be expressed exactly.

EXAMPLE: One third can be expressed as "1 3", three-eights as "3 8" and three times larger as "3 1".

The precision of representation and the *exact* property for proportion values is defined to be: each *integer value item* that defines the proportion should be represented according to the precision and exact property for integer values, as defined in clause 6.2.2.2.

6.2.3.11 Size

Values of the *size* data type shall represent the axis aligned rectangular size of a visual component. Values are constructed from a pair of integers as defined in table 16. In the table, "integer" refers to the corresponding spaces and representations in the integer definitions from clause 6.2.2.2.

Table 16: Size data type

Name	Size
Value space	{ (integer × integer) }
Lexical space	< (\+|-)? [0-9]+ \p{Zs} (\+|-)? [0-9]+ >
Canonical lexical rep.	< (0 | -? [1-9] [0-9]*) \p{Zs} (0 | -? [1-9] [0-9]*) >
Constraining facets	Integer sequence of length 2. No white space allowed within integer values.

The first number in the pair shall represent the inclusive size of a visual component in pixels when measured along the horizontal axis. The second number of the pair shall represent the inclusive size of a visual component when measured along the vertical axis.

Zero and negative values for both the horizontal and vertical size of a visual component shall be permitted. Any such representation shall result in the visual component not being displayed to a user. A zero or negative size shall not be used to represent the linear transformation of a visual component, including by rotation or reflection

NOTE 1: The size and position data types have been defined separately, rather than as a "pair of integers" data type, to assist a transcoder with validation and network specific transformations.

NOTE 2: A size value may be defined that specifies that a component is larger than the reference screen area.

EXAMPLE: The default size of the reference screen area is written as "720 576".

The precision of representation and the *exact* property for *size value items* is defined to be: each integer value that defines the *size* should be represented according to the precision and *exact* property for integer values, as defined in clause 6.2.2.2.

6.2.3.12 Time

Values of the *time* data type represent an instant of time that recurs every day. The time data type shall represent a value according to the "time" data type defined in clause 3.2.8 of XML Schema Part 2 [4]. The time data type is defined in table 17.

Table 17: Time data type

Name	time
Value space	{ All zero duration daily time instances. }
Lexical space	< [0-2] [0-9] : [0-5] [0-9] : [0-5] [0-9] (\. [0-9] [0-9]? [0-9]?)? (((\+|-) [0-1] [0-9] : [0-5] [0-9]) | Z)? > Values are written as: hours, minutes and seconds in the 24 hour clock, separated by colons (":"); followed by an optional decimal fraction of a second, with the default value when not present of ".0"; followed by an optional timezone that can be expressed as "Z" for UTC or a value between +14:00 to -14:00 for an offset from UTC.
Canonical lexical rep.	As lexical space, with the restrictions that if the decimal fraction of a second is ".000", then it shall not be shown, otherwise the "." and all three digits shall be shown. The timezone shall be shown in the normalized form, ranging from +12:00 to -11:59. The canonical form of the zero-length timezone +00:00, -00:00 or "Z" shall be "Z".
Constraining facets	No white space is allowable within a time. The minimum inclusive time is "00:00:00+12:00". The maximum inclusive time is "23:59:59.999-11:59".

If the *exact* property of a *time value item* is set to false, the decimal fraction of a second may be approximated.

6.2.3.13 Timecode

Values of the *timecode* data type shall represent a temporal reference to a point in a linear media stream, e.g. to identify a frame in a sequence of video. Timecode values shall be consistent with the EBU [17] / SMPTE [18] timecode specifications, as defined in table 18.

Table 18: Timecode data type

Name	Timecode
Value space	References to frames of a sequence of video or film that is anything up to 24 hours in length.
Lexical space	< [0-2] [0-9] : [0-5] [0-9] : [0-5] [0-9] \. [0-5] [0-9] > Timecode shall be written in the following order: hour, minute, second and frame through the second.
Canonical lexical rep.	As the lexical representation.
Constraining facets	Timecode shall not contain white space. The minimum inclusive timecode is "00:00:00.00". The maximum inclusive timecode is "23:59:59.59".

NOTE: The maximum frame number component of a timecode depends on the video system in use, e.g. PAL or NTSC, as described in [17] and [18]. PCF does not provide a data type for the representation of the time base of timecode values as this is resolvable from associated linear video and audio. Whenever linear video and audio content is transcoded from one timebase to another, PCF timecode values should be converted to appropriate values accordingly.

Timecode values cannot be approximated and so the *exact* property of a *timecode value item* shall be ignored.

6.2.3.14 URI

The *URI* data type shall represent values that are valid Uniform Resource Identifiers according to "anyURI" data type in clause 3.2.17 of the XML Schema specification [4], which is based on RFC 2396 [9]. The data type is defined in table 19.

Table 19: URI data type

Name	URI
Value space	See clause 3.2.17 of [4].
Lexical space	See clause 3.2.17.1 of [4].
Canonical lexical rep.	As lexical representation.
Constraining facets	A URI value shall not contain white space.

See clause 6.5.2 for more details on the use of URIs within PCF service descriptions.

EXAMPLE: The following are valid URIs according to the PCF data type:

```
http://www.etsi.org/
ftp://ftp.sourceforge.net/pub/sourceforge/
urn:x-dvb-pcf:bbc.co.uk:bbc-one
```

6.2.3.15 User keys

The *user key* data type shall represent virtual keys available to a user. A PCF service author may use any virtual key and assume that a PCF tool will match it to an appropriate key or feature available to a user. However, not all platforms will provide a mapping from a virtual key to a platform-specific key or feature, and where no suitable mapping for a key is available then a PCF tool should not render the service and shall report an error.

The virtual keys that shall be supported by PCF are listed in annex K and are based on those available in MHP [7] and WTVML (see bibliography). The user key data type in defined in table 20.

Table 20: User keys data type

Name	userkey
Value space	All virtual keys listed in table 37.
Lexical space	Tokens matching those shown in table 37. Lower case versions shall not be acceptable.
Canonical lexical rep.	All values in the lexical space.
Constraining facets	No white space is allowed within user key names.

NOTE 1: Transcoders should implement a mapping table from virtual keys to the most appropriate key on a device such as a remote control. The names of the virtual keys will not necessarily match with those on a remote control and, in this case, a transcoder should try to find the nearest equivalent. The most appropriate key may change depending on which component currently has focus.

EXAMPLE: A transcoder for one network maps the "VK_CANCEL" virtual key in a PCF service description to an actual remote control key labelled "backup" in one network. Another transcoder for a different network maps the same virtual key to an actual remote control key marked "cancel".

All virtual keys are approximated by the platform-specific equivalent. The *exact* property of *user key value items* shall be ignored.

Keys entered by a user that are not listed in annex K, such as the keys on a remote control keyboard, shall be represented by the value "VK_UNKNOWN".

NOTE 2: The user key event described in annex K contains a user key value and an actual platform-specific key code. For all "VK_UNKNOWN" keys, an author can access the platform specific key code. However, it is anticipated that complex components, such as those for text entry, will manage text entry and other advanced user key features in a platform native way.

6.2.4 Octet data items

6.2.4.1 Octet data introduction

Many items of data for PCF will be provided as sequences or streams of *octet data*, for example sound clips and image data. Rather than creating specific PCF data types for each type of data, the PCF provides a framework for handling all forms of octet data in a consistent way. The type of data is an attribute attached to the data rather than part of the data itself. This mechanism is based on the Multipurpose Internet Mail Extensions (MIME) [1], [2] and [3] and allows data to be:

- embedded within a PCF service description;
- in a separate local file;
- by reference to an external resource, such as a file or stream.

6.2.4.2 Octet data model

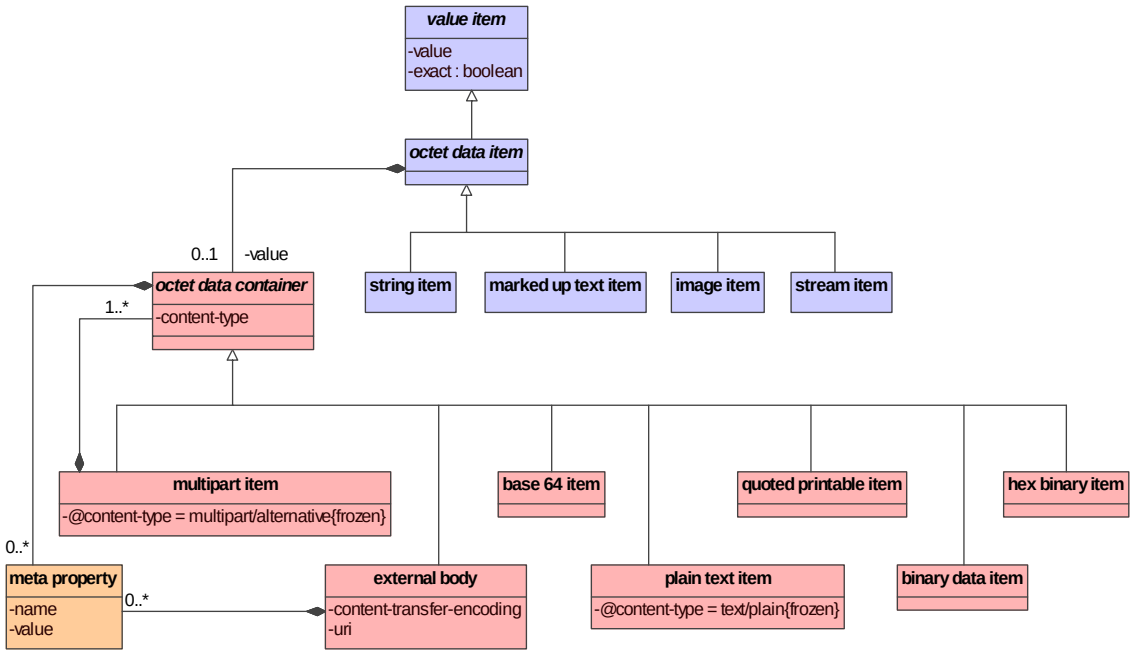

Figure 4: Octet data item model

Figure 4 shows the *octet data item* model. PCF has seven different kinds of *octet data container* that can represent the value of an *octet data item* and these are described in clause 6.2.4.3. PCF has four data types that can be represented by *octet data items* and these are described in clause 6.2.4.4.

The PCF octet data item provides an interface between value items and octet data. Octet data items shall be kinds of value items where the value property of the value item is provided by a sequence or stream of octet data. This interface allows description of the metadata associated with the octet data, including what type or types of data the octet data item provides and the data itself, either in an embedded form or by reference.

For any octet data item where a value is provided by octet data, the value provided shall be preferred over any other mechanism for specifying a value for the item.

EXAMPLE: In the following valid value for a string specified according to the PCF schema associated with the present document, the preferred value of a string that shall be used by any transcoder is "I am preferred.".

```
<String name="whichOne" value="I am ignored.">
  <PlainTextData>I am preferred.</PlainTextData>
</String>
```

Octet data items can also contain a collection of zero or more *meta property items*, as defined in clause 6.2.4.3.2.

6.2.4.3 Octet data containers

6.2.4.3.1 Portable MIME types

The PCF portable MIME types are a set of MIME types [1] that all PCF systems shall support. A service description containing *octet data items* with content-type compatible with the portable MIME types shall be known as a *portable service description*.

Octet data containers may have a "content-type" property. The value assigned to this property shall observe the following:

- It shall be accurate with respect to the encoding of the octet data.
- It should be one of the MIME types [1] supported by PCF, as specified in table 21, to ensure portability. If the encoding of the octet data is not known, such as in the case of an external resource, then the "content-type" property may either be omitted or may be assigned a generic MIME type, such as "application/octet-stream".

NOTE: If the data contained or referred to within an *octet data item* is encoded in a format other than one of those specified in table 21 then the service description shall not be considered portable with respect to the present document.

- It shall be appropriate for the *octet data item* containing the *octet data container*. In most cases this is obvious and there are a number of "peer" MIME types that are equally valid data representations for a particular type of octet data item. For example, image/png and image/jpeg for the image item. However, in some cases potential MIME types are not peers and some mapping will need to be specified. For example, text/x-dvb-pcf and text/plain for the *marked up text item*.

For the "text" major MIME type, character set specification using the optional "charset" parameter within the content-type property shall be supported. See the example in clause 6.2.4.4.1.

PCF defines a set of MIME types for the portable interchange of octet data between PCF systems. Table 21 shows the portable MIME types that shall be supported by a PCF system. All other MIME types may be used within a PCF service description but any service description that includes an unsupported MIME type shall not be considered as a portable service description.

Table 21: PCF portable MIME types

MIME type	Description
text/plain	Plain text data with no formatting commands or directives, as defined in clause 4.1.3 of [1].
text/x-dvb-pcf	Text content according to the PCF's own mark up notation, as defined in clause 6.6.
image/png	Octet data in the PNG image format [12].
image/jpeg	Octet data in the JPEG format [13].
image/ x-mpeg-iframe	Octet data that is an MPEG Intra-Frame (I-Frame), as defined in the MPEG-2 specification [15].
audio/mpeg	A video elementary stream in MPEG format [14].
video/mpeg	A video elementary stream in MPEG format [1].
audio/basic	Octet data in the basic audio format defined in clause 4.3 of [1].
multipart/alternative	See clause 6.2.4.3.9.
application/octet-stream	An octet stream of arbitrary binary data.
video/x-MP2PS	An octet stream that represents an MPEG-2 Program Stream.
video/x-MP2TS	An octet stream that represents an MPEG-2 Transport Stream.
video/x-MP2TS-P	An octet stream that represents an MPEG-2 Program within an MPEG-2 Transport Stream.
video/x-MP2ES	An octet stream that represents an MPEG-2 Elementary Stream of undefined type.

6.2.4.3.2 Meta property items

Meta property items are used for an author to provide additional information to the PCF system about how to interpret the octet data provided. *Meta property items* themselves shall have the following properties:

- **name** - The name of the meta property, that is a member of the PCF name data type defined in clause 6.2.3.8.
- **value** - The value associated with the meta property. This shall be a string according to the string data type defined in clause 3.2.1 of XML Schema Part 2 [4].

The names of all meta-property items contained within an *octet data item* shall be unique. Systems that do not support meta property items or recognize the name of a particular meta property item shall ignore that meta property item.

NOTE: Meta property items are provided to allow authors to provide hints to transcoders as to how they may want to interpret some octet data. Meta property items allow an author with knowledge of a particular set of transcoders to provide these with additional information to enable them to optimize transcoding. It should always be possible to ignore all meta property items and achieve a valid rendering of a service.

EXAMPLE: See the example in clause 6.2.4.3.9.

6.2.4.3.3 Embedded plain text data

An embedded plain text data item shall contain octet data that is encoded as plain text according to the "text/plain" MIME type as specified in the MIME specification [1]. Plain text data shall have a character encoding that matches that of the PCF source document in which they are declared. When represented in an XML document, such as a PCF service description, plain text cannot include XML mark up.

NOTE: In most cases, a *value item* of the PCF string type is more appropriate than using an octet data item that contains plain text data. The *plain text data* item is provided for compatibility with other systems.

EXAMPLE: The following two fragments of XML written according to the PCF XML schema are effectively equivalent.

```
<String name="headline1" value="Tsunami Early Warning"/>

<StringData name="headline1">
  <PlainTextData>Tsunami Early Warning</PlainTextData>
</StringData>
```

6.2.4.3.4 Embedded binary data

An embedded *binary data item* shall contain octet data that has no encoding and is 8-bit clean.

NOTE: Binary data items are difficult to encode as part of an XML representation. They are included so that other PCF models, such as relational database representations of PCF, can represent binary data items in a system-native binary representation. For example, the Binary Large Objects supported in SQL (see bibliography).

6.2.4.3.5 Embedded base64 data

An embedded base64 data item shall contain octet data that is encoded ccording to the method specified in clause 6.8 of part 1 of the MIME specification [1].

NOTE: Base64 encoded data can be safely embedded into an XML document as it is guaranteed not to contain left or right angle brackets "<" and ">", or any other special characters according to the XML specification [5].

EXAMPLE: Image data for an upward pointing arrow can be encoded within a PCF service description using the base64 encoding as follows:

```
<ImageData name="up_arrow">
  <Base64Data content-type="image/png">
  iVBORw0KGgoAAAANSUhEUgAAAAoAAAAKCAIAAAACUFjqAAAABmJLR0QA/wD/AP+gvaeTAAAANUlE
  QVR4nGP8//8/A27AhMxhZGRkZGTEKY1PN1wfsgHE6UazEs4loJsBj8f+///PSIK/MQEANb4MF+Di
  rHMAAAAASUVORK5CYII=
  </Base64Data>
</ImageData>
```

6.2.4.3.6 Embedded hexadecimal binary data

An embedded *hexadecimal binary data item* shall contain octet data that is encoded according to the method allowed by the "ietf-token" production of the MIME specification (Part 1) [1] is set to the value "hex". The "hex" encoding of binary data is defined in clause 3.3 of [26].

NOTE: Hexadecimal encoded data can be safely embedded into an XML document as it is guaranteed not to contain left or right angle brackets "<" and ">", or any other special characters according to the XML specification [5].

EXAMPLE: The string "PCF" in the UTF-8 character encoding is represented as "514346" as hexadecimal binary data, where each character in the original string is mapped to two hexadecimal characters in the encoded string.

6.2.4.3.7 Embedded quoted printable data

An embedded quoted printable data item shall contain data that is encoded according to the method specified in clause 6.7 of part 1 of the MIME specification [1].

NOTE: It is possible to represent data in a quoted printable form in such a way that it is safe to embed it within an XML document. Care must be taken to ensure that all special characters according to the XML specification [5] are encoded in their hexadecimal representation.

EXAMPLE: The following XML fragment is simple example of quoted printable encoding of data to the UTF-8 character set [6]:

```
<QuotedPrintableData>
  This is a UK currency =C2=A3 symbol, and for =3D measure ...
</QuotedPrintableData>
```

6.2.4.3.8 External body items

An external body item shall be used when octet data is defined in an external resource to the PCF service description, rather than embedded within the service description. External body items are intended to provide a similar function within PCF to the "message/external-body" MIME type defined in clause 5.2.3 of part 2 of the MIME specification [2].

External body items shall have the following properties:

- **content-transfer-encoding** - Defines the encoding used within the external body of the octet data, with these encodings being those registered with IANA and as specified in part 4 of the MIME specification [3]. The default value for this property shall be set to "binary", and "binary" should be treated as the default value when no other encoding is specified. The encodings are consistent with those available within PCF for embedded data, as specified in clauses 6.2.4.3.3 to 6.2.4.3.7

- **content-type** - Type of content contained within the file. This is any MIME type registered with the Internet Assigned Numbers Authority (IANA) as specified in part 4 of the MIME specification [3]. This property should be specified but may be left blank for PCF systems to automatically determine the content type of the external body's octet data.

- **uri** - Identity of the resource containing the octet data for the external body item. This may be specified by a PCF URN value, as defined in clause 6.5.2.

An external body item may contain one or more meta property items that indicate the location of the resource containing the external octet data.

An external body item may also contain other meta property items to provide system-specific data about the external body to a PCF transcoder. The names of all meta property items within an external body item shall be unique within the external body item.

EXAMPLE: The following XML fragment illustrates an octet data item that contains an external body item reference to the ETSI logo.

```
<ImageData name="logo">
  <ExternalBody content-type="image/png"
    content-transfer-encoding="binary"
    uri="http://www.etsi.org/images/logos/etsi-white-small.png">
    <MetaProperty name="comment" value="small ETSI logo"/>
  </ExternalBody>
</ImageData>
```

6.2.4.3.9 Multipart data item

Multipart data items shall contain one or more *octet data items*, each of which shall be considered as part of the octet data item in which the multipart data item is contained. The *content-type* property for a multipart data item shall be "multipart/alternative".

NOTE 1: Through the multipart data item, PCF supports the concept of the "multipart/alternative" MIME type. This allows a number of different versions of the same asset to be declared in a PCF service description. PCF tools can provide a service author with the ability to group a selection of alternative assets, such as set of images, to enable a transcoder to choose the most appropriate version of an asset to present to a viewer on a particular platform.

For a multipart data item to be considered portable, at least one of the octet data items it contains shall be of one of the supported PCF MIME types, as listed in table 21. If this is not the case, it may not be possible for a service description to be rendered.

NOTE 2: For each multipart data item, a PCF transcoder should use meta property items defined within octet data items to select the most appropriate asset for the target platform. For portability, it is recommended that these properties are named in a platform independent way so that a service can be targeted at platforms that were not considered during the original authoring of a service.

EXAMPLE: A service is to be targeted at three platforms with different colour models. The first platform has a fixed palette, the second has a dynamically assigned palette and the third supports full 24-bit colour palettes. A service author provides two images: one image will work with the first and second palette-based platforms; the other image is a true colour image that is suitable for rendering on the third platform. This is illustrated in the following XML fragment:

```
<ImageData name="logo">
  <MultipartData content-type="multipart/alternative">

    <ExternalBody content-type="image/png"
      content-transfer-encoding="binary"
      uri="file:///c:/images/logo_pallette.png">
```

```
      <MetaProperty name="palette" value="30"/>
     </ExternalBody>

     <ExternalBody content-type="image/png"
       content-transfer-encoding="binary"
       uri="file:///c:/images/logo_original.png">
      <MetaProperty name="colours" value="full"/>
     </ExternalBody>

  </MultipartData>
</ImageData>
```

6.2.4.4 Octet data item types

6.2.4.4.1 String octet data items

Value items of the PCF *string* data type, as defined in clause 6.2.2.4, may have their *value* property represented by an *octet data container*. The data provided in the octet data container shall have a *content-type* with the major MIME type "text".

The only portable content type for *string octet data items* shall be MIME type "text/plain", with the same character set encoding as the PCF source document in which the *string value item* is declared. A PCF system may transform any other character set encoding of a string to the PCF string representation in a platform-dependent way. The behaviour of a PCF system when a character set is specified and not known by a PCF system is undefined in the present document.

EXAMPLE: The following string value item, which is represented according the PCF XML schema associated with the present document, is a string octet data item defined by content in a file outside of the containing PCF source document.

```
<String name="outside">
  <ExternalBody content-type="text/plain; charset=utf-8"
    content-transfer-encoding="binary"
    uri="file:external_text.txt"/>
</String>
```

6.2.4.4.2 Marked up text octet data items

Value items of the PCF *marked up text* data type, as defined in clause 6.2.3.7, may have their *value* property represented by an *octet data container*. The data provided in the octet data container shall have a *content-type* property of "text/x-dvb-pcf" or "text/plain".

If the *content-type* property is set to "text/plain" then the octet data shall be treated as if it were a body element, containing a single paragraph element, containing the octet data itself, with each Carriage Return (0x0D) replaced by a line break element.

EXAMPLE: The following marked up text value item, which is represented according the PCF XML schema associated with the present document, is encoded using the base 64 [1] encoding. The value of this octet data item is equivalent to that shown in the example in clause 6.2.3.7.

```
<MarkedUpText name="markedUp">
  <Base64Data content-type="text/x-dvb-pcf">
    PGJvZHkgeG1sbnM9XFxcXFxcXCJodHRwOi8vd3d3LmR2Yi5vcmcvcGNmL3gtZHZiL
        XBjZlxcXFxcXFwiPg0KICA8cD5FeGFtcGxlIHRleHQgd2l0aCA8ZW0+ZW1waG
        FzaXM8L2VtPi48L3A+DQogPC9ib2R5Pg0K</Base64Data>
</MarkedUpText>
```

6.2.4.4.3 Image octet data items

The image data type in PCF shall represent image media data according to the "image" MIME type as specified in the MIME specification [1]. Image data shall be represented by an octet data container and shall not have any other form of lexical representation in PCF.

As defined in clause 6.2.4.3.1, the only two portable MIME types for PCF octet data items are "image/png" and "image/jpeg". A PCF system may be able to transform or make use of other types of data , as described in clause 6.2.4.3.1.

EXAMPLE: The following image octet data item, which is represented according the PCF XML schema associated with the present document, represents a small bitmap image arrow encoded as base 64 data.

```
<ImageData name="upArrow">
  <Base64Data content-type="image/png">
    IVBORw0KGgoAAAANSUhEUgAAAAoAAAAKCAIAAAACUFjqAAAABmJLR0QA/wD/AP+gv
    aeTAAAANUlEQVR4nGP8//8/A27AhMxhZGRkZGTEKY1PN1wfsgHE6UazEs4loJsBj8
    f+///PSIK/MQEANb4MF+DirHMAAAAASUVORK5CYII=
  </Base64Data>
</ImageData>
```

6.2.4.4.4 Stream octet data items

The *stream* data type in PCF shall represent a composition of one or more elementary media streams, such as video, audio, ancillary or other media data in a streamed format. Stream data shall be represented by an *octet data container* and shall not have any other form of lexical representation in PCF.

As defined in clause 6.2.4.3.1, the portable MIME *content-type* values for stream data shall be "video/mpeg", "audio/mpeg", "audio/basic", "video/x-MP2PS", "video/x-MP2TS", "video/x-MP2TS-P", "video/x-MP2ES" and the generic "application/octet-stream".

NOTE: The generic MIME type "application/octet-stream" shall only be considered portable for stream data items in PCF that identify an external resource, such as a broadcast TV service. It is assumed that such stream data will be available within the platform in a format that is known to the platform-specific transcoder.

EXAMPLE: The following stream octet data item, which is represented according the PCF XML schema associated with the present document, represents a broadcast TV service ("BBC One") that is delivered as part of an MPEG-2 Transport Stream. The broadcast TV service is identified by a PCF-compatible URN (see clause 6.5.2).

```
<StreamData name="tv">
  <ExternalBody content-type="video/MP2TS-P"
    content-transfer-encoding="binary"
    uri="urn:x-dvb-pcf:bbc.co.uk:bbc-one"/>
</StreamData>
```

6.2.5 Compound types

PCF *compound types* represent data types that contain collections of other PCF items.

6.2.5.1 Compound data type

The *compound* data type of the PCF is the set of all containers for PCF items. The generic compound data type is specialized into other compound data types by:

- providing restrictions that describe the data types that can be contained in a container;
- specifying structures within the containers that enable reference to, and manipulation of, values in these containers.

Two high-level compound data types are included with the PCF, map and array, as shown in figure 5. The map data type is described in clause 6.2.5.2 and the array data type in clause 6.2.5.3.

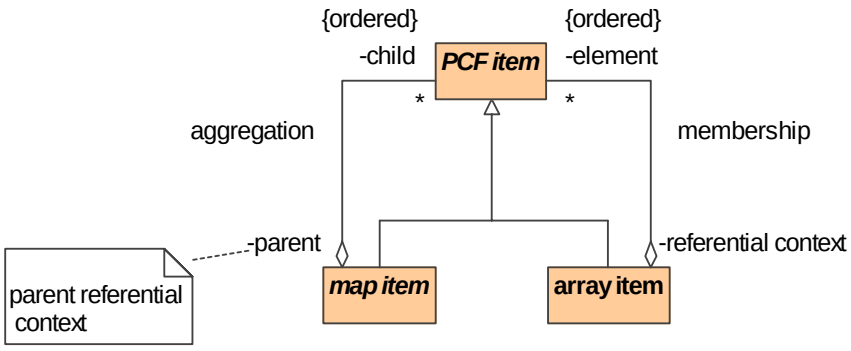

Figure 5: Compound data type

The structural elements of a PCF service description that are described in clause 6.3 are derived from the compound data type.

Every PCF item that is an instance of the PCF compound data type shall have:

- All the properties of a PCF item as defined in clause 6.2.1.1.
- A **content** property that contains zero or more PCF items. PCF items represented by the content property shall be known as the items within an instance of the PCF compound data type.
- A **length** property that is the number of items in the content property of the map item container. This property shall be a read-only property.

A PCF item that is of a compound data type shall provide a **referential context** for all the items that it contains, as described in clause 6.4.

6.2.5.2 Map type and item

The **map data type** shall be a compound data type that represents all possible sequences containing name and PCF item pairs. A PCF **map item** is a sequence of zero or more name and PCF item pairs and is an instance of the PCF map data type.

NOTE 1: The PCF does not define a maximum length for the sequence of items within a PCF map item. The behaviour of a transcoder when it receives a map item containing more PCF items that the transcoder's maximum is undefined.

Every PCF item within a map item shall have a name associated with it, specified by the PCF item's name property. That name shall be of the PCF name data type. All names within a map item shall be unique.

By means of PCF active description, it shall be possible to insert a PCF item into a map item at: the start of the sequence; the end of the sequence; before another sequence item; after another sequence item.

NOTE 2: The ordering feature is supported to allow a z-ordering for the layout of visual components to be specified in terms of the order that the components are defined within map items. **Scene** items are a kind of map item and are described in clause 6.3.5.

Items shall be initially ordered within the sequence of a map item according to their order of declaration.

A map item shall have a read-only **first** property that represents the first PCF item in the sequence within the map item. If the map item has zero items within it, the first property shall be set to a PCF item with its type property set to **boolean**, value property set to "false" and **nil** property set to "true".

A map item shall have a **parameter-list** property with a value that is a list of value parameter items passed into the map item, as described further in clause 6.4.5.

It shall be possible to iterate through the items contained within the content property of a map item according to their relative order, starting with the first item.

It shall be possible to reference a PCF item within a map item using its unique name. As map items can be PCF items contained within other map items, a path specification can be used to reference items in nested map item descriptions, as described in clause 6.4.2.2.

It shall be possible to test references to two PCF items contained in a map item to determine the order in which the PCF items occur in the map item.

6.2.5.3 Typed array data type and array items

The **array** data type shall be a set of all possible sequences containing PCF items that are each of the same type. Values of the array data type are called **array items**. An array item shall have within it a fixed-length sequence of zero or more PCF items. All PCF items within the array item shall have the same data type.

The length property of an array item is a final parameter, as defined in clause 7.2.2.2, which is therefore read-only in active description. The parameter is treated according to the following rules.

- If the length parameter is declared, the array declaration shall contain not more than this number of value items, or an error shall be reported.
- If the length parameter is declared and the number of items is less than the length, the PCF items declared shall be placed from the start of the array and the end of the array shall be padded with nil value items, as defined in clause 6.2.1.3.
- Where the length parameter is not declared, the length of the array shall be the same as the number of PCF items declared inside the array.

All array items are themselves PCF items and as such have a type parameter. This parameter shall be set to the name of the data type of all of the PCF items within the array item with the word "array" appended onto the end. This parameter is mandatory and final.

EXAMPLE 1: An array item containing PCF items of type string shall itself have the type name "stringarray". An array of string arrays shall have the type name "stringarrayarray".

NOTE 1: Arrays can be of zero length.

It shall not be possible to have an array of map items.

NOTE 2: As array items are PCF items, it is possible to have an array of child array items. In this case, each child array item shall have the same type parameter and length parameter.

PCF items within an array item shall be ordered and indexed according to their order of declaration. Index values shall be inclusive and range from 0 for the first item in the list up to the length of the array minus 1.

By means of PCF active description, it shall be possible to:

- read the value of any items of an array by an index into the array;
- iterate through all the items in an array;
- change the value of any PCF item in an array to the value of another PCF item of the same type.

From active description, it shall not be possible to change either the type of the array or the length of the array.

EXAMPLE 2: The following fragment of XML shows two equivalent array item declarations of the string type and is consistent with the PCF XML schema.

```
<StringArray name="no_nil_values" length="4">
    <String value="First"/>
    <String value="Second"/>
</String>

<StringArray name="nil_values_included" length="4">
    <String value="First"/>
    <String value="Second"/>
    <String nil="true"/>
    <String nil="true"/>
</String>
```

NOTE 3: In the PCF XML Schema, array items are represented by elements with an element name that starts with the type parameter and ends with "Array". For example, array items of the integer type are represented by the XML element "IntegerArray". An "integerarrayarray" type is represented by an "IntegerArray" element that contains only other "IntegerArray" elements.

6.3 Service description structure

PCF services are described in terms of PCF items, with certain high-level structural description items that must always be present. The description items that make up a PCF service are introduced in clause 6.3.1 and described in more detail in clause 6.3.2 to 6.3.6. Finally in this clause, scoping rules that define access to values in active description are defined in clause 6.3.7.

6.3.1 Description items

The description items of a PCF service description are service items, scene items, other component items and collection items. The relationship between these items is shown in the UML class diagram of figure 6.

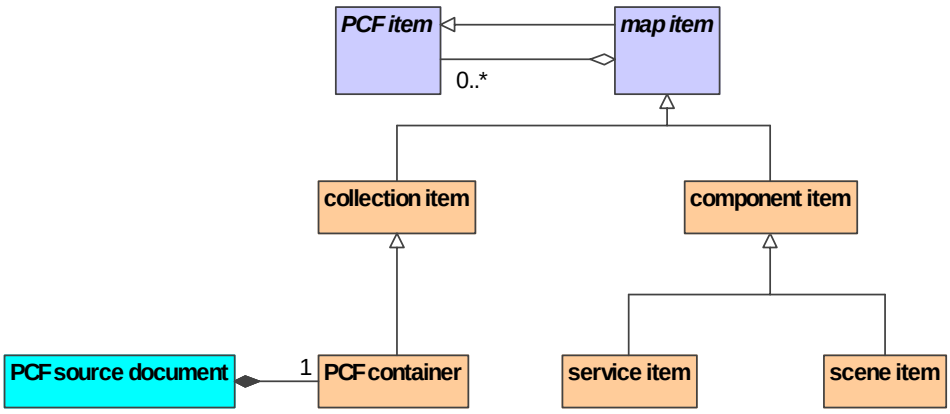

Figure 6: The PCF structural items

All PCF service description shall be contained within a PCF container. A PCF container shall be the top-level item in a PCF source document. Every PCF source document shall have exactly one PCF container. PCF containers are defined in clause 6.3.4.

All the structural items of a PCF service description, PCF container, component item, collection item, Scene item and service item, shall be kinds of map item, as described in clause 6.2.5.2.

EXAMPLE: An XML document stored in a file and with its schema set to be the PCF XML schema is a PCF source document. It must have as its root element a PCF container, which is represented as the element named "PCF". This is illustrated in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>

<PCF xmlns="http://www.dvb.org/pcf/pcf"
  xsi:schemaLocation="http://www.dvb.org/pcf/pcf pcf.xsd">

  <String name="example"
    value="Only PCF item in this PCF source document."/>
</PCF>
```

Component items provide structured containers that correspond to a component description, according to the component model described in clause 7. Component items are described in clause 6.3.2.

The low level containers of PCF service descriptions are collection items. These allow an author to group items into structures of their choosing in a way that is useful for their particular service description. Collection items are described in clause 6.3.3.

Service items and scene items are the highest level of structured description and can provide the entry points of a user-experience of a PCF service. Service items are described in clause 6.3.6 and scene items in clause 6.3.4.

6.3.2 Component items

Component items provide a structured description of instances of PCF component types. PCF components are defined in clause 7. PCF systems should interpret component items when rendering of a PCF service description. As such, component items shall be known as **renderable items**.

A renderable item can be active or inactive and can be activated and deactivated by active service description. When a renderable item is active, it should be presented to a viewer and/or available within PCF active description according to the rules defined for the renderable item.

NOTE: The rules for presenting each PCF component item type are defined in A and the model for defining these rules in clause 7.2.2.

Component items shall be members of the PCF map data type that is defined in clause 6.2.5.2. As well as providing a referential context for the PCF items within a component item, a component item also provides a **rendering context** for all the PCF items they contain.

6.3.3 Collection items

Collection items are groups of PCF items. Collections may be defined by service authors to enable modular service descriptions that provides hierarchical referential context and facilitates referencing, as described in clause 6.4.

Collection items shall be members of the PCF map data type.

EXAMPLE: An example of a PCF collection according to the PCF schema is shown below. The example shows a collection of default properties to be applied to news story items throughout a service.

```
<Collection name="news_style">
  <Color name="textcolor" value="#DFDFDF"/>
  <String name="content" value="More news follows shortly ..."/>
  <Integer name="border-width" value="3"/>
</Collection>
```

NOTE: Collection items shall not be renderable items. Their only purpose is to structure data and provide referential context.

Collection items shall not provide rendering context to the items they contain. A collection item shall provide the same rendering context to the PCF items contained within it as the rendering context of the collection item itself.

6.3.4 PCF container

A PCF container shall be a collection item that is the root item of any PCF source document.

A PCF source document may be identified as a resource describing an interactive service that is to be rendered. In this case, the PCF container in the PCF source document should contain within it exactly one service item that provides the **root rendering context** for the service.

NOTE: The behaviour of a system interpreting a PCF container that does not contain exactly one service is undefined in the present document.

6.3.5 Scene items

The purpose of the scene item is to describe a spatially and temporally coordinated unit of viewer experience description. Scene items shall be component items and, therefore, are renderable items.

When a renderable item is active, it is presented to a viewer and/or available to participate in viewer interaction through active description.

Only one scene in an interactive service description shall be active at any one time. When a PCF scene item is activated, it shall set its own state to active and activate all renderable items within its rendering context. When a PCF scene item is deactivated, it shall deactivate all renderable items within its rendering context.

PCF items contained within a scene item shall only contribute to the viewer experience when the scene item they are contained within is active. The state of such PCF items shall be maintained only whilst this scene item is active. Their state shall be re-initialized on each activation of the scene item.

NOTE: PCF items that need to have their state maintained independently of scene item activation must be contained in a service item (see clause 6.3.6).

A scene item or any of a scene item's descendants shall not contain any service items or scene items.

A scene item is declared using the Scene component as specified in clause A.1.1.2.

6.3.6 Service items

A service item represents the complete description of an interactive service. A service item is a component item and therefore is a renderable item.

A service item shall contain, either directly or by reference:

- all scene items that are part of the viewer experience of the service being described;
- any other PCF items that are common to all scene items.

The state of viewer experience session at a point in time shall be defined by the combination of all renderable items within the rendering context of the service item and all the renderable items within the currently active scene item.

Exactly one scene item shall be active at any point in time. Navigation to another scene item shall cause the source scene item to be deactivated and the target scene item to be activated.

A service item shall not contain any other service items within itself or any of its descendants. All PCF items within a service item except for scene items shall have their state maintained independently of scene item activation for the duration of the service session.

Component items that contribute to the viewer experience described by **all** scene items should be included within the service item. A service item can provide the root rendering content of a service description. When a PCF service is initialized, the associated service item shall be activated and all renderable items, apart from scene items, within the service item's rendering content shall be activated. These renderable items remain activated throughout an interactive service session until the session exits. When the service exits, all descendant renderable items and the service item shall be deactivated.

NOTE: The inclusion of a PCF component item within a service item shall be interpreted by a PCF transcoder as an indication that any contribution made to presentation by the component item shall be uninterrupted by any navigation between scene items.

EXAMPLE: A Video component can be declared within the service item to describe seamless presentation of video during navigation between scenes.

A service item is declared using the Service component as specified in clause A.1.1.1.

6.3.7 Scoping rules

The PCF scoping rules determine the way in which components can access and manipulate properties within other components in the component hierarchy using active description. The scoping rules reflect the run-time scope of variables and component properties: the lifetime of a component item within a scene item shall be equal to the lifetime of its parent scene. A scene, and therefore its descendant renderable items, are active only when the scene itself is currently active.

The scoping rules are:

- Behaviour within a component item may access its own properties and those of all of its descendant component items.

- Behaviour at the service level shall not access properties of its child scenes or any components within those scenes.

NOTE: Behaviour at the service level may access properties of any non-scene child components.

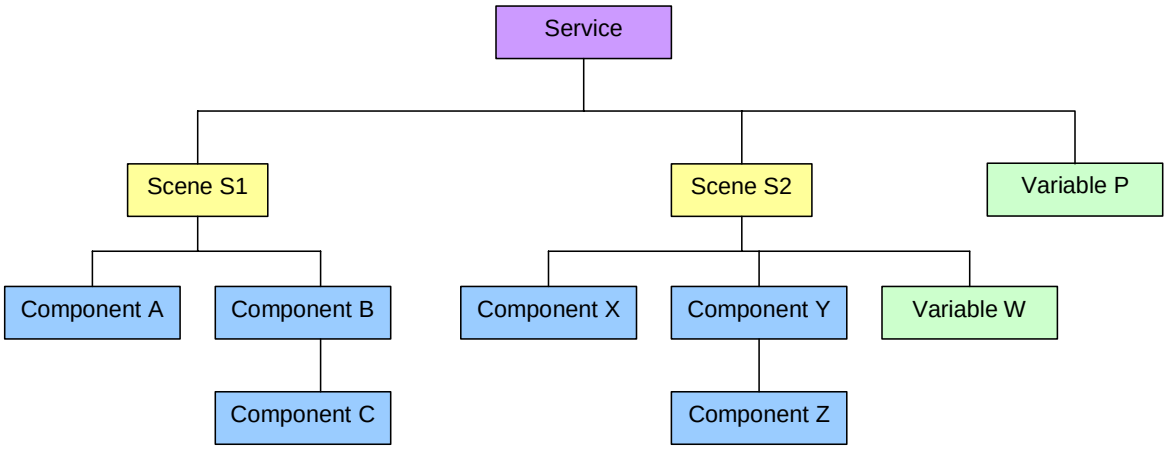

Figure 7: An example component hierarchy in PCF

Figure 7 illustrates a component hierarchy. A service contains two **scene**s, S1 and S2, and a service-level variable component, P. Scene S1 contains components A and B, where B contains component C. Scene S2 contains components X and Y, where Y contains component Z, and a scene-level variable component W. The scoping rules stipulate that component A is visible to component S1, and that components B and C are visible to S1. However, C cannot access properties in either B or A.

EXAMPLE 1: Component C has a property called fillcolor. Component B can access the value of C.fillcolor using appropriate action language.

None of the components in scene S1 are visible to S2. Should the author require such manipulation, the information can be stored at the service-level, in this case in the variable component P.

EXAMPLE 2: When scene S1 is active, action language can write the value of C.fillcolor to the variable component P. When scene S2 is active it cannot directly access the value of C.fillcolor from S1. However, it can read the value of variable component P. In this way, information can be passed from one scene to another.

Moreover, a variable component can be placed anywhere in the component hierarchy. A variable component can be used to store values shared by two or more components provided it is declared in a scope outside of the participating components.

EXAMPLE 3: Variable component W is declared within the scope of S2. Component X and Y can manipulate variable W in order to share information because variable W is declared outside of component X and Y. Variable W has a lifetime that coincides with scene 2. Should this information also be required by component A in scene S1, then a variable component outside of the scene scope should be used, e.g. variable component P.

6.4 Reference and navigation

Clause 6.4.1 introduces the PCF referencing model. Clauses 6.4.2 to 6.5.2 specify the mechanisms for making and resolving references that should be supported by PCF tools.

6.4.1 Referencing model

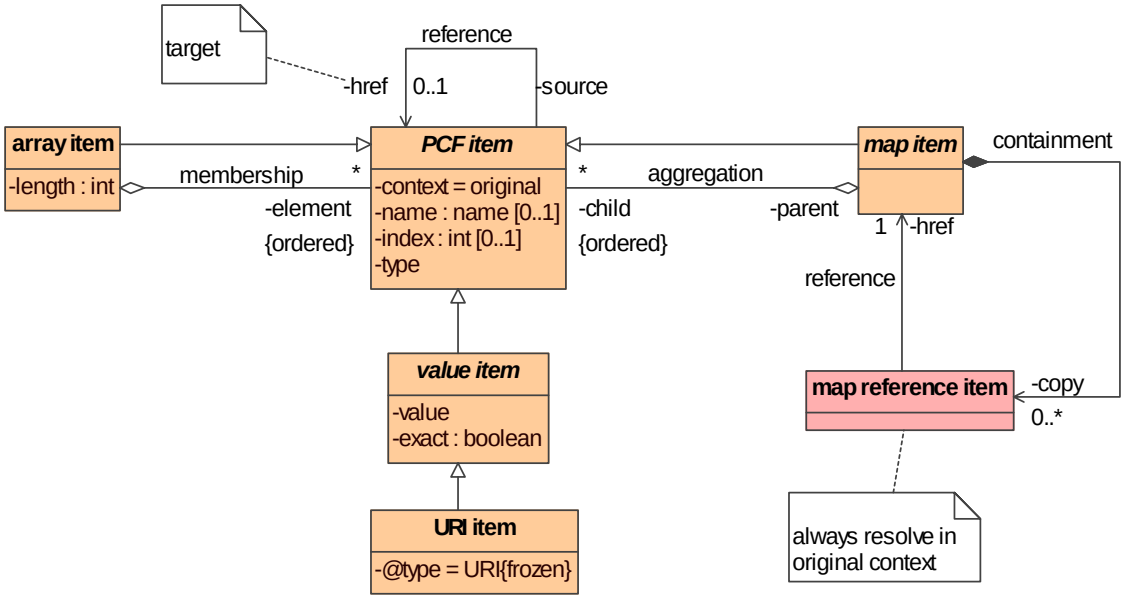

Figure 8: PCF reference and navigation model

Figure 8 shows the relationships between PCF items that are part of the PCF referencing and navigation model.

6.4.2 Typed reference

6.4.2.1 PCF item references

PCF item references allow any PCF item to be specified by reference rather than by an explicitly stated value. This shall be achieved using the "href" property that is available for all PCF items, as described in clause 6.2.1 and illustrated by the reference relationship in figure 8.

References are made from a PCF item and this item shall be known as the **source item**. References are made to a PCF item and this item shall be known as the **target item**.

A PCF item reference shall be any PCF item with a non-empty "href" property. If the "href" property is non-empty, the value of the PCF item shall be defined by reference and any locally specified value shall be ignored. The value of a PCF reference item shall be determined by embedding the content of target of the reference at the position of the source reference.

PCF reference items are strongly typed. The source and target of a PCF item reference shall be of the same type.

If a source item has no name specified and the target item does have a name specified, the name of the source item shall be the same as the name of the target item on resolution of the reference. Where both the source item and target item have a name, the source item shall be known by its own name, not that of the target, after reference resolution.

NOTE: Both the source item and target item may not have names specified where they are declared as elements of an array.

Clause 6.4.2.2 defines the path format of these references and clause 6.4.3 defines the rules for the contextual resolution of these references.

6.4.2.2 Reference path format and resolution

The reference path format enables an author to uniquely identify any item in a PCF service description by a path name. As a service description may be partitioned into a number of PCF source documents, the PCF reference path format allows references both within and between PCF source documents. This is achieved through the use of URIs [9].

The aim of resolving a path shall be to determine the target item that the path refers to or that the path is invalid. A value that represents a path in PCF shall be known as a **PCF path**.

PCF items defined by reference have a **href** property that is specified by a "ref_path" production. The map item that is the closest ancestor by containment to the source item is known as its **parent item** and all the items contained by a map item are known as its **child items** or **children**.

The service item that is the furthest ancestor by containment from the source item in the current resolution context shall be known as the **root service item**.

The PCF container that is associated with the PCF source document according to the current resolution context is known as the **root container item**.

The root service item and root container item may change depending on the context for the resolution of the reference, as defined in clause 6.4.3.

NOTE 1: Source items in arrays have their closest ancestor that is a map item defined as their parent item. The array in which the source item is contained shall not be considered as its parent item.

The format of a path, specified using Extended Backhaus-Naur Form [*], shall be:

```
ref_path  ::= (uri_part)? "#" ("/" | "~/")? path
uri_part  ::= absoluteURI | relativeURI
path      ::= path_item ("/")+ path | path_item
path_item ::= name (index)? | "." | ".."
index     ::= "[" integer "]"
```

Missing productions in the path format grammar above shall be defined as follows:

- "integer" - A non-negative integer value consistent with the PCF integer data type defined in clause 6.2.2.2.
- "name" - A name value consistent with the PCF name data type defined in clause 6.2.3.8.
- "absoluteURI" and "relativeURI" - as defined in [8].

A PCF path shall be resolved in a sequence of steps. At each step, the path shall be resolved with respect to a specific PCF item, known as the **relative item,** determined in the previous step. Therefore, the first step shall be to determine the initial relative item.

Where the "uri_part" of the "ref_path" is present, it explicitly identifies the PCF source document that contains the target item (the **target PCF source document**) and the PCF container of this target PCF source document shall be the initial value for the relative item. In this case, the optional "/" literal may be present but has no effect on the choice of relative item. The optional "#~" literal shall not be allowed.

Where the "uri_part" is not present, the target PCF source document shall be the containing PCF source document in the context of the resolution of the reference (see clause 6.4.3). The relative item shall then be set according to one of the following rules:

- If the optional "/" literal is present, the initial value for the relative item shall be the root container item.
- If the optional "~/" literal is present, the initial value for the relative item shall be the root service item.
- If the "/" or "~/" literals are not present, the initial value for the relative item shall be the parent item of the source item.

EXAMPLE: The comments in the following XML document, written according to the PCF XML schema, show how elements in a PCF source document called "myservice.xml" can be referred to:

```
<!-- refer to PCF container as:
        - "myservice.xml#/" from outside or inside the source doc.
        - "#/" from inside the source document -->

<PCF>

<!-- refer to following integer "age" as:
        - "myservice.xml#/age" from outside or inside the source doc.
        - "#/age" from inside the source document -->
  <Integer name="age" value="32"/>
```

```
  <Service name="older">

<!-- refer to following integer "age" as:
        - "myservice.xml#/older/age" from out/inside the source doc.
        - "#/older/age" from inside the source document
        - "#age" locally
        - "#~/age" from in context derived from service "older" -->
    <Integer name="age" value="33"/>

<!-- transclude an age value from parent context - same as using
     "#/age" in original context, may differ in derived context -->
    <Integer name="oldage" href="#../age" context="original"/>
  </Service>
</PCF>
```

If the "path" includes another "path" then the "path_item" shall be resolved to a new relative item and then the remaining "path" shall be resolved with respect to this as a next step.

If the "path" is made up of only a "path_item" then this "path_item" shall be resolved as the last step of path resolution and the PCF item that it resolves to shall be the target item that the path refers to.

The way in which a "path_item" is resolved shall depend on how it is specified, according to the following rules:

- a "name" and optional "index", that shall resolve to one of the children of the relative item as follows:
  - Where an "index" is provided, the name shall refer to an array item or the path is invalid. The "index" is a non-negative and shall refer to a PCF item in the array by its order in the sequence of items and the "path_item" shall resolve to this item. If the index is greater than the length of the array, the path shall be considered invalid.
  - Where no "index" is provided, the path shall refer to a PCF item that is a child item of the relative item by matching the name property of the PCF item.
- a full-stop character ("."), where the "path_item" resolves to the current relative item;
- a double full-stop ("..") to indicate that the "path_item" shall resolve to the parent of the relative item. If no parent item exists then the path shall be considered invalid.

NOTE 2: By this definition, empty paths are invalid. Paths must always contain a hash "#" character and at least one "path_item". However, as an empty path is used to indicate that a PCF item is not a PCF reference item, PCF interpreters should not report an error.

6.4.3 Contextual resolution

The paths specified by the "href" properties of a PCF item reference are either resolved in their originally declared context or a context derived from where they are used. As defined in clause 6.2.1.1, every PCF item has a "context" property, the value of which is an enumeration set to either "original" or "derived". The default value for this property shall be "original".

When the "context" property of a PCF item reference is set to "original", then:

- The root container item shall be the PCF container within which the PCF item reference is defined.
- The root service item shall be the furthest ancestor service component from the PCF item reference that is a service item. If no ancestral service item can be found, the PCF path defining the PCF item reference is invalid.
- The path shall be resolved in the hierarchical referential context provided by the root container item and all its descendants. This is the PCF container of the source document where the PCF item reference is declared.

If the "context" property of a PCF item reference is set to "derived", a service item shall be activated and the reference resolved with respect to this service item, known to be active. Contextual resolution shall then use the following rules:

- The root container item shall be the PCF container that is furthest ancestor by declaration of the activated service item.
- The root service item shall be the activated service item.

- The path shall be resolved in the hierarchical referential context provided by the root container item and all its descendants.

NOTE 1: The main reason for using "derived" references is to allow the reuse of structures that have already been declared. When a PCF item of compound type is reused in a new context, it is copied in to replace a PCF reference item, often of original type. All the child items that are themselves PCF reference items of "derived" type are then resolved relative to the location where it has been copied to rather than their original context.

NOTE 2: A chain of references is established when one PCF item reference depends on the resolution of another. If a source item with original context depends on the resolution of its target item, where the target item is a derived context PCF item reference, the source item should be replaced by embedding the unresolved target item with respect to the root container item and then the derived item can be resolved with respect to the root service item.

EXAMPLE: The following XML fragment written according to the PCF XML schema shows the reuse of a TextBox component in two scenes, using both original and derived references:

```
<PCF>
  <String name="title" value="this is not resolved here"/>
  <TextBox name="heading">
    <!-- ref cannot be resolved here: root service item unknown -->
    <String name="content" href="#../title" context="derived"/>
    <!-- other properties here -->
  </TextBox>

  <Service name="reuse">
    <!-- service properties omitted -->
    <Scene name="scene1">
      <String name="title" value="First scene"/>
      <TextBox href="#../../heading" context="original"/>
    </Scene>

    <Scene name="scene2">
      <String name="title" value="Second scene"/>
      <TextBox href="#/heading" context="original"/>
    </Scene>

  </Service>
</PCF>
```

When all references are resolved, the following XML fragment is equivalent:

```
<PCF>
  <Service name="reuse">

    <Scene name="scene1">
      <TextBox name="heading">
        <String name="content" value="First scene"/>
        <!-- other shared properties here -->
      </TextBox>
    </Scene>

    <Scene name="scene2>
      <TextBox name="heading">
        <String name="content" value="Second scene"/>
        <!-- other shared properties here -->
      </TextBox>
    </Scene>

  </Service>
</PCF>
```

6.4.4 Map reference items

A **map reference item** enables all the content of one map item, the **target map item**, to be copied into the content of another, the **source map item**. A map item may contain zero or more reference items, allowing the content of zero or more map items to be copied into the content of another.

A map reference item shall have a **href** property with a value that is a PCF path, as defined in clause 6.4.2.2, that shall resolve to the target map item.

All PCF map reference items shall be resolved in their original context.

NOTE: An author can structure a service using collection items defined by PCF item references with a derived context to achieve the effect of a map reference item being resolved in a derived context.

All child items in the target map item shall be copied into source map item in the order they are declared so that they replace the map reference item. If the target map item is empty, resolution of the map reference item shall remove the reference item from the source map item.

If, as part of the map reference item resolution items, more than one PCF item has the same name within the source map item, the rules below shall apply before the resolution shall be considered complete. For each set of PCF items with the same repeated name within the source map item:

- the item with the repeated name with highest index shall remain in the source map item in the same relative position in the sequence to other PCF items;
- all other PCF items with that name shall be removed.

EXAMPLE: The following fragment of XML, written according to the PCF XML schema, shows a map reference item represented as the "Copy" element being used to combine two collections into the properties into a component.

```
    <Collection name="textinfo">
      <String name="content" value="Press red"/>
      <Color name="textcolor" value="#FF0000"/>
    </Collection>

    <Collection name="boxinfo">
      <Size name="size" value="100 30"/>
      <Position name="origin" value="150 210"/>
    </Collection>

<TextBox name="instruction">
      <Copy href="#../textinfo">
      <Copy href="#../boxinfo">
      <!-- next property overides that from "boxinfo" Copy -->
      <Position name="origin" value="600 60"/>
</TextBox>
```

6.4.5 Parameter items

Behaviour within a component item may only access components within the same scope as the enclosing component item (see clause 6.3.7). **Parameter items** provide a means to allow behaviour to access PCF components outside its enclosing context.

A component declared as a peer of a second component can be passed into in the context of the second component using parameter items. Parameter items appear as if they are locally declared within the component they are passed into, thus they may be accessed from within the second component's context. Each parameter item shall be given a new name, an alias, that must be unique within its enclosing component and which shall be used to refer to the item within the component's context.

The PCF component has a **parameter-list** property as introduced in clause 6.2.5.2. This is a space delimited list of **parameter pairs** that contain names that map external components to their aliases. The first item in each parameter pair is the path to the first component. The second item in the parameter pair is the name that shall be used for the item within the second component. The two items shall be separated by a colon.

To enable strong typing and validation, any component that uses parameter items shall explicitly declare each parameter item alias and component class that it expects to be provided by the enclosing context.

NOTE: In the PCF XML schema, parameter items are represented by the "Parameter" element. Parameter item declaration allows parameterized component instances to be validated independently of their enclosing context and allows a transcoder to re-use the implementation of such items within their platform-specific output.

Each declared parameter item shall be provided with a reference to a component instance in the enclosing context and shall have the same component class as the component instance it refers to. It is an error if either of these conditions fails.

Within a component, a parameter item can be accessed exactly as if it had been declared locally as a child component.

EXAMPLE 1: The following XML fragment shows how parameter items can be used to share a common variable between two scenes. Behaviour in both scenes can access the variable, which does not lose its value on scene transition.

```
<Service name="interactive_quiz">
    <IntegerVar name="score"/>
    <Scene name="level1" parameter-list="score:running_total">
        <Parameter name="running_total" class="IntegerVar/>
        <!-- Behaviour here can now use running_total as if it was local -->
    </Scene>
    <Scene name="level2" parameter-list="score:running_total">
        <Parameter name="running_total" class="IntegerVar/>
        <!-- Behaviour here has access to the same variable as in level1 -->
    </Scene>
</Service>
```

EXAMPLE 2: The following XML fragment illustrates passing a parameter down the PCF component hierarchy to make it accessible by a child component.

```
<Service name="game">
    <StringVar name="userName"/>
    <Scene name="beginner" parameter-list="userName:name">
        <Parameter name="name" class="StringVar"/>
        <StaticELC name="score_banner" parameter-list="name:name">
            <Parameter name="name" class="StringVar"/>
            <!-- The userName variable in the service can be read and written
here -->
        </StaticELC>
    </Scene>
</Service>
```

EXAMPLE 3: The following XML fragment illustrates the use of a PCF path to pass an item within a Collection into a component. It also shows a parameter list containing two parameter pairs separated by a space.

```
<Service name="beginner">
    <Collection name="scoreboard">
        <StringVar name="name"/>
        <IntegerVar name="score"/>
    </Collection>
    <Scene name="scene1" parameter-list="scoreboard/name:name
scoreboard/score:score">
        <Parameter name="name" class="StringVar"/>
        <Parameter name="score" class="IntegerVar"/>
        </StaticELC>
    </Scene>
</Service>
```

6.4.6 Navigation reference items

Navigation reference items represent navigable links between PCF items or external items to a PCF service description.

Navigation reference items shall be members of the URI data type defined in clause 6.2.3.14. The value property of a navigation reference item shall be known as the **navigation target**. The navigation target shall be defined by a PCF path, as defined in clause 6.4.2.2, that can be resolved to a PCF item.

The scene navigation action defined in clause J.1.1 can be used to change the currently active scene during an interactive service session. Calling the scene navigation action results in the following steps:

- Checking that the navigation target resolves to a scene item.
- Deactivating the current scene item.
- Activating the navigation target scene item.

A navigation reference item can be resolved during an interactive session and shall always resolve in the hierarchical referential context of the active service item that is its ancestor.

NOTE: Navigation reference items are PCF items, so a navigation reference item can be defined by a PCF item reference and this can be resolved in a derived context.

EXAMPLE: The following XML fragment, which is written according to the PCF XML schema, shows how navigation reference items can be combined with PCF item references.

```
<PCF>
  <ELC name="template1">
    <SceneNavigate>
      <URI name="target" href="#../../content/next" context="derived"/>
    </SceneNavigate>
  </ELC>

  <Service>
    <Scene name="scene1">
      <ELC href="#/template1"/>
      <Collection name="content">
        <URI name="next" value="#~/scene2"/>
      </Collection>
    </Scene>
  </Service>
</PCF>
```

6.5 Uniform Resource Identifiers

6.5.1 General usage

The PCF supports the URI [9] scheme for making links between items of content and external resources to a service description. The Uniform Resource Identifier specification provides a means to identify a resource in a location-specific manner, using the Uniform Resource Locator (URL) syntax, or in a location-independent manner, using the Uniform Resource Name (URN) syntax.

URNs may be used to identify a resource whose location will vary from platform to platform, for example broadcast video and audio streams. The URN provides a unique name for a resource but does not define its location, so achieving portability. However, this does require a target platform to provide a means to resolve a unique name to an actual resource on the target platform.

6.5.2 URN syntax in the PCF

All PCF URNs shall be encoded according to the following syntax:

```
"urn:x-dvb-pcf:" <providerID> ":" <date> ":" <contentID> (":" <revision>)?
```

All PCF URNs shall start with "urn:" as required by the URN specification [10]. The namespace identification syntax of the urn shall always be "x-dvb-pcf" and this shall always be followed by a colon ":".

The next item in a PCF URN shall be a **provider identifier** ("providerID"). This shall be an Internet domain name that is owned by the provider at the date on which a service description is published. This shall provide a globally unique identifier for the source of the content. This provider identifier shall not itself contain a colon. Provider identifiers shall be followed by a colon ":". If the provider identifier is not explicitly defined then the default provider identified "dvb.org" shall be assumed.

The next item in a PCF URN shall be a **date** ("date") value that indicates the date of publication of the resource referred to. This can be used to distinguish between different daily episodes of a news programme. This value shall either be empty or a date value consistent with the PCF date data type, as described in clause 6.2.3.3. An empty date value shall indicate that the content referred to is the most recent version or a live stream. Dates, even empty dates, shall be followed by a colon ":".

The next item in a PCF URN shall provide a **content identifier** ("contentID"). This, along with its publication date and revision number, should uniquely identify the item of content referred to by the URN to the content provider. The content identifier shall not itself contain a colon.

The final item of a PCF URN is an optional **revision** number ("revision") specification. This is a non-zero positive integer that, for any two items with the same URN without a revision number specified, shall be higher number for the more recent version. Where a revision number is specified, a colon shall be used to separate it from the preceding content identifier. A URN with no revision number specified shall match with the most recent version of a piece of content with the same base URN and a revision number specified.

EXAMPLE 1: The following PCF URN could be used to reference a live video stream from the Centre Court at the Wimbledon Tennis Championships 2005. No date field is specified as the URN refers to a live stream.

```
urn:x-dvb-pcf:bbc.co.uk::Wimbledon2005_CentreCourt_LiveFeed
```

EXAMPLE 2: The following PCF URN refers to an audio clip that should be played out in response to a user action within a PCF service. The **date** field distinguishes the audio clip from other clips that may be available from the provider.

```
urn:x-dvb-pcf:sky.com:2005-03-21:SMS_Send_Audio
```

EXAMPLE 3: The following PCF URN refers to a collection of updated scores during a rugby championships. The revision number updates every time the resource is updated with new score information.

```
urn:x-dvb-pcf:bbc.co.uk::six_nations_2005_latest_scores:145
```

6.6 Marked up text representation

Marked up text value items allow PCF service authors to apply styling to sections of flowed text content by incorporating mark up tags, as defined in annex F. Marked up text tags allow both inline styling and tabulated structuring of text.

Where visual component content is rendered from a marked up text value item, the component's styling shall be inherited as the base properties for the initial rendering of the text and then can be overridden by locally specified mark up tags.

NOTE 1: Where visual component content is represented as a string value item, the styles described in the properties of the component will be used to present the entire string.

The rendering of inline mark up tag presentation shall not be compulsory.

NOTE 2: Many platforms are not able to support a change of styling part way through a flow of text. A transcoder may choose which inline mark up tags are rendered. Transcoders may implement alternative approaches for rendering inline styling properties.

EXAMPLE 1: Text with unsupported inline styling may be rendered as an image at the transcoder.

Marked up text tags do not have an equivalent of the **name** or **exact** properties of a value item, as specified in clause 6.2.1.3. No error shall be reported if a particular styling property cannot be achieved. The PCF referencing model shall not be part of the marked up text representation and so no reference can be made to a style tag to, for example, reuse it.

EXAMPLE 2: The following fragment of XML shows how data for a table could be represented as a single marked up text item.

```
<MarkedUpText>
    <body xmlns="http://www.dvb.org/pcf/x-dvb-pcf">
        <table>
            <tr><th>Team</th> <th>Played</th> <th>Points</th></tr>
            <tr><td>Wales</td> <td>4</td> <td>8</td></tr>
            <tr><td>Ireland</td> <td>4</td> <td>6</td></tr>
        </table>
    </body>
</MarkedUpText>
```

Marked up text value items shall be represented as XML documents [5] that conform to the schema "x-dvb-pcf.xsd".

NOTE 3: Where appropriate, the relevant structures and element names of HTML [8] have been adopted for the PCF.

7 General component specification

7.1 Overview

Components are the functional building blocks within a PCF service description. All standard presentational and non-custom behavioural aspects of a PCF service are described by combining different types of components. The PCF defines a standard toolkit of components to implement commonly used interactive service functionality.

The complete set of standard PCF components is introduced in clause 7.5 and is fully specified in annex A.

PCF also defines a behaviour notation sufficiently complex to allow components from the standard toolkit to be extended and combined. This behaviour notation is described fully in clause 9.

Components are initialized and manipulated using a standard interface model and a standard set of scoping rules. Whilst the present document defines **what** the intended visual appearance and behaviour of each component type shall be, it does not specify **how** this should be implemented. The actual implementation may vary significantly on a platform-by-platform basis, depending upon that platform's underlying API and resources.

Each PCF component shall be implemented to provide an author's intended user experience according to the component's specification. This is to ensure portability of PCF services.

The PCF component object model consists of two separate, but closely associated, object models:

- The **Component Specification Model** is described in clause 7.2. This is a meta-model used to define which component classes are available for use within the PCF. It is not part of the PCF model for describing interactive services; rather, it describes the interface characteristics of each component type.
- The **Component Instantiation Model** is described in clause 7.3. It comprises the set of objects that are used to declare and manipulate components within a PCF document. This is part of the PCF model for describing interactive services and provides the core building blocks of PCF service descriptions.

An XML schema representation of the component specification model, called the **Component Definition Syntax**, is provided with the present document (see clause G.2.1).

The definition of a component class shall consist of:

- a provider and a name, together being a unique identifier of the component class;
- its interface definition according to the component specification model;
- a textual description of the details of this definition;
- a description of the behaviour that the component implementation is required to exhibit.

7.2 Component specification model

7.2.1 Overview

The standard PCF component types are defined in clause 7.5. In each case the formal definition consists of up to three sections. These are: the **Interface Definition** in the **Component Definition Syntax** (described in 7.2.2); a **Textual Description** section that expands upon the interface definition; and for elements with intrinsic behaviour, a **Behaviour Specification**.

These sections are defined in detail in clauses 7.2 to 7.7

Target platforms shall implement the standard PCF components in accordance with their component specifications in annex A.

New component types that are added to future versions of the PCF specification shall also be defined using this component specification format.

It is recommended that the component specification format is used to define any custom components that may be created. Defining components in a standard format will aid compatibility with other PCF tools, help authors to understand the component, and facilitate standardization if the custom component is later adopted as a standard PCF component.

7.2.2 Interface definition

The component specification model is the meta-model used to define the characteristics of each **component class**. This object model, and its associated component definition syntax, shall be used to specify the component classes that are available for use within a PCF service description.

NOTE: The component specification model is not part of the actual PCF for describing interactive services, but has been defined within the present document to provide a solid foundation upon which the PCF Component Model is built. However, as component type definitions in component definition syntax are schema-conformant XML they may be used by a transcoder for PCF validation.

This model is illustrated in figure 9.

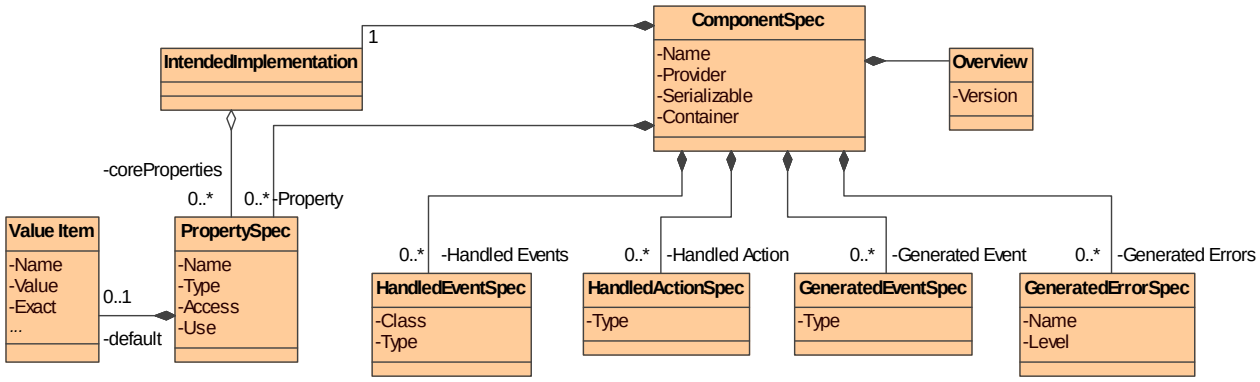

Figure 9: Component specification model

This model is implemented as the PCF **Component Definition Syntax** (CDS). The CDS schema is defined in file "component-syntax.xsd", which is provided with the specification. Objects within the component specification model are used to define component classes, and the interface characteristics of those component classes. The objects defining these interface characteristics, shown in figure 9 are as follows:

- A **ComponentSpec** item shall define a specific **Component Class**. A ComponentSpec item shall have these attributes:
  - A **provider**. This shall identify the creator of the component. Components defined within the present document shall have the default provider, "dvb.org".
  - An identifying **name**. This shall be unique amongst all component class names from the provider.
  - A **serializable** flag. This shall define whether the component class shall have a serialized form that may be used for data exchange.
  - A **container** flag. This shall define whether instances of the component class may contain child components.

A **ComponentSpec** item shall contain:

- A **Properties** item containing zero or more **PropertySpec** items. This defines the set of properties that may be set and/or read on an instance of the component class. These properties may be either defined directly using **PropertySpec** elements, or included as part of a property group using **PropertyGroupRef** elements.
- An optional **HandledEvents** item containing zero or more **HandledEventSpec** items. This defines the set of events that a component instance of this class may respond to during its lifetime. These events may be either defined directly using **HandledEventSpec** elements, or included as part of a handled event group using **HandledEventGroupRef** elements.

- An optional **HandledActions** item containing zero or more **HandledActionSpec** items. This defines the set of actions that a component instance of this class may respond to during its lifetime. These events may be either defined directly using **HandledActionSpec** elements, or included as part of a handled action group using **HandledActionGroupRef** elements.

- An optional **GeneratedEvents** item containing zero or more **GeneratedEventSpec** items. This defines the set of events that a component instance of this class may generate during its lifetime. These events may be either defined directly using **GeneratedEventSpec** elements, or included as part of a generated event group using **GeneratedEventGroupRef** elements.

- An optional **GeneratedErrors** item containing zero or more **GeneratedErrorSpec** items. This defines the set of error events that a component instance of this class may generate during its lifetime. This set shall contain all the errors that are generated as part of any inherent behaviour defined for that component. These errors may be either defined directly using **GeneratedErrorSpec** elements, or included as part of a generated error group using **GeneratedErrorGroupRef** elements.

- An **Overview** item. The overview item contains attributes detailing the component class's version. It is envisaged that some component types may evolve over time, so a version number allows different versions of the same component to be identified.

- An **IntendedImplementation** item. This has a single **coreProperties** attribute, and this attribute shall be a list of the properties that **must** be implemented on a target platform. These are the properties of a component that are required to ensure that all implementations provide an equivalent user experience.

7.2.2.1 Groups

Many component types have features in common. Different component types may:

- have common properties, such as colour or position;

- respond to the same events, such as key-press events;

- respond to the same actions, such as show/hide actions;

- generate the same events, such as OnSelect;

- generate the same errors, such as OnInvalidMediaType.

When this is the case, then instead of including the PropertySpec, HandledEventSpec, HandledActionSpec etc. items directly within the ComponentSpec object, the common features shall be declared separately from the component specification, as part of a group. This group shall then be included by reference from within the component specification. The component specification object model showing the ability to group items is illustrated in figure 10.

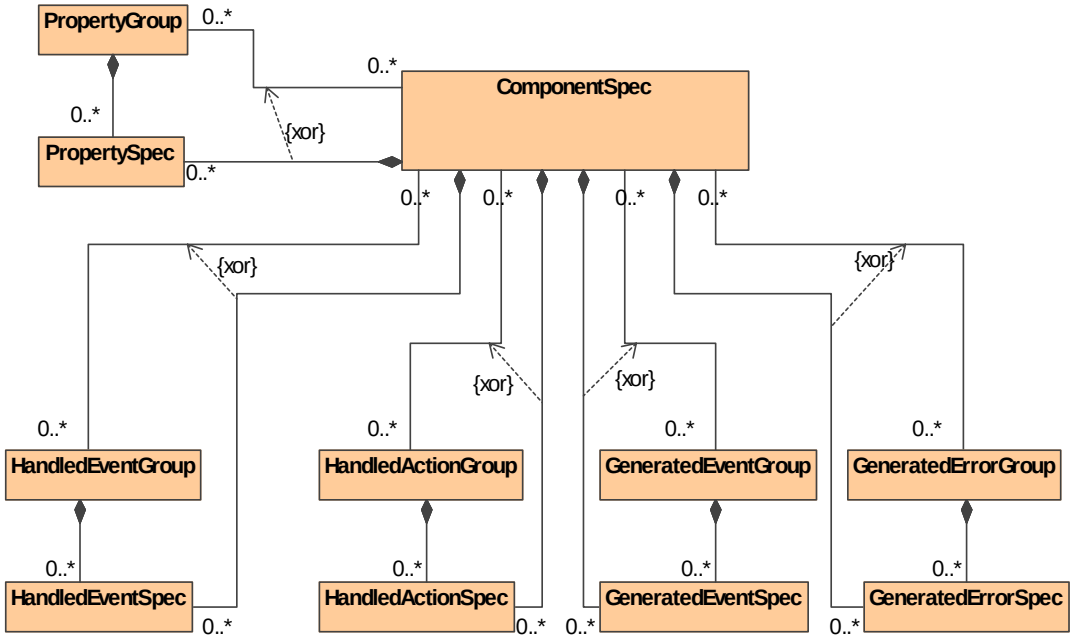

Figure 10: Component specification object model showing groups

The use of groups achieves consistency across component types. In the case of properties, it also allows property values to be cascaded, as described in clause 7.3.3.

- Every property defined within a property group shall have a name that is unique across all property groups.
- Every event defined within an event group shall have a name that is unique across all event groups.
- Every action defined within an action group shall have a name that is unique across all action groups.
- Every error defined within an error group shall have a name that is unique across all error groups.

PCF defines a set of property, event, action and error groups in annexes A and B. Standard PCF component type specifications use these groups in addition to individually defined items.

Custom components should use standard groups where appropriate. Custom groups can be defined and may be used in the definition of custom components.

NOTE: Custom groups shall observe the requirements for unique naming across all property groups.

EXAMPLE 1: All components that have a border, such as Rectangle, Polygon, TextBox and Button include the **border_properties** property group within their set of properties. The border-properties property group contains properties such as **linestyle** and **border-width**. This means that equivalent border properties can be referred to consistently across all component types. It also simplifies the specification of each type, as instead of specifying each border property for each component type, the border properties are specified once in the property group and then the group is included by each component specification.

When a component instance is declared within a PCF service, the properties that may be declared for that instance are the union of all properties within property groups specified for that component type, and any other individual properties specified for that component class.

A special feature of property items defined as part of a property group is that component instance declarations can inherit values for those items from an ancestor component item. This is explained in clause 7.3.3.

Property groups are used only as a means to declare properties within component type specifications and are not used within a PCF service description itself. Therefore when setting property values in an instance of a component the only distinction between properties from property groups and class specific properties is the inheritance which is allowed for property group properties.

EXAMPLE 2: A **Polygon** component has properties from the **border_properties** property group and also a class-specific property, **vertices**, specifying the vertices of the **Polygon**. The definition of an instance of the **Polygon** component type may list its property items in any order, intermingling border-properties group property items such as linestyle with the vertices property item and property items from other property groups used by the **Polygon** class.

7.2.2.2 Property specifications

The **ComponentSpec** item shall contain a **Properties** item. Within the **Properties** item there may be **PropertySpec** items, which declare properties specific to the component class, and **PropertyGroupRef** items, which reference properties contained in **PropertyGroup** items declared outside of the component.

PropertySpec items have the following attributes:

- A **name**. If the **PropertySpec** is part of a property group, the name shall be unique amongst all properties which are contained in property groups.
- A **type**. This shall specify the **value item** type, from the set of available PCF value types (integer, boolean, string, imageData etc) that shall be used to provide this property value within a component instance declaration. If a PropertySpec is of type **enumeration** then it shall contain or reference an **EnumerationSpec** item that defines the allowable values for the property. Enumeration specifications are defined in clause 7.2.2.3.
- A **use** specifier. This shall have a value of either "required" or "optional". If the use is specified as required, then all component instance declarations of this component type must be provided with a value for this property. If the use is optional then a **default** value may be specified within the property's specification.
- An **access** specifier. This specifies the level of accessibility and mutability of this property over the lifetime of a component. This attribute shall be one of these values:
  - **initializeOnly:** A property item that shall only be used to specify the initial value of the property. Such property items cannot be accessed from within the PCF action language;
  - **final:** A property item that can be used to specify the initial value of the property, and can be read but not modified from within the PCF action language;
  - **readWrite:** A property item that can be used to specify the initial value of the property, and can be read and modified from within the PCF action language;
  - **readOnly:** A property item that can only be read from within the PCF action language. Such property items are used to access dynamic component state.

Where the **use** is set to "optional", a default value may be provided through the inclusion of a default value item within the **PropertySpec**. The name of this item shall be **default** and its value is the chosen default value. If a component instance will not be complete without a value being available for a particular property then either the **use** shall be specified to "required", or a default value shall be supplied in the **PropertySpec**.

NOTE: See clause 7.5.2 for a naming convention for properties which are associated with particular component states.

7.2.2.3 Enumeration specifications

The enumeration data type is defined in detail in clause 6.2.2.3. **EnumerationSpec** items allow enumerations to be declared. They may be declared directly within the **PropertySpec** item in which they are used, or outside of any **ComponentSpec** items, and then referenced from within **PropertySpec** items using an **EnumerationRef** item. The relationship between **PropertySpec** items and **EnumerationSpec** items is illustrated in figure 11.

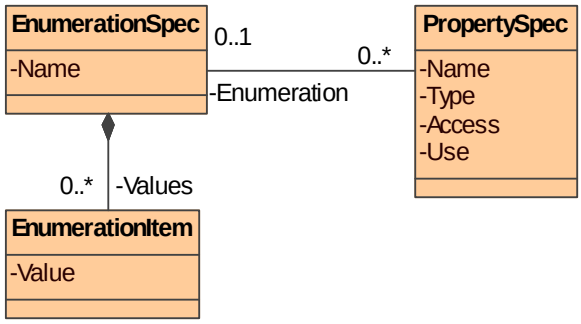

Figure 11: Enumeration Specification Objects

An EnumerationSpec shall:

- have a **name**;
- contain one or more **EnumerationItems**, each of which defines a possible value;
- have their default, and any values used in component instances, provided using a String value item.

7.2.2.4 Handled event specifications

Components can handle and generate events. Clause 7.4.5 describes how generated events are declared in a component class specification. Figure 12 illustrates how events are modelled in the component class specification.

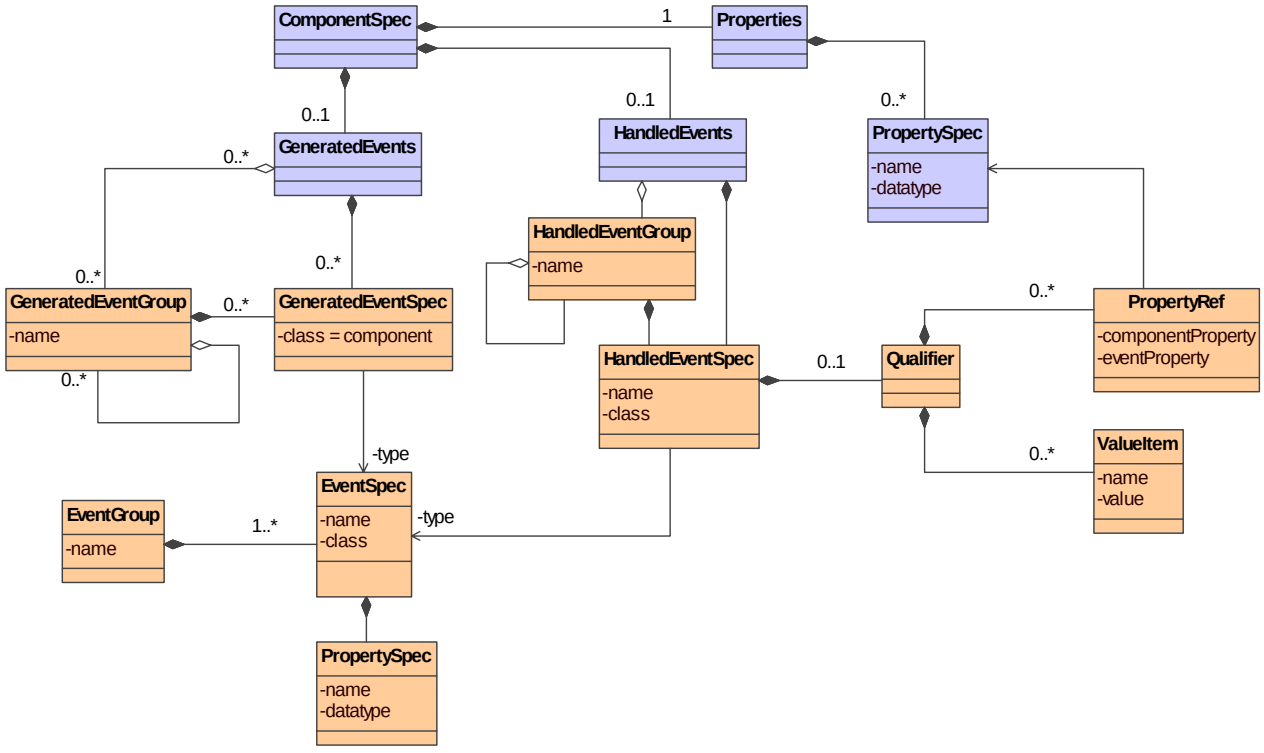

Figure 12: Component event model

The **ComponentSpec** item may contain a **HandledEvents** item. Within the **HandledEvents** item there may be **HandledEventSpec** items, which declare handled events specific to the component class, and **HandledEventGroupRef** items, which reference handled events contained in **HandledEventGroup** items declared outside of the component. The **HandledEventSpec** items associated with a **ComponentSpec** shall specify all the events external to the component that trigger behaviour within the inherent behaviour defined for that component class.

NOTE 1: **HandledEventSpec** items do not define new event types themselves; the specifics of individual event types, such as cause and associated data, are defined within annex B. **HandledEventSpec** items are similar to the **Trigger** item defined in clause 9.6.3.2; they refer to an existing event type and may define qualifier values against the properties of this event type.

HandledEventSpec items have the following attributes:

- A **class** specifier. This specifies what class of event this HandledEventSpec refers to. Components shall be able to handle events of class **user** or **system**. Components shall not be able to handle events generated by other components.

- A **type** specifier. This specifies the event type and associates the HandledEventSpec with an event specification in annex B.

A HandledEventSpec item may contain a qualifier, which restricts the events which are handled based on property values in the event. The qualifier shall contain:

- Zero or more **PropertyRefs**. These give the name of the property in the event and the name of a property within the component. The property within the component must be of the same type as the event property identified.

- Zero or more **ValueItems**. These give the name of the property in the event and a value. The value must be of the same type as the event property.

An event shall be handled only if the values of its properties match all of those specified in the handled event qualifier.

Where the qualifier contains or refers to an array value, an event shall be handled only if its property matches at least one of the values specified in the array.

NOTE 2: Unlike **Trigger** qualifiers, **HandledEventSpec** qualifiers may refer to component properties that can be manipulated at run-time.

7.2.2.5 Generated event specifications

The **ComponentSpec** item may contain a **GeneratedEvents** item. Within the **GeneratedEvents** item there may be **GeneratedEventSpec** items, which declare generated events specific to the component class, and **GeneratedEventGroupRef** items, which reference generated events contained in **GeneratedEventGroup** items declared outside of the component. The **GeneratedEventSpec** items associated with a **ComponentSpec** shall specify all the component events that are generated as part of the behaviour defined by the inherent behaviour for that component class. Figure 12 shows how events are included in a component class specification.

NOTE 1: GeneratedEventSpec items do not describe the events themselves; events are described fully in clause 9.2 and the specifics of individual events, such as cause and associated data, are defined within annex B.

GeneratedEventSpec items have the following attribute:

- A **type** specifier. This specifies the event type and associates the **GeneratedEventSpec** with an event specification in annex B.

NOTE 2: In contrast with **HandledEventSpec** items, **GeneratedEventSpec** items do not have a **class** attribute. This is because all generated events are of class **component**.

7.2.2.6 Handled action specifications

- The **ComponentSpec** item may contain a **HandledActions** item. Within the **HandledActions** item there may be **HandledActionSpec** items, which declare handled actions specific to the component class, and **HandledActionGroupRef** items, which reference handled actions contained in **HandledActionGroup** items declared outside of the component. The **HandledActionSpec** items belonging to a **ComponentSpec** shall specify all the actions that trigger behaviour within the inherent behaviour defined for that component class. **HandledActionSpec** items do not describe the actions themselves; actions are described fully in clause 9.4.

7.2.2.7 Generated error specifications

The **ComponentSpec** item may contain a **GeneratedErrors** item. Within the **GeneratedErrors** item there may be **GeneratedErrorSpec** items, which declare generated errors specific to the component class, and **GeneratedErrorGroupRef** items, which reference generated errors contained in **GeneratedErrorGroup** items declared outside of the component. The **GeneratedErrorSpec** items belonging to a **ComponentSpec** shall specify all the component errors that are generated as part of the behaviour defined by the inherent behaviour for that component class. **GeneratedErrorSpec** items do not describe the error events themselves; error events are described fully in clause 9.3.5, and the specifics of individual errors, such as cause and associated data. Commonly generated errors are defined within clause B.4 and component specific errors are defined within annex A.

GeneratedErrorSpec items have the following attribute:

- A **type** specifier. This specifies the error type and associates the **GeneratedErrorSpec** with an error specification in clause B.4.

7.2.2.8 Intended implementation

The **ComponentSpec** item shall contain an **IntendedImplementation** item. The **IntendedImplementation** item has:

- a **coreProperties** attribute. This is a list of the names of the component's properties which shall be implemented as a minimum by a target implementation.

It is the list of core properties that specifies the **degrees of freedom** with which a component type may be implemented. The core properties define what a component implementation **has** to implement; all other properties define what a component **should ideally** implement. As long as a target platform implements a level of component functionality somewhere between the mandatory level and the ideal level then that target platform shall be considered to implement the component.

These degrees of freedom allow lower-specification platforms to implement the minimum set of functionality whilst not constraining higher-specification platforms to this minimum.

7.2.2.9 Overview item

The **ComponentSpec** item shall contain an **Overview** item. The **Overview** item has:

- A **version** attribute. This indicates the version number of the component class which is being declared by the **ComponentSpec** item.
- It is envisaged that some component types may evolve over time, so the version number allows different versions of the same component to be identified.

7.2.3 Textual description

The textual description section of a component's class specification shall fulfil two purposes:

- to describe how the properties specified within the interface definition affect the appearance and/or behaviour of an instance of the component class;
- to provide a human readable description of the component class.

Each property specified within the interface definition shall have some associated text within the textual description section to explain what the significance of the property is for this component type.

The explanation shall be sufficiently detailed so that it, in conjunction with any relevant aspects of the behaviour specification, completely specifies how the initial value, and subsequent modifications to the value, of the property affect the appearance and behaviour of the component instance to which the property applies.

In addition to the description of individual values there shall be a description of the overall component. This shall provide an overview of the component's appearance and behaviour and shall be in sufficient detail so that it, in conjunction with the property descriptions and the behaviour specification, provides a complete description of the appearance and behaviour of a component instance of that class.

7.2.4 Behaviour specification

The behaviour specification of a component shall define the behaviour of the component class, both in terms of how an instance of the component class interacts with the rest of the service across its interface and in terms of how this interaction is reflected in the component instance's appearance and the nature of any subsequent interactions.

Where possible, a model of the component class's behaviour should be described using a UML **statechart**. The description of behaviour in this way does not mean that an implementation of the component class on a target platform should actually contain the statemachine defined by the statechart; it means that the component shall behave **as though** it contained the statemachine.

The statechart shall define:

- the internal states that the component may be in;
- the component's response to external events and actions;
- the state transitions that may occur in response to internal or external events and actions;
- changes to appearance in response to events, actions and or state transitions;
- the conditions under which the component will generate **component events** and **component errors**.

Where necessary the statechart should be accompanied by additional descriptive text.

If the internal behaviour of a component class is too complex for complete description, then it is acceptable to only provide a low-level description of how the component behaves at its interface. This shall include a description of all the events and actions that shall be responded to, and all the component events that shall be generated, and under what circumstances. In such a case the internal behaviour of the component does not need to be modelled as a statechart; however, it must still be described in sufficient detail to portray what the intended user experience of the component's internal behaviour.

EXAMPLE: The internal behaviour of a component to implement a Tetris-like game may be too complex to practically model as a statechart. In such a case the behaviour may be explained as a selection of screen-shots with accompanying text.

The omission of a statechart for a component class will inevitably lead to more scope for different implementations to provide differing user experiences. It is therefore recommended that statecharts are used wherever it is practicable to do so.

7.3 Component instantiation model

The second aspect of the PCF component model, the **component instantiation model**, is part of the PCF. Items from the model are used to declare component instances within a PCF service description. The component instantiation model is implemented within the PCF XML schema.

The **component specification model** is related to the component instantiation model in that the former defines what is expected and valid within the latter. The relationships between items in the two models are illustrated in figure 13.

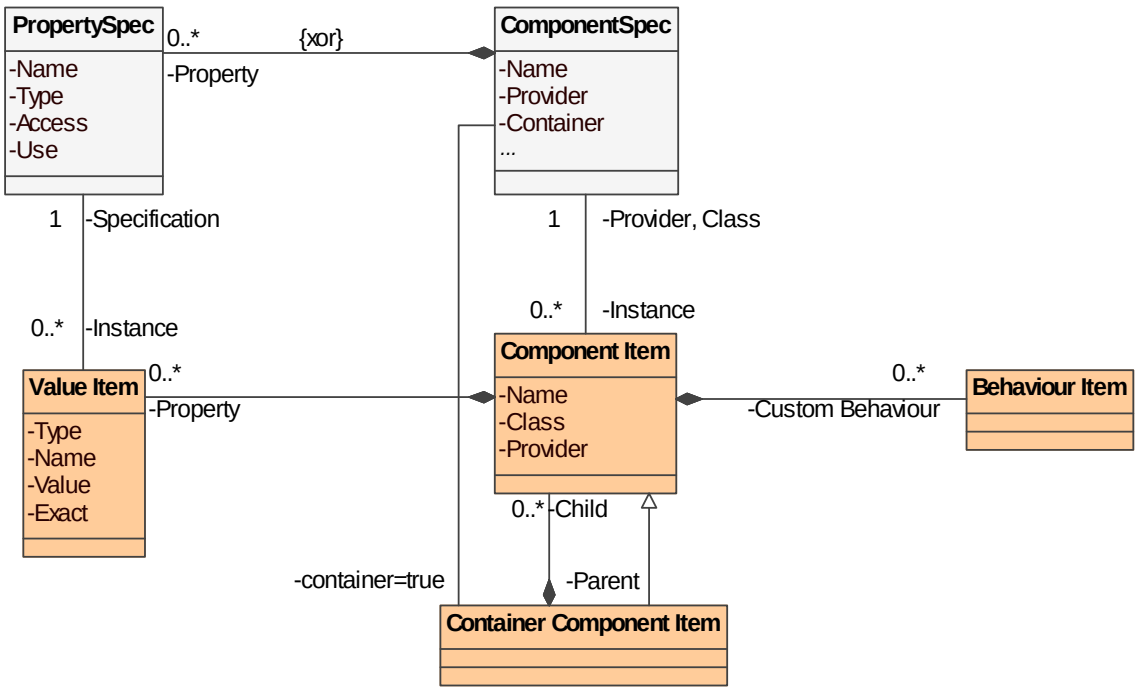

Figure 13: Relationship between component specification and instantiation items

7.3.1 Component

All components instances shall be of the ordered map data type, as defined in clause 6.2.5.2. This allows PCF items to be contained within the component. Every component instance shall have a **name** that uniquely identifies it within the scope of its declaration.

All components shall have a component **class** and optionally a **provider**. If the provider is not specified it defaults to "dvb.org". Together the component **provider** and **class** link a component instance with its associated **component class specification**, as defined by a ComponentSpec item. A component instance shall adhere to the rules specified by its associated ComponentSpec item.

A component may contain:

- zero or more **value** items to define the initial values of its properties;
- zero or more **behaviour** items to define custom extensions to the inherent behaviour of the component instance;
- zero or more child component items if the component item is a container component. This will be indicated in the ComponentSpec item for the component class, as described in clause 7.2.2.

7.3.2 Properties

Each **ComponentSpec** component class definition specifies a set of properties that affect the appearance and behaviour of component instances of that class.

Within a component instance declaration the properties are defined by a set of value items that specify the initial property values, and hence the initial state, of that component instance.

- A value item has a **name** that may match the name of a PropertySpec item defined within the component's class specification. If it does, then the value item shall provide the value of that property.
- A value item has a **data type**. This shall be one of the data types defined by the PCF architecture in clause 6.2.
- A value item has a **value**.

- The data type of a value item with a particular name must match the data type specified by the **PropertySpec** item with the same name in the component's class specification. If the **PropertySpec** data type is an enumeration, then the value item shall be of type **String**, and the value must be one of the values given in the **EnumerationSpec**.

Properties which need initial values may obtain these from one of four sources. The value is obtained using the following precedence order. If no value is specified then the next option is used:

1) A value is declared in the component instance.

2) If the property is defined in a PropertyGroup then if any of the component instances in the ancestral hierarchy has a declared value for the property this shall be used. If more than one ancestor declares a value, the value declared in the most immediate ancestor shall be used. See clause 7.3.3 for more information on cascaded properties.

3) If a default value has been specified for the property in the PropertySpec then this shall be used.

4) If the property **name** has a "-focus" or "-disabled" suffix then the value for the property named similarly, but without the suffix shall be used.

NOTE 1: If the **use** attribute of the **PropertySpec** item defining the property is set to "required" then a value must be obtained in step 1 or step 2. If no value has been obtained after step 2 then the component instance is invalid.

If a property needs an initial value, but one has not been obtained then the component instance is invalid.

It shall be possible to modify and query a defined subset of these properties using the PCF's action language. This shall be to determine and change the state of a component instance. The ability to initialize, modify and query a specific property of a component instance is defined by the **access** attribute of that property's PropertySpec item.

NOTE 2: All aspects of a component instance are specified by its property items. Some of these properties may conventionally, in a language such as HTML, be considered content. Other properties specify presentational aspects such as colour, styling etc. PCF does not make an explicit distinction between content and presentation. All property items can use the PCF's flexible referencing and item grouping mechanism, as specified in clause 6.4.1, to allow the service author to partition a service description in the most appropriate way.

7.3.3 Cascaded properties

It shall be possible for property items that are specified as a member of a PropertyGroup to be cascaded through the component hierarchy within a service description, so that child components may inherit properties that are defined within their ancestral hierarchy.

If a component class contains a property that is specified within a PropertyGroup, and an instance of this component does not contain a value item explicitly providing a value for the property, then the property shall be inherited from the nearest component in the ancestral hierarchy providing a value. This inheritance takes place in the context of where the component is being used; this is the derived context.

EXAMPLE: A **Rectangle** component is declared within a **StaticELC**. The **Rectangle** instance has no **fillcolor** or **linestyle** property values specified. The container in which the **Flow** is presented has a fillcolor property value of red (#FF0000) specified but no linestyle. In this case the **Rectangle** will inherit the parent container's fillcolor property, and so will also have a fillcolor of red. As it is unable to inherit the linestyle property (assuming linestyle is not defined higher up the ancestral hierarchy) it will use the default linestyle property value of "solid".

7.3.4 Component implementation tolerance

PCF is intended to be portable across many platforms, each with their own characteristics. As such it is accepted that on different platforms certain features may not be rendered exactly the same, but will always provide an equivalent user experience. The permissible tolerance within which features may be rendered is defined by the **IntendedImplementation** element within the component class specifications in annex A.

The IntendedImplementation element defines the minimum implementation that a target platform must provide for this component class. It has a single **coreProperties** attribute. This attribute lists the properties of the component that are essential to providing an equivalent user experience.

Properties in the coreProperties list of a supported component shall be implemented by a target platform.

All properties that are specified for a component class but not included in the coreProperties list are non-mandatory. These may be optionally implemented by a target platform.

NOTE 1: To provide an optimal user experience a target platform should implement as many non-mandatory properties as possible.

A service author may specify that a feature should by implemented exactly as specified by setting the **exact** flag of the relevant property value item to **true**.

A property value item with the exact flag set to true must be implemented exactly by a target platform, even if that property is not specified as a core property of the component class.

NOTE 2: The ability to specify that a property must be rendered exactly is provided so that, in exceptional circumstances, a service author can prevent a target platform from approximating or ignoring a property where this would be inappropriate. Forcing properties to be rendered exactly will reduce the portability of a service.

7.4 Component behaviour

7.4.1 Behaviour overview

The PCF generic component model allows the state of a component instance to be modified during its lifetime. The model also allows a defined subset of the inherent behaviour of a component instance to be extended. These aspects of custom behaviour are described using one or more **behaviour items**. The behaviour items available within PCF, i.e. statemachines, event handlers and action items, are described fully in clause 9. Custom behaviour may be used to:

- extend the behaviour of individual components;
- connect the behaviour of a number of components together within a container or **Scene**.

It is possible to modify the state of a component instance in two ways:

- by changing one or more property values;
- by invoking an **action**.

In both cases these changes would be defined using the PCF **action language**.

A component may generate **events** to provide notification of a restricted set of internal state transitions. This allows the behaviour of a component instance to be extended. Generated events may be responded to by behaviour items declared either within the component instance itself, or within the component's ancestors.

7.4.2 Accessing component properties

It shall be possible to read, using the PCF action language, the component property values that are defined within their property specification that have an **access** type of **final**, **readOnly** and **readWrite**.

It shall be possible to modify, using the PCF action language, the component property values that are defined within their property specification to have an **access** type of **readWrite**.

When a component instance has one or more of its property values updated, the underlying implementation of the component instance shall update itself to reflect this change.

Property read and write operations shall be synchronous: when a component instance has a property value modified and then immediately read, then the underlying component implementation shall ensure that the value returned from the read operation shall be the value set in the modify operation.

EXAMPLE: If the fillcolor property of a **TextBox** is modified from green (#00FF00) to blue (#0000FF) and then immediately read, the value returned from the read shall also be blue (#0000FF). This shall be the case even if the underlying graphics implementation means that there is a delay before the representation on the screen is refreshed with the new fillcolor.

The syntax and mechanism for modifying and reading property values is described fully in the behaviour section, clause 9.4.6.

7.4.3 Handled events

Some component types have inherent behaviour that may respond to **events**.

The events that a component of a particular type may respond to shall be defined by the set of **HandledEventSpec** items within the component's class specification.

The set of events that are understood by each component class are detailed in annex A.

The target platform shall implement components so that they respond to events appropriately, as defined by the behaviour specification within the component's class specification.

EXAMPLE: A **Button** component will respond directly to a user key-press without the need for any additional behaviour to be specified; it will change its appearance to show that it is in an "active" state.

7.4.4 Handled actions

Component **actions** are commands that are sent to a specific component instance, and which direct that component instance to perform some operation and/or state change.

The actions that may be sent to a component of a particular type are defined by the set of **HandledActionSpec** items within the component's class specification.

Actions may have zero or more parameters to provide the component with additional data to perform that action. These are specified by **parameter** items within the HandledActionSpec item.

The target platform shall implement components so that they respond to actions appropriately, as defined by the behaviour specification within the component's class specification.

The syntax and mechanism for sending actions to components is described fully in the behaviour section, clause 9.4.

The set of actions that are understood by each component class are detailed in annex A.

EXAMPLE: The ReturnPath component implements a **request** action to submit a transaction to a remote server. This action expects a **Transaction** component as a parameter.

7.4.5 Generated events

In addition to responding to external stimuli from the PCF service such as events, property changes and actions, some components shall also be able to generate **component events**. This enables them to provide notification of internal state transitions to the wider PCF service in which they are contained.

The events that a component of a particular type may generate are defined by the set of **GeneratedEventSpec** items within the component's class specification.

The events generated by each class of component are specified in annex B.

The target platform shall implement components so that they generate events appropriately, as defined by the behaviour specification within the component's class specification.

EXAMPLE: In addition to changing its visual appearance, the **Button** component, when selected, will also generate an **OnSelect** component event. This event may then be handled by a custom behaviour item elsewhere in the containing **Scene** to add some additional custom behaviour - for instance by invoking a transfer action on a **ReturnPath** component elsewhere in the **Scene**.

Component events may have zero or more parameters to provide additional data about the event.

7.4.6 Generated errors

As well as generating events during their normal lifetime, some component types may also generate errors to provide notification of an internal problem.

The errors that a component of a particular class may generate are defined by the set of **GeneratedErrorSpec** items within the component's class specification.

The errors generated by each class of component are specified in annex A.

The target platform shall implement components so that they generate errors appropriately, as defined by the behaviour specification within the component's class specification.

EXAMPLE: An **Image** component will generate an **OnInvalidMediaType** error if the type of media does not match on of the supported image/* types.

7.4.7 Component scope

The scope within which components are modifiable by behaviour items is strictly defined by the PCF. These rules are defined fully in clause 6.3.7.

The order in which events generated by components are made available to other components is governed by the component event propagation model. This is defined in clause 9.3.4.

7.5 Defined PCF component classes

7.5.1 Overview

Components in the PCF can broadly be grouped into the following categories:

- **Visual components:** These components contribute to visual aspects of the viewer experience.
- **Non-visual components:** These components have no visual appearance but provide some aspect of service functionality.
- **Container components:** These components may contain a number of other components. Container components may contain custom behaviour to specify interactions between their child components.

The PCF does not specify components in a class hierarchy in the object-oriented sense. This is because the PCF does not specify how a component should by implemented, only how the component is intended to appear and behave. As such, it is possible that the implementation on a particular platform uses a completely different class hierarchy.

7.5.2 Visual components

Table 22 lists the visual components available as standards within PCF:

Table 22: PCF visual components

Component Type	Description
AxisLine	Draws a horizontal or vertical line.
Background	Draws an image or a solid colour as the background to a scene.
Button	Implements the appearance and behaviour of a button, radio button or check-box.
Clock	Implements the appearance and behaviour of a clock.
ConnectStatusImage	Defines a location within the reference screen where platform specific graphics may be rendered to represent the connection state of the return path component.
Ellipse	Draws an ellipse.
HintTextBox	Draws plain text "hints" within a defined rectangular area in response to focus events on visible components.
Image	Displays an image.
ImageAnimated	Displays a set of images in sequence to create an animation.
ImageScalable	Displays an image that is scalable to the size available.
Line	Draws a line between two arbitrary points.
Menu	Implements the appearance and behaviour of a menu.
NumericNavigator	Allows a user to enter a number and then navigates to a scene linked to the number.
PickList	Draws a "pop-up" containing a list of textual items, and allows the user to select an item.
Pixel	Draws a pixel.
Polygon	Draws a polygon based on an arbitrary set of points.
RadioButtonGroup	Groups a set of buttons into a mutually-exclusive select group.
Rectangle	Draws a rectangle.
SpinControl	Implements the appearance and behaviour of a spin control.
TextBox	Draws a text box.
TextInput	Implements the appearance and behaviour of a text input field.
Ticker	Implements the appearance and behaviour of a marquee ticker.
Subtitles	Controls the presentation of subtitles, defined as an elementary stream of a Stream component.
Video	Controls the presentation of a video, defined as an elementary stream of a Stream component.

All visual components shall be in either the "Visible" or "Hidden" state (see figure 14).

By default, all visual components are able to take focus, however this may be overridden either in the component's class specification or in the instance declaration by setting the focusable property to false.

Components that are focusable shall implement the statechart defined in figure 14.

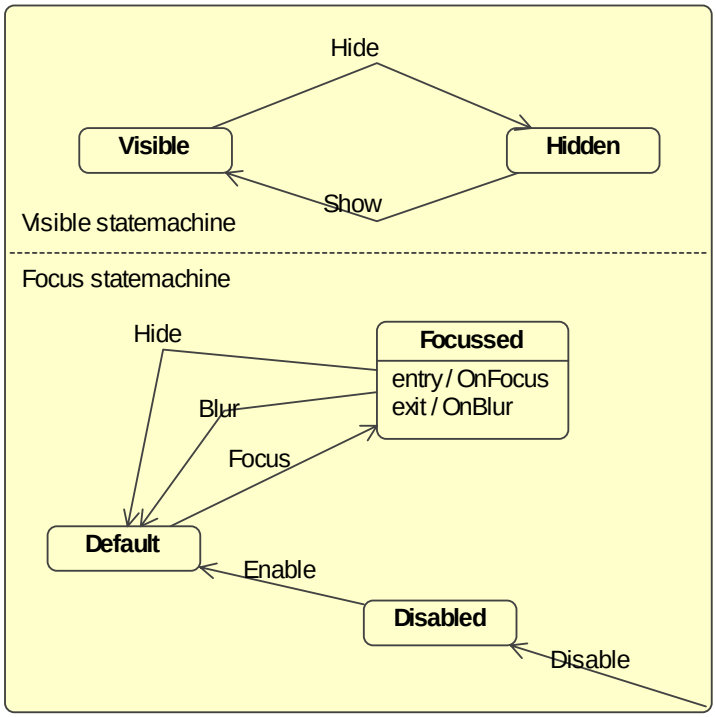

Figure 14: Visual component behaviour

For a non-focusable component, the Focus statemachine in figure 14 is reduced to just the "Default" state.

When a focusable component is in the "Disabled" state it can still have a visual appearance, if one is defined, but it shall not take focus.

When a focusable component is in the "Default" state it shall have a visual appearance and shall be focusable.

The only dependency between the two statemachines shall be that when a component is in the "Hidden" state it shall not take focus.

The "Show" event shall occur when the **visible** property is set to true. The "Hide" event shall occur when the **visible** property is set to false. The "Enable" event shall occur when the **enabled** property is set to true. The "Disable" event shall occur when the **enabled** property is set to false. The "Focus" event shall occur when the component is given focus. The "Blur" event shall occur when the component loses focus.

Only a visible component with a **focusable** property set to true shall be able to take focus, and the point at which it takes focus shall be determined by its **tabindex** property.

Some component classes may be specified to automatically alter their appearance depending on their state without the need for any action description. In order to allow the author of a service description to control such changes in appearance, a component specification may associate properties with each component state. This association is achieved through the use of standard suffixes to the property name:

- To associate a property with the "Default" state there shall be no suffix.
- To associate a property with the "Focused" state the suffix "-focus" shall be used.
- To associate a property with the "Disabled" state the suffix "-disabled" shall be used.

There is no requirement for the properties associated with a particular component state to completely describe all aspects of a component's visual appearance.

For each aspect the value is taken from the first property found with a value (either explicitly declared or provided as a default in the component class specification) using the order of precedence defined in table 23.

Table 23: Order of selection for properties associated with focus state of visual components

Current state	Default	Focused	Disabled
First choice	(default)	-focus	-disabled
Second choice		(default)	(default)

7.5.3 Non visual components

7.5.3.1 Functional components

Table 24 lists the functional non-visual components available as standard within PCF.

Table 24: PCF functional non-visual components

Component Type	Description
Audio	Controls the presentation of an audio, defined as an elementary stream of a Stream component.
CurrentTime	Used to obtain the current time.
Indicate	Provides a simple voting mechanism using a return path.
Random	Used to obtain a random number.
ReturnPath	Provides access to a return path.
SecureReturnPath	Provides access to a secure return path mechanism.
Stream	Controls the connection to a composition of elementary media streams (video, audio, subtitles etc.) that are presented in synchronization.
StreamEvent	Controls a source of stream events from a data elementary stream of a Stream component.
Timer	Used to generate a timer notification at a specified point in the future.
Transaction	Controls the data exchange over a return path.

7.5.3.2 Variable and cookie components

Table 25 lists variable and cookie components available as standard within PCF.

Table 25: PCF variable and cookie components

Component Type	Description
BooleanVar	Used to store and retrieve a single boolean value for the lifetime of the encapsulating Container component.
DateTimeVar	Used to store and retrieve a single date and time representation value for the lifetime of the encapsulating Container component.
IntegerVar	Used to store and retrieve a single integer value for the lifetime of the encapsulating Container component.
StringVar	Used to store and retrieve a single string value for the lifetime of the encapsulating Container component.
BooleanCookie	Used to store and retrieve a single boolean value such that it may persist beyond the duration of a service session.
DateTimeCookie	Used to store and retrieve a single date and time representation value such that it may persist beyond the duration of a service session.
IntegerCookie	Used to store and retrieve a single integer value such that it may persist beyond the duration of a service session.
StringCookie	Used to store and retrieve a single string value such that it may persist beyond the duration of a service session.

Each variable or cookie has a single property called **value**, which is of the appropriate type. Variables store the value for the lifetime of the component's parent container. Cookies persist for at least the lifetime of the user's session.

7.5.4 Container components

Figure 15 shows the conceptual hierarchy of component classifications within the standard set of PCF components:

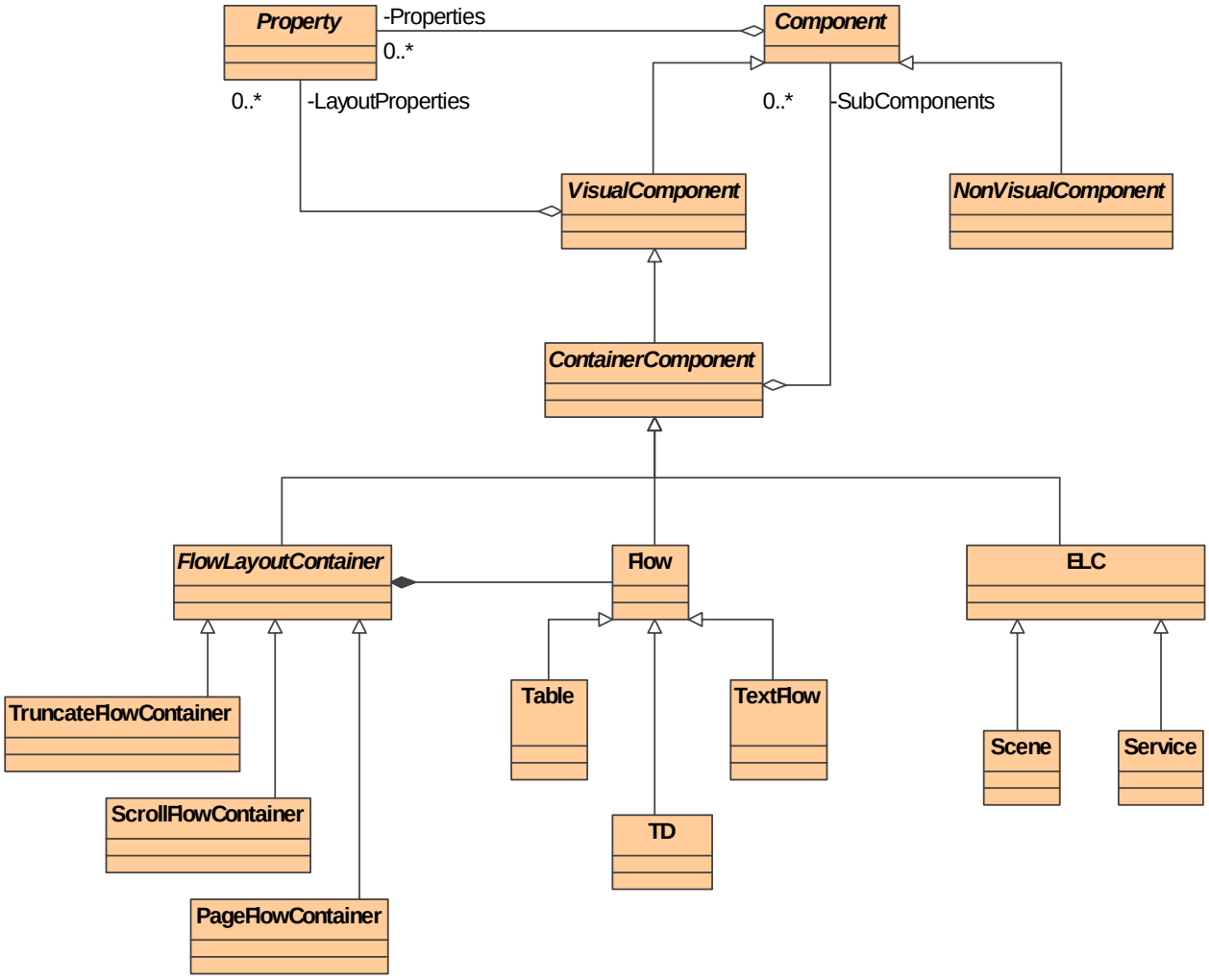

Figure 15: Component object model

NOTE: This is not a class hierarchy in the strict object-oriented sense. PCF does not specify how a component should by implemented, only how the component is intended to appear and behave. As such, it is possible that the implementation on a particular platform uses a completely different class hierarchy.

The hierarchy in figure 15 is derived from the commonality of facets at the component interface level - the use of common property groups, the ability to contain child components and layout characteristics etc, and as such provides a useful approximation of a class hierarchy.

Table 26 lists the container components available as standard within PCF.

Table 26: PCF standard container components

Component Type	Description
Service	This is the top-level component. A Service positions all its immediate sub-components using the same layout rules as a StaticELC.
Scene	Positions all its immediate sub-components using the same layout rules as a StaticELC.
ELC StaticELC	Explicit Layout Container: positions its sub-components using explicitly defined size and position properties.
SFC StaticSFC	ScrollFlowContainer: presents a Flow within a fixed size rectangular area, with a scroll control to reveal Flow content that cannot be directly accommodated in the presentation area.
PFC StaticPFC	PageFlowContainer: presents a Flow within a fixed size rectangular area, with a paging control to reveal Flow content that cannot be directly accommodated in the presentation area.
TFC StaticTFC	TruncateFlowContainer: presents a Flow within a rectangular area, with limited "stretch" governed by a maxsize control. Flow content that cannot be directly accommodated in the presentation area at maximum size will be truncated.
Flow	Positions its subcomponents using the flow layout algorithm specified in the present document.
TextFlow	Specialization of Flow which is limited to contain only a single instance of SimpleMarkedUpText.
Table	Presents content in a tabular form using child components specified in the present document. Tables can exist only within a Flow or within a FlowLayoutContainer.

Static forms of these components restrict the manipulation of certain properties at run-time. This provides components which will be portable to a wider range of platforms.

The layout algorithms underlying these components are specified within clause 8.

7.6 Custom components

In addition to the component types defined as standard within the PCF specification, it shall be possible to create custom components to implement special functionality or to exploit specific features available on a target platform.

As PCF uses a generic component interface for its standard components, instances of custom components shall be declared within a PCF service in exactly the same way.

The present document defines the interfaces for standard component types using the PCF's **Component Definition Syntax** and specifies what the perceived behaviour of that standard component type should be. It shall be the responsibility of the custom component creator to define the interface of the custom component type, although this must still be using the CDS to define the interface elements.

The present document places no restrictions on how the underlying component implementation should appear or behave.

It is the responsibility of the transcoder for the target platform for which the custom component has been written to interpret a declaration of an instance of that component and implement it appropriately.

It will only be possible to render a custom component on platforms that support that specific custom component type. As such, services that contain custom components will be less portable than services that contain only standard PCF components.

In the case where the same custom component is available on multiple platforms, it shall be the responsibility of the custom component creator to ensure that the component type provides an equivalent user experience on each of these platforms.

To prevent naming collisions, the creator of a custom component should set the provider of the component to their domain name. They can then choose the name of the component, ensuring that it is unique within components which are provided by them. If necessary the provider name may have a path after the domain to further differentiate between components.

EXAMPLE 1: Foo Components has produced an interactive game, and will name it as follows:

```
provider="foocomponents.com" name="Game"
```

EXAMPLE 2: The Big Broadcasting Company has produced two quizzes, and names them as follows:

```
provider="bigbroadcastingcompany.com/gameshows/strongestchain " name="quiz"
provider="bigbroadcastingcompany.com/gameshows/testthecontinent" name="quiz"
```

NOTE: The creation of custom events is not permitted. Custom components shall use only events defined by the present document.

7.7 Schema components

The generic component model applies to all component classes. As such, it shall be possible to declare a component instance of any class using the generic component model element **Component** and specify that component instance's properties with the various **ValueItems**.

In addition to this method of declaration it shall be possible to declare standard PCF components using class-specific declaration elements. This is to allow a PCF author to write less verbose, but functionally equivalent PCF.

The components that may be used in this way are collectively referred to as **schema components**. Elements for specifically declaring each type of schema component are included within the PCF schema. The element for each schema component shall be the same as that component's class name.

EXAMPLE: A **Rectangle** component may be declared generically using the following PCF fragment:

```
<Component class="Rectangle">
   <Color    name="fillcolor"  value="#0000FF"/>
   <Color    name="linecolor"  value="#FFFFFF"/>
   <Position name="origin" value="100 100"/>
   <Size     name="size"   value="50 30"/>
</Component>
```

Alternatively, the functionally equivalent, but type-specific declaration would be as follows:

```
<Rectangle>
   <Color    name="fillcolor"  value="#0000FF"/>
   <Color    name="linecolor"  value="#FFFFFF"/>
   <Position name="origin" value="100 100"/>
   <Size     name="size"   value="50 30"/>
</Rectangle>
```

NOTE: Custom component declarations shall always use the generic form.

8 Layout specification

8.1 Introduction

The PCF layout specification governs the overall positioning and appearance of the visible PCF components that comprise the layout of a PCF service.

Layout of visible components in PCF can be controlled in two ways:

- explicit layout;
- flow layout.

Explicit layout supports pixel-accurate positioning of graphical and textual objects on screen within an explicit layout container, as described in clause A.1.1.4.

Flow layout supports presentation of content that is flowed into a defined rectangular area according to an automated layout algorithm. There are three kinds of **Flow**:

- Flow: Flowed content can include text, inline graphics and all other visual components. A Flow component can contain child visual components. A Flow component can contain MarkedUpText content.
- TextFlow: TextFlow is a subset of Flow where only SimpleMarkedUpText content is supported. A TextFlow shall not contain any child components.
- Table: Tables are a subset of Flow where flowed content is laid out in a two dimensional tabular structure. The table layout algorithm in PCF is based on the CSS2 table specification

Flowed content is presented in one of three kinds of Flow container:

- **TFC** (Truncate Flow Container): A rectangular bounded area in which a Flow of content can be presented. The TFC may stretch, within specified limits, to accommodate the Flow. Content that cannot be accommodated within the TFC shall be truncated.
- **SFC** (Scroll Flow Container): A rectangular bounded area in which a Flow of content can be presented. The SFC presents a scrollbar to enable the viewer to scroll up and down through content that cannot be accommodated in the SFC.
- **PFC** (Page Flow Container): A rectangular bounded area in which a Flow of content can be presented. The PFC presents left and right navigation controls to enable the viewer to page back and forth through content that cannot be accommodated in the PFC.

Flow layout is specified in clause 8.3.

PCF can be deployed to a wide variety of platforms, and it is possible that information about certain aspects of specific destination platforms may not be available to the service author. These may include:

- screen resolution;
- OSD registration;
- safe areas;
- fonts;
- colour depth.

A service author may not know the screen resolution of the target platform on which a service is to be rendered. Specification of explicit screen coordinates may result in distortions to or clipping of the intended appearance. To avoid this potential problem, all screen coordinates shall be specified with respect to a reference screen model. The reference screen model is described in detail in clause 8.7.

NOTE: A PCF transcoder can map a reference screen location to an actual screen location on a specific target platform.

An intrinsic layout issue faced by service authors is achieving accurate registration, or overlay, of OSD graphics above the video plane. This is only possible to reliably achieve when both screen resolution and aspect ratio are known. The PCF does not attempt to address this industry-wide issue. Issues relating to OSD registration are described in clause 8.8.

Many interactive TV receivers have limited built-in font resources. Because PCF service authors may not have knowledge of font resources available on target platforms, PCF follows the CSS2 [22] approach. This approach allows font selection criteria to be expressed by the service author, but font selection is ultimately determined by the capabilities of, and resources available to, a specific target platform. Font selection is described in clause 8.10.

8.2 Explicit layout

8.2.1 Introduction

A PCF **explicit layout container** component is a container component that allows its child components to be precisely positioned and sized using explicit co-ordinate and size values. An explicit layout container is a container used to group child components. The child components are then positioned relative to the origin of the explicit container. The origin of the explicit container may be offset from the origin of the screen co-ordinate system.

An explicit layout container has no visual appearance or size of its own. It has an **origin** position property that defines the zero point of the co-ordinate system relative to which all the container's child components are positioned. Components within an explicit layout container do not affect each other's horizontal or vertical position. Child components higher up the z-plane display stack will appear on top of child components further down.

8.2.2 Explicit layout container elements and characteristics

There are two types of explicit layout container available: static (**StaticELC**) and dynamic (**ELC**). The static container allows its child components to be laid out at initialization, after which point the container cannot subsequently be repositioned or hidden. In contrast, the dynamic container can be repositioned, hidden or shown, like any other visual component; by doing so all its child components are themselves repositioned, hidden or shown.

The relationship between these and other related components is shown in figure 16.

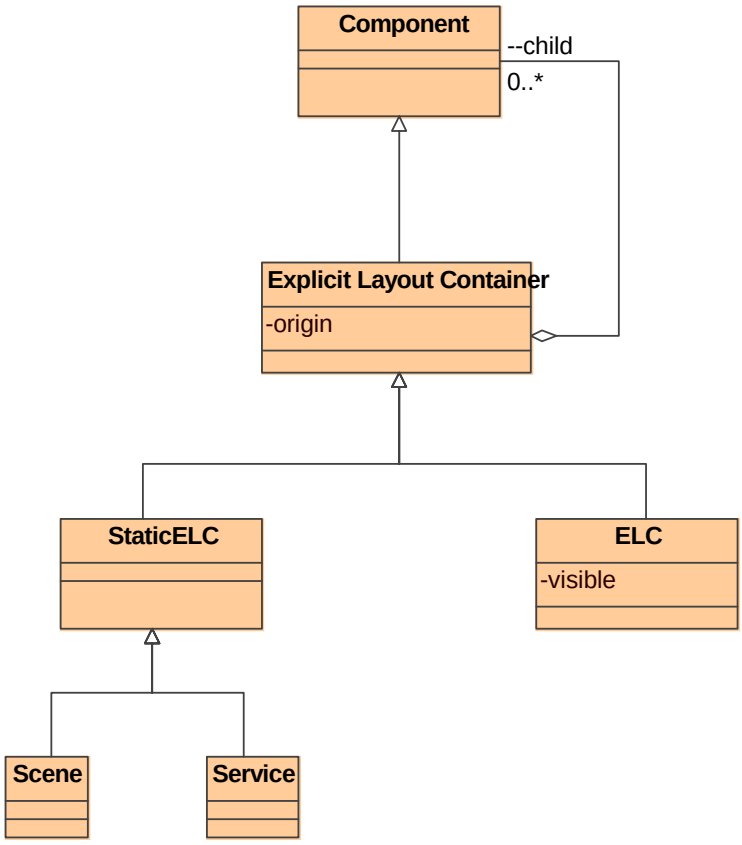

Figure 16: Explicit layout component hierarchy

An explicit layout container is a container for components. Child components shall be positioned relative to the origin of the explicit layout container.

EXAMPLE 1: An explicit layout container has an origin position of 100,100. The container has a single child component, a **Rectangle**, which has an origin property of 80,50. The position of the **Rectangle** on the screen will be relative to the layout container's origin, so will therefore be at screen co-ordinates of 180,150. This is illustrated in figure 17.

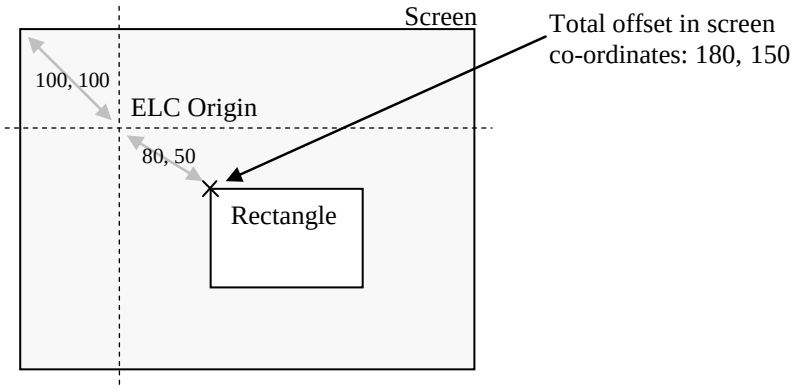

Figure 17: Relative positioning of explicit layout child components

A StaticELC is one that can only be initialized. Once initialized it cannot be moved or hidden. It shall still be possible to show, hide and move each individual child component.

A Scene is a StaticELC. All scenes shall have a fixed origin at screen co-ordinate 0,0.

A dynamic ELC has all the properties of a StaticELC. In addition it shall be possible to show and hide the container, move its origin, and reposition it in the display stack. When a dynamic explicit layout container is shown, hidden or moved, all its child components shall also be shown, hidden or moved accordingly.

It shall be possible for an explicit layout container to contain further explicit layout containers. In each case the origin co-ordinate of the child container defines the zero point for its children. However, it shall not be possible for a StaticELC to be located within an ELC. This is because the static container cannot move, and so would be unable to move if its dynamic parent container moved.

Explicit layout containers may contain Flow layout containers. However, explicit layout containers shall not be contained within Flow containers.

If one or more components within an ELC are moved individually, either in the X and Y planes or up and down the display stack, and the ELC is subsequently moved, then the child components shall retain their new relative positions within the ELC.

EXAMPLE 2: Figure 18 shows an ELC that contains three child components: a **Rectangle**, an **Ellipse** and a triangle (made using the **Polygon** component). Initially they are ordered, as illustrated in A, from back-to-front: the rectangle, triangle and ellipse.

Subsequently, the triangle component is brought to the front of the display stack, and both the triangle and the ellipse are moved. This is illustrated in B.

Finally, the ELC is moved to the right. As can be seen in C, the modified relative spatial relationships between the container's child components are maintained after the parent is moved.

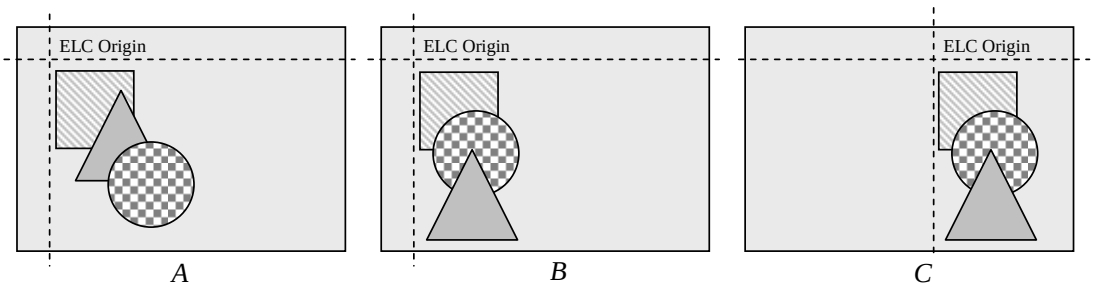

Figure 18: Moving child components within a (dynamic) ELC

Upon initialization, components shall be positioned within the z-plane in the order in which they are declared, those components declared first being at the back. If two components occupy the same space on screen, the component declared later, and thus higher up the display stack, will obscure the component lower down the display stack.

EXAMPLE 3: An explicit layout container has two child components, a **Rectangle** and a **Polygon**. The **Rectangle** is declared before the **Polygon** in the PCF description. The **Polygon** will therefore be nearer the front of the z-plane display stack. This is illustrated in figure 19.

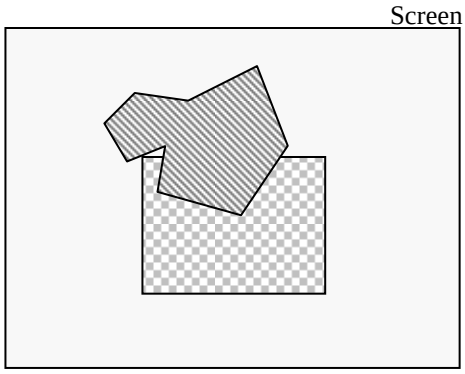

Figure 19: Display stack positioning of explicit layout child components

Child components within an explicit layout container shall occupy the same band of the display stack. If two explicit layout containers are children of a parent explicit layout container, then these children shall each occupy their own separate band of the display stack. All the children of the rear child layout container shall be positioned underneath all the children of the front child layout container. For further explanation of the display stack see clause 8.9.

EXAMPLE 4: This is illustrated in figure 20. In diagram *A* there are two Explicit Layout Containers, ELC 1 and ELC 2. Both have three components at different z-order positions, which overlap. ELC 2 is in a z-order position in front of ELC 1. If the containers are repositioned so that they overlap, as in diagram *B*, the rearmost component of ELC 2 is still in front of the front-most component of ELC 1.

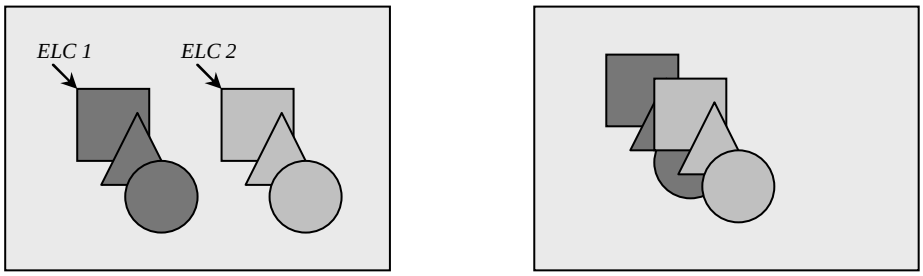

Figure 20: Explicit layout containers overlapping each other

8.3 Flow layout

8.3.1 Introduction

PCF flow layout provides a mechanism whereby components and content that do not require the layout precision of an explicit layout container can be "flowed" into a rectangular area and positioned relative to each other according to the Flow layout algorithm.

A **Flow** component lays out its child elements in essentially the same way as is common for HTML. Character sets requiring left-to-right and right-to-left flow are supported, and these are handled on an element-by-element basis. Within this basic framework, more precise positioning control is achieved by allowing the alignment, spacing and block-separation of individual elements to be specified.

The layout algorithm used by the Flow component is based directly on the layout algorithm specified by the W3C Cascading Style Sheets, level 2 CSS2 Specification. The present document defines a flexible set of rules for applying formatting to HTML documents. The requirements for flow layout in PCF differ from those for HTML web pages in that:

- They are less extensive - much of the more subtle formatting control available within CSS2 is unnecessary in PCF flow layout components.
- PCF already achieves some of the more complex aspects of layout using its own mechanisms - the explicit layout component achieves an equivalent effect to CSS2's **absolute** and **fixed** position types.

Because of these differences, the layout facilities available within Flows are a restricted subset of CSS2, and the syntax for controlling this layout is significantly different. However, the underlying flow layout algorithm is identical.

The PCF flow layout is not a component itself; instead, a number of container components are defined that implement the flow layout algorithm. These present, and allow navigation across, the laid out area in a variety of different ways.

8.3.2 Flow layout elements

A PCF **Scene** description forms a description tree.

A **Flow** is a node on this description tree.

A Table is a specialization of Flow, and is described in clause 8.5.

A **Flow** will incorporate both the visual components and marked up text beneath it in the description tree hierarchy. These are called **PCF flow layout elements** and their relationships are illustrated in figure 21. Within clause 8.3 these are referred to simply as **elements**.

These elements form a subset of the nodes within the description tree (as the tree may contain non-visual components which are ignored by the **Flow**), and this subset may be considered equivalent to the document tree described in the CSS2 specification.

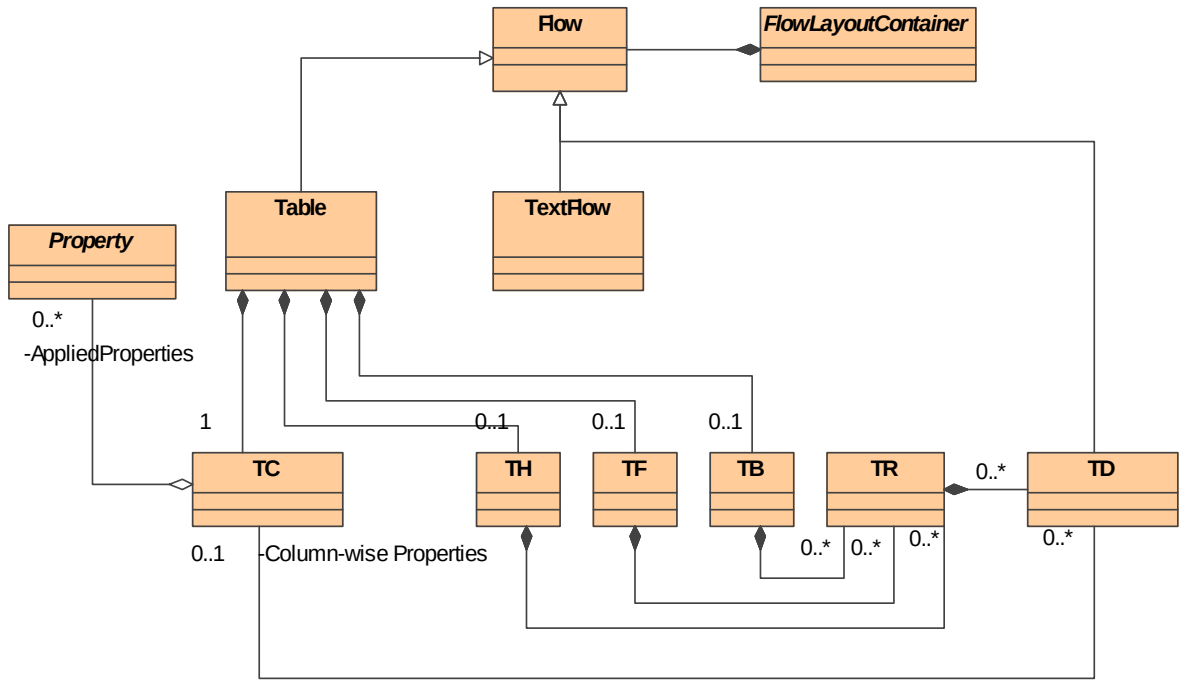

Figure 21: Flow layout component hierarchy

8.3.3 The flow layout box model

A PCF flow layout component shall lay out its child components and content using a conceptual **box model**. In this model each separately identifiable child element is treated as though it were contained within a box. Boxes are flowed across the content area of their containing box using the visual formatting model described in clause 8.3.4.

Each conceptual box has the attributes shown in figure 22.

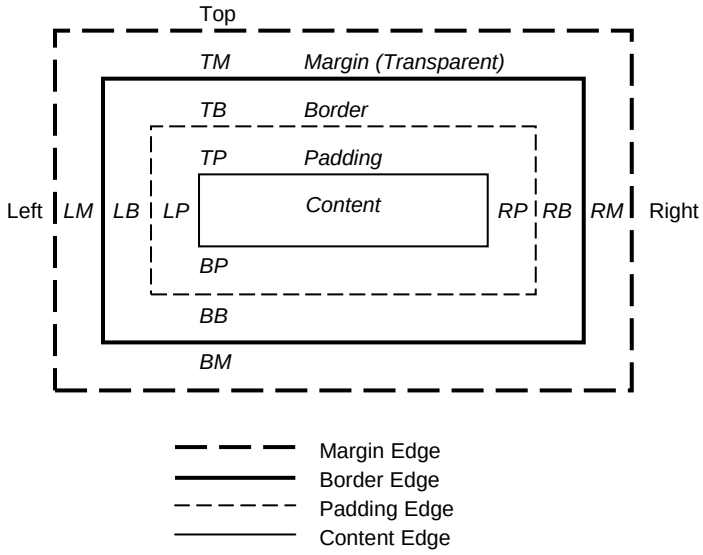

Figure 22: Layout box attributes

The layout box for an element has four concentric areas: the **content area**, the **padding area**, the **border area**, and the **margin area**. The perimeter of each area is called an "edge", so each box has four edges:

- The **content edge:** this edge surrounds the element's rendered content. By default its height and width are the minimum required to contain the content, although this can be overridden with explicit values. The background style of the content area is the **background** property of the generating element.
- The **padding edge**: this edge surrounds the box's padding. If the padding width is 0 then this edge is the same as the content edge. The background style of the padding area is the **background** property of the generating element.
- The **border edge**: this edge surrounds the box's border. If the border width is 0 then this edge is the same as the padding edge. The style of the padding area is specified by the **border** properties of the generating element.
- The **margin edge**: this edge surrounds the box's margin. If the margin width is 0 then this edge is the same as the border edge. Margins are always transparent, and allow the content area of the containing parent element to show through.

The properties relating to each of the edges may be specified using value items referring to the **top**, **bottom**, **left** and **right** segments in isolation.

8.3.4 Flow layout box types

8.3.4.1 Overview

The layout algorithm of the flow uses a number of virtual box types. These are not PCF service description items; they are conceptual objects used to describe the model for laying out PCF components and marked up text within a Flow component.

These objects, and the relationships between them, are illustrated in figure 23.

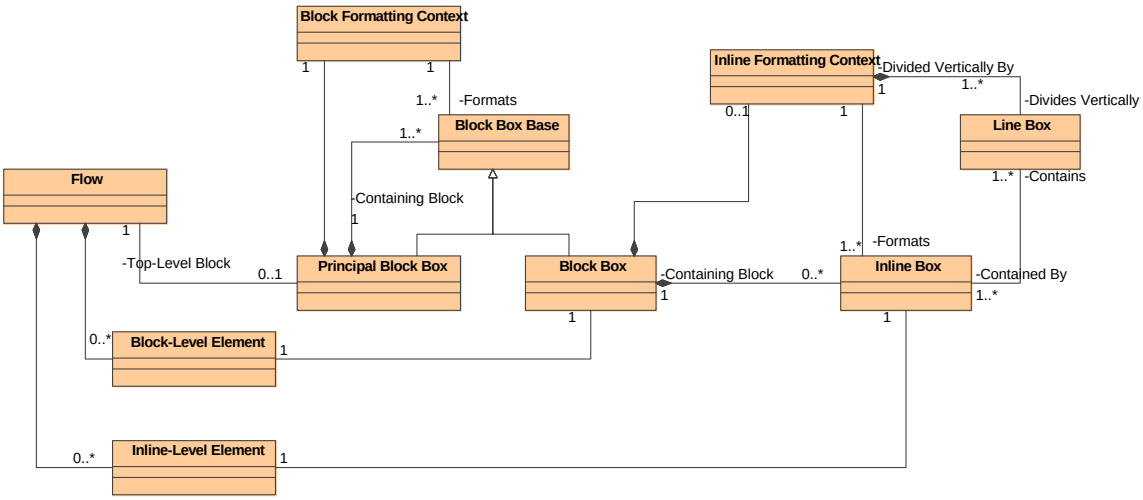

Figure 23: Flow layout model objects

The three left-most items within figure 23 represent elements within a PCF description tree. These are:

- Flow: the Flow component itself. This contains the child elements that must be laid out using the flow layout model.
- Block-level element: any child element that requires formatting visually as a separate block. For example: paragraph and horizontal rule.
- Inline-level element: any child element that does not form a new block of content. For example: emphasized text within a paragraph, inline images and inline components.

The remaining items within the diagram are conceptual items used to define the flow layout model. These are:

- Block box: a layout box used to contain a block-level element.
- Principle block box: block-level elements shall generate a principal block box that directly contains only block boxes. The principal block box establishes the containing block for descendant boxes.
- Block formatting context: principal block boxes establish the block formatting context used for laying out descendent block boxes.
- Inline box: a layout box used to contain an inline-level element. Inline boxes are contained within a block box, which establishes their containing block. An anonymous containing block box shall be generated if none is explicitly defined.
- Inline formatting context: the containing block box establishes the inline formatting context for laying out its descendent inline boxes.
- Line box: an inline formatting context lays its inline boxes out over one or more horizontal lines. The formatting context generates a line box to represent the rectangular area that contains the boxes that form a single line.

These items are defined in detail within clauses 8.3.4.2 to 8.3.4.6.

8.3.4.2 Containing blocks

Elements within a Flow are contained within a hierarchical tree structure: the Flow contains a number of child elements; some of these child elements may themselves contain child elements.

Within this structure, the element at each node of the tree is laid out either by a block box or by an inline box. In either case an element's layout box is positioned relative to, and within, that box's **containing block**.

The containing block of a descendent box shall be the content-edge of the nearest block-level ancestor box.

The width of the top-level containing block shall be constrained by the width of the container in which the Flow is to be presented. The height of this block may grow to accommodate all descendent boxes.

8.3.4.3 Block-level elements

Block-level elements are those elements, such as paragraphs, that require formatting visually as a separate block. A block-level element shall be given sole occupancy of the vertical area in which it is located within its containing block. Figure 24 shows three block-level elements, labelled 1, 2 and 3, within their containing block, labelled C.

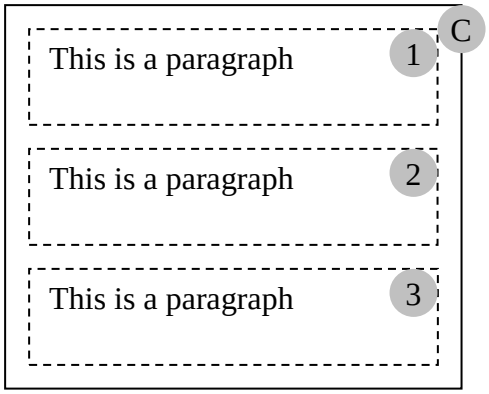

Figure 24: Layout of block-level elements

Block-level elements shall generate a **principal block box**, which shall contain only block boxes, and these block boxes shall be used to lay out the block level elements.

If a Flow contains only inline-level elements, then an anonymous block box shall be generated to contain them.

If block-level elements and inline-level elements exist at the same level within the element tree then an anonymous block box will be generated to contain each contiguous sequence of inline-level elements. Thus, if a block box has at least one block box inside it, it shall be forced to have **only** block boxes inside it.

EXAMPLE: Given the following marked-up text:

```
<body>
    This is some text
    <p>This is some more text</p>
</body>
```

The body element has both an inline and a block-level element. To make it easier to define the formatting an anonymous block box shall be generated around "This is some text". This would then be laid out as shown in figure 25.

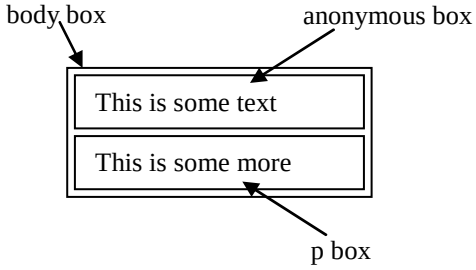

Figure 25: Layout of anonymous block boxes

8.3.4.4 Block formatting context

In a **block formatting context** block boxes shall be laid out vertically, one after the other, beginning at the top of the containing block.

The vertical distance between sibling boxes shall be determined by the margin properties of those siblings.

Vertical margins between adjacent block boxes within a block formatting context shall collapse so that the margin height separating the boxes is equal to the maximum of the bottom margin of the upper box and the top margin of the lower box.

In a block formatting context each block box's vertical outer edges shall touch the vertical edges of the containing block.

8.3.4.5 Inline-level elements

Inline-level elements do not require a separate formatting block. They shall be distributed into lines, with each successive element following on horizontally from its previous sibling. Figure 26 illustrates three inline elements, Text Box 1, Text Box 2 and Image 1, being laid out within their containing block.

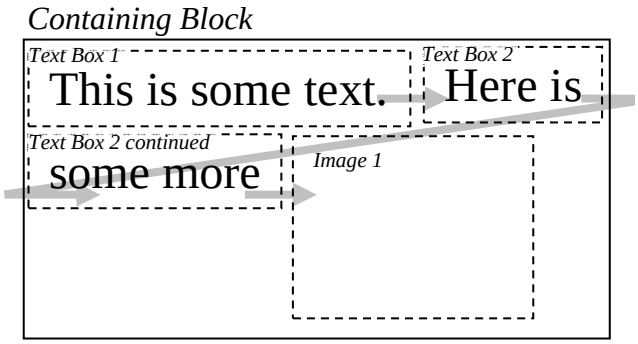

Figure 26: Layout of block-level elements

In figure 26 the second element, labelled **Text Box 2**, has been split across two lines as otherwise it would overrun the right-hand edge of its containing block.

Within a containing block box, some inline boxes will be generated explicitly by inline-level elements such as **em**, **big**, or **small**. In cases where text is declared directly within the block-level element, an **anonymous inline box** will be generated to contain it.

EXAMPLE: Given the following marked-up text:

```
<p>This is <em>emphasized</em> text</p>
```

The **p** generates a block box with three inline boxes inside it. The box for "emphasized" is an inline box generated by the **em** inline element. The other inline boxes, for "This is" and "text" are generated by the block-level element, **p**, and are anonymous inline boxes.

8.3.4.6 Inline formatting context

In an **inline formatting context**, boxes shall be laid out horizontally, one after the other, beginning at the top of a containing block. Horizontal margins, borders, and padding shall be respected between these boxes.

Boxes shall be aligned vertically depending upon their **vertical-align** property: their bottoms or tops may be aligned, or the baselines of text within them may be aligned. The rectangular area that contains the boxes that form a line is called a line box.

The width of a line box shall be determined by its containing block. A line box shall always be tall enough for all of the boxes it contains. However, it may be taller than the tallest box it contains (if, for example, boxes are aligned so that baselines line up).

When the height of a box B is less than the height of the line box containing it, the vertical alignment of B within the line box shall be determined by the vertical-align property.

When several inline boxes cannot fit horizontally within a single line box, they shall be distributed among two or more vertically-stacked line boxes. Thus, a paragraph is a vertical stack of line boxes. Line boxes shall be stacked with no vertical separation and no overlap.

The left edge of a line box shall touch the left edge of its containing block and the right edge shall touch the right edge of its containing block. Line boxes in the same inline formatting context may vary in height.

EXAMPLE: One line might contain a tall image, requiring a taller line box than other lines that contain only text.

When the total width of the inline boxes on a line is less than the width of the line box containing them, their horizontal distribution within the line box shall be determined by their "text-align" property. If that property has the value "justify", the user agent may stretch the inline boxes as well.

An inline box may not exceed the width of a line box so long inline boxes shall be split into several boxes and these boxes distributed across several line boxes. When an inline box is split, margins, borders and padding shall have no visual effect where the split occurs. Formatting of margins, borders, and padding may not be fully defined if the split occurs within a bidirectional embedding.

By default, text within an inline box shall have its white space collapsed. If the inline box overruns its line box, then the inline box shall be split at the white space nearest to the right edge (for a left-to-right character set) and still within the line box. If an inline box occupies and overruns an entire line box, and due to no appropriate white space cannot be split, then the overrunning content shall be truncated.

If non-default white space handling is required, the **white-space** property may be used to modify the rules for collapsing white space and splitting inline boxes.

8.3.5 Layout properties

The following clauses describe the properties that a child component within a flow layout may define that have specific side-effects when defined within a flow-layout.

8.3.5.1 General properties

The **v-align** property defines the vertical alignment of a child element. If the element is an inline-level element then the v-align property shall define the element's alignment within its containing line-box. This property shall have no effect if the child element is a block-level element. The v-align property may have the following values:

- **Baseline:** align the baseline of the box with the baseline of the parent box. If the box does not have a baseline, align the bottom of the box with the parent's baseline.
- **Middle:** align the vertical midpoint of the box with the baseline of the parent box plus half the x-height of the parent.
- **Sub:** lower the baseline of the box to the proper position for subscripts of the parent's box. (This value has no effect on the font size of the element's text.)
- **Super:** raise the baseline of the box to the proper position for superscripts of the parent's box. (This value has no effect on the font size of the element's text.)
- **Text-top:** align the top of the box with the top of the parent element's font.
- **Text-bottom:** align the bottom of the box with the bottom of the parent element's font.
- **Top:** align the top of the box with the top of the line box.
- **Bottom:** align the bottom of the box with the bottom of the line box.

The **h-align** property defines the horizontal alignment of child elements within a block-level element. Given that a block-level element contains a stack of one or more line boxes, the h-align property shall define how the inline-level elements within each line box are aligned. The h-align property may have the following values:

- **Left:** inline-level elements shall be aligned to the left-hand edge of their containing line box.

- **Right:** inline-level elements shall be aligned to the right-hand edge of their containing line box.
- **Center:** inline-level elements shall be aligned so that they are centred within their containing line box.
- **Justify:** the inline-level elements may be stretched so that they span the entire width of the containing line box.

The **white-space** property defines how whitespace inside an element is handled. Values have the following meanings:

- **Normal:** this value directs user agents to collapse sequences of whitespace, and break lines as necessary to fill line boxes. Additional line breaks may be created by occurrences of the `<br>` element.
- **Pre:** this value prevents user agents from collapsing sequences of whitespace. Lines are only broken at newlines in the source, or at occurrences of the `<br>` element.
- **Nowrap:** this value collapses whitespace as for "normal", but suppresses line breaks within text except for those created by the `<br>` element.

8.3.5.2 Side-specific properties

The following properties may be used in their general form, in which case the property is applied to all four sides (top, bottom, left and right) of the box in question, or they may be applied to individual sides by adding the appropriate suffix.

NOTE: When an individual side is specified in conjunction with the general form (applied to all sides), the value for the individual side overrides that of the value for the general form for the given side.

The **margin** property defines the width of the margin area of a box. The "margin" property itself is a shorthand for setting the margin width for all four sides to the same value; these may also be set individually using **margin-top**, **margin-bottom**, **margin-left** and **margin-right**. All of these properties specify the margin width in pixels.

The **padding** property defines the width of the padding area of a box. The "padding" property itself is a shorthand for setting the padding width for all four sides to the same value; these may also be set individually using **padding-top**, **padding-bottom**, **padding-left** and **padding-right**. All of these properties specify the padding width in pixels.

The set of **border** properties define the width, colour and style of the border area. The border properties prefixed with just "border" apply the property in question to all four sides, whereas border properties may be applied individually using properties prefixed with **border-top**, **border-bottom**, **border-left** and **border-right**. The following border properties may be set:

- **Width:** the width, in pixels, of the border.
- **Color:** the default colour of the border.
- **Focuscolor:** the colour of the border when the element has focus.
- **Disabledcolor:** the colour of the border when the element is disabled.
- **Activecolor:** the colour of the border when the element is active.
- **Idlecolor:** the colour of the border when the element is idle.
- **Shadowcolor:** the colour to use as shadow when the border is using a multicoloured style.
- **Highlightcolor:** the colour to use as highlight when the border is using a multicoloured style.
- **Linestyle-singlecolor:** the single-colour linestyle to use for the border. The value may be **solid**, **dashed**, **dotted** or **double**.
- **Linestyle-multicolor:** the multi-colour linestyle to use for the border. The value may be **groove**, **ridge**, **bevelled-outset** and **bevelled-inset**.

8.4 TextFlow

The TextFlow component is a specialization of the Flow component and obeys all the rules of the Flow component, but its content type shall be SimpleMarkedUpText, and it shall not contain any child elements.

8.5 Table layout

8.5.1 Introduction

Table layout is a specialization of Flow layout, which provides a mechanism where child components and content are positioned in a two dimensional grid structure.

Tables in the PCF are based upon the specification for tables in the W3C's HTML4 [8] specification and Cascading Stylesheets 2 [22] specification.

As in HTML4 / CSS2, tables in the PCF support the grouping of table rows into header, footer, and body sections. Column groupings are also supported. Row and column groupings enable flexible rendering of tables in ways that emphasize the table content and enhance usability.

EXAMPLE 1: Row groupings provide for the possibility of a PCF implementation that scrolls the table body, but keeps header and footer always visible, or that repeats the header and footer rows across multiple screens of paged table content.

EXAMPLE 2: Column groupings enable contrasting styling and formatting to be applied to specific columns, such as the "total" column in an accounts grid.

As in HTML4 / CSS2, table cells in the PCF may span multiple rows and columns.

The **Table** component defines the algorithm for laying out the table grid, and exists independently of the Flow layout container objects that are used to present Flows and tables.

The layout of content within each cell of a **Table** obeys the rules defined for flow layout. See clause 8.3.

The visual representation of a **Table** is handled by one of the three flow containers:

- **SFC** (Scroll Flow Container);
- **PCF** (Page Flow Container);
- **TFC** (Truncate Flow Container).

The flow layout containers are specified in clause 8.6.

8.5.2 Table layout algorithms

The PCF does not define any optimal table layout algorithm. PCF implementations may make use of any suitable layout algorithm, including those that prioritize speed over precision. The only exception to this is where the service author has specified that the "fixed" table layout algorithm should be used.

The fixed table layout algorithm is specified in clause 8.5.2.1.

An automatic table layout algorithm is specified in clause 8.5.2.2. This clause is optional.

8.5.2.1 Fixed table layout

Under the "fixed" table layout algorithm, the horizontal width of the table and columns shall not depend on the content of the cells. Instead, table width depends only on the presentation container width, and column width depends on declared column widths, borders and cell spacing.

The overall width of the table shall be derived from the available internal width of the flow container, i.e. actual container width, minus any padding and, in the case of the SFC, minus any allocation for the scroll bar.

The width of each column in the table shall be determined as follows:

1) A column element **width** property with a value other than "auto" sets the width for that column.

2) Where column **width** has not been specified, a cell in the first row of the column with a value other than "auto" for the **width** property sets the width for that column. If the cell spans more than one column, the width is divided evenly over the columns spanned.

3) Any remaining columns equally divide the remaining horizontal table space, minus borders and cell spacing.

Where the available width of the presentation container is greater than the sum of the values of the **width** properties for each of the columns plus borders and cell spacing, the **Table** shall be positioned within the container according to the flow alignment for that container.

Where the available width of the presentation container is less than the sum of the values of the **width** properties for each of the columns plus borders and cell spacing, the PCF implementation shall return an error.

NOTE: The fixed table layout algorithm allows PCF implementations to begin layout as soon as the entire first row has been received. Subsequent rows do not affect column width.

Cell content is flowed within each cell according to the PCF flow layout algorithm specified in clause 8.3.

Cell content may be truncated according to the value of the **wrap** property for that cell, as specified in clause A.1.2.3.5.

If the containing row for a cell has a **row-height** property value defined, each cell in that row shall be this height. Otherwise, cell height shall be stretched to accommodate the content and row height shall be determined according to the algorithm specified in clause 8.5.2.4.

8.5.2.2 Automatic table layout (optional)

Under the "automatic" table layout algorithm, the horizontal width of the table is derived from the available internal width of its flow container. This is a departure from CSS2.

The automatic table layout algorithm shall adjust the width of the table columns to fill the width of the presentation container.

The automatic table layout algorithm shall only apply where one or more columns do not have a width property specified.

Column widths are determined as follows:

1) The minimum content width for each cell in the **Table** shall be calculated according to the rules specified in clause 8.5.2.1. If the specified width for the cell is greater than minimum content width, then the specified width defines the minimum cell width. It is an error if the specified width is smaller than the minimum content width. A value of "auto" for the **width** property of a cell means that minimum content width defines the minimum cell width.

2) The maximum cell width shall be calculated by formatting the content in each cell without breaking any lines, except where explicit line breaks are specified.

3) For each column, a minimum and a maximum column width shall be determined from the cells that span only that column. The minimum column width is the largest minimum cell width of the cells in that column. The maximum column width is the largest maximum cell width of the cells in that column.

4) If a width property is specified for a column and the width property is greater than the minimum column width, then the width property shall define the column width. It is an error if the width property for the column is smaller than the minimum column width.

5) For each cell that spans more than a single column, the sum of minimum column widths for the spanned columns must be greater than the minimum cell width.

6) If condition 5 is not met, then the minimum column width for each spanned column that does not have a width property specified shall be increased evenly so that condition 5 is met. The PCF does not define how to increase column widths in situations where the number of additional pixels required is not a multiple of the number of columns being increased. It is an error if no minimum column widths can be increased.

This provides a minimum and maximum width for each column. The presentation container width shall determine final column widths as follows:

1) If the width of the presentation container is greater than the sum of the maximum widths required by all the columns plus borders and spacing then the final width for each column shall be the maximum column width plus an additional width. The additional width shall be the difference between the sum of all maximum column widths and the width of the presentation container, distributed evenly across all columns that do not have a width property specified.

2) If the width of the presentation container is less than the sum of the maximum widths required by all the columns plus borders and spacing, and more than the sum of the minimum widths required for all the columns plus borders and spacing, then the final width for each column shall be the minimum column width plus an additional width. The additional width shall be the difference between the sum of all minimum column widths and the width of the presentation container, distributed evenly across all columns that do not have a width property specified.

3) It is an error if the available width of the presentation container is less than the sum of the minimum widths required by all columns plus borders and spacing.

8.5.2.3 Table height algorithm

Table height is determined by the sum of the row heights, plus borders and spacing.

8.5.2.4 Row height algorithm

Row height is determined as follows:

1) Where the declared **height** property for the row has a value other than "auto", row height shall be determined by the height property.

2) Where row **height** is undeclared, or has a value of "auto", row height is calculated after all cells in that row have been laid out, and shall be equal to the minimum height required by the tallest cell in the row. See clause 8.5.2.5 for details of the cell height algorithm.

8.5.2.5 Cell height algorithm

Table cells may have a **height** property declaration, or may inherit a **height** property declaration from the parent table row.

Where a cell **height** is declared or inherited, content in that cell shall not wrap, unless the declared cell height is sufficiently large to accommodate the wrapped content. Overflowed content that cannot fit within the bounds of the cell declared height shall truncate. Where cell **height** and row **height** has not been declared, or has a value of "auto", content in that cell shall be flowed using the flow algorithm specified in clause 8.3, and the cell height shall be the minimum height required by the content in that cell.

Where a cell spans two or more rows, cell height shall be derived from the sum of the spanned row **height** property declarations. If the spanned rows do not have **height** property declarations, or have **height** property values of "auto", then the sum of the row heights must be great enough to encompass the minimum height of the cell spanning the rows.

Where row height is undeclared, and must be calculated, the calculation is contingent upon vertical alignment of cells within that row.

The vertical-align property of each cell determines its alignment within the row. Each cell's content has a baseline, a top, a middle and a bottom, as does the row itself. The values for the **vertical-align** property are as follows:

- **Baseline** - the cell is positioned such that the baseline of the cell's content is aligned with the baseline of the containing row, or of the baseline of the first of several spanned rows.

- **Top** - the top of the cell box is aligned with the top of the containing row, or the top of the first of several spanned rows.

- **Bottom -** the bottom of the cell block is aligned with the bottom of the containing row, or the last of several spanned rows.

- **Middle -** the centre of the cell is aligned with the centre of the containing row, or the centre point of several spanned rows.

The baseline of a cell is the baseline of the first line of content in that cell. If there is no text in the cell, the baseline shall be the baseline of whatever content is in the cell. If the cell is empty, the baseline shall be the bottom of the cell block. The maximum distance between the top of the cell block and the baseline over all cells that have a **vertical-align** property of "baseline" is used to set the baseline for the row.

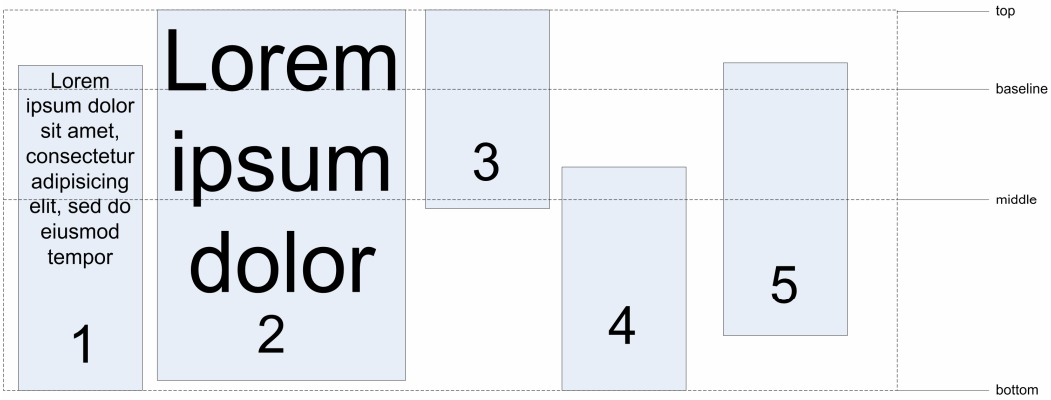

Figure 27: vertical alignment of cells

EXAMPLE: In figure 27, cell blocks 1 and 2 are baseline aligned. Cell block 2 has the largest height above the baseline, and so determines the baseline for the row.

NOTE: If no cell block has baseline alignment, then the row will not have, and does not need, a baseline.

To avoid ambiguity, cell alignment shall proceed as follows:

1) Cells that are baseline aligned are positioned first. This establishes the baseline for the row.

2) Cells that have top alignment are positioned second. The row now has a baseline, a top, and a provisional height, which is the distance from the top to the lowest bottom of the cells positioned thus far.

3) Bottom and middle aligned cells are positioned. If any of these have a height which is larger than the current row height of the row, then the row height shall be increased by lowering the bottom.

4) Cell blocks that are smaller than the height of the row receive extra top or bottom padding.

8.5.2.6 Intra-cell content alignment

8.5.2.6.1 Horizontal alignment

Horizontal alignment of cell content shall be determined by the **horizontal-alignment** property for the cell. The values for the horizontal-alignment properties are:

- **left** - the content is aligned left in the cell;
- **centre** - the content is centred in the cell. Where the cell contains wrapped content, each line of content shall be centred;
- **right** - the content is aligned right in the cell;
- **justify** - the content is justified, with extra spacing such that the text string fills the available cell width.

8.5.2.6.2 Vertical alignment

Vertical intra-cell alignment shall be determined by the **vertical-alignment** property for the cell. The values for the **vertical alignment** properties are:

- **baseline** - the baseline of the content shall be aligned;
- **top** - the top of the content shall be aligned with the top of the cell;
- **bottom** - the bottom of the content shall be aligned with the bottom of the cell;
- **middle** - the middle of the content shall be centred vertically in the cell.

8.5.3 Borders

Border width interacts with table width, cell width and cell padding and spacing. The relationship is described by the following equation, which holds true for every table row:

Row width = $(0.5 * \text{border-width}_0) + \text{margin-left}_1 + \text{padding-left}_1 + \text{width}_1 + \text{padding-right}_1 + \text{margin-right}_1 + \text{border-width}_1 + \text{border-width}_2 + \text{margin-left}_2 + \ldots \text{margin-right}_n + (0.5 * \text{border-width}_n)$

In this equation, *n* is the number of cells in the row, and border-width_i refers to the border between cells *i* and +1.

NOTE 1: Only half of the two exterior borders are included within the table width. The other half lies in the margin area of the table.

NOTE 2: PCF implementations must manage situations where there are odd numbers of pixels specified for borders. The PCF makes no explicit specification for how this issue should be handled.

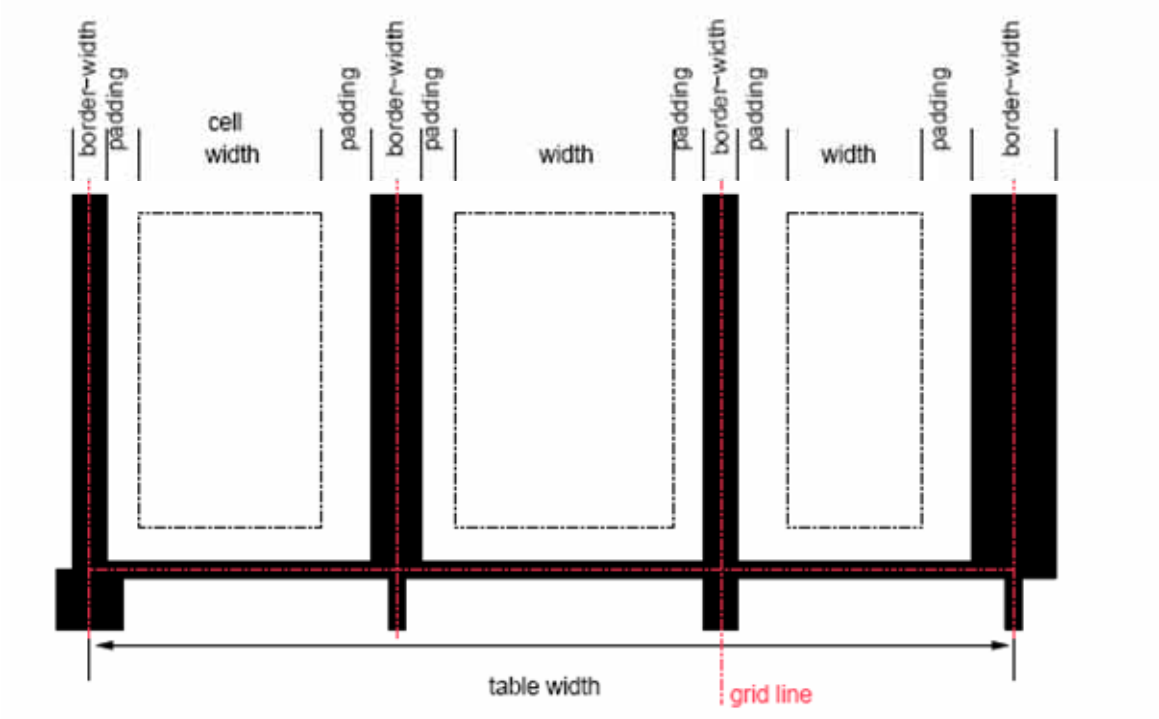

Figure 28: PCF border drawing diagram

Borders may be specified for tables, rows, row groups, columns, column groups and cells. It is therefore possible that border segments within a table structure may have conflicting width, style and colour definitions declared in elements whose borders coincide.

Border definition conflicts are resolved as follows:

1) Borders with border style of "hidden" take precedence over all conflicting definitions.

2) Borders with the style of "none" have lowest priority. A border will be omitted only if all coinciding borders have the style "none".

NOTE: The default border style definition in "none".

3) If none of the conflicting border definitions is "hidden" and at least one of them is not "none", then narrower borders are discarded in favour of wider borders.

4) If all widths in conflicting border definitions are the same, the styling is resolved in the following order of preference: "double", "solid", "dashed", "dotted", "ridge", "outset", "groove" and "inset".

If the border definitions conflict only in colour, then the colour is resolved according to a hierarchy: cell colour, row color, row group colour, column colour, column group colour, and table colour.

8.6 Flow layout container components

The top-level block of all Flows will have a fixed width, defined as the available content area width of the container in which the Flow is to be presented. However, the height of the top-level block will grow to fit the child elements that are contained within the Flow.

The height of the top-level block may exceed the available content area height of the container component. When this occurs, it may be desirable to follow a number of different policies:

- truncate the content that overflows the available area;
- increase the height of the flow layout container to accommodate the extra height, up to a certain maximum size;
- provide a scroll bar to allow the user to scroll down to the overflowing content;
- provide a paging mechanism to allow the user to page to the overflowing content.

These four policies are implemented using three different flow layout container components:

- **SFC** (Scroll Flow Container);
- **PCF** (Page Flow Container);
- **TFC** (Truncate Flow Container).

The **SFC** component supports vertical scrolling of the content. If the flowed content exceeds the available content area, the user may scroll through the content using a pair of pre-defined up/down keys.

The **PFC** component splits the Flow into a number of pages, and then allows the user to navigate through these pages using a pair of pre-defined previous/next keys.

Both the SFC and PFC may reserve portions of their screen area to present appropriate navigation controls.

EXAMPLE: A PCF compliant implementation may reserve a strip of screen area along the right-hand edge of a ScrollFlowContainer in order to visually present a scroll bar graphic. A strip of screen area along the botton edge of a PFC may similarly be reserved to present "prev" and "next" button graphics. In both cases, the effective area available for presentation of flowed content will be affected.

NOTE: Inclusion of a scroll bar graphic on the right hand edge of a ScrollFlowContainer will reduce the available content area width for the ScrollFlowContainer. This will impact on calculation of the Flow.

The **TFC** component implements a non-paging, non-scrolling flow layout. By default a TFC component truncates any content that exceeds the available content area. However, an optional maximum height property can be defined that specifies the amount that the bottom edge of the TFC layout container can move if it needs to accommodate content that exceeds the default available content area.

Content that overflows the maximum height of a TFC shall be truncated.

Maximum height of a TFC shall be constrained by the available height between the top of the TFC and the bottom of the reference screen. TFC that include a maximum height definition that would result in the bottom of the container falling off the bottom of the reference screen shall be treated as if the maximum height definition were the maximum height that can be accommodated within the reference screen area.

8.7 Reference screen model

8.7.1 The reference screen

There is considerable variation in the display resolution of potential target devices.

NOTE: Standard definition TV resolution in Europe is typically 720 x 576 pixels, regardless of aspect ratio. Some widescreen TV formats support 1280 x 720 resolution. Smaller devices such as handheld computers or mobile phones may also be delivery targets for PCF services.

To support the definition of exact pixel locations on screen, despite the possibility of delivery to platforms with different resolutions, PCF has adopted a **reference screen model**.

All PCF screen locations are defined with respect to a reference screen. This is a bounded rectangular co-ordinate system that comprises the reference screen in which all screen locations within the service description shall be located. The **referenceScreen** property of the PCF service item defines the size of the reference screen for a particular service description.

The mapping of the reference screen to a target device is the responsibility of a PCF transcoder. If such mapping needs to accommodate a difference in resolution between that of the reference screen and that of the display of the target device it shall observe the rules defined in clause 8.7.2. These rules make use of properties of the PCF service item, as defined in clause A.1.1.1.

8.7.2 Mapping the reference screen to a target device

8.7.2.1 Target device display resolution same as reference screen

When the resolution of the display of the target device and that of the reference screen are the same, pixel co-ordinates in the reference screen shall map directly to their equivalent pixel co-ordinates in the display.

A PCF transcoder may ignore the **referenceScreenMapping**, **referenceScreenAlignment** and **referenceScreenSurround** properties of the PCF service item.

8.7.2.2 Target device display resolution different to reference screen

If the resolution of the display of the target device is different to that of the reference screen the referenceScreenMapping property of the PCF service item is used to indicate how a PCF transcoder shall handle the resolution mis-match.

If the referenceScreenMapping property is set to "display-anamorphic" then the reference screen shall be scaled to completely fill the display of the target device. However, since the scaling applied horizontally may be different to that applied vertically, the resulting presentation may be anamorphic.

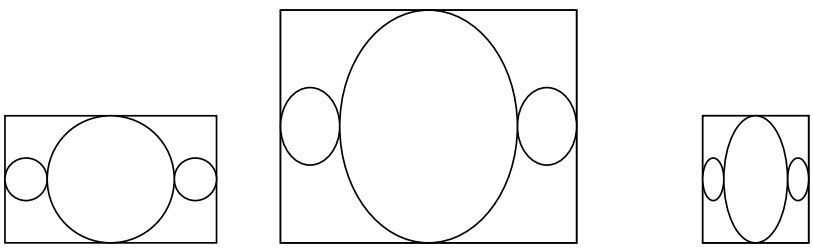

Figure 29: ReferenceScreenMapping property set to "display-anamorphic"

If the referenceScreenMapping property is set to "display-preserve" then the reference screen shall be scaled to fill the display of the target device as far as possible whilst preserving its aspect ratio, i.e. the scaling applied horizontally shall be the same as that applied vertically. The implication of this is that the reference screen may only fill the display in one dimension. The referenceScreenAlignment property of the service item shall be used to position the scaled reference screen within the display and the referenceScreenSurround property of the service item shall be used to fill any part of the display not mapped to by the reference screen.

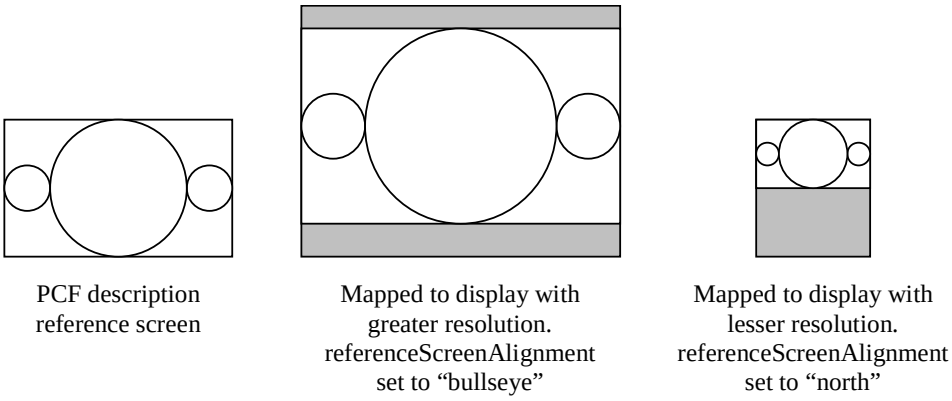

Figure 30: referenceScreenMapping property set to "display-preserve"

If the referenceScreenMapping property is set to "pixel" and the resolution of the display is greater than that of the reference screen then the reference screen shall not be scaled. The referenceScreenAlignment property of the service item shall be used to position the reference screen within the display and the referenceScreenSurround property of the service item shall be used to fill any part of the display not mapped to by the reference screen.

The mapping rules for the case where the referenceScreenMapping property is set to "pixel" and the resolution of the display is less than that of the reference screen is not defined by the present document.

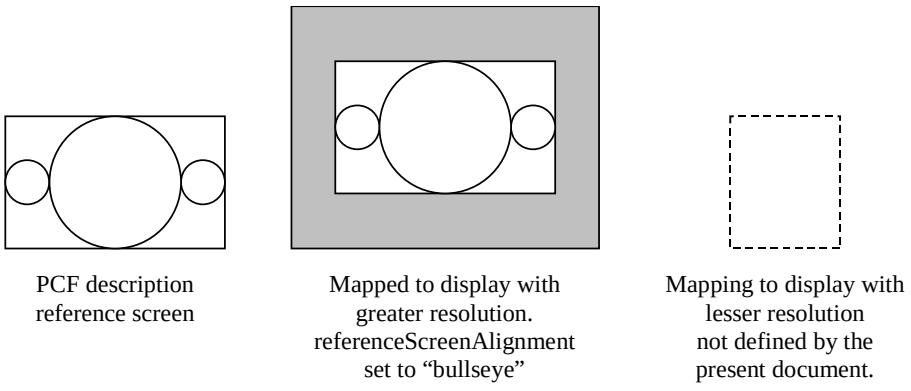

Figure 31: referenceScreenMapping property set to "pixel"

8.7.2.3 Scaling the reference screen (informative)

The PCF does not specify the algorithm to be used when mapping coordinates from the reference screen to the display of a target device in cases where their resolution is not the same. PCF transcoders may implement scaling algorithms as appropriate for specific target platforms. Distortions may occur when visual components are scaled for display on non-reference screens.

NOTE: The reference screen model provides service authors with a portable means to describe the visible aspect of a service so as be able to achieve equivalent appearance on displays with different resolutions. However, mapping between resolutions may require approximation and rounding of exact coordinate values. It will also require the scaling of certain content formats, such as images, with potentially variable results in quality depending on the particular implementation provided by the transcoder. To achieve precise control over a service's appearance at different screen resolutions, service authors can provide separate sets of screen positions, sizes (and if necessary assets) for each target screen resolution, using the approach described in annex L.

8.8 Registration of video and graphics

The PCF makes no assumptions regarding the physical implementation of target devices and handles all visible components within a single co-ordinate space, i.e. the reference screen. In this model registration of graphics to a particular point in the video is always achieved. However, in practice video and graphics may be handled differently within the receiver and the requirements of achieving the correct aspect ratio for any video and maintaining registration of video and graphics can not always be met. See annex P for further information on aspect ratio handling.

The **serviceAspectRatio** and **videoHandlingPriority** properties of the PCF service item may be used by a PCF transcoder to optimize the presentation of a service on a particular target device. The use of these by a PCF transcoder is not defined in the present document

8.9 Display stack model

8.9.1 Initializing the display stack

The PCF employs a layering model, known as the display stack. The initial ordering of visual components and containers within the display stack is implicit from the order in which they are declared in service and scene items.

At the start of the service session the display stack shall be initialized in an empty state. Then any visual components in the service item shall be added, one at a time and in their order of declaration, to the top of the display stack. Then any visual components in the first scene item shall be added to the display stack in the same manner. This results in the first visual component declared within the service item at the bottom of the display stack and the last component declared in the first scene item at the top.

When navigation between scenes occurs the current scene item is deactivated and all visual components declared within it shall be removed from the display stack. This leaves the display stack consisting of just the visual components declared within the service item in their current order.

NOTE 1: The order of these remaining components in the display stack may be different to their order of declaration in the service item due to use of action language.

NOTE 2: Service item ordering is maintained on scene transition.

NOTE 3: The entire target scene will appear on top of Service items after scene transition.

When the target scene item is activated all visual components shall be added to the display stack in the same manner as for the first scene item.

A special case is the handling of the Background component. Declared Background components shall not be added to the display stack (see clause A.2.1).

The initial order of visual components in the display stack shall be local to the layout container in which they are declared. Within a container, each child visual component shall be displayed in front of its preceding sibling. The visual component declared last is displayed on top. This is illustrated in figures 32 and 33 where the numbers indicate the order in which components are added to the display stack.

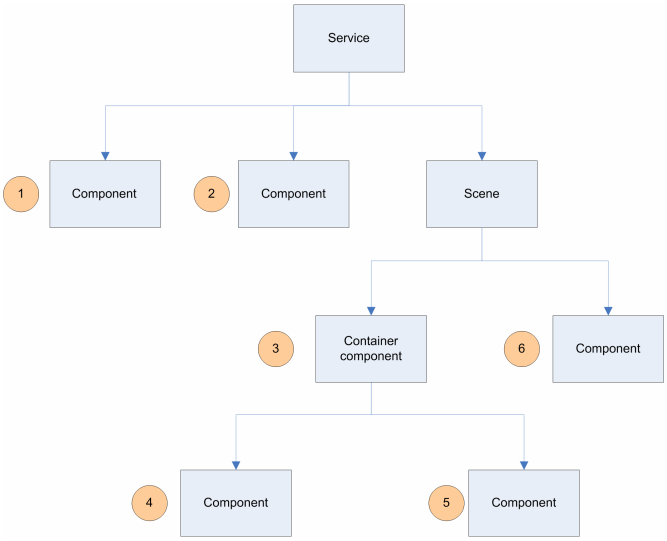

Figure 32: Order of component rendering

Figure 33: Display stack layering model

8.9.2 Manipulating the display stack

Visual components may be moved up and down in the display stack using the following actions:

- **infront** - to move one object in front of another;
- **behind** - to move one object behind another;
- **top** - to move an object to the topmost position on the stack;
- **bottom** - to move an object to the bottom of the stack.

When a visual component is moved in this way, other visual components may be partially or wholly obscured or uncovered.

Container components may also be moved up and down the display stack where this is permitted in their action definition.

EXAMPLE: Figure 34 illustrates the display stack model. In diagram *A* there are two Explicit Layout Containers, ELC 1 and ELC 2. Both have three overlapping components, whose ordering is implicit from the order in which they are declared. ELC 2 was declared after ELC 1, and therefore its child components are displayed above ELC 1 in the display stack. This is illustrated in diagram *B*, where the rear-most component of ELC 2 is in front of the front-most component of ELC 1.

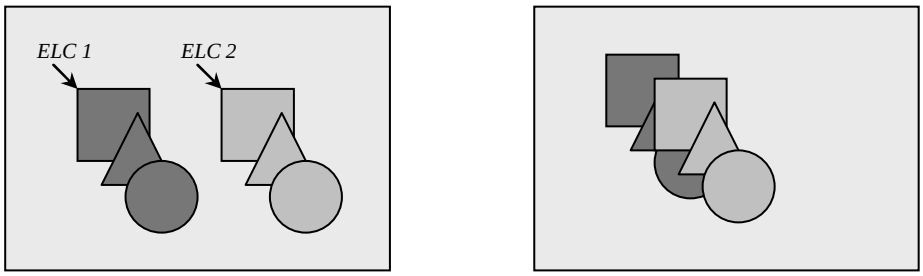

Figure 34: Display stack behaviour

Similarly, when an Explicit Layout Container is hidden or shown, all of its sub-components are hidden or shown as well.

8.10 Font selection

Font selection in the PCF is modelled closely on the font selection approach of CSS2 [22].

NOTE: Service authors are advised to refer to clause 15.1 of the CSS2 specification that informed the specification of fonts in the PCF.

The PCF does not support font definitions in service descriptions. Rather, it follows the CSS2 model where fonts are requested through definition of a series of font properties. The specified set of font properties in the service description provide the basis for a font selection exercise performed by the PCF implementation.

The PCF provides a rich set of font specification criteria to enable a very granular font specification. The PCF supports the following font selection criteria:

- **font-family** - specifies the font family to be used to render the text;
- **font-style** - specifies whether the font is to be displayed using a normal, italic or oblique face;
- **font-emphasis -** specifies emphasis for fonts;
- **font-variant** - specifies whether the font should be rendered using normal or small-caps glyphs;
- **font-weight** - specifies the boldness or lightness of the glyphs used to render the text;
- **font-stretch** - specifies the degree of condensing or expansion in the glyphs used to render the text;
- **font-size** - specifies the size of the font from baseline to baseline when set solid, i.e. when the font size and line height properties have the same value.

Font properties are defined in clause C.9.

9 Behaviour specification

At the core of an interactive service is an event-driven system, allowing external events to determine the route through the system rather than following some internally expressed route. The complexity of an interactive service stems from event/action patterns often coupled with concurrency and timing aspects. The PCF behaviour format provides a succinct means of capturing these intricacies in a way that can be easily and reliably implemented, analysed and verified.

The specification of the behaviour for PCF starts in clause 9.1 with an introduction to the behavioural concepts of PCF. This is followed by detailed accounts of the event types available to PCF (see clause 9.2), their propagation through the component hierarchy (see clause 9.3.4.2), the PCF action language (see clause 9.4), and the use of statemachines (see clause 9.6).

9.1 Introduction

The PCF behaviour provides for a service author:

- a description of the different **event** types and how they propagate within the service structure;
- the ability to describe a particular component as having **focus** thus influencing the event propagation method;
- an **action language** to describe the response to an event;
- a method to describe specific moments in the lifetime of an interactive service can be captured and described as a **state;**
- a method to describe the sequences of states and actions that occur in response to discrete events based on **statemachines;**
- a graphical representation of statemachines in the form of **statecharts;**
- an XML-based textual representation of statemachines.

The behaviour model is high-level, freeing a service from containing details of implementation and enabling service authors to succinctly express their intent.

NOTE: The high-level model enables transcoders to implement an optimized transition for each target platform.

The PCF provides the service author a variety of components with intrinsic behaviour. The behaviour of these components complies to the event propagation model and focus mechanism. For many PCF services, intrinsic component behaviour will be sufficient.

For occasions when these component do not provide the behaviour required, the service author shall describe additional behaviour using a combination of statemachines, action language and events, taking advantage of the event propagation model and focus mechanism.

9.1.1 Intrinsic component behaviour

The PCF provides a comprehensive set of components that a service author may use to build a service. Some of these components have intrinsic behaviour that causes them to react to certain events without the need for any additional PCF behaviour declarations.

EXAMPLE 1: In response to the UP and DOWN remote control keys, the highlight on a Menu component will move up and down the list of menu labels.

Some of the component's properties provide an interface to the behaviour description enabling the service author to affect certain aspects of the perceived behaviour.

EXAMPLE 2: The service author can control the colour associated with the highlighted menu label by providing a value for the **textcolor-focus** property of the Menu component.

9.1.2 Independent behaviour

When the PCF components do not provide the necessary behaviour to create a particular intended user experience, the service author may add their own behaviour. This shall be achieved using a combination of PCF statemachines and PCF action language descriptions by:

- creating custom components with the desired behaviour, as described in clause 7.6;
- attaching independent statemachines, as described in clause 9.6, within the component hierarchy.

9.2 Events

9.2.1 Run-time event model

Events in PCF are items that have a transient existence at run-time, and can only be accessed from action language that are triggered by that event.

An event has an **eventtype** that defines a set of named properties as specified in annex B.

Within a component declaration, interest in an event is declared through the **trigger** item within an **onEvent** item or a state **transition** item. The onEvent item or state transition item defines a **name** for the event whilst it exists in the component's content. For the purposes of action language, events that exist in a component's context shall be considered to be components themselves declared in the same context. Thus the event name may be used to directly access the properties contained within the event.

NOTE 1: OnEvent and state transition names shall, therefore, be unique within the context of the component that declares them.

An event may contain a **source** property that indicates the source component responsible for propagating the event into the context in which the action language is now executing.

NOTE 2: **Source** is therefore a reserved word and shall not be used as the name of an event property item in event type specifications.

NOTE 3: The value of **source** thus changes each time an event propagates up a level in the component hierarchy.

Two special values are defined to identify the source of the event:

- "this" identifies the source as the component in which the action language is defined;
- "system" identifies the source as the run-time environment itself.

NOTE 4: The values "this" and "system" are therefore reserved words and shall not be used to identify components in the PCF.

9.2.2 Event access declaration

Trigger declarations may contain a qualifier declaration. This provides a means to declare specific values for the properties of an event that must be met in order for the event to trigger a state transition.

Qualifier property values are tested for equality with their respective event properties and such tests must all evaluate to "true" in order for the event to occur.

Each qualifier value item shall provide the name of the event property and the literal value to test it against.

9.3 Event propagation model

The event propagation model is necessary within PCF in order that a transcoder is able to properly realizePCF specified behaviour. Included in this abstract model are the notions **system events**, **user input events** and **component events** and how they are influenced by focus.

The specification of the event propagation model for PCF starts in clause 9.3.1 with an introduction to the relationships with PCF that determine the event propagation model. The technical specification follows in clauses 9.3.2 to 9.3.5.

9.3.1 Introduction

9.3.1.1 Object model

The object model shown in figure 35 illustrates the relationship between an event and the service structure.

A single statemachine is interested in zero or more events.

A component is associated with a statemachine. It follows that a component is interested in those events associated with its statemachine.

A container is a type of component and is associated with zero or more components.

A scene is a container and, therefore, a type of component.

A service is associated with zero or more components.

A **focus manager** will be used to identify which component within a service currently has focus. No more than one component shall have focus at any given time during the lifetime of a service being rendered on a receiver. This component may be declared at scene-level, container-level or basic component-level.

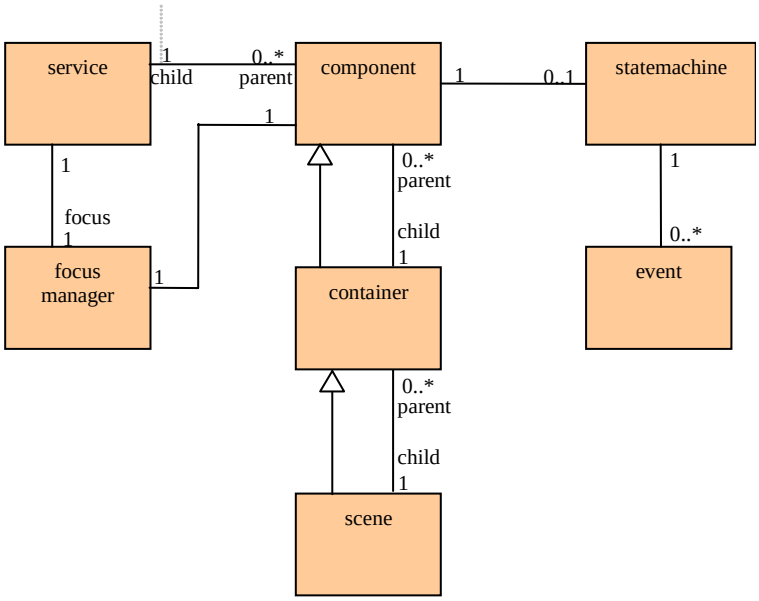

Figure 35: Object model showing how events relate to other aspects of a service.

9.3.1.2 Component containment hierarchy

A scene shall be constructed from a combination of logical groupings of components. This results in a hierarchy of components. There is a parent-child containment relationship between components at different levels within the hierarchy.

EXAMPLE: The component hierarchy depicted in figure 36 shows the containment relationship within the service called "root". For example, the container "group" is the parent of "comp3" and "comp4". It is the child of "scene1". The descendents of a component include all the generations of containment. Therefore, the descendents of "scene"1 are "comp1", "group", "comp2" and the children of "group", "comp3" and "comp4".

All components within the hierarchy of a scene are active when the scene is active. The event propagation model shall determine how events are routed around this hierarchy.

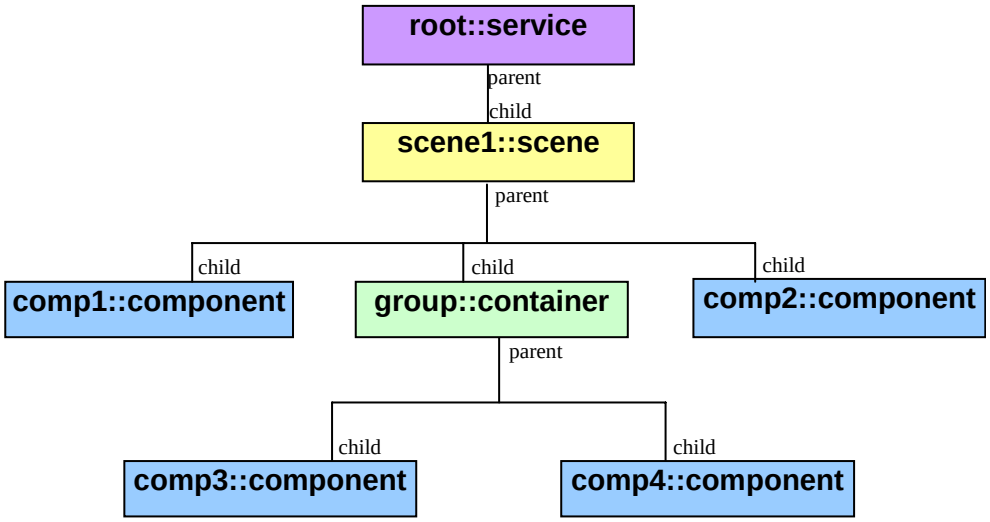

Figure 36: Abstract component hierarchy (not UML)

9.3.1.3 Event propagation

A PCF event is an occurrence within the lifetime of a service that may invoke a response in some way.

Event types include system events (see clause 9.3.2), user input events (see clause 9.3.3) and component events (see clause 9.3.4).

The event propagation model specifies the rules that determine the routing of events around the component containment hierarchy. The model provides a consistent solution for all three event types whilst acknowledging their different purposes.

9.3.2 System events

9.3.2.1 Overview

System events are independent of the user's interaction with the service and so are not linked to the currently focused component. Instead, a response may be gained from an interested component within the component hierarchy.

System events include:

- file/object updates in the transmission stream;
- synchronization with streams;
- in-band triggers.

9.3.2.2 System event propagation rules

An event shall initially target the Service component.

- The event shall target the active scene and all its descendents in an unspecified. All components that have an interest in this event may respond.
- The event shall target all descendents of the **Service** component excluding any non-active scene components.

EXAMPLE: There are five components interested in a particular system event in the component hierarchy shown in figure 37 (indicated using a red border). The system event targets each of the three components within the active scene (scene1) and targets the direct child components of the service indicated by the red arrows. The highlighted component contained within scene2 is not targeted. The order in which components are targeted is not specified.

NOTE: The mechanism by which events are made available to components' children can be made efficient through careful implementation. The transcoder need not examine all components within the component hierarchy in order to locate those with an interest in a particular event. In particular:

- If a component has knowledge of the events of interest to its descendents then the distribution of an event can be restricted to those branches of the component hierarchy where there is interest in the current event.
- If a component is aware of exactly which of its descendents has interest in a particular event then those components could be targeted directly.

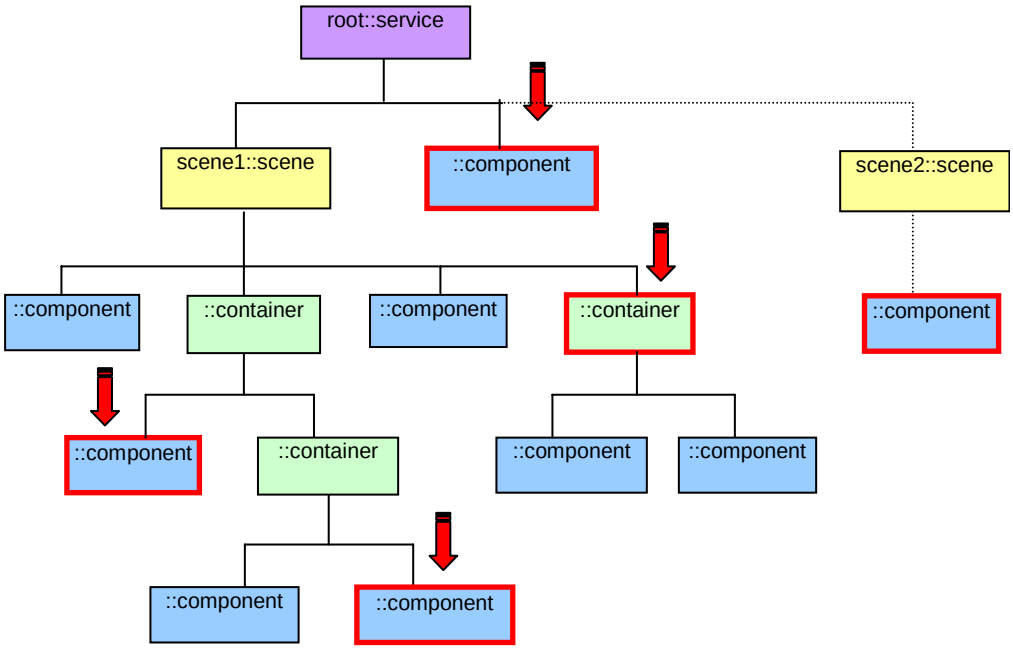

Figure 37: Events target service level components, and those of the active scene (scene1)

9.3.3 User input events

9.3.3.1 Overview

User input events are generated as a result of user interaction. The scope of PCF considers only:

- Remote control key press (virtual keys).

It is the concept of focus that makes the event propagation model for user input events different to those of system events.

9.3.3.2 Focus control

No more than one component shall have focus at any given time. A user event shall initially target the component that currently has focus.

If no component has been identified as having focus, then by default, the scene will assume focus and a user event shall initially target the scene.

The PCF action language will facilitate the movement of focus between components.

EXAMPLE: The component hierarchy shown in figure 38 includes a Menu and a SpinControl that respond to DOWN user key events. If the Menu has focus when a DOWN event is generated then the event first targets the Menu component. The Menu responds to the event. The SpinControl, despite having an interest in this event, does not respond.

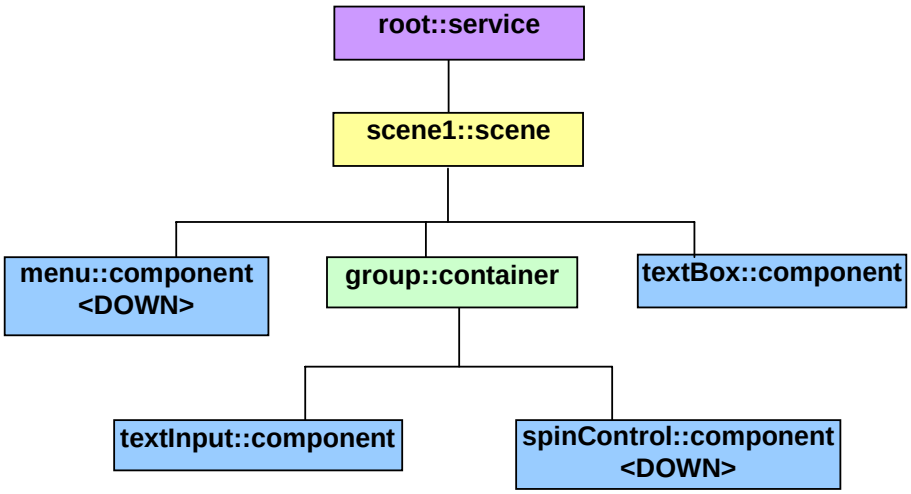

Figure 38: Hierarchy of components within an example scene

9.3.3.3 User input event propagation rules

1) An event shall initially target the focused component.

2) If the component is interested in the event then the event propagation shall terminate and the component may respond to the event.

3) If the component is not interest in the event then the event shall target the parent component of the current component.

4) Steps 2 and 3 repeat until either:

   - a component has an interest in the event and propagation terminates (step 2); or

   - the target component is the Service component.

5) If the Service component is not interested in the event then the event propagation continues as if it were a system event.

EXAMPLE: The component with focus in figure 39 is highlighted with a yellow border. Those components interested in a particular event have been drawn with a red border. The event first targets the focused component. This component has no interest in this particular event and so the event targets its parent component (a container). None of the components on the path to the Service component have an interest in this event. When the event reaches the Service component it propagates as if it were a system event. It targets all those components within the currently active scene that have an interest in this particular event. This is illustrated in figure 40 where the event targets the three components within scene1. Notice that the event does not target the component in scene2 despite its interest in the event.

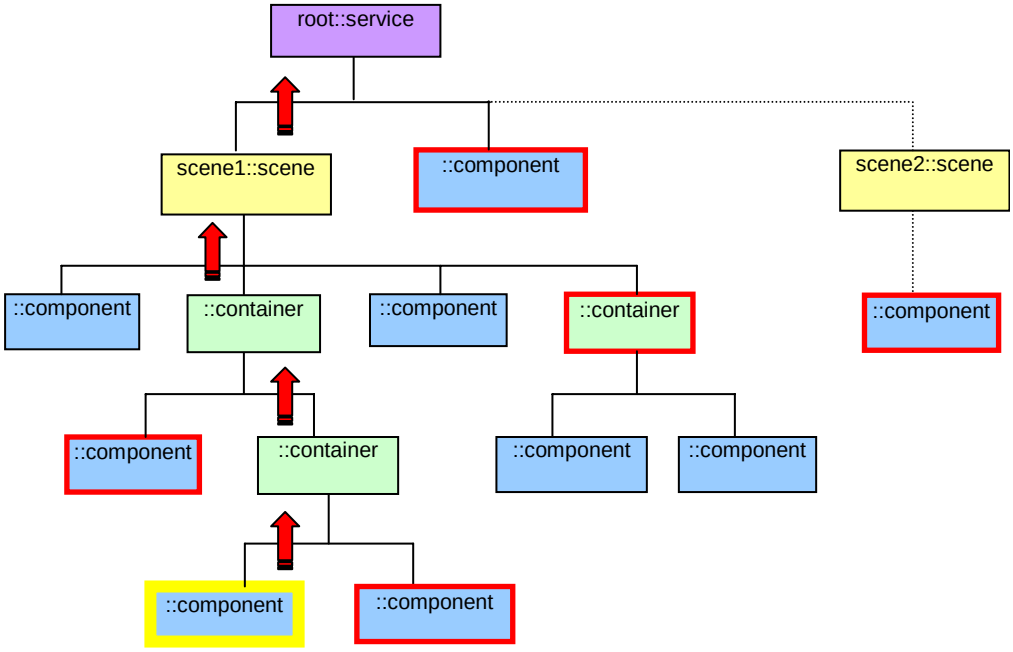

Figure 39: Event propagation from focused component towards the Service component

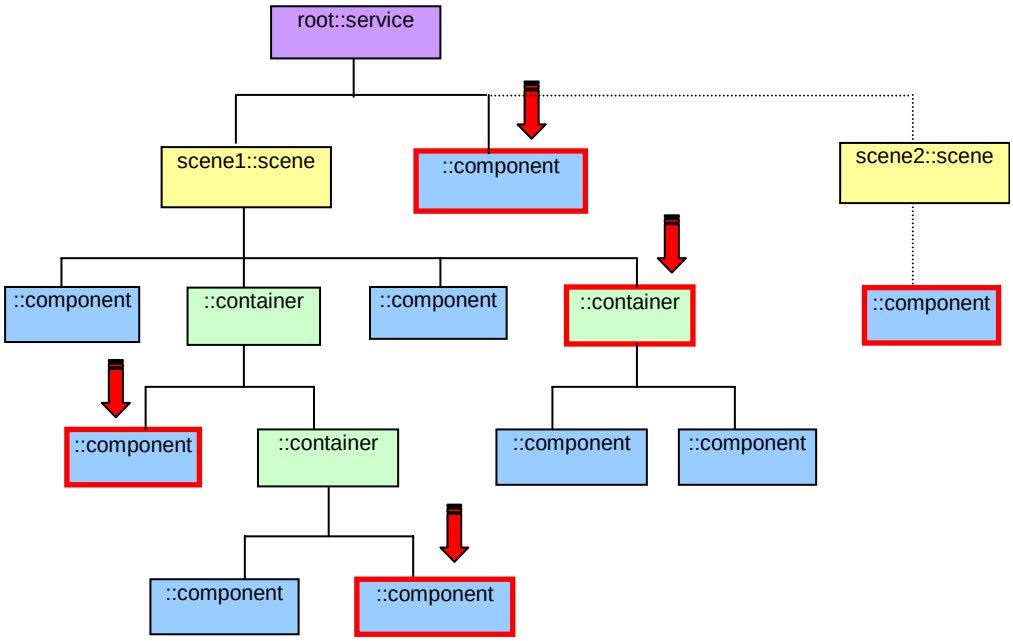

Figure 40: Event propagation after reaching the scene component.

NOTE: The mechanism by which events are made available to the descendents of the scene component can be made efficient through careful implementation. In particular:

- The transcoder need not reconsider those components that declined the event whilst propagating towards the root of the component hierarchy.

9.3.4 Component events

9.3.4.1 Overview

A component generates component events to provide information about a change in its internal state. Not all internal state changes of a particular component type shall be reported, but the component type specifications in annex A shall specify which events a particular component type will generate, and under what conditions these events will occur.

EXAMPLE 1: A **Button** component generates an **OnSelect** event when the user presses the select key whilst the button is active and has focus.

The purpose of component events is twofold:

- To provide notification hooks, thus allowing the intrinsic behaviour of a component to be extended with custom behaviour items.
- To allow parent container components to perform some action in response to an event generated by one of their children. By this means a parent component can link the behaviours of two or more of its children.

EXAMPLE 2: When the user changes the selected item on a **Menu** component the **Menu** generates an **OnSelect** event. If a scene has a **Menu** and a **TextBox** as two of its children it could link the behaviour of these two components together by responding to the **Menu**'s **OnSelect** event by changing the text within the **TextBox**.

Behavioural access to PCF components is defined by a set of scoping rules (see clause 6.3.7). Only behaviour items declared within a component itself, or within a component's parental hierarchy, may manipulate that component. It follows that this scoping applies to the visibility of events generated by a component: the only behaviour items that may respond to a component event are those declared within the generating component, or those declared within the generating component's parental hierarchy.

9.3.4.2 Component event propagation rules

When a component generates an event, event handlers shall be given the opportunity to respond to that event in the following order:

1) An event shall initially target the generating component.
2) If the component has custom behaviour items that are interested in the event then the event propagation shall terminate and the behaviour items may respond to the event.
3) If the component is not interested in the event then the event shall target the parent component of the current component.
4) Steps 2 and 3 repeat until either:
   - a component has an interest in the event and propagation terminates (step 2); or
   - the target component is the **Service** component.
5) If the event propagates up to the service and is not handled then it is discarded.

NOTE: Unlike user interface events, component events that have propagated up to the service level are **not** broadcast to the rest of the scene.

EXAMPLE: The component generating an event is highlighted with a yellow border. The parent component interested in that event has been drawn with a red border. In this case it is the current scene. The event first targets the generating component. Behaviour declared within this component has no interest in this particular event and so the event targets its parent component (a container). None of the components on the path to the scene component have an interest in this event. When the event reaches the scene component, behaviour items declared within the scene react to the event.

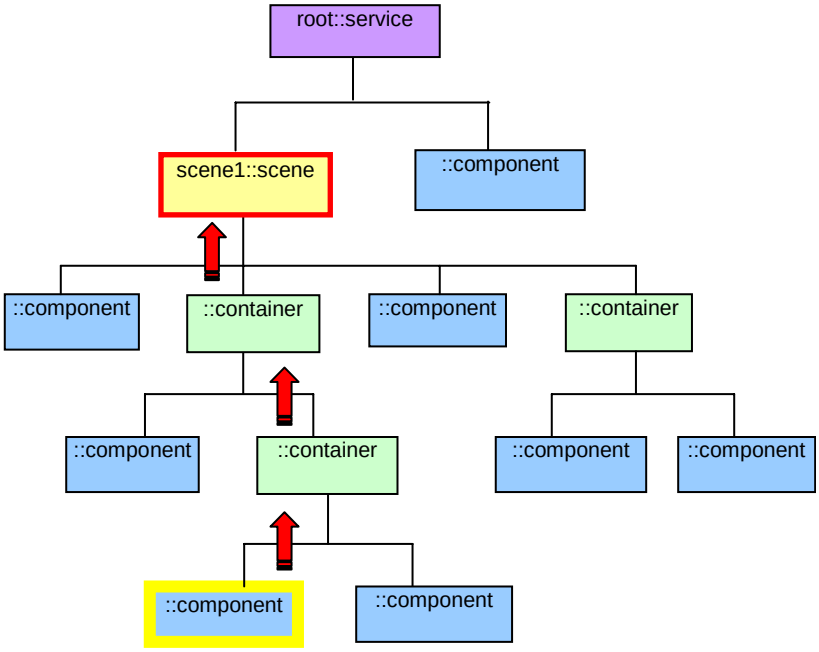

Figure 41: Event propagation from generating component towards the Service component.

9.3.5 Error Events

An error event is a specialization of a component event, and shall behave exactly as a component event for purposes of propagation and declaration of trigger items to receive such events. The definition of the error event is in annex B

Conditions under which error events are generated by components are declared in the component specification annex A. This annex also defines the values contained in the properties of each error event.

9.3.5.1 Execution error levels and default responses

Errors that propagate through the service description are identified as being at one of three levels. Their level determines the default service session response if they are not handled explicitly within the service description provided by the author.

The levels are:

- **error -** an error event of this level propagating through to the **Service** component is considered fatal and shall cause termination of the service session.
- **warning -** an error event of this level propagating through to the **Service** component is recorded by the service session for later examination, but does not affect the service session.
- **notice -** an error event of this level propagating through to the **Service** component is discarded and does not affect the service session.

9.3.5.2 Error types

A PCF service description can suffer:

- **reference errors -** when a reference is not resolvable at run-time;
- **component errors -** as defined in clause B.4.1;
- **guard failures -** the condition cannot be evaluated;
- **action language errors -** numerical or semantic errors during execution.
- A reference error shall result in the generation of a **ReferenceError** event.

EXAMPLE 1: At run-time, a receiver may discover that a piece of the PCF service description cannot be found when activating a scene that references it.

A component error shall result in the generation of an error event with the property values as specified by the GeneratedError definitions given in the component specification.

A guard failure and an action language error are equivalent. Both are failures in execution of action language description, and shall result in the generation of an **ExecutionError** error event from the component containing the action language description with a **level** property as specified in clause 9.3.5.1

EXAMPLE 2: An action language error can occur if arrays indexes are out of bounds, or if a type conversion fails, or if there is an attempt to divide a number by zero.

A guard failure shall also result in the guard condition evaluating to "false".

An action language failure of level "error" shall result in the action language execution being abandoned at the position where the error occurred and the action language context being exited.

An action language failure of level "warning" shall not have any effect on subsequent execution of the action language description.

An action language failure of level "notice" shall not have any effect on subsequent execution of the action language description.

9.4 Action language

9.4.1 Introduction

The action language allows a PCF service author to describe a sequence of actions to be taken during the lifetime of a PCF service, where these actions test and modify the current state of an interactive service. The action language is designed to be similar to ECMA-script (see bibliography) wherever possible.

EXAMPLE: The following fragment of action language sets the fill colour of a Rectangle when the loadComplete variable is true.

```
if (loadComplete == true)
    statusRectangle.fillcolor = #7f7f7f;
```

The action language supports flexible partitioning of execution between head-end and receiver. As such, the action language and its run-time model are not specified in terms of what a machine should do to carry out a sequence of actions, but rather what an author can expect to have happened once the execution of an instance of action language has completed.

NOTE: The PCF action language is expressed as an EBNF grammar provided in annex I. It should also be considered as a semantic action model that could be represented by alternative notations. An XML [5] representation of the action model has been developed to validate the principle; however, this is not currently part of the specification.

The action language provides the following features:

- strongly typed values that are compatible with PCF static service descriptions, as described in clause 6.2.1.1, and a mapping between values in a static service description and values in an active service description;
- a library of functions for use in expressions that can be used to manipulate values of PCF data types, as described in clause 9.4.9;
- a library of system actions that can be called from any PCF action language, as described in clause 9.4.7.

9.4.2 Representation and execution

The PCF action language is represented by the normative EBNF grammar provided in annex I. The productions of this grammar represent the PCF action language **run-time execution model**, as described in clause 9.4.6.

Action language has two uses in PCF:

- to define sequences of actions to be executed in response to an event or state transition, as described in clause 9.6.3.1;
- to define guard conditions, as described in clause 9.6.3.3.

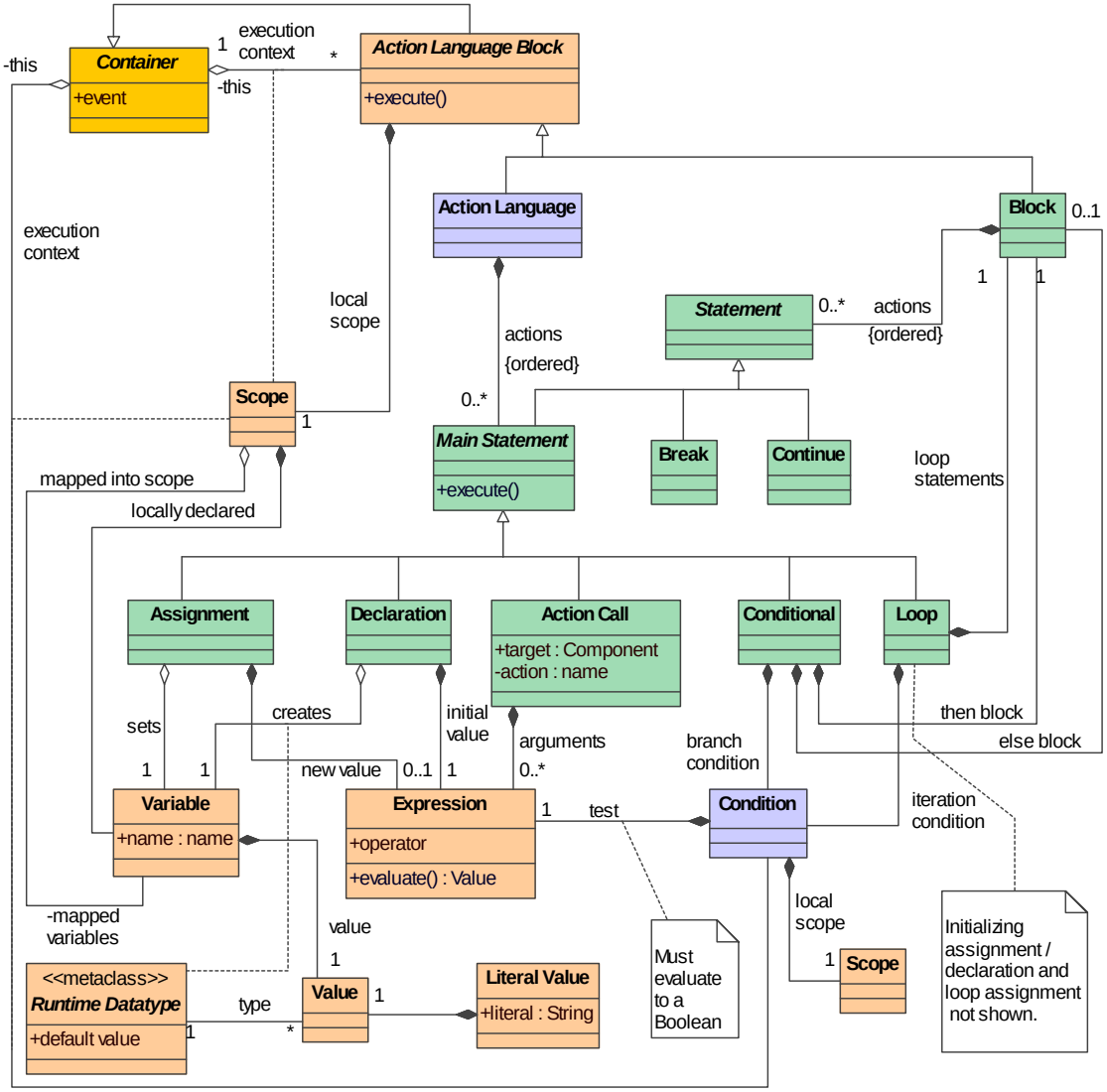

Figure 42: Action language model

The model for representing action language is shown in figure 42. The top level items of the action language execution model are **action language** and **condition** items, which are collectively known as **action language items**. These are contained within a generic **container** item, such as a guard condition in a statemachine, which provide the **execution context** of an action language instance. The items in this model are described below:

- **Action language block** items represent all sequences of actions that can be executed. They can act as a container for other action language blocks.
- **Action language** items are kinds of Action Language Block items and contain a sequence of **Statement** items, where each statement item is considered as an action to be executed in the order in which they are declared. The different kinds of statement are described in clauses 9.4.6.1 to 9.4.6.6.
- Action language block instances have a **scope**, known as their **local scope**, which contains variables mapped in from the execution context, as described in clause 9.4.5.2, and any locally declared variables.

- **Run-time datatype** items represent the data types available for values in any action language instance. Each run-time data type is a set of possible values, as described in clause 9.4.5.2
- **Value** items represent values that are members of the run-time data types.
- **Literal value** items are literals that represent values of a given run-time data type item.
- **Variable** items are mutable named values contained within the scope of an action language instance. Variable items exist within a scope for one of two reasons:
  - They have been declared locally using the declaration statement described in clause 9.4.6.3. Variable items declared this way are known as **local variables**.
  - They have been mapped into the scope as described in clause 9.4.5. Variable items available within an action language instance by mapping are known as **mapped variables**.
- **Expression** items represent an evaluation that can be carried out as part of a statement or condition and return a value that is calculated based on the structure of the expression. An output value shall be calculated by carrying out the evaluation operation for the expression, as described in clause 9.4.7. The evaluation result is a value that is a member of one of the run-time data types.
- **Condition** items consist of an expression item that evaluates to a Boolean value. Conditions items are also used in guard conditions of statemachines, as described in clause 9.6.3.3.
- **Block** items are a sequence of statement items that are executed within their own local scope. The execution context for a block is its nearest ancestor action language block container, known as its parent block.

9.4.3 Valid action language

To be executed in a run-time environment as part of a user experience, action language items shall be valid. This means that they are syntactically correct with respect to the action language notation grammar in annex I and semantically correct according to the rules stated in clause 9. If action language is invalid, it shall not be executed and an error shall be reported. This error shall be known as a **transcode-time error**.

NOTE: The action language is designed in such a way as to minimize run-time errors. A transcoder may check the validity of action language items and report where errors are found prior to the rendering of a service.

9.4.4 Action language data type and action language items

The **action language data type** shall represent the set of all possible instances of the action language model. Members of the action language data type are **action language items**. Action language items shall be PCF items as defined in clause 6.2.1 of the architecture.

NOTE 1: Action language items can benefit from the reuse permitted by the PCF architecture's reference and navigation model, as defined in clause 6.4.

NOTE 2: The scope of any action language block contains the local variables and mapped in variables in the scope of the parent action language block, according to the mapping process described in clause 9.4.5.

EXAMPLE: Action language items are represented by the "ActionLanguage" element in the PCF XML schema. The following fragment of XML illustrates the reuse of an action language item.

```
<ActionLanguage name="incrementScore">
  score.value++;
</ActionLanguage>

<Scene name="scoreDisplay">
    <IntegerVar name="score">
        <Integer name="value" value="0"/>
    </IntegerVar>

    <OnEvent name="select">
        <Trigger eventtype="KeyEvent">
            <UserKey name="key" value="VK_ENTER"/>
        </Trigger>
        <ActionLanguage href="#../incrementScore" context="original"/>
```

```
    </OnEvent>
</Scene>
```

9.4.5 Run-time data mapping

9.4.5.1 Execution context

Action language items are declared in a rendering context provided by the item's closest ancestor component item, known as the parent component item. The parent component item provides all the **mapped variables** into the execution context provided by the container.

All mapped variables for the local scope of an action language block with its associated parent component item shall be determined prior to the execution of that action language block. Mapped variables shall be determined by the following rules:

- All renderable items in the rendering context of the container that are active at the same time as this container shall become mapped variables. This includes all items mapped into the scope of the parent component items using the parameter lists defined in clause 6.4.5. The run-time data type of these items shall be "component". In this case, the name of the mapped variable shall be the same as the PCF item to which it refers.

- Renderable items shall include all renderable items described within descendent collection items of the parent component item. In this case, the name of the mapped variable shall be the path from the parent component item to the PCF item to which it refers. The path shall consist of a sequence of one or more collection names separated by a period character ".", followed by another period character and the name of the item itself.

EXAMPLE 1: The following XML fragment, which is written according to the PCF XML schema, shows how a component item within a collection item can be referenced from an action language item.

```
<Scene>
    <Collection name="variables">
        <IntegerVar name="counter">
            <Integer name="value" value="1"/>
        </IntegerVar>
    </Collection>
    <OnEvent name="select">
        <Trigger eventtype="KeyEvent">
            <UserKey name="key" value="VK_ENTER"/>
        </Trigger>
        <ActionLanguage name="jumpCounter">
            variables.counter.value += 10;
        </ActionLanguage>
    </OnEvent>
</Scene>
```

NOTE 1: It shall not be possible to access or manipulate a collection item from action language.

- All properties of the parent component item with "access" specifiers set to "readOnly", "final" and "readWrite" shall become mapped variables. Table 27 shows the mapping between the "declaration type" of a PCF item type name and the "run-time type" name of the mapped variable.

Table 27: Equivalent declaration and run-time data types

declaration type	run-time type	declaration type	run-time type
boolean	boolean	boolean array	boolean[]
color	color	color array	color[]
currency	currency	currency array	currency[]
date	date	date array	date[]
dateTime	datetime	dateTime array	datetime[]
integer	integer	integer array	integer[]
markedUpText	markeduptext	markedUpText array	markeduptext[]
position	position	position array	position[]
time	time	time array	time[]
proportion	proportion	proportion	proportion[]
timecode	timecode	timecode array	timecode[]
size	size	size array	size[]
string	string	string array	string[]
URI	URI	URI array	URI[]

NOTE 2: When mapping component item properties and renderable items into the execution context of the action language, a name clash cannot occur as the mapped in items shall have been declared within the same referential context.

EXAMPLE 2: The following XML fragment, which is written according to the PCF XML schema, illustrates a component item of "readWrite" access type being mapped into an action language item and can be modified. Note also the use of the XML CDATA tag [5] as the action language example contains an element-like tag that is not part of the PCF schema.

```
<Service>
    <Proportion name="serviceAspectRatio" value="4 3"/>
    <OnEvent name="aspect_shift">
        <Trigger eventtype="KeyEvent">
            <UserKey name="key" value="VK_ENTER"/>
        </Trigger>
        <ActionLanguage>
            <![CDATA[
                serviceAspectRatio = <proportion>16 9</proportion>;
                ]]>
        </ActionLanguage>
    </OnEvent>
</Service>
```

- All descendent renderable items and properties within a mapped item of "component" shall become mapped variables, named according to the component, followed by a period character "." and determined by applying the rules in the rest of this clause.

9.4.5.2 Run-time data types

Other than the "component" run-time type already discussed in clause 9.4.5.1, the run-time data types of the action language model shall be defined to have the names specified in the "run-time type" columns of table 27. Authors can assume that the sets of values represented by equivalent declaration and run-time types are the same.

NOTE: Transcoders can convert values from the acceptable values for PCF data types to platform-specific data types for the execution of action language. Platform-specific data representation of data should not affect the outcome of the execution of action language items, as defined in clause 9.

Enumeration values of the enumeration data type defined in clause 6.2.2.3 can be mapped into the action language but shall not be declared within the action language. Enumerations can be written in the action language grammar using the "EnumerationLiteral" production (see clause I.2.3). The system action library in annex J defines enumerations that can be used as arguments to specific action calls. The component specifications in A define enumerations that can be assigned to specific component item properties.

All value items, irrespective of data type, can have a nil value in PCF service description. The "NilLiteral" token in the action language grammar shall represent equivalent nil value for run-time data types. Any expression that contains a nil value shall evaluate to nil.

9.4.6 Run-time execution model

9.4.6.1 Statements

Action language blocks consist of a sequence of **statement** items, each of which shall be executed in the order in which it is declared, starting with the first in the sequence. Statements are represented by the "Statement" production (see clause I.3.1) in the action language notation. In the notation, statements within an action language block are separated by semi-colon characters "`;`".

The run-time execution model does not state how long the execution of a statement item should take or whether the next statement must start immediately following completion of the end of the previous one.

NOTE 1: An author can expect that a PCF transcoder will create a rendering of a PCF service that tries to execute a sequence of actions as fast as practically possible on a given platform. This should giving the appearance to the user that execution of an action language item is atomic and as near to instantaneous as possible.

The main statements of the PCF action language are assignment, declaration, action call, conditional and loop. These are described in the clauses 9.4.6.2 to 9.4.6.6 in terms of the change of state they cause at the completion of the statement's execution. Some statements generate run-time errors or warnings, as defined in clause 9.4.6.7.

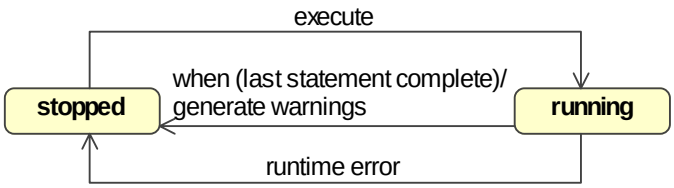

Figure 43: Action language execution states

An action language item has two execution states, "stopped" and "running", as shown in figure 43. When an action language item is executed, the first statement it contains should be executed immediately and the action language item shall transition to the state "running".

When the last statement in the sequence has completed its execution, the action language item shall transition to the "stopped" state. At this point, any warnings generated during the execution of the action language items can be posted as component events, as described in clause 9.3.4. When a run-time error occurs in the execution of some action language, execution of the action language shall stop, returning the action language instance to its "stopped" state.

NOTE 2: The PCF action language does not have user-definable functions or data structures as in languages such as C. An author must work within the constraints of the action language, making use of the modular representation of the PCF architecture's referencing model, to achieve modularity and reuse of code.

9.4.6.2 Assignment statement

An **assignment** statement uses a **variable** item and, depending on the type of the assignment statement, has an optional expression item. Seven types of assignment statement are defined. An assignment statement is represented by the "Assignment" production (see clause I.3.2) in the PCF action language notation.

Where an expression is provided, the execution of an assignment statement shall have completed when both:

- the expression has been evaluated;
- the value of the variable has been set to the result of the evaluation, according the type of the assignment statement.

The five types of assignment statement for which an expression is required are:

- Direct assignment ("="): The result of the evaluation shall be set as the new value of the variable.
- Assignment by addition ("+="): The result of the evaluation shall be added to the current value of the variable, using the add operator that is described in clause 9.4.7.2, and assigned as the new value of the variable.

NOTE 1: For any variable v and expression e, an assignment by addition "`v += e`" is equivalent to the assignment "`v = v + e`".

- Assignment by subtraction ("`-=`"): The result of the evaluation shall be subtracted from the current value of the variable, using the subtract operator that is described in clause 9.4.7.2, and assigned as the new value of the variable.

NOTE 2: For any variable v and expression e, an assignment by subtraction "`v -= e`" is equivalent to the assignment "`v = v - e`".

- Assignment by multiplication ("`*=`"): The result of the evaluation shall be multiplied with the value of the variable, using the multiply operator that is described in clause 9.4.7.2, and assigned as the new value of the variable.

NOTE 3: For any variable v and expression e, an assignment by multiplication "`v *= e`" is equivalent to the assignment "`v = v * e`".

- Assignment by division ("`/=`"): The result of the evaluation shall be divided by the value of the variable, using the divide operator that is described in clause 9.4.7.2, and assigned as the new value of the variable. If the expression is the literal value 0, the PCF service shall be invalid. If the expression evaluates to 0 at run-time, a "division by zero" error shall be generated.

NOTE 4: For any variable v and expression e, an assignment by multiplication "`v /= e`" is equivalent to the assignment "`v = v / e`".

For all direct assignments, the run-time data type that is the result of evaluating an expression shall be the same as the run-time data type of the variable to which the value is assigned. This can be checked prior to execution of the action language, and if the types are not the same then the action language instance shall be considered as invalid.

For all other assignment types that have an expression, either:

- the expression shall evaluate to the integer data type and the variable shall be of the integer data type;
- or the action language instance shall be considered as invalid.

The PCF action language shall not permit the implicit casting of a variable's value to a different run-time data type on assignment.

Where no expression is provided, the assignment statement is either an assignment by increment or assignment by decrement statement, as described below:

- Assignment by increment ("`++`"): A value of 1 shall be added to the value of the variable.

NOTE 5: For any variable v, an assignment by increment statement "`v++`" is equivalent to "`v = v + 1`".

- Assignment by decrement ("`--`"): A value of 1 shall be subtracted from the value of the variable.

NOTE 6: For any variable v, an assignment by increment statement "`v--`" is equivalent to "`v = v - 1`".

For assignment by increment/decrement statements, the value shall be of the integer data type or the action language shall be considered invalid.

Assignment to a mapped variable that is a property of a component item with **access** specifier "initializeOnly", "readOnly" or "final" shall result in the action language being invalid.

NOTE 7: It shall only be possible to assign values to mapped variables derived from properties with **access** specifier "readWrite".

9.4.6.3 Declaration statement

A declaration statement uses a run-time data type item to create a new variable item and has an optional expression that can be used to initialize the variable's value. A declaration statement is represented by the "Declaration" production (see clause I.3.4) in the PCF action language notation.

The execution of a declaration statement shall be considered complete when a new variable has been created within the local scope of the action language instance containing the declaration statement. The variable shall have the name specified in the declaration statement.

Where an optional expression is provided, this shall be treated as a declaration statement followed by an assignment statement. This assignment statement has as its variable the newly created variable and the declaration's expression. The declaration shall be considered complete once the assignment statement has completed.

Where no expression is provided with the declaration, the variable shall be assigned the nil value for the run-time data type associated of the declaration.

EXAMPLE: The following two action language sequences are equivalent. Firstly, a declaration with an expression:

```
integer x = 3;
```

Secondly, a declaration followed by an assignment:

```
integer x;
x = 3;
```

When a new variable is declared, it shall exist within the local scope of the action language block in which it is declared. The variable shall be available for all statements after the declaration until the end of the current sequence of actions. Local declaration shall override any mapped variables of the same name.

Any reference to the variable that is made by a statement before the declaration statement shall be treated as follows:

- If the reference is to one of the mapped variables in the current local scope, the reference shall be resolve to this before to the declaration statement.
- Otherwise the action language shall be considered as invalid.

Independently of the run-time data type of the variable, the name of the variable shall be unique within all variables declared in the local scope. If a variable is declared with the same name as another variable that has been previously been explicitly declared in the same action language block, the action language instance shall be considered invalid.

9.4.6.4 Action call statement

An action call statement has two forms:

- Calling a system-level action, known as a **system action**, as specified in annex J.
- Using a target component that is in scope, calling an action specified the component's specification, known as a **component action**, as described in annex A.

In both cases, the action shall have a name and may have a sequence of expressions to be evaluated to provide arguments for the action.

The execution of an action call statement shall have completed when:

- any expressions provided as arguments for the actions have been evaluated and passed to the action;
- for system actions and component actions, the action has completed as specified for the zero or more values passed in.

A system action can be distinguished from a component action, as no path to a particular component is provided.

- For system actions, if no action of the specified name exists as listed in annex H then the action language shall be considered as invalid.

For component actions, the component specification for the type of component referenced in the action language shall provide an action of the given name, or the action language shall be considered as invalid.

Each action has a signature that describes a sequence of data types that shall match to an acceptable sequence of values of those types as arguments to the action call. All the expressions in a sequence that make up the arguments to an action call shall evaluate to values of data types that match the signature, in order, item-for-item, or the action language shall be considered as invalid. The order of declaration of any Parameter items within the component action specification defines the signature of the action.

NOTE: The signature may be a sequence of zero length. In this case, it shall be acceptable for a sequence of arguments to be of zero length.

EXAMPLE 1: The following system action call statement causes a service to transition to a new scene:

```
SceneNavigate(<uri>#~/homepage</uri>, <enum>mark</enum>, nil);
```

EXAMPLE 2: The following component action hides the presentation of a **TextBox**.
help_text.hide().

9.4.6.5 Conditional statement

The conditional statement shall use the evaluation of a condition to choose whether to execute a block of statements or not. Where the block is not executed and an optional alternative "`else`" block of statements is provided, this shall be executed instead. A conditional statement is represented by the "Conditional" production (see clause I.3.5) in the PCF action language notation.

The execution of a conditional statement shall have completed when:

- the condition expression has been evaluated;
- if the condition evaluates to true, execution of the "`then`" statement block has been executed and is completed;
- if the condition evaluates to false:
  - if an "`else`" expression block is provided, this statement block has been executed and is completed;
  - if no "`else`" expression block is provided, the conditional statement is complete.

NOTE: A block has completed executing when each statement within that block has completed executing.

EXAMPLE: The following action language statement shows a conditional statement, which is represented by the keyword "`if`".

```
if ((a < b) && (b == 10)) a++;
else { a = 0; b = 0; }
```

9.4.6.6 Loop statement and loop control

A loop statement is represented by the "Loop" production (see clause I.3.6) in the PCF action language notation.

The loop statement shall consist of:

- an optional initial declaration statement;
- a required loop condition;
- a single loop statement or block of loop statements;
- an optional loop assignment.

Execution of a loop statement shall take the form of a sequence of iterations. Before the first iteration, the optional initial declaration statement shall be executed in the same scope as the loop statement. An iteration shall consist of:

1) The iteration starts with the evaluation of the loop condition. If the condition evaluates to false, the loop shall terminate. Termination shall result in: the end of the current iteration; no further iterations of the current loop; execution of the loop statement to have completed. If the condition evaluates to true, the iteration continues with the next step.

2) Next, the single loop statement or block of loop statements is executed. A block of loop statements may contain loop control statements as described below.

3) Next, if the optional loop assignment is present, this shall be executed.

4) The execution completes and a new iteration of the loop shall start, as defined in the first step above.

The loop block and its descendant blocks can contain the loop control statements break and continue. These shall be interpreted as follows:

- **break** - execution of the break statement shall stop the execution of the current loop block, stop the current iteration and terminate the loop.
- **continue** - execution of the continue statement shall stop the execution of the current loop block and jump to the next step of the iteration (evaluation of the optional loop assignment).

NOTE: Any declaration statements within descendant blocks of the action language notation shall re-initialize the variable they declare within the local scope.

EXAMPLE: The following action language statement shows a loop statement, which is represented by the keyword "for". The example illustrates action language that could be used to fade in the appearance of some text ("text3") to a level that does not exceed an author's defined limit ("maxIntensity").

```
for ( int i = 0 ; i < 255 ; i += 4 ) {
  if (i > maxIntensity) break;
  integer[4] colorNums = [i, i, i, 255];
  color fontColor = integerToColor(colorNums);
  text3.textcolor = fontColor;
}
```

9.4.6.7 Execution errors

Run-time execution errors result in "execution error" events as defined in clause 9.3.5.

Run-time execution errors of "error" level are:

- divide by zero during the evaluation of an expression;
- array index out of bounds;
- failure of a action call;
- failure of a function in the expression function library, such as a type conversion function;
- platform-specific fatal errors, such as out of memory errors.

Run-time execution errors of "warning" level are:

- integer underflow and overflow in expression evaluation;
- platform-specific warning messages.

The "errorstring" associated with the error event should contain a description of the error. The "errorstring" may provide information for an author to be able to locate the action language item the caused the problem, and the statement within the action language block in particular.

9.4.7 Expressions and conditions

9.4.7.1 Evaluation

Expressions are structured as a tree of sub-expressions. Expressions can be evaluated as part of the execution of an action language statement where required. Evaluation of an expression shall be complete once a value of a known run-time data type has been determined using the rules in this clause.

The leaf nodes of the tree and what they shall evaluate to shall be:

- literal values, which evaluate to a corresponding value of a run-time data type;
- array declaration expressions, which evaluate to the elements of an array of a specific run-time data type by the evaluation of each sub-expression within the array declaration expression;
- path references that do not includes indexes, representing the names of variables in the current scope, which evaluate to the value of the variable referred to within the current local scope;
- path references including indexes, representing an element of a named array within the current scope, which evaluate to the element of the array at the index given by evaluation the index expression;
- action language expression library function calls, which evaluate to the result of calling the named function, where the arguments to the function are determined by first evaluating all the expressions in the argument list.

EXAMPLE: The following action language shows four declaration statements corresponding to the four leaf node types listed above:

```
string stringLiteral = "Example text.";
integer[3] birthday = [4/2, 10+1, 1972];
integer[3] anotherBirthday = birthday;
integer birthYear = birthday[3];
date birthDate = integerToDate(birthday);
```

Conditions shall be defined by expressions that shall evaluate to the boolean run-time data type. All expressions that evaluate to the boolean run-time data type may not necessarily be conditions.

Non-leaf nodes of action language expressions shall be defined by arithmetic and logical expressions that are evaluated by performing an operation on the result of the evaluation of their non-leaf child nodes. These operations are defined in clauses 9.4.7.2 to 9.4.7.4

NOTE: The action language grammar is defined in an unambiguous way that shall determine operator precedence during the construction of expression trees.

9.4.7.2 Arithmetic operators

The operators listed below perform arithmetic operations. These operations are defined for the integer run-time data type only. An expression with an arithmetic operator with children that evaluate to a data type other than integer shall result in a transcode-time error.

The following arithmetic operators are defined:

- **add** (+) - Evaluation of the add operator shall result in an integer value that is the sum of two integer-valued child expressions.
- **subtract** (-) - Evaluation of the subtract operator shall result in an integer value that is the integer-valued right-child of the operator node subtracted from the integer-valued left-child.
- **multiply** (*) - Evaluation of the multiply operator shall result in an integer value the multiplicative product of the two integer-valued child expressions.
- **divide** (/) - Evaluation of the divide operator shall result in an integer value quotient from the division of the integer-valued left-child by the integer-valued right side.
- **modulus** (%) - Evaluation of the modulus operator shall result in an integer value remainder from the division of the integer-valued left-child by the integer-valued right side.
- **unary minus** (-) - Evaluation of the unary minus operator shall result in an integer value that has the opposite sign to that of the integer-valued child.
- **increment (**++**) -** Evaluation of the increment operator shall result in an integer value that is one greater than the integer-valued child.
- **decrement (**--**) -** Evaluation of the decrement operator shall result in an integer value that is one less than the integer-valued child.

9.4.7.3 Logical operators

The operators listed below perform logical operations. These operations are defined for the boolean run-time data type only. An expression with a logical operator with children that evaluate to a data type other than boolean shall result in a transcode-time error.

- **and** (&&) - Evaluation of the and operator shall result in a boolean value that is true if and only if both of the boolean-valued children are true.
- **or** (||) - Evaluation of the or operator shall result in a boolean value that is false if and only if both of the boolean-valued children are false.
- **not** (!) - Evaluation of the not operator shall result in a boolean value of true if and only if the boolean-valued child is false.

9.4.7.4 Relative operators

Relative operators compare the value of two items, to see if they are equal, not equal, lesser or greater than one another. The value that results from the evaluation of a relative operator is always of the boolean run-time type.

The following relative operators are defined for all run-time data types:

- **equals** (==) - Evaluation of the equals operator shall result in a boolean value that is true if and only if the left-child is the same run-time data type as the right-child and the value of both children is the same.
- **not equals** (!=) - Evaluation of the not equals operator shall result in a boolean value that is false if and only if the left-child is the same run-time data type as the right-child and the value of both children is the same.

NOTE: When values of the component run-time type are compared, they shall only be considered equal if they are the same component.

These following relative operations are defined for integer-valued children only. An expression with a relative operator with children that evaluate to a data type other than integer shall result in a transcode-time error. The following relative operators compare integer values:

- **less** (<) - Evaluation of the less operator shall result in a boolean value that is true if and only if the integer-valued left-child is less than the integer-valued right-child.
- **less or equal** (<=) - Evaluation of the less or equal operator shall result in a boolean value that is true if and only if the integer-valued left-child is less than or equal to the integer-valued right-child.
- **greater** (>) - Evaluation of the greater operator shall result in a boolean value that is true if and only if the integer-valued left-child is greater than the integer-valued right-child.
- **greater or equal** (>=) - Evaluation of the greater or equal operator shall result in a boolean value that is true if and only if the integer-valued left-child is greater than or equal to the integer-valued right-child.

9.4.8 System action library

The system actions of the PCF action language are defined in clause I.1.

9.4.9 Expression function library

The system actions of the PCF action language are defined in annex G.

9.5 Action language shortcuts

Alongside the PCF action language syntax are a number of "shortcuts" that provide an alternative way of describing certain commonly used, actions. These are described in annex J.

NOTE: Importantly, transcoders that do not implement the PCF action language rely on these shortcuts to describe the required behaviour.

9.6 Statemachines

A statemachine is a description of behaviour that encapsulates the events outlined in clause 9.2 and the action language described in clause 9.4. Statemachines are often associated with components but can be described independently for use within a PCF service.

The semantics and notation of statemachines used within PCF are closely related to those used by the OMG Unified Modelling Language (UML) which also uses statecharts to model behaviour.

9.6.1 Introduction

The sequences of states and actions that occur in response to discrete events can be expressed using a **statemachine**.

Statemachines can be modelled using a **statechart**, a well-documented and accepted graphical representation of behavioural intent.

Statecharts provide a logical way of describing the intended user-experience of behaviour that is complementary to the content, navigation, component and layout description of a PCF service. They describe the event-driven aspects of a system independently to its data flow or structural aspects.

The PCF behaviour model specifies a number of information items that form part of the architecture information set as described in clause 6.1.3.

The information items significant to the description of behaviour are represented in the UML object model in clause 9.6.1.3. The specification of each information item is given in clause 9.6.2 to clause 9.6.5. The description includes its graphical representation on a statechart. Where appropriate, the PCF XML representation of each information item is outlined.

9.6.1.1 State definition

A state is an identifiable period of calm during the lifetime of an object or an interaction during which it:

- satisfies some condition;
- performs some **action**; or
- waits for some **event**.

There may be many different states associated with a particular service.

A change of state may occur as the result of a particular event. This is indicated on a statechart by a transition arrow directed away from the currently active **source** state towards the **target** state. The transition includes details of the event, any **guard** conditions under which the transition should not take place and any action that must take place before the target state becomes active.

9.6.1.2 PCF state types

The UML model identifies fourteen different state types that can be used in combination with to form a statemachine. These are listed in the first column of table 28.

The PCF model is based on the UML description of statecharts diagrams. Only six of the UML state types are required to fully describe statemachines for interactive services. The PCF information items are given in the second column alongside the equivalent UML name. The third column indicates the clause in which each information item is defined.

Table 28: The UML states that are utilized by PCF.

UML name	PCF name	Described in clause:
initial state	initial state	9.6.4.1
final state	final state	9.6.4.2
deep history state	Not used	NA
shallow history state	history state	9.6.4.3
branch state	choice state	9.6.5.3
junction state	junction state	9.6.5.2
simple state	Not used	NA
composite state	state	9.6.5.1
synchronization state	Not used	NA
join state	Not used	NA
fork state	Not used	NA
submachine state	Not used	NA
stub state	Not used	NA

9.6.1.3 Object model

Figure 44 shows a UML diagram to represent the PCF object model for statemachines. Each UML class corresponds to a single information item.

Although the model is independent of the PCF representation, there is a direct correlation between the class names and the elements used in the PCF XML representation of the information set.

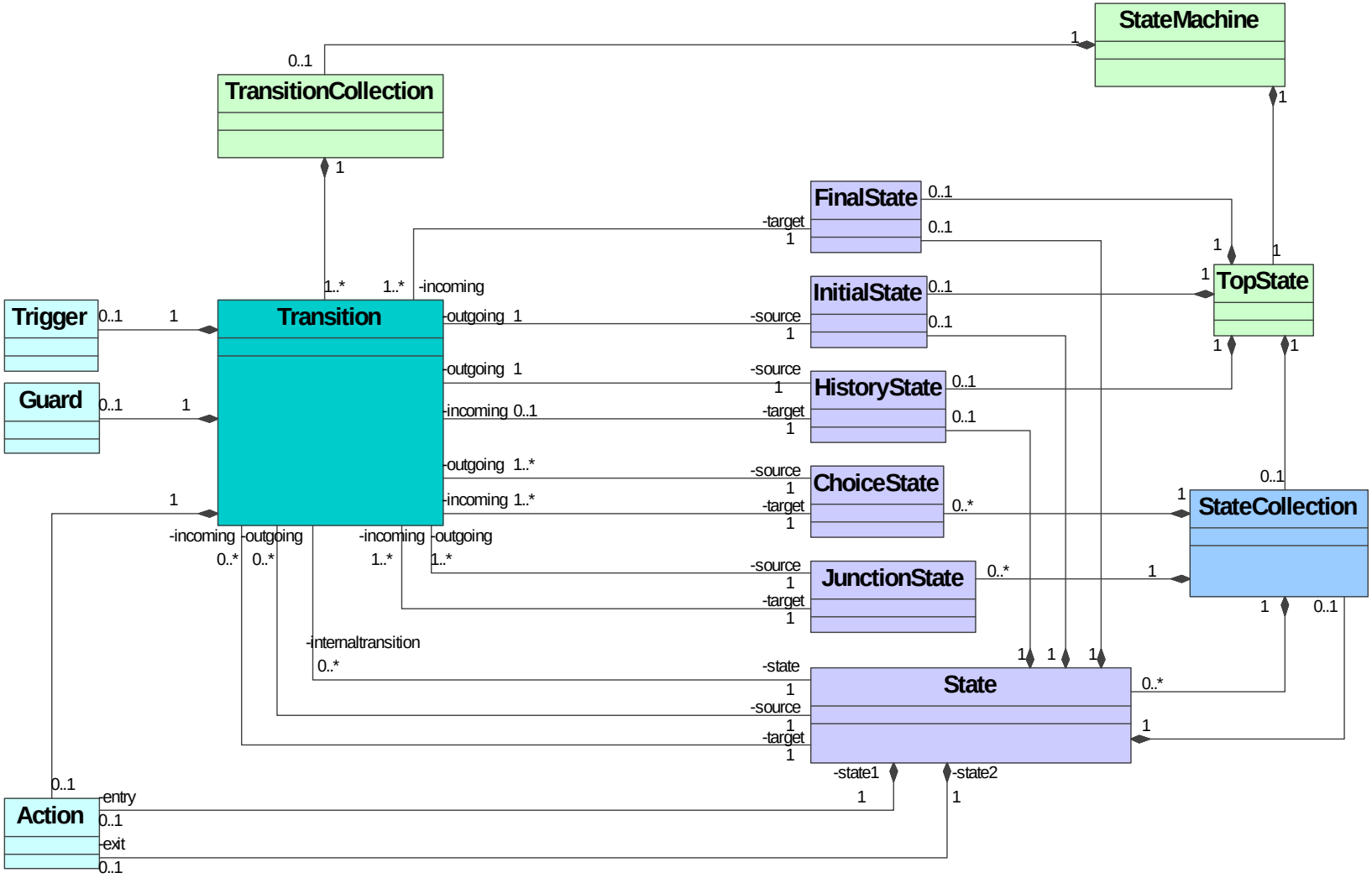

Figure 44: The PCF Statemachine object model

9.6.1.4 Transition and onevent object model

Figure 45 shows a UML diagram to represent the PCF object model for a state transition item and onEvent item. Both derive from the abstract class *TransitionBase*.

The TransitionBase has:

- at most one Trigger;
- at most one Guard;
- at most one piece of ActionLanguage.

Transition items are pieces of behaviour that cause a state change from the **source** state to the **target** state as described in clause 9.6.3.1. OnEvent items are descriptions of behaviour that do not cause a change in state. They may be used within internal transitions as described in clause 9.6.5.1.3 or may be used at any point in the PCF description as a "shortcut" - a piece of behaviour without the need for a statemachine.

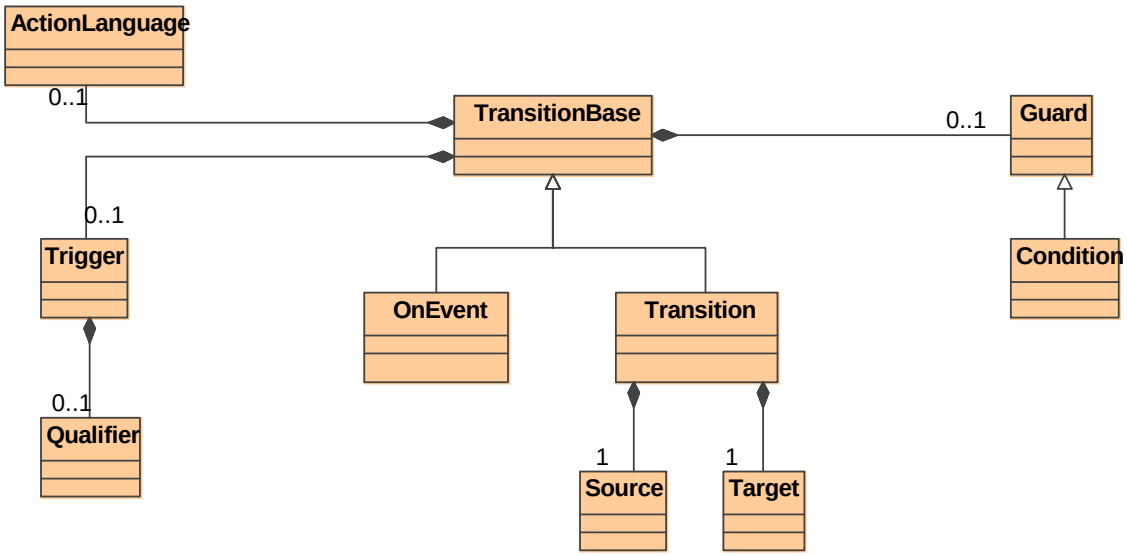

Figure 45: Transition and OnEvent object model

9.6.2 Statemachine

A statemachine item represents sequences of states and actions that occur in response to discrete events.

The properties of a statemachine item may include:

- a name that is unique within the enclosing parent item.

The description of a statemachine item shall contain, in no specific order:

- at most one state transition collection item;
- exactly one top state item.

A statechart is a graphical representation of a statemachine item.

EXAMPLE: A statemachine item is represented in the PCF XML format using an element with name StateMachine:

```
<StateMachine>
    <TopState>
        <!--state information -->
    </TopState>
    <TransitionCollection>
        <!--transition information -->
```

```
    </TransitionCollection>
</StateMachine>
```

9.6.3 Transition collection

The transition collection item contains a list of state transition items available to the enclosing statemachine.

The properties of a transition collection may include:

- a name that is unique within the enclosing statemachine item.

The description of a state transition collection item shall contain:

- at least one transition item.

EXAMPLE: PCF XML outline for a transition collection.

```
<TransitionCollection>
    <Transition ...>
    <Transition ...>
</TransitionCollection>
```

All transitions between different states shall be declared within the TransitionCollection.

9.6.3.1 Transition

A state transition item is a directed relationship between two state items; the source state and the target state. A state item shall be any of the PCF state types listed in table 28.

NOTE: A state transition item is distinct from a scene transition item as described in clause J.1.1.

The properties of a state transition item shall include:

- an optional name that is unique within the enclosing statemachine item;
- exactly one source state item;
- exactly one target state item.

The source state item and target state item shall identify the name of the state item within the enclosing statemachine to which they refer. The transition item forms the description of an outgoing transition item of the source state item and as an incoming transition item of the target state item.

The description of a state transition item shall contain, in no specific order:

- at most one trigger item;
- at most one guard item;
- at most one action item.

The guard item shall be evaluated before any action is executed. The action shall be executed only if the guard condition evaluates to true.

When a state transition item is represented on a statechart it shall take the form of an arrow as shown in figure 46. The arrow shall point away from the source state and towards the target state.

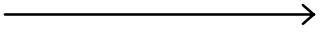

Figure 46: Statechart representation of a transition item

EXAMPLE: A state transition item is represented in the PCF XML format using an element with name Transition. The source state and target state are represented by the source and target attributes and they refer to the states named "state1" and "state2" respectively. This transition is the outgoing transition for the state with name "state1" and the incoming transition for the state with name "state2".

```
<Transition name="tran1" source="state1" target="state2">
    <Trigger ... />
    <Guard .../>
    <ActionLanguage .../>
</Transition>
```

Transitions whose source and target state are the same state may be declared within the TransitionCollection, but more commonly are declared as an InternalTransition within the state itself, as described in clause 9.6.5.1.3.

9.6.3.2 Trigger

The trigger item indicates the event that causes a state transition.

The properties of a trigger item shall include:

- an optional name that is unique within the enclosing parent item;
- an eventtype from an enumerated list of event types listed in the event schema.

The description of a trigger item shall contain, in no specific order:

- zero or more PCF items to qualify the value of the eventtype. The name attribute of these PCF items shall match a corresponding property of the eventtype that is being qualified.

EXAMPLE: A trigger item is represented in the PCF XML format using an element with the name Trigger. The example below describes a Trigger that filters KeyEvents, specifically the VK_LEFT forms the value for the "key" property of the KeyEvent event.

```
<Trigger name="trigger1" eventtype="KeyEvent">
    <UserKey name="key" value="VK_LEFT"/>
</Trigger>
```

9.6.3.3 Guard

The guard item contains a conditional expression. A transition with a guard shall take place only when this expression evaluates to true.

The description of a guard item shall contain:

- a Condition item from the PCF action language schema.

EXAMPLE: PCF XML outline for a guard.

```
<Guard>
    <Condition .../>
</Guard>
```

9.6.3.4 Action

The action item is a piece of PCF action language. The PCF XML format uses a piece of action language from the action language schema.

9.6.4 Top state

A top state item is the root of the state containment hierarchy.

The description of a top state item shall contain:

- exactly one initial state item;
- at most one final state item;
- at most one history state item;
- exactly one state collection item.

A top state item shall not be the source of any transitions and, therefore, not contain any outgoing transitions.

When a top state item is represented on a statechart it shall take the form of a rectangle with rounded corners as shown in figure 47.

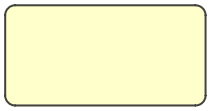

Figure 47: Statechart representation of a top state item

EXAMPLE 1: The statechart in figure 48 shows the top state item containing one initial state item, one final state item and two state items.

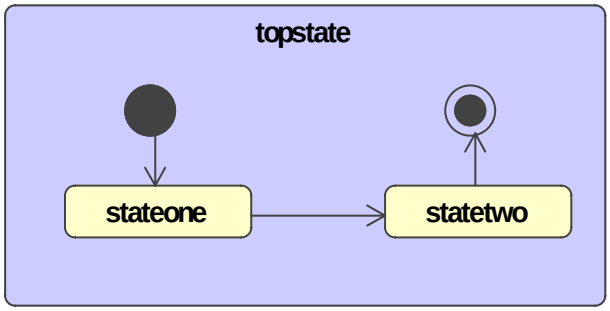

Figure 48: The top state item is drawn explicitly

EXAMPLE 2: The graphical representation of a top state item on a statechart is optional. The statechart shown in figure 49 is equivalent to that in figure 48 .

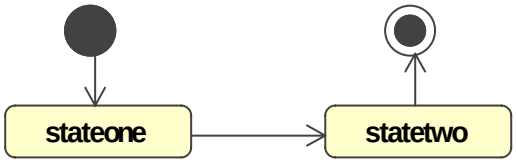

Figure 49: The top state item is implicit

EXAMPLE 3: A top state item is represented in the PCF XML format using an element with name TopState:

```
<TopState>
    <InitialState name="I_initial" .../>
    <FinalState name="F_initial" .../>
    <StateCollection name="otherstates" .../>
<TopState>
```

9.6.4.1 Initial state

An initial state item is the source state of single transition that indicates the default internal state of the enclosing state.

The properties of an initial state item may include:

- a name that is unique within the enclosing state item.

An initial state item shall be associated with:

- exactly one outgoing transition item where no trigger item is specified.

An initial state item shall not be the target of any state transitions and, therefore, shall not be associated with any incoming transition items.

NOTE: The outgoing transition item is not described within the initial state item, but is contained within the transition collection item, as described in clause 9.6.3.

When an initial state item is represented on a statechart it shall take the form of filled circle as shown in figure 50.

Figure 50: Statechart representation of an initial state item

EXAMPLE: An initial state item is represented in the PCF XML format using an element with name InitialState:

```
<InitialState name="initial"/>
```

A transition drawn to the enclosing state boundary is equivalent to the transition being drawn to the initial state in the enclosing state.

9.6.4.2 Final state

A final state item indicates that the enclosing state is completed. If the enclosing state is the top state (item) then this indicates that the entire statemachine is completed.

The properties of a final state item may include:

- a name that is unique within the enclosing state item.

A final state item shall be associated with:

- at least one incoming transition item.

A final state item shall not be the source of any transitions and, therefore, shall not be associated with any outgoing transitions.

NOTE 1: The incoming transition item is not described within the final state item, but is contained within the transition collection item, as described in clause 9.6.3.

When a final state item is represented on a statechart it shall take the form of an unfilled circle enclosing a smaller filled circle as shown in figure 51.

Figure 51: Statechart representation of a final state item

EXAMPLE: A final state item is represented in the PCF XML format using an element with name FinalState. This example gives the outline PCF XML for a final state item with two incoming transition items.

```
<FinalState name="final_2"/>
```

NOTE 2: A state that does not contain a final state item will exit when the scene exits.

9.6.4.3 History state

A history state item can be used instead of, or in conjunction with the initial state item to indicate "enter-by-history". It is the source state of single transition item that indicates the default internal state item of the enclosing state item. Specifically, it enables the default state item to become the most recently visited child state.

NOTE 1: This behaviour is based on the UML **shallow history pseudostate**. The properties of the UML deep history pseudostate can be mimicked using nested history state items.

The properties of a history state item may include:

- a name that is unique within the enclosing state item.

A history state:

- shall be associated with exactly one outgoing transition item where no trigger is specified;
- should be associated with at least one incoming transition item.

NOTE 2: The transition items are not described within the history state item, but are contained within the transition collection item, as described in clause 9.6.3.

When a history state item is represented on a statechart it shall take the form of an open circle enclosing the letter **H** as shown in figure 52.

Figure 52: Statechart representation of a history state item

The first time the enclosing state is visited the child state that becomes active shall be the "default state".

If the transition terminates on the history state then the default state is indicated by the outgoing transition from the history state. A transition that terminates on the edge of the enclosing state is equivalent to a transition terminating on the initial state. When entering the enclosing state through the initial state then the default state is indicated by the outgoing transition from the initial state.

Subsequent visits to the enclosing state through the history state will cause the most recently visited child state to become the active state. If the most recently visited child state was the final state then the active child state shall be the default state.

Subsequent visits to the enclosing state through the initial state shall cause the default state indicated by the initial state to become the active state.

If no initial state exists then all transitions to the enclosing state shall terminate on the history state.

EXAMPLE 1: A history state item is represented in the PCF XML format using an element with name HistoryState. This example shows the PCF XML outline for the history state item drawn in figure 53.

```
<HistoryState name="example1"/>
```

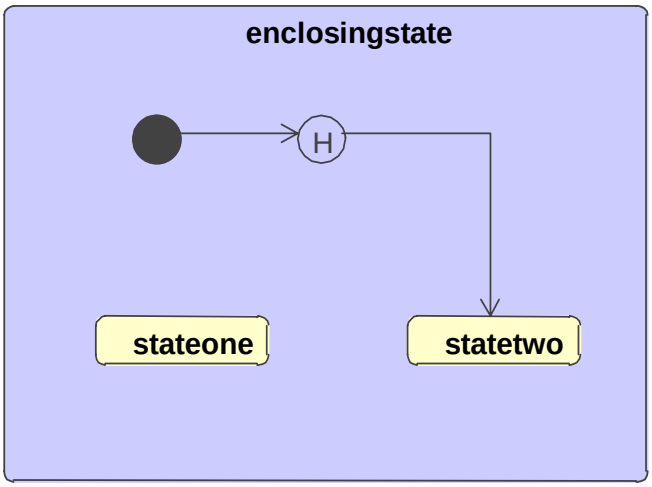

Figure 53: All visits to enclosingstate exploit the behaviour of the history state

EXAMPLE 2: When a history state exists without an accompanying initial state the enclosing state can only be activated through transitions directed to its history state.

The history state in figure 54 is not accompanied by an initial state. The transitions 1 and 2 are directed to the history state. The outgoing transition from the history state, transition 3, indicates that "stateone" is the default child state of "enclosingstate". This shall be the active child state the first time "enclosingstate" is visited. Subsequent visits to "enclosingstate" result in the most recently visited child state becoming active.

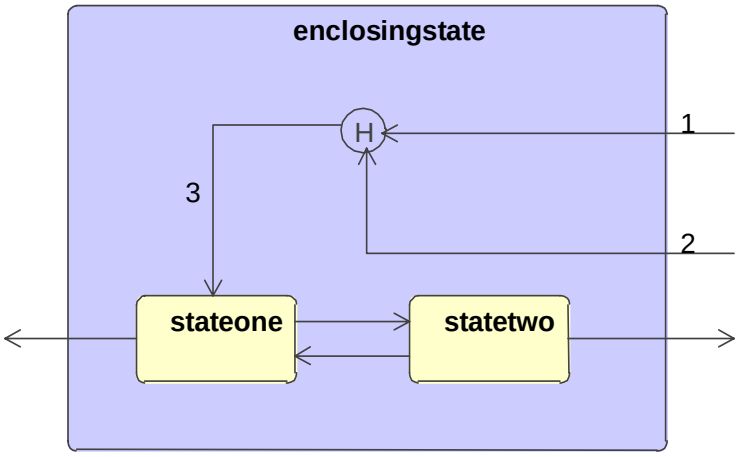

Figure 54: History state may be used without an accompanying initial state

EXAMPLE 3: Figure 55 shows a history state in conjunction with an initial state.

If the first visit to "enclosingstate" is via transition 1 then the active child state is determined by the outgoing transition from the initial state, transition 4. The active child state is "stateone". If the first visit is via transition 2 or 3 then the active child state is determined by the outgoing transition from the history state, transition 5. The active child state is "statetwo".

Subsequent visits to "enclosingstate" via transition 1 to the initial state shall result in "stateone" becoming active.

Subsequent visits via transition 2 or 3 to the history state shall result in the most recently visited child state becoming active.

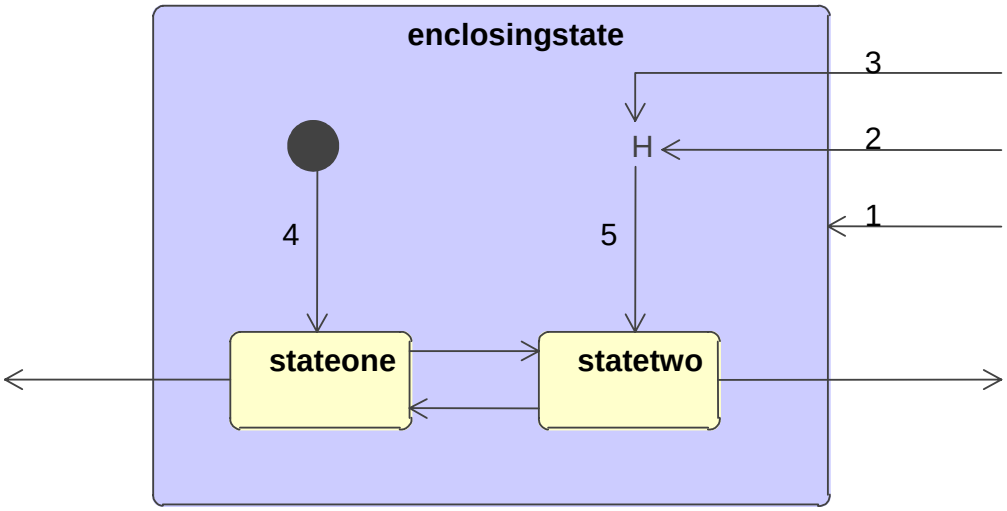

Figure 55: Both the history and initial state shall have an outgoing transition

9.6.5 State collection

A state collection item is used to group states of which there can be more than one instance within the enclosing state. These include state items, junction state items, and choice items but exclude initial state items, final state items and history items.

The properties of a state collection item may include:

- a name that is unique within the enclosing state item.

The description of a state collection item shall contain, in no specific order:

- zero or many state items;
- zero or many junction state items;
- zero or many choice state items.

A state collection item has no graphical representation on a statechart.

EXAMPLE: A state collection item is represented in the PCF XML format using an element with name StateCollection. This example shows the PCF XML outline of a state collection item containing two state items, an initial state item and a final state item.

```
<StateCollection>
        <State name="stateA" ... />
        <State name="stateB" ... />
        <JunctionState name="junctionA" .../>
        <ChoiceState name="choiceA" .../>
        <ChoiceState name="choiceB" .../>
</StateCollection>
```

9.6.5.1 State

The properties of a state item may include:

- a name that is unique within the enclosing state item;

The description of a state shall contain, in no specific order:

- at most one state entry action item;
- at most one state exit action item;
- zero or more internal state transition items;
- at most one initial state item;
- at most one final state item;
- at most one history state item;
- at most one state collection item.

A state item shall be associated with:

- zero or more incoming transition items;
- zero or more outgoing transition items, each with a trigger item specified.

NOTE: The transition items are not described within the state item, but are contained within the transition collection item, as described in clause 9.6.3. A state's outgoing transition items can be identified as those transition items whose source state item has the same value as the name of the state. A state's incoming transition items can be identified as those transition items whose target state item has the same value as the name of the state.

When a state item is represented on a statechart it shall take the form of a rectangle with rounded corners as shown in figure 56.

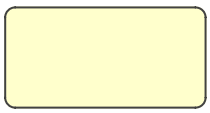

Figure 56: Statechart representation of a state item

EXAMPLE: A state item is represented in the PCF XML format using an element with name State. This example shows the PCF XML outline of the state item "stateone" shown in figure 57. The transitions to and from stateone, and those between the child states of stateone are described within the transition collection element.

```
<State name="stateone">
    <StateEntry>
        <ActionLanguage> <!-- dothis --> </ActionLanguage>
    </StateEntry>
    <StateExit>
        <ActionLanguage> <!-- dotheother --> </ActionLanguage>
    </StateExit>
    <InternalTransitions>
        <OnEvent>
            <Trigger eventtype="KeyEvent"></Trigger>
            <ActionLanguage> <!-- something --> </ActionLanguage>
        </OnEvent>
    </InternalTransitions>
    <InitialState name="stateone_i"/>
    <FinalState name="stateone_f"/>
    <StateCollection>
        <State name="stateA"/>
        <State name="stateB"/>
    </StateCollection>
</State>
```

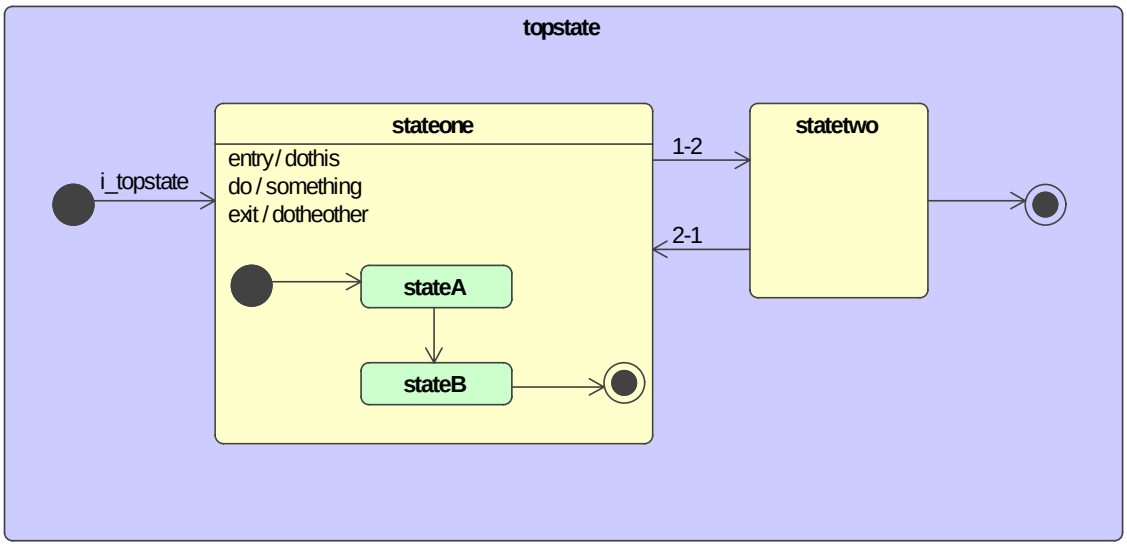

Figure 57: The description for "stateone" includes transitions and child states

9.6.5.1.1 State entry

A state entry item contains a piece of action language that shall be activated upon entry to the enclosing state.

The description of a state entry item:

- shall contain exactly one action item.

NOTE: There is no guard item associated with the state entry item.

EXAMPLE 1: A state entry item is represented in the PCF XML format using an element with name StateEntry.

```
<StateEntry>
    <ActionLanguage ... />
</StateEntry>
```

Actions associated with the state entry item shall take place when the enclosing state becomes active, and before any actions associated with internal states.

EXAMPLE 2: In figure 58, the entry actions associated with stateone take place before those associated with the initially activated child state statetwo.

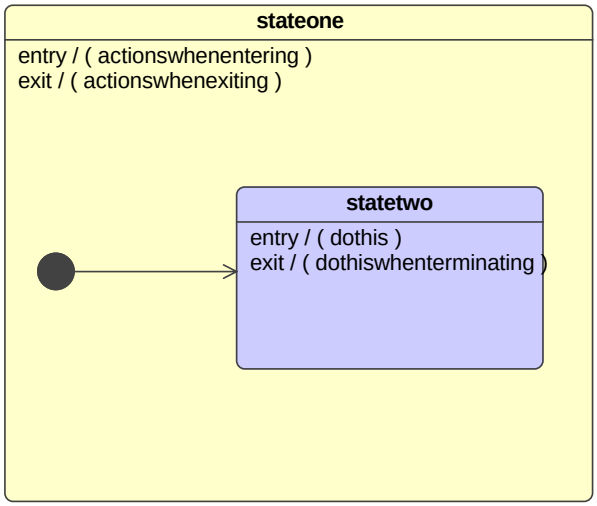

Figure 58: Nested states with state entry and state exit

9.6.5.1.2 State exit

A state exit item contains a piece of action language that shall be activated upon exit from the enclosing state. There is no guard item associated with the state exit item.

The description of a state exit item:

- shall contain exactly one action item.

NOTE: There is no guard item associated with the state exit item.

EXAMPLE 1: A state exit item is represented in the PCF XML format using an element with name StateEntry.

```
<StateExit>
    <ActionLanguage ... />
</StateExit>
```

Actions associated with the state exit item shall take place when the enclosing state is being terminated, and after any actions associated with terminating internal states.

EXAMPLE 2: In figure 58, when stateone terminates, the exit actions associated with statetwo take place before those associated with stateone.

9.6.5.1.3 Internal transitions

An internal transition item does not cause a change of state.

The properties of an internal transition item may include:

- a name that is unique within the enclosing statemachine item.

The description of an internal transition item:

- shall contain a trigger item;
- may contain a guard item;
- may contain an action item.

EXAMPLE: The PCF XML representation of an internal transition takes the form of the OnEvent element. One or more OnEvent elements can be described within an InternalTransitions element.

```
<State name="thisstate">
    <InternalTransitions>
        <OnEvent name="pressselect" source="thisstate" target="thisstate">
            <Trigger eventtype="KeyEvent">
                <UserKey name="key" value="VK_ENTER"/>
            </Trigger>
            <ActionLanguage> <!—actions → </ActionLanguage>
        </OnEvent>
    </InternalTransitions>
</State>
```

Internal transition items should be described within the state item. This ensures that this characteristic of the state is retained if the state be reused by other parts of the service. However, internal transition items may be described within the transition collection item. A transition item whose source and target refer to the same state describes an internal transition for this state.

9.6.5.2 Junction state

A junction state item shall be used to join one or more incoming transitions to one or more outgoing transitions. Specifically, it shall be used when no action is associated with one or more of the incoming transitions.

The properties of a junction state item may include:

- a name that is unique within the enclosing state item.

A junction state item shall be associated with:

- at least one incoming transition item;
- at least one outgoing transition item.

The description of incoming transition items associated with a junction state item:

- may contain a trigger item;
- may contain a guard item;
- shall not contain an action item.

The description of outgoing state transition items associated with a junction state item:

- shall not contain a trigger item;
- may contain a guard item;
- may contain an action item.

NOTE 1: The transition items are not described within the junction state item, but are contained within the transition collection item, as described in clause 9.6.3.

When a junction state item is represented on a statechart it shall take the form of a filled circle as shown in figure 59.

Figure 59: Statechart representation of a junction state item

If none of the guard items evaluate to true then no state transition takes place.

If more than one guard item evaluates to true then one transition shall be selected arbitrarily.

The representation of a junction state item on a statechart is graphical shorthand for multiple simple state transitions that share certain guard conditions. The statechart in figure 60 uses a junction state item and is equivalent to the statechart in figure 61 that uses multiple simple transitions.

NOTE 2: A transcoder may choose to implement a junction state item as multiple single transitions for ease of processing.

EXAMPLE: A junction state item is represented in the PCF XML format using an element with name JunctionState. This example shows the PCF XML outline of the junction state item and the associated transition items shown in figure 60.

The guard condition [b<0] is evaluated. If true, the guards on the outgoing transitions are evaluated. The condition that evaluates to true determines the new state. Should [a=6] (no outgoing transition is configured to be true when [a=6]) then there is no transition from state one.

```
<JunctionState name="junction"/>
    ...
<TransitionCollection>
    <Transition name-"in1" source="stateone" target="junction">
        <Guard <!-- b<0 -->    />
        <Trigger .../>
        <!--no action for incoming transition-->
    </Transition>
    <Transition name="out1" source="junction" target="statetwo">
        <Guard <!--a>7 --> />
        <!—no trigger for outgoing transition -->
    </Transition>
    <Transition name="out2" source="junction" target="statethree">
        <Guard <!--a=5 --> />
        <!—no trigger for outgoing transition -->
    </Transition>
    <Transition name="out3" source="junction" target="statefour">
        <Guard <!-- a<5 --> />
        <!—no trigger for outgoing transition -->
    </Transition>
</TransitionCollection>
```

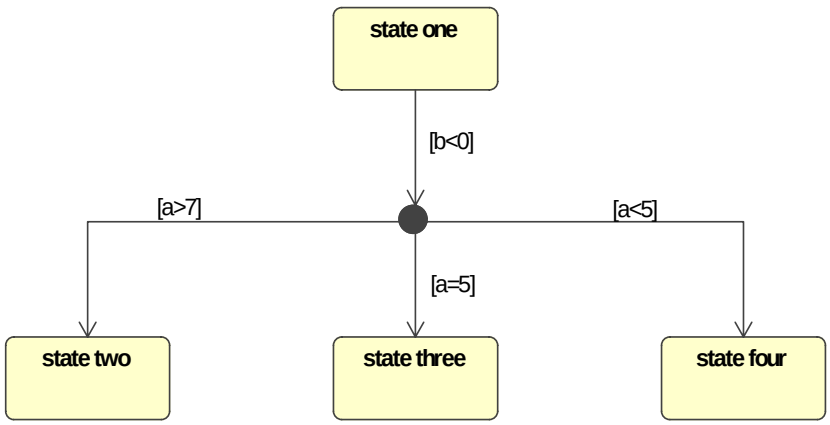

Figure 60: Junction state item used to simplify multiple guarded transitions

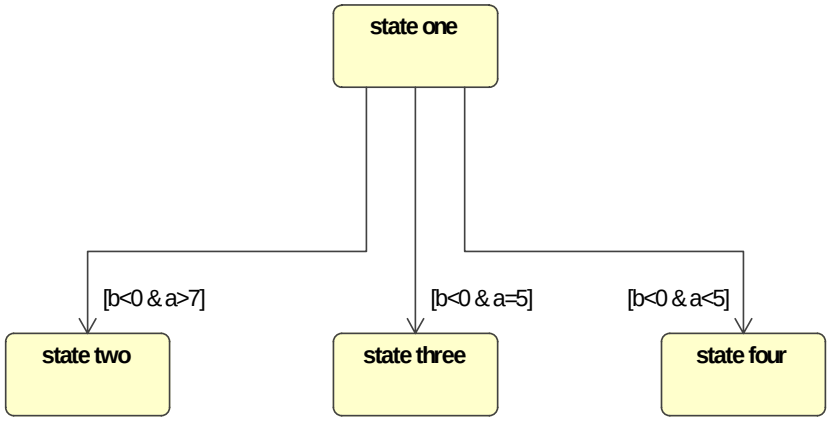

Figure 61: Multiple guarded transitions

9.6.5.3 Choice state

A choice state item shall be used to join one or more incoming transitions to one or more outgoing transitions. Specifically, it shall be used when an action is associated with one or more of the incoming transitions.

The properties of a choice state item may include:

- a name that is unique within the enclosing state item.

A choice state item shall be associated with:

- at least one incoming transition item;
- at least one outgoing transition item.

The description of incoming transition items associated with a choice state item:

- shall contain a trigger item;
- can contain a guard item;
- should contain an action item;

The description of outgoing transition items associated with a choice state item:

- shall not contain a trigger item;
- can contain a guard item;
- can contain an action item.

NOTE 1: The transition items are not described within the state item, but are contained within the transition collection item, as described in clause 9.6.3.

When a choice state item is represented on a statechart it shall take the form of an unfilled circle as shown in figure 62.

Figure 62: Statechart representation of a choice state item

The representation of a choice state item on a statechart is not graphical shorthand for multiple simple transitions. The statechart in figure 63 that uses a choice state item is not equivalent to the statechart in figure 64 that uses multiple simple transitions.

The actions associated with a simple transition are evaluated after all the associated guard conditions have been considered. The compound transition through a choice state item is the only way to express that an action takes place before a guard condition is considered.

The action associated with the incoming transition may affect the value of the condition to be tested on the outgoing transition. As a result, it is not possible to test the guard conditions associated with the outgoing transitions until the choice state has been reached. The transition cannot be abandoned at the choice state and so there should be exactly one outgoing transition whose guard condition evaluates to true for each valid value of the test variable. In the case where more than one of the outgoing guards is true then one shall be selected arbitrarily. There should be one outgoing transition labelled with the condition [else] to ensure all possible values are captured.

NOTE 2: A choice state item should be used only when an action is associated with one or more of the incoming transitions. Otherwise, a junction state item should be used to enable the transcoder to take advantage of the easier implementation.

EXAMPLE 1: A choice state item is represented in the PCF XML format using an element with name ChoiceState. This example shows the PCF XML outline of the choice state item shown in figure 63 that joins together a single incoming transition with three outgoing transitions. A guard is associated with each transition. An action is associated with the incoming transition.

The guard condition [b<0] is evaluated. If true, the action a:=f(m) is executed. Only after the action has been executed is the guard on the outgoing transition evaluated. The [else] condition ensures all possible values of "a" are considered.

```
<ChoiceState name="choice"/>
…
<TransitionCollection>
    <Transition name="in1" source="stateone" target="choice">
        <Guard ... <!-- b<0 -->    />
        <Trigger .../>
        <Action ...<!-- a:=f(m) --> />
    </Transition>
    <Transition name="out1" source="choice" target="statetwo">
        <Guard ... <!-- else --> />
    </Transition>
    <Transition name="out2" source="choice" target="statethree">
        <Guard ... <!-- a=5 --> />
    </Transition>
    <Transition name="out3" source="choice" target="statefour">
        <Guard ... <!-- a<5 --> />
    </Transition>
</TransitionCollection>
```

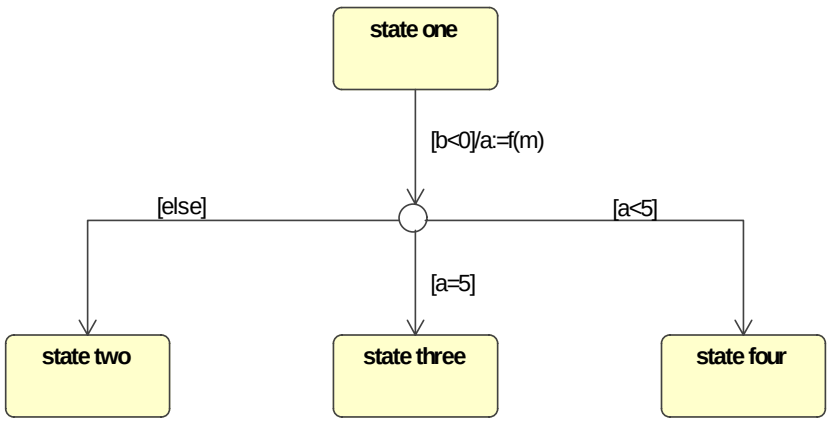

Figure 63: Choice state item used with action on incoming transition

EXAMPLE 2: The single transition from state one to state four in figure 64 is not equivalent to the transitions from state one to state four through the choice state in figure 63. Figure 64 demands "if [b<0], do a:=f(m), then test if [a<5]…" whereas figure 63 denotes "only if both [b<0] and [a<5] are true then do a:=f(m)". These expressions are not necessarily equivalent because the action a:=f(m) may influence the value of "a" thus affecting the test result of the second guard condition.

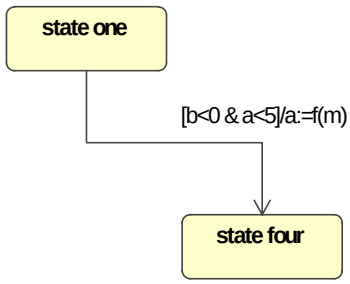

Figure 64: A single transition cannot replace the transitions through a choice state item

9.7 OnEvent - statemachine shortcut

Alongside the PCF statemachine model is a shortcut that provides an alternative way of describing behaviour that involves trigger, guard and action language items.

The onEvent item is synonymous with an internal transition, as described in clause 9.6.5.1.3, and is appropriate for simple behaviours where a change in state does not occur.

The following examples show how OnEvent can be used:

EXAMPLE 1: When "scene1" is active and a `VK_ENTER` keypress is received, the scene navigates to "next_scene".

```
<Scene name="scene1">
    <!--visual components etc -->
    <Button name="button1">
        <!--Button properties -->
    </Button>
    <Button name="button2">
        <!--Button properties -->
    </Button>
    <OnEvent name="pressselect">
        <Trigger eventtype="KeyEvent">
            <UserKey name="key" value="VK_ENTER"/>
        </Trigger>
        <SceneNavigate>
            <URI name="target" value="#../next_scene"/>
        </SceneNavigate>
    </OnEvent>
</Scene>
```

EXAMPLE 2: When "scene2" is active, the scene navigates to "next_scene" when either "button1" or "button2" is selected (the OnEvent declared at scene-level captures an OnSelect event generated by either Button component).

```
<Scene name="scene2">
    <!--visual components etc -->
    <Button name="button1">
        <!--Button properties -->
    </Button>
    <Button name="button2">
        <!--Button properties -->
    </Button>
    <OnEvent name="pressselect">
        <Trigger eventtype="OnSelect"/>
        <SceneNavigate>
            <URI name="target" value="#../next_scene"/>
        </SceneNavigate>
    </OnEvent>
</Scene>
```

EXAMPLE 3: When "scene3" is active, the scene navigates to "next_scene" only when "button1" is selected (the OnEvent being declared in the scope of "button1").

```
<Scene name="scene3">
     <!--visual components etc -->
     <Button name="button1">
         <!--Button properties -->
```

```
            <OnEvent name="pressselect">
                <Trigger eventtype="OnSelect"/>
                <SceneNavigate>
                    <URI name="target" value="#../next_scene"/>
                </SceneNavigate>
            </OnEvent>
        </Button>
        <Button name="I_do_nothing">
            <!--Button properties -->
        </Button>
    </Scene>
```

9.8 User-defined behaviour

9.8.1 Scope of user-defined behaviour

User-defined behaviour can be introduced at any level of the component hierarchy, either using statemachines or events.

User-defined behaviour shall:

- extend any intrinsic behaviour, not override it;
- obey the same scoping rules as PCF components; it may only manipulate child components within the scope of declaration.

EXAMPLE 1: In order to change the label text of a Button component to that of another Button component, the user-defined behaviour shall be described within a container component that contains both Button components.

EXAMPLE 2: In order to change the label text of a Button component, where the replacement text does not depend on any other component, then the user-defined behaviour may be defined within the scope of the Button component itself.

9.8.2 Event propagation involving user-defined behaviour

Event propagation takes place as described in clause 9.3.

For a component that has both intrinsic behaviour and user-defined behaviour, an event shall initially target the intrinsic behaviour. Only if the intrinsic behaviour does not respond shall the event target the user-defined behaviour.

Where more than one instance of user-defined behaviour responds to the same event, the order in which the behaviour is executed is undefined.

EXAMPLE 1: Figure 65 shows the intrinsic behaviour of a focused component respond to a user input event. Event propagation of the user input event terminates.

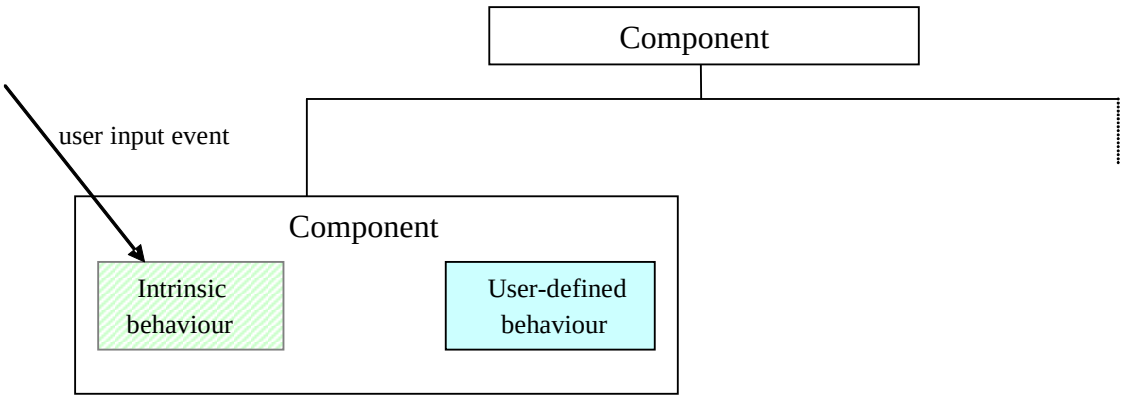

Figure 65: Event propagation terminates at intrinsic behaviour of focused component

EXAMPLE 2: Figure 66 shows a user input event initially target the intrinsic behaviour of a focused component. The intrinsic behaviour does not respond to this event and so the user input event propagates to the user-defined behaviour. The user-defined behaviour responds to the user input event which terminates. But the user-defined behaviour generates a component event which continues propagating up the component hierarchy. The component event propagation continues according to the rules in clause 9.3.4.

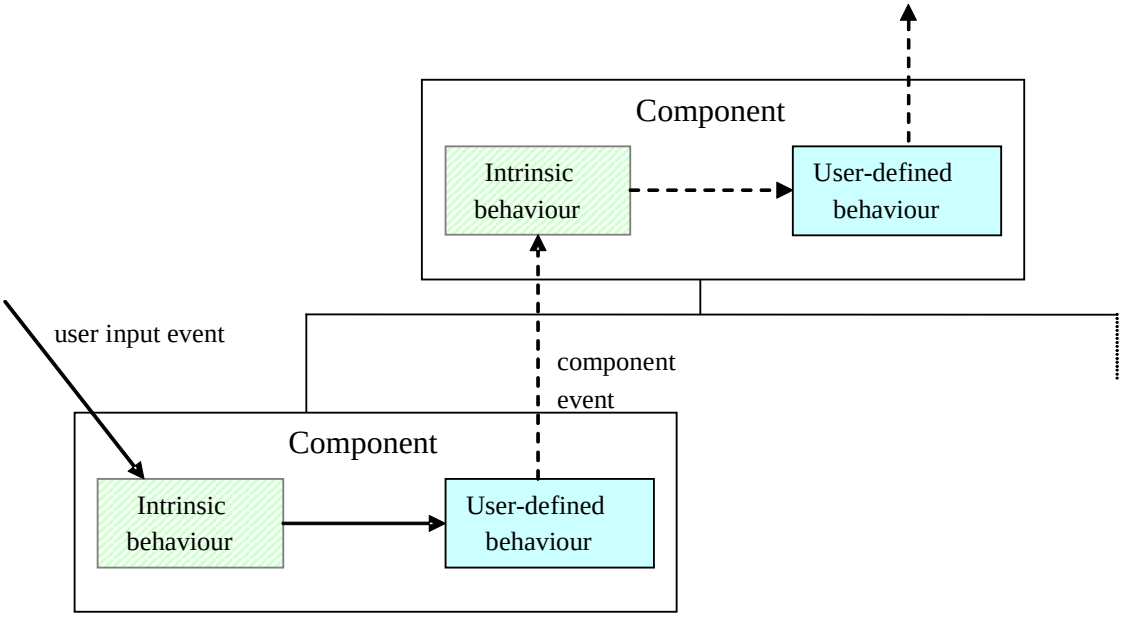

Figure 66: Event propagation with intrinsic behaviour in focused component

EXAMPLE 3: Figure 67 shows a focused component with no intrinsic behaviour defined. The user input event initially targets the user-defined behaviour of the focused component. This behaviour does not respond to this event and so it is passed up the component hierarchy. The parent component's intrinsic behaviour responds to the user input event and generates a component event which passes first to any user-defined behaviour in this component. Only if this behaviour does not respond does the component event propagate up the component hierarchy as described in clause 9.3.4.

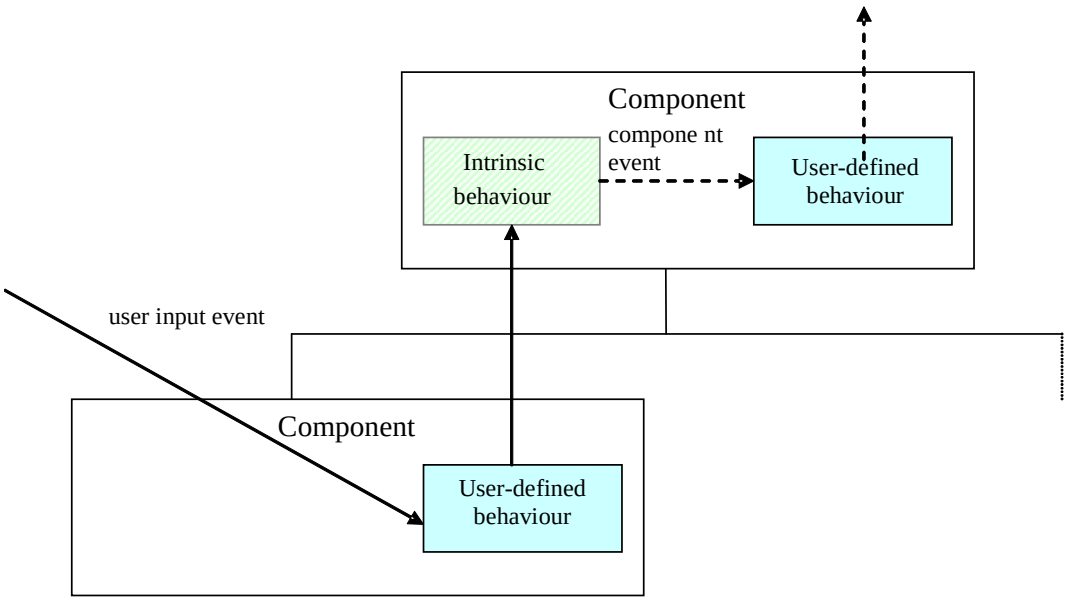

Figure 67: Event propagation with no intrinsic behaviour in focused component

10 Return path

10.1 Introduction

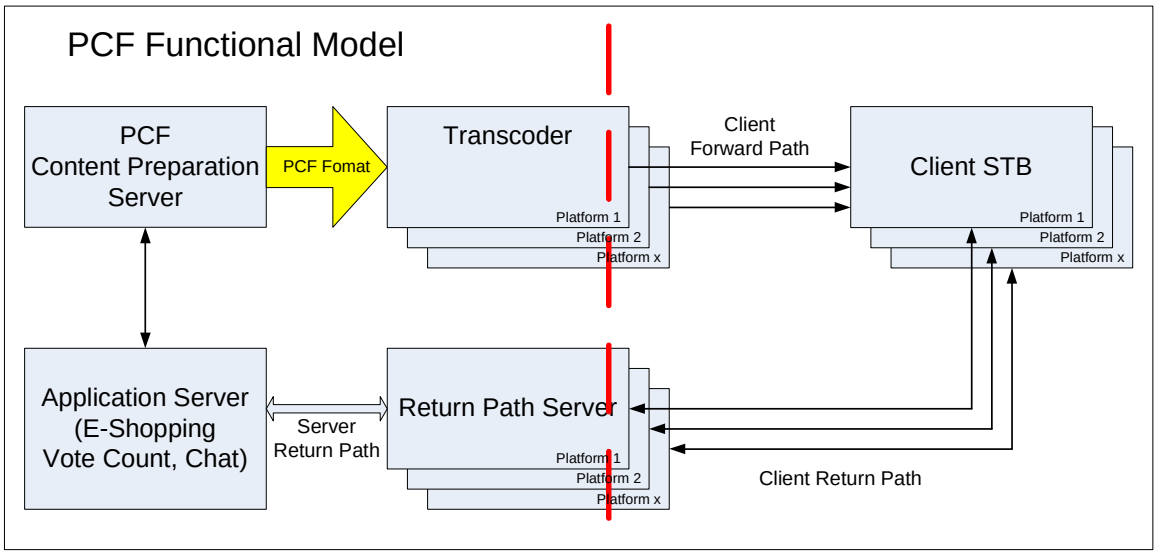

Figure 68: PCF High Level Functional Model / Logical Architecture

NOTE: The logical architecture depicted in figure 68 is an example logical architecture and is not mandated by the PCF specification.

The PCF return path mechanism provides a means to transfer data to and from the content provider under the author's control - "Client Return Path + Server Return Path" in figure 68. This should be distinguished from transfer of resources, such as images, referenced through a URI [ref 7 in Architecture Section 2.3.14] via the "Client Forward Path" in figure 68. Usage of the return path implies direct electronic communication without consumer intervention to allow the information exchange to take place, i.e. it does not include instructing a user to communicate via SMS. The return path mechanism enables the exchange of serializable component types (e.g. integer, string, date, array)

The standard return path implementation is built around three items:

- a **ReturnPath** component;
- a **Transaction** component;
- a **Transfer** collection.

A fourth component, the indicate component, is a cut down version of the **ReturnPath** component, whose purpose is to a enable very simple return path functionality whereby a connection is made but no transaction occurs, such as is used in a very simple voting application. A fifth component, the **SecureReturnPath** component allows for the secure transfer of data to take place.

10.2 Return path components

10.2.1 Returnpath component

The **ReturnPath** component embodies the return path itself and manages the actual data transfer exchange process. Its properties incorporate information to define the target application server, via its connectionTarget (URI) property and the current status of the return path connection (i.e. closed, open, opening, closing) via its state property. At run-time the **ReturnPath** component is passed a reference to a data **Transaction** component that contains the source and/or destination data transfer collections. The **ReturnPath** component statechart is shown in figure 69.

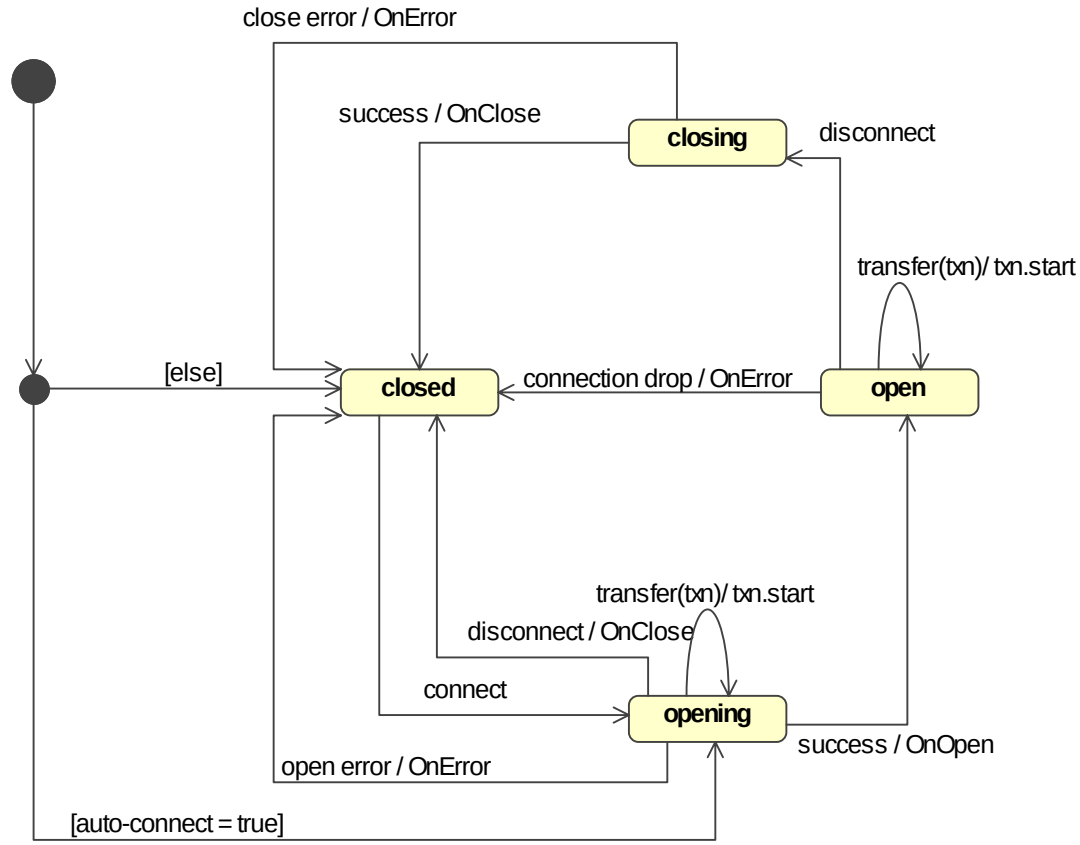

Figure 69: ReturnPath Component Statechart

The auto-connect property defines the start up behaviour of the **ReturnPath** component. Setting auto-connect to "true" requires that the **ReturnPath** component starts opening its connection as soon as it is in scope, otherwise it initiates into the closed state.

Calling the connect action shall cause the **ReturnPath** component to start opening its connection. A successful outcome will result in the return path becoming open and generating an OnOpen event.

Calling the disconnect action shall cause the **ReturnPath** component to start closing its connection. If the **ReturnPath** component is in its open state then it shall wait for all outstanding busy transactions to complete before becoming closed. If the **ReturnPath** component is in its opening state then it shall abort any busy transactions and return directly to the closed state. In either case the **ReturnPath** component shall generate an OnClose event.

It shall be possible to submit a transaction by calling the transfer action of the **ReturnPath** component when the return path is in either the opening or open states. Upon submission of the transfer action the associated transaction shall become busy. The data transfer associated with the transaction shall only be carried out after the return path has reached its open state.

The **ReturnPath** component may not always close in a well defined manner and can abort at any point. This is depicted by the "OnError" events in figure 69, the **ReturnPath** components statechart. "OnError" events always result in the return path being closed and any busy transactions failing.

EXAMPLE: A PCF code snippet might look like:

```
<ReturnPath name="rp">
    <URI name="connectionTarget" value="urn:dvb-pcf:www.pizza.tv:orderline"/>
    <Boolean name="autoConnect" value="true"/>
</ReturnPath>
```

NOTE: The ConnectionTarget value is an abstract URI that shall be resolved by a transcoder on a target platform. For example a dial up capable platform could resolve to a telephone number whereas a broadband platform could resolve to a URI.

10.2.2 Transfer collection

The data transfer collection defines a sequence of information to be transferred over the return path. Only serialized data can be transferred through the return path. Serialized data is a restricted set of data types, appropriate for transfer through the return path, for example integers, strings, variables, dates but NOT **Rectangle**s, **Button**s, **Menu**s etc. The PCF specification defines whether or not a component type is serializable. The transfer of graphics images through the return path is not required in the initial version of the PCF and therefore images are not defined as serializable.

EXAMPLE: A PCF code snippet might look like:

```
<TransferCollection name="request">
        < String name="pizzatype" value="pescatore"/>
        < Integer name="pizzasize" value="12"/>
        < Integer name="quantity" value="1"/>
</TransferCollection>
```

NOTE: The data Transfer Collection is generic enough to be used in other situations, such as data transfers to a storage device.

10.2.3 Transaction component

The **Transaction** Component embodies the status of an actual data transfer exchange process. (i.e. idle, busy) via its state property, and upon completion it shall generate an outcome event which defines the success (OnComplete) or failure (OnError) of the transaction. The transaction can be aborted under application control by using the "abort" action, which shall generate an OnAbort event. The **Transaction** component statechart is shown in figure 70.

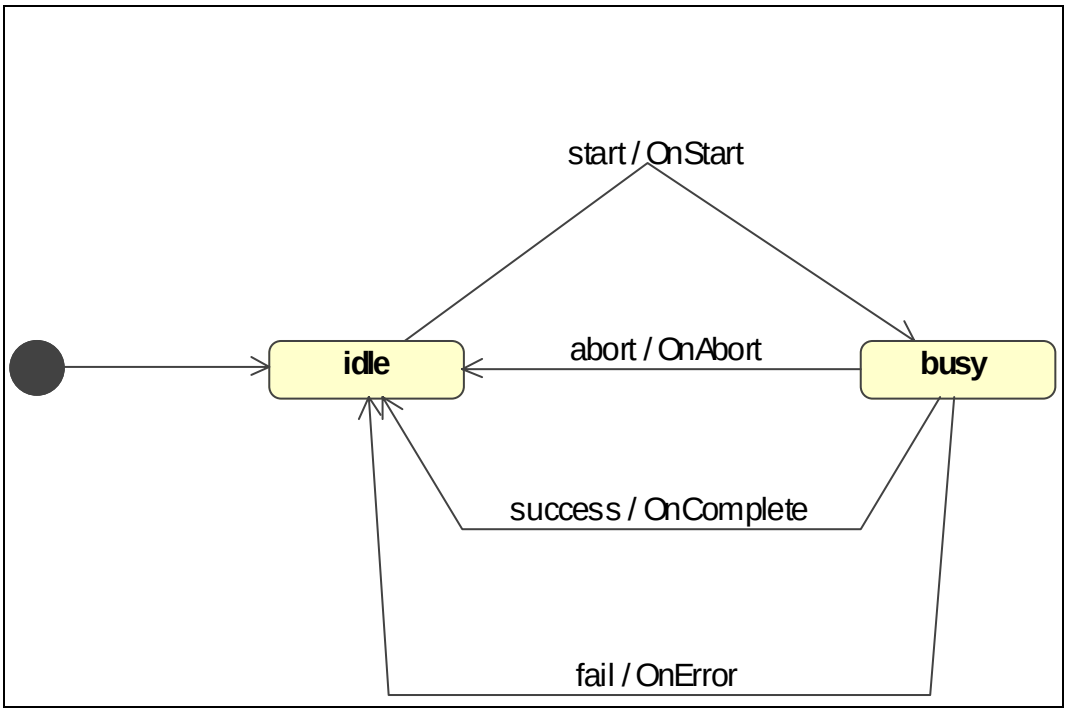

Figure 70: Transaction Component Statechart

The request (Client STB to Application Server) transfer collection specifies the order of components to serialize and send. The response (Application Server to Client STB) transfer collection specifies the order of serialized components to receive and interpret.

EXAMPLE: A sample PCF fragment might look like:

```
        <TextBox name="userFeedback">
            <String name="content" value="Please press the send button to order your
pizza!"/>
        </TextBox>
        <Transaction name="pizzaTransaction">
            <TransferCollection name="request">
                <StringVariable name="pizzatype" value="pescatore"/>
```

```
        <IntegerVariable name="pizzasize" value="12"/>
        <IntegerVariable name="quantity" value="1"/>
    </TransferCollection>
    <TransferCollection name="response">
        <StringVariable name="promoMessage" value=""/>
    </TransferCollection>
</Transaction>
<OnEvent name="completeEvent">
    <Trigger eventtype="OnComplete"/>
    <ActionLanguage><![CDATA[
        string temp[64] = strncat("Thank you. Your pizza is on its way!",
                          pizzaTransaction.response.promoMessage, 64);
        userFeedback.content = temp;
    ]]>
    </ActionLanguage>
</OnEvent>
<OnEvent name="errorEvent">
    <Trigger eventtype="OnError"/>
    <ActionLanguage><![CDATA[
        string temp[64] = strncat("Sorry. Your order has failed,
                          please telephone us instead: ", errorEvent.errorstring, 64);
        userFeedback.content = temp;
    ]]>
    </ActionLanguage>
</OnEvent>
```

10.2.4 Indicate component

The indicate component is an alternative to the **ReturnPath** component. The indicate component is not associated with a **Transaction** component and therefore it does not transfer any PCF defined data. Its purpose is purely to make a connection and then close.

EXAMPLE 1: Indicate components can be used to implement a simple voting service; one component for "yes" and another for "no". On a dial up modem platform the implementation may dial one telephone number for the "yes" and another for the "no." On a broadband always-on platform the implementation may call one URL for the "yes" and another URI for the "no". An application running on the return path server would perform the necessary counting function on behalf of the author.

The indicate component statechart is shown in figure 71.

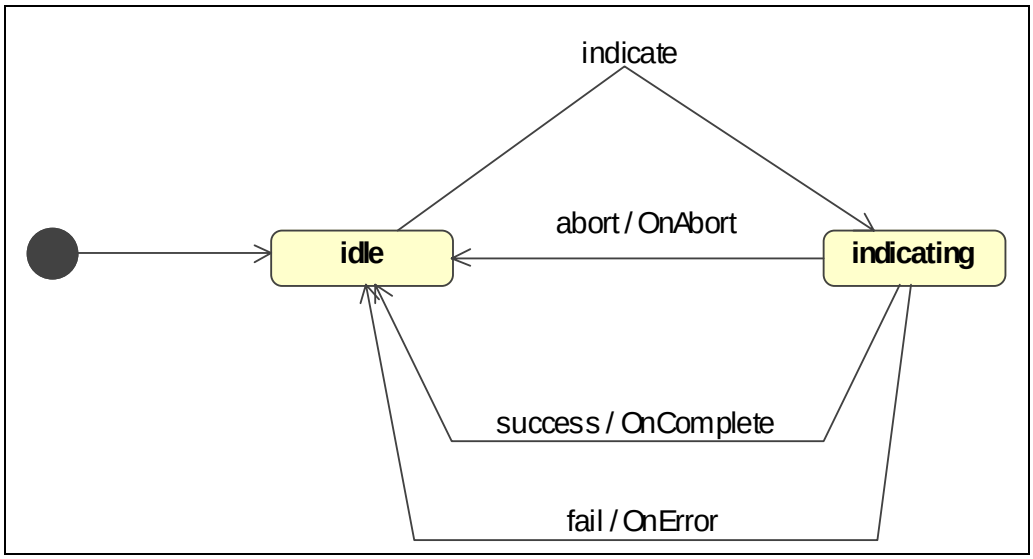

Figure 71: Indicate Component Statechart

An indication is initiated by invoking the indicate action of the indicate component from the action language. Upon completion it shall generate an outcome event which defines the success (OnComplete) or failure (OnError) of the indication. The indication can be aborted under application control by using the "abort" action, which shall generate an OnAbort event.

NOTE 1: The indicate component abstracts the implementation away from any particular return path technology. A platform with a dial up modem would probably implement a drop call to a specified telephone number whereas a platform with an always on broadband connection would probably connect to a specified URL. In each case the transcoder would derive the appropriate functionality.

NOTE 2: Due to potential resource contention, a platform with a dial up modem may disconnect any previously established return path connections, when the indicate action is invoked on an indicate component.

EXAMPLE 2: An Example PCF code snippet might look like :

```
<Indicate name="YesVote">
        <URI name="connectionTarget"
                value="urn:pcf:broadcasterX.co.uk/quizvote/yes"/>
</Indicate>
<Indicate name="NoVote">
        <URI name="connectionTarget"
                value="urn:pcf:broadcasterX.co.uk/quizvote/no"/>
</Indicate>
```

10.2.5 Secureretumpath component

The statechart for the **SecureReturnPath** component is identical to the **ReturnPath** component in clause 10.2.1.

The **SecureReturnPath** component is not considered open until a secure channel has been fully established.

If the **SecureReturnPath** component fails to establish the secure channel, an open error will follow.

10.3 Return path transfer process

Data transfers are initiated by invoking the transfer action of an open or opening **ReturnPath** component within the action language. The return path transfer action shall include a reference to a previously defined transaction.

The interaction between a **ReturnPath** component and the **Transaction** component is shown in figure 72.

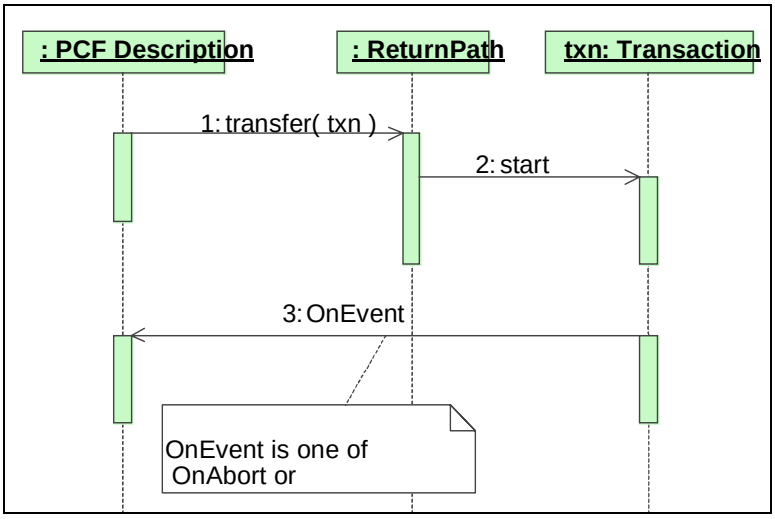

Figure 72: Return Path Sequence of Events

EXAMPLE: An example PCF code fragment might look like:

```
<Button name="sendOrder"/>
<OnEvent name="sendAction">
    <Trigger eventtype="OnSelect"/>
    <ActionLanguage><![CDATA[
        rp.transfer(<uri>pizzaTransaction</uri>);
    ]]></ActionLanguage>
</OnEvent>
```

10.4 Return path object model

ReturnPath, Transfer Collection and **Transaction** components and their associated data objects may be defined at the service level and/or at the scene level. Should a **Transaction** component go out of scope whilst it is busy it shall be aborted (for example closing a scene that incorporates **ReturnPath** components). Should a **ReturnPath** component go out of scope whilst it is not in a closed state then any associated busy transactions shall be aborted. The author can control against this happening, if required, by declaring the components at a higher level in the hierarchy, or by defining some explicit scene behaviour.

The transmission order of any pending transactions is not guaranteed. If this is important to the application author then he will manage this in his code, for example, by explicitly performing status / error message checks on pending / committed transactions.

A given **Service** or scene may declare more than one **Transaction** component e.g. one for reading and one for writing, or alternatively one for reading one type of data and another for reading a different type of data. A given platform may support multiple open return path connections, so a **Service** may contain more than one **ReturnPath** component at a time.

Each **Transaction** component shall contain 0 or 1 transfer collection named "request" that defines the outgoing data of the transaction (from the client's perspective).

Each **Transaction** component shall contain 0 or 1 transfer collection named "response" that defines the incoming data of the transaction (from the client's perspective).

The Domain model of the components related to the Return path is shown in figure 73.

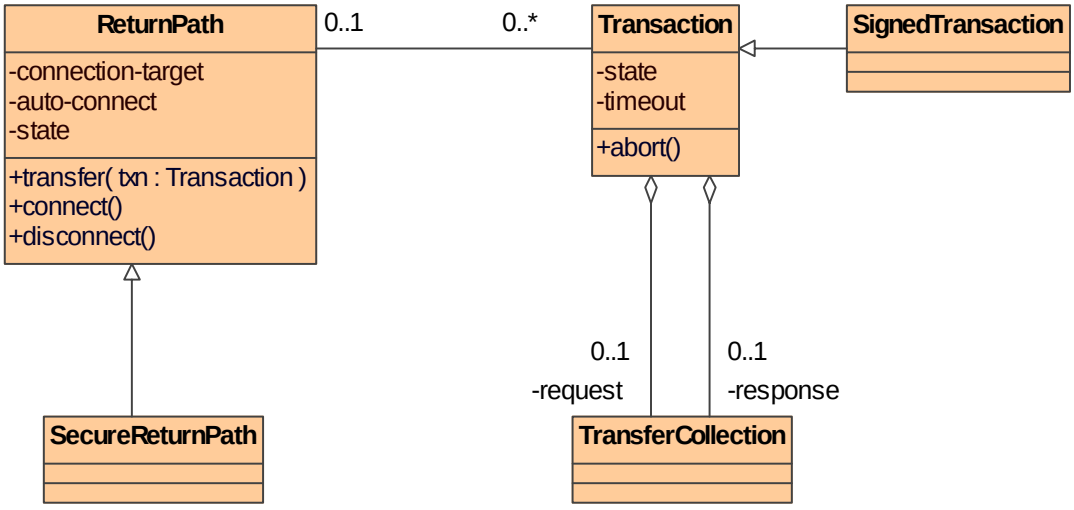

Figure 73: Return Path Components Domain Model

10.5 Security of return path data

10.5.1 Introduction

An author can request that security mechanisms occur within the platform. The specification of security algorithms is outside the scope of the PCF.

An author will be able to specify that any particular items of data that are exchanged across the return path may be a combination of signed or secure. The implementation of each of these methods may require the sharing of secrets, but how this is realized is also outside the scope of the PCF. However, an author can expect the following minimum behaviours.

10.5.2 Signed data

The purpose of signing is to ensure that there is a reasonable degree of assurance that the data being transferred has not been tampered with in transit.

EXAMPLE: An MD5 signature may be applied.

10.5.3 Secure data transfer

The purpose of encryption is to securely and privately transfer data through the Client Return Path (see figure 1 above). Physically the Return Path Server could reside with the application author's systems in order to achieve full end-to-end data security. The PCF does not specify any security or encryption protocol.

NOTE: An example encryption protocol might be SSL3.0, however which protocol and how secure that protocol may be is outside the scope of the PCF and is for the content author to determine its suitability independently for each target platform.

In order to realize this, the PCF defines the **SecureReturnPath** component which transfers data in an encrypted format.

Once declared within a PCF **Service**, the **ReturnPath** component and the **SecureReturnPath** component provide the same interface for transferring data.

10.6 Return Path Transaction Format (RPTF)

The PCF does not specify the format of data on the Client Return Path. The Return Path Transaction Format (RPTF) provides a standardized, but non mandatory, platform independent representation of the data exchanged on the Server Return Path.

NOTE: The RPTF does not specify a means by which data is transported or encapsulated.

The RPTF consists of RPTF transaction items that correspond to RP **Transaction** items as defined in the authored PCF **Service**. These **Transaction** items will comprise zero or more RPTF TransferCollections. Figure 74 below shows how the above Return path Object Model is extended to incorporate these RPTF elements.

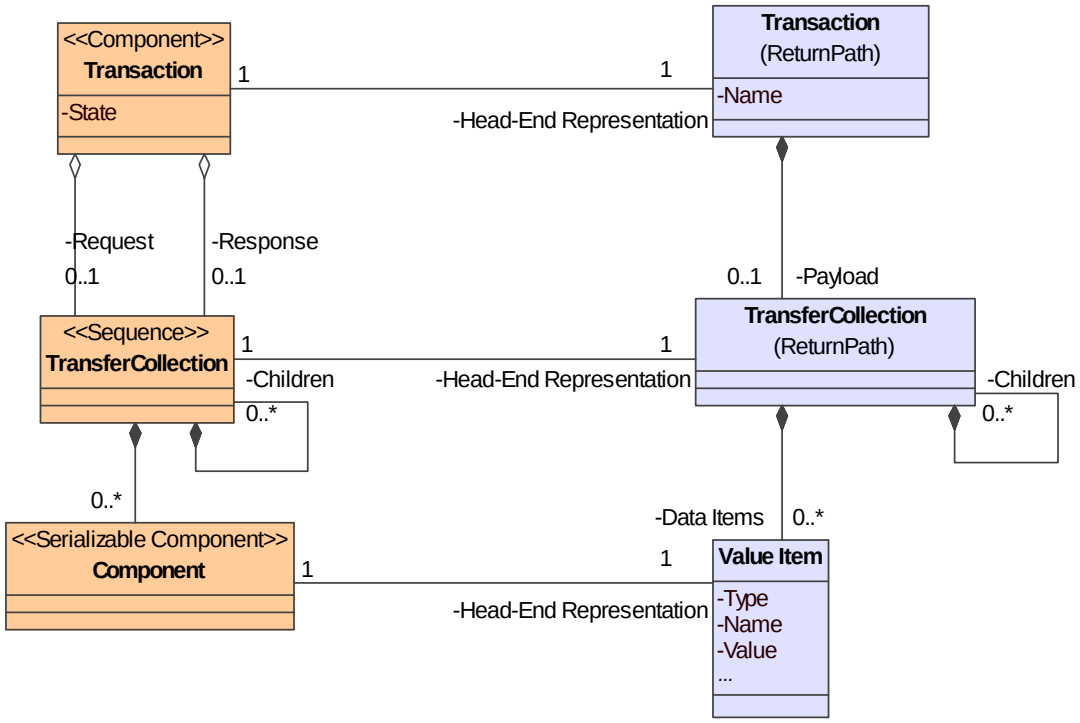

Figure 74: Return Path Transaction Format Mapping

Left = PCF Description Right = RPTF Representation

The RPTF model in figure 74 shall be serialized as an XML representation according to the schema defined below:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.dvb.org/pcf/rptf" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:pcf="http://www.dvb.org/pcf/pcf" elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:import namespace="http://www.dvb.org/pcf/pcf" schemaLocation="pcf.xsd"/>
    <xs:element name="Transaction">
        <xs:annotation>
            <xs:documentation>container for RPTF transaction</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:choice minOccurs="0" maxOccurs="unbounded">
                <xs:element ref="String"/>
                <xs:element ref="Integer"/>
                <xs:element ref="Boolean"/>
            </xs:choice>
            <xs:attribute name="name" type="xs:NCName" use="required"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

10.7 Connection usage display to viewer

Typical return path usage will involve communicating to the viewer platform specific information relating to the availability, cost, status and security of the return path connection.

The presentation of static information regarding the use of the return path connection, for example cost per minute, may be achieved using standard PCF visible components, such as **TextBox** and **Image**, where the **content** property is defined as a well known **ExternalBody**. The referenced external body can then be provided on a platform specific basis as appropriate.

The presentation of dynamic information regarding the status of the return path connection may be achieved using the **ConnectStatusImage** component (see clause A.2.4).

11 Profiles

11.1 Introduction

The purpose of profiles is to provide an easy way to describe the minimum capabilities, i.e. the *minimum profile*, that a platform must provide in order to achieve the authorial intent captured in the PCF description of an interactive service.

This is so that an author can have confidence that their service description will work on a particular target platform, or more precisely a particular target device, without having to meticulously match up every feature they have used with the features provided by that target.

As not all platforms will be able to support the full range of PCF features the present document defines a number of standard (DVB) profiles (see annex D). These DVB profiles have been defined to reflect pragmatic sub-sets of the current version of the PCF given both the fundamental ability for platforms to support specific PCF features and the ease of implementing a supporting PCF transcoder. It is possible that in the future additional DVB profiles may be defined in response to feedback from practical experience of using the PCF and/or further evolution of the PCF specification itself.

Individual platforms or content providers may independently create and publish proprietary profiles that may be of use in optimizing a PCF description of a service. This activity is outside of the scope of the DVB.

To simplify the interchange of profile definitions (DVB or proprietary) between parties a standard representation may be used.

Platforms that claim to conform to a particular profile (DVB or proprietary) shall provide at least the minimum capabilities identified for the profile. In some cases this implies that specific hardware resources are present in the platform.

A standard mechanism for associating a profile with a PCF service description is provided.

11.2 Profile definition

A PCF profile shall be defined by reference to one or more *profile packages*. Each profile package embodies a set of related PCF features, e.g. behaviour. The definition of a profile package shall be described by reference to the normative clauses within a specific version of the PCF specification that describe the relevant PCF features. Each profile package shall be assigned a unique identifier formed according to the URI specification.

The PCF features within a profile package shall be grouped into one or more *levels*. Each new level shall consist of a superset of PCF features with respect to the previous level and shall be identified by an incrementing integer count, starting with 1.

NOTE 1: The purpose of levels is to reflect both the inherent boundaries in platform capabilities (e.g. availability or not of **SecureReturnPath**) and significant steps in the complexity of PCF transcoder implementation.

Profile packages should be defined so as to achieve the coarsest possible sub-setting of the PCF specification. Furthermore, the number of levels within a profile package should be minimized wherever possible. A standard (DVB) set of profile package definitions is specified in annex D.

NOTE 2: It is only by minimizing the number of profile packages and levels that the profiling mechanism is able to meet its stated purpose, i.e. to provide an easy way to describe the minimum capabilities that a platform must provide.

So in fact a profile is defined by referencing one or more profile packages at a particular level. Within a particular profile each profile package may be at a different level.

A standard syntax for the declaration, and so interchange, of profile definitions is specified in annex D. This shall be used to declare a profile using one of two possible methods:

- A profile may be declared as an explicit list of profile packages at a particular level.

EXAMPLE 1:

```
<profileDef name="mycom.com/myprofile">
    <package id="dvb.org/pcf/package/architecture" level="1"/>
    <package id="dvb.org/pcf/package/behaviour" level="1"/>
    <package id="dvb.org/pcf/package/layout/explicit" level="2"/>
</profileDef>
```

- Alternatively a profile may be declared as an extension to another existing profile.

EXAMPLE 2:

```
<profileDef name="mycom.com/mybetterprofile">
  <baseProfile id="mycom.com/myprofile"/>
  <package id="dvb.org/pcf/package/returnpath" level="1"/>
</profileDef>
```

If the declaration of a profile, by either method, includes more than one instance of a particular profile package once all profile instances have been expanded, then all instances of the profile package other than the one with the greatest value of level are redundant and may be ignored.

EXAMPLE 3: The following fragment.

```
<profileDef name="mycom.com/mybetterprofile">
  <BaseProfile id="mycom.com/myprofile"/>
  <package id="dvb.org/pcf/package/architecture" level="2"/>
</profileDef>
```

Expands to:

```
<profileDef name="mycom.com/mybetterprofile">
  <package id="dvb.org/pcf/package/architecture" level="1"/>
```

```
  <package id="dvb.org/pcf/package/behaviour" level="1"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="2"/>
  <package id="dvb.org/pcf/package/architecture" level="2"/>
</profileDef>
```

Which is equivalent to:

```
<profileDef name="mycom.com/mybetterprofile">
  <package id="dvb.org/pcf/package/behaviour" level="1"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="2"/>
  <package id="dvb.org/pcf/package/architecture" level="2"/>
</profileDef>
```

11.3 Profile association

A PCF profile is associated with a PCF service description by its inclusion in a PCF service digest that is referenced from within the Service element of the service description itself. The mechanism for referencing a PCF service digest is described in clause A.1.1.1.

The PCF service digest may contain *profile aliases* defined using the profile Alias element. Each defined profile alias is uniquely identified within the scope of the service description by the name attribute of the profileAlias element. If a profile alias is defined with the name "minimum" this specifies the minimum capabilities that will be required of a platform to render the intended viewer experience. The use of profile aliases with any other name is not defined in the present document.

It is not mandatory for a PCF service description to have any associated profiles. This will occur if either no PCF service digest is referenced from the service description, or if the referenced digest includes no profile alias definitions.

NOTE: If no default profile is associated with a PCF Service then a PCF transcoder can only assume that any aspect of the PCF may have been used in the description of the service. How a PCF transcoder chooses to deal with this situation (i.e. reject the service description, attempt to process but issue a warning) is outside the scope of the present document.

12 Service digest

12.1 Introduction

The purpose of the PCF service digest is to provide summary information that may be interchanged independently of the actual PCF service description itself.

NOTE: This is expected to be useful during the initial stages of any business-to-business interchange.

12.2 Digest definition

The digest does not form part of the PCF service description itself. However, it may be referenced from a PCF service description as an external resource (see clause A.1.1.1).

The digest shall be encoded as an XML document according to clause G.1.6.

The root element of a digest, pcfServiceDigest, shall contain the following information items:

- A name attribute. This shall be used to provide a human-readable textual name for the service, e.g. "BBC Olympics 2008".
- A provider attribute. This shall be a registered Internet domain name, e.g. "bbc.co.uk". (See RFC 1591 [29] for DNS name registration.) The provider attribute is case insensitive and must be a fully qualified name according to the rules defined by RFC 1591 [29].
- A pcfSpecVersion attribute. This shall identify the version of the PCF specification with which the service description is conformant, i.e. it shall be the same as the value encoded within the Service element of the PCF service description itself.

A pcfServiceDigest may also contain the following information items:

- A version attribute. This shall be used to provide a version number for the PCF service description.
- A ServiceID child element. This shall be used to provide a unique identifier for the service according to some identification scheme.
- A Description child element. This may be used to provide a free-form textual description of the PCF service.
- One or more ServiceEntryPoint child elements. These identify alternative entry scenes for navigating to the service in addition to that defined by the "firstScene" property of the Service item itself.
- One or more ProfileAlias child elements. See clause 12.3.

12.3 Profile alias definition

The profileAlias child element within a pcfServiceDigest element shall be used to define a profile alias for the relevant PCF service description. A profileAlias element shall contain the following information item:

- A name attribute. Each defined profile alias shall be given a name that is unique within the scope of the service description. The name "minimum" shall be assigned to a profile alias that identifies the minimum capabilities that will be required of a platform to render the intended viewer experience. Other profile aliases may be assigned any name as convenient.

A profileAlias may also contain the following information items:

- One or more baseProfile child elements. Each instance shall identify a specific PCF profile (see clause 11.2).
- One or more package child elements. Each instance shall identify a specific PCF profile package.

A profileAlias shall contain at least a one baseProfile or package child element.

If the declaration of a profileAlias includes more than one instance of a particular profile package once all profile instances have been expanded, then all instances of the profile package other than the one with the greatest value of level are redundant and may be ignored. This is handled in the same way as for profile definitions (see clause 11.2).

So for the simplest case where a service is authored according to a single profile the association might look like the following:

```
<profileAlias name="minimum">
  <baseProfile id="dvb.org/pcf/profiles/core"/>
<profileAlias>
```

For the situation where the core of a service is authored according to one profile but part of the service is authored by a third party according to another profile the association might look like the following:

```
<profileAlias name="minimum">
  <baseProfile id="dvb.org/pcf/profiles/core"/>
  <baseProfile id="my.com/pcf/profiles/abitmore"/>
</profileAlias>
```

NOTE: This is really a convenience since at the two profiles could have been merged into a more optimized form in advance. However, it was agreed that the work required of a transcoder to manage this was minimal.

Another example is where a standard profile is extended with one (or more) packages, as follows:

```
<profileAlias name="whizzy">
  <baseProfile id="dvb.org/pcf/profiles/core"/>
  <package id="dvb.org/pcf/package/behaviour" level="3"/>
</profileAlias>
```

And of course it would be possible to simply define a profile alias only using packages, i.e. no base profile:

```
<profileAlias name="another">
    <package id="dvb.org/pcf/package/behaviour" level="3"/>
        … other package defs …
    <package id="dvb.org/pcf/package/explicitlayout" level="2"/>
</profileAlias>
```

12.4 Example PCF service digests

EXAMPLE 1: A simple service digest.

```
<pcfServiceDigest name="BBC Olympics 2008" provider="bbc.co.uk" pcfSpecVersion="1.0"/>
```

EXAMPLE 2: A service digest that provides both service entry points and profiling information.

```
<pcfServiceDigest name="BBC Olympics 2008" provider="bbc.co.uk" version=1.0a" pcfSpecVersion="1.0">
    <ServiceID type="">bbc.co.uk/olympics2008</ServiceID>
    <Description>No use of return path</Description>
    <ServiceEntryPoint>index_scene</ServiceEntryPoint>
    <ServiceEntryPoint>scoreboard_scene</ServiceEntryPoint>
    <ProfileAlias name="default">
        <BaseProfile id="dvb.org/pcf/profiles/core"/>
    </ProfileAlias>
</pcfServiceDigest>
```

13 Mechanism for transport and packaging (optional)

The PCF transport and packaging mechanism specifies a semantic model for transfer of PCF service descriptions between any source and sink entities. It is intended that this be used for delivery of PCF service descriptions to PCF-compliant transcoders. The description that follows is expressed in terms of these entities. This does not preclude its use for exchange between alternative entities.

This part of the specification is optional. A PCF-compliant system may choose to implement an alternative means for exchange of PCF service descriptions.

Should a PCF-compliant system adopt this part of the specification, all clauses contained herein are normative.

13.1 PCF data exchange model

13.1.1 Assets, transactions and acceptability

A **PCF service** shall be described in a collection of separate data units known as **PCF assets**. Each asset shall be identified by a URI that is unique within the scope of the service registration. PCF assets shall contain either PCF-compliant XML or other octet data in a supported content format. Subsets of this collection shall be delivered to the **transcoder** as a series of **transactions** over time. Update and remove operations shall be supported at the transcoder interface to insert, refresh or remove individual assets from the set currently loaded onto the transcoder.

In order to allow transactions to be processed in distinct contexts at a specific transcoder, each service shall be registered with that transcoder by the **service provider**. All transactions shall then take place within the context of this **registration**.

NOTE 1: This allows the same service to be registered one or more times with the same or separate transcoders.

NOTE 2: Transactions may be initiated by any party that has access to the service registration. This is allowed to be an entity other than the service provider.

A transaction message shall pass a set of PCF asset URIs into a service context on the transcoder. This shall declare this set of URIs to be part of the service context. This message may also contain the content of zero or more assets, where each asset shall be identified by a URI that is unique within the service registration context.

Transaction messages may be delivered synchronously with the transcoder, in which case all processing shall be completed before a response message is generated. Messages may be delivered asynchronously with the transcoder, in which case a response message shall be generated immediately after receipt of the message, and the status of the transaction may be monitored through subsequent message requests and responses.

The transcoder shall resolve all supplied URIs and acquire asset contents according to the update models described in clauses 13.1.2 to 13.1.4. All assets so acquired become the **loaded assets** of that service within that registration context.

The transcoder may then test the acceptability of the content using a combination of XML syntax validation , PCF semantic correctness, compliance with the PCF profile in use and content transcodability as bounded by the exactness declarations within the content

NOTE 3: The business rules applied to determine acceptability of content are considered to be outside the scope of the present document, for example a failure to achieve exactness for a particular value item may not invalidate the content i.e. it may still be accepted as part of a transfer. Instead the business rules shall be determined by the tool provider.

13.1.2 Push update model

If the URI can be resolved to another part of the transaction message, the content shall be retrieved from this part. If the content is not acceptable, the transaction shall fail.

13.1.3 Pull update model

If the URI cannot be resolved to another part of the transaction message and the URI is not a PCF indirect URN as defined in clause 6.2.3.14, the URI shall be used to retrieve the content from a remote resource. Retrieval shall happen once for each asset. If the content request fails, or the content is not acceptable the transaction shall fail.

13.1.4 Online update model

If the PCF asset has its **is_volatile** flag set to "true", retrieval of content shall occur once initially and may occur intermittently thereafter during the **lifetime** of that asset. Thus the asset should remain available at this URI throughout this time. The pattern of retrieval requests is platform-specific, and not defined by this specification. A portable hint can be specified to define the **validity period** of a **volatile asset**, and the platform shall not cache the asset for longer than this period. If the initial content request fails, or the content is not acceptable the transaction shall fail. If any subsequent retrieval of a volatile asset fails, or any instance of the content is not acceptable, the transcoder shall reject the asset and an error event shall be generated in the run-time environment of any device accessing this asset.

13.1.5 Asset lifetime

The lifetime of the assets loaded by each transaction is bounded by the lifetime of the service registration. When the service is unregistered, the service shall become unavailable and any remaining assets associated with that service registration may be discarded by the platform.

13.1.6 Service packaging and references

One or more PCF assets, known as **PCF source documents**, shall contain the PCF description of the service. Exactly one PCF source document shall be identified as the **master asset**, and shall be used by the transcoder as the starting point for examining the description hierarchy of the loaded assets of a PCF service.

EXAMPLE 1: Figure 75 illustrates a set of loaded assets in a registration context on a transcoder.

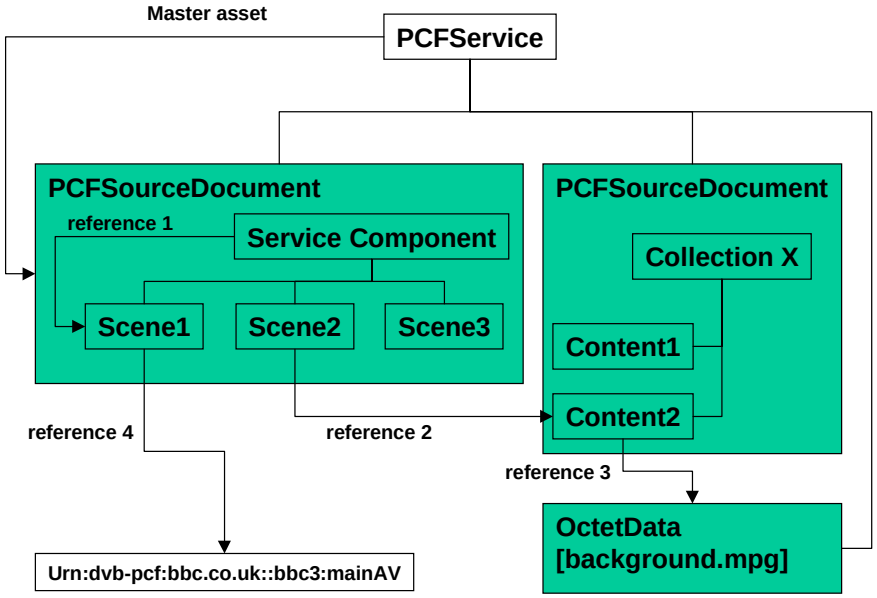

Figure 75: Simple PCF service example

PCF source documents may contain references to PCF entities that form part of the same service definition. Each entity shall be encapsulated by a PCF asset. The referenced entity may be within the same PCF asset (e.g. reference 1 in figure 75) or in a separate PCF asset (e.g. reference 2 in figure 75) or may be the entire asset (e.g. reference 3 in figure 75). All forms are known as **direct references**.

PCF source documents may contain abstract references to entities which are beyond the scope of the PCF service definition. Such references are known as **indirect references**. They shall be implemented using the PCF URN syntax (see annex Q).

EXAMPLE 2: Indirect references are used for:

- Abstract media service and **Service** component references, for service tuning and component selection such as "urn:x-dvb-pcf:bbc.co.uk::bbctwo".
- PCF service entry point references, for inter-PCF service navigation such as "urn:x-dvb-pcf:skyinteractive.com::SkyActive".
- PCF return path configuration references, for establishing return path sessions such as "urn:x-dvb-pcf:www.foo.bar::myvotingserver".

NOTE: URNs are managed independently by each content provider. This includes registration with platform operators and distribution to service authors.

13.1.7 Service coherence

Throughout the service registration lifetime, all direct and indirect references within the set of assets loaded into the service context must be resolvable. It is the responsibility of the transcoder to test this condition for each new transaction. This is known as the **coherence validation rule** and ensures that the service is always in a coherent condition. If this condition is invalidated by a transaction, this transaction shall be rejected by the transcoder and an error returned to the service provider. Thus, the order in which assets are supplied via a sequence of transactions is important and shall be maintained by the transcoding process.

13.1.8 Transcoder hints

Hints may be provided to the transcoder with each transaction to assist the transcoder in processing the assets. Annex E defines a set of portable hints that shall be supported by all transcoder implementations that adopt this clause of the specification. Other hints, known as transcoder directives, are not portable and may be implemented by a transcoder to extend the hinting capability in a vendor-specific manner. PCF specifies a syntax for the transport of all hints.

13.2 Detailed model specification

Figure 76 illustrates the domain model for the physical format of a PCF service description and the interacting objects through which such a description is transported to a PCF transcoder. The UML containment association has deliberately been used to specify entities whose lifetimes are related.

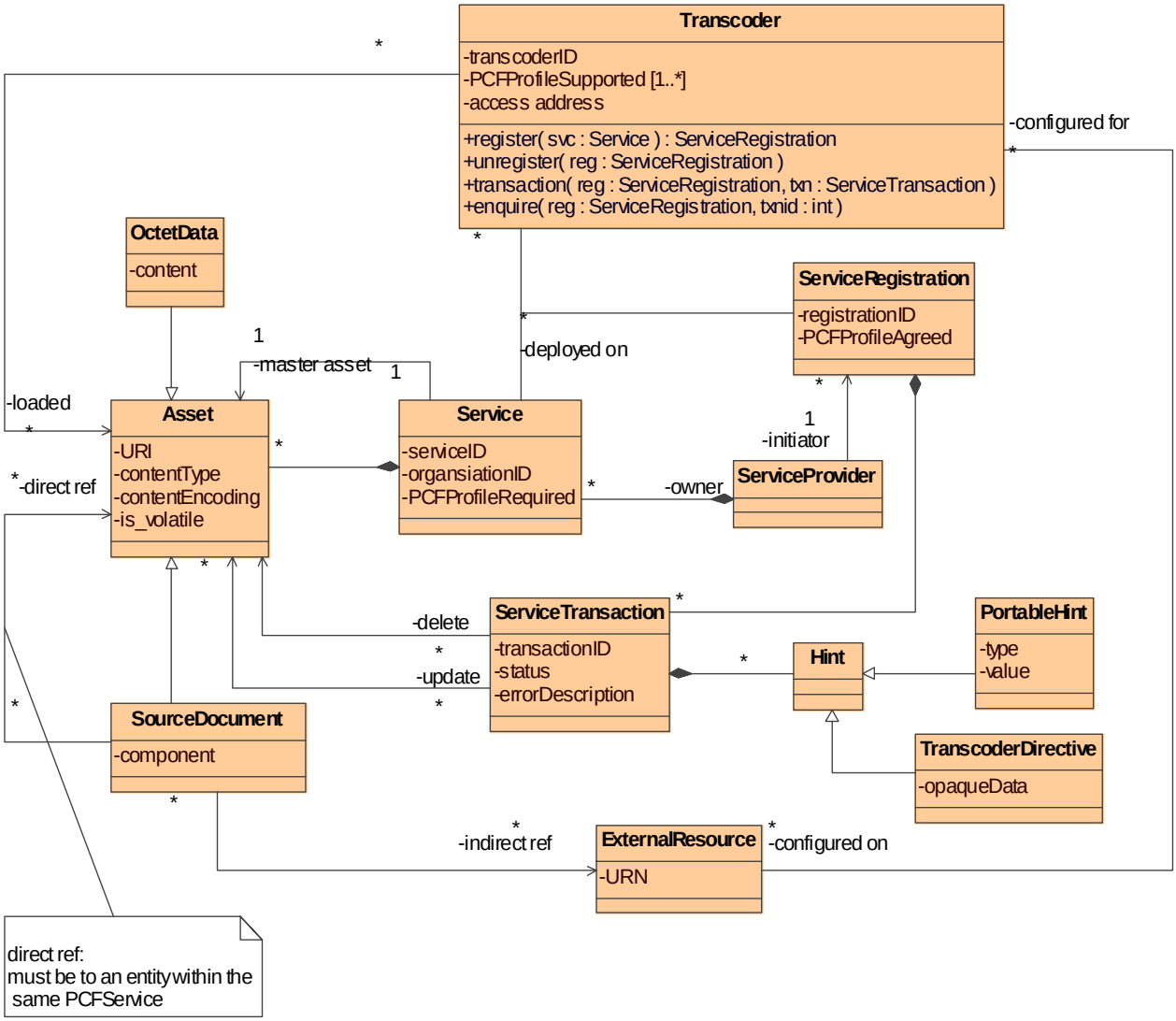

Figure 76: Transport and Packaging domain model

13.2.1 PCFTranscoder

This object encapsulates the public interface of a network operator's transcoder. Associations, attributes and operations are as defined in table 29.

Table 29: Transcoder interface specification

Associations:	PCFService: by association class ServiceRegistration a PCFTranscoder will be responsible for transcoding the service descriptions contained in all associated PCFServices. Transcoding occurs within the context of one or more PCFServiceRegistration instances. Thus a PCFService may be registered one or more times allowing temporal isolation between transmissions of the same PCFService. A ServiceRegistration is returned by a successful call to the register() operation.
	PCFAsset: as role loaded each PCFTranscoder will reference a set of PCFAssets. These assets are those that have been successfully delivered by the transaction() operation in the context of a ServiceRegistration.
Attributes:	transcoderID: a unique identifier for a transcoder instance.
	PCFProfileSupported: each PCFTranscoder declares its transcoding capability through a set of acceptable profile values. When a register() operation is called, the PCFService::PCFProfileRequired value is checked for existence within this set.
	access address: Each PCFTranscoder will be exposed on a well-known access address (e.g. a SOAP URI).
Operations:	ServiceRegistration register(PCFService): a service provider must register PCFServices with PCFTranscoders. This operation shall result in a ServiceRegistration, thus it shall allocate a registrationID and resolve the PCFprofileAgreed value that shall be used when transcoding the service. It must be the first action taken during the service lifetime. If no acceptable PCFProfileAgreed value can be found, the operation shall fail.
	unregister(ServiceRegistration): this operation shall destroy the ServiceRegistration and any loaded PCFAssets associated to it via successful transaction() operations. It must be the last action taken during the service lifetime.
	transaction(ServiceRegistration, ServiceTransaction): this operation shall update the state of a service description, within the context of the ServiceRegistration, with a set of new PCFAssets. It may also remove PCFAssets from the loaded set. NOTE: This operation applies the *coherence validation rule* across the loaded PCFAssets. See ServiceTransaction for details and error reporting.

13.2.2 ServiceRegistration

This object encapsulates the association between a PCFTranscoder and a PCFService. Associations, attributes and operations are as specified in table 30.

Table 30: Service registration interface

Associations:	ServiceTransaction: all updates to a service description occur within the context of a ServiceRegistration.
	ServiceProvider: a service provider shall be the initiator of a ServiceRegistration via the register() operation.
Attributes:	registrationID: allocated by the PCFTranscoder to identify this context. Unique within the scope of the PCFTranscoder.
	PCFProfileAgreed: the profile value agreed by the registration operation, to be used in transcoding the PCFService

13.2.3 ServiceTransaction

This object encapsulates a set of PCFAssets to be applied to a PCFTranscoder through the transaction() operation. Associations, attributes and operations shall be as specified in table 31.

NOTE: This is distinct from the return path transaction that is described in clause 10.2.3.

Table 31: Service transaction interface

Associations:	ServiceRegistration: this provides the context on the PCFTranscoder within which transactions will be applied and validated.
	PCFAsset: in roles "update" and "delete" the set of PCFAssets to be changed or removed from the service context. NOTE 1: In order to adhere to the *coherence validation rule* it is necessary that the first transaction supplies the PCFAsset identified as the *master asset* in the PCFService.
	Hint: metadata to assist the transcoder in operating on the PCFAssets referenced in this transaction.
Attributes:	transactionID: allocated by the transcoder to identify this transaction. Unique within the scope of the ServiceRegistration.
	Status: the state of the transaction in the PCFTranscoder. Values are: 1) accepted: transaction accepted for processing; 2) validated: transaction has passed coherence test and service context state has been updated; 3) completed: transaction processing complete; 4) failed: transaction failed during processing, see errorDescription. NOTE 2: States 1 and 2 will only be visible for an asynchronous transaction request. Once a transaction has completed processing and has been applied to the ServiceRegistration context it may be removed from the transcoder. This is indicated by the completed state. A transaction can fail at any point prior to reaching completed. This is indicated by the failed state. Failed transactions can also be removed from the transcoder.
	errorDescription: machine and human readable error value to identify cause of failure.

13.2.4 PCFService

This object encapsulates the existence of a PCF service description and associated assets. Associations, attributes and operations shall be as defined in table 32:

Table 32: Pcf service description interface

Associations:	ServiceRegistration: PCF services may be registered multiple times to different transcoders. The service registration holds the context in which the service is deployed.
	PCFAsset: a PCF service description is a collection of PCF assets. One of these assets is indicated as the *master asset* and is used as the starting point for service validation and transcoding.
Attributes:	serviceID: ID allocated by service provider to identify this PCF service description. Unique within scope of organizationID.
	organizationID: globally unique ID allocated by PCF Naming Authority to service providers. Equivalent to MHP organizationID in [20].
	PCFProfileRequired: minimum level PCF profile value against which this PCF service can be deployed. Failure to meet this level within the target transcoder will prevent registration.

13.2.5 PCFAsset and specializations

PCFAsset is an abstract interface to a PCF source document of other octet data. PCF assets shall be passed across the interface as the atomic units of data transfer. PCF assets thus define the atomic units of a service description that may be modified over time.

Specific asset types are also described. Associations, attributes and operations shall be as defined in table 33.

Table 33: PCF assets interface

Associations:	PCFSourceDocument(specialization): Unit containing PCF service description source language. Contains a PCF Component in XML format. This unit may be associated with zero or more other PCF assets or ExternalResources.
	OctetData(specialization): Unit containing octet data in a PCF-compliant content format. Transcoder support for content formats is determined by profile level agreed, e.g. graphics formats, text markup formats. This unit shall be entirely self-contained.
Attributes:	URIPath: identifier for PCFAsset and optionally locator for asset contents. May be used by transcoder to retrieve the asset (pull and online update models). Shall be unique within scope of PCFService.
	contentType: content format indicator for transcoder use, as specified by "content-type" header in [21]. Supported values are determined by agreed profile level. NOTE 1: Where pull model content delivery is used, this value represents the expected content type and shall be used to validate the actual content type when that asset is received.
	contentEncoding: content encoding indication for transcoder use, as specified by "content-encoding" header in [21]. Supported values are determined by agreed profile level. NOTE 2: Use for pull model content delivery validation as noted for contentType field.
	is_volatile: flag to indicate that an asset will be updated intermittently at source and should be refreshed by the transcoder. URIPath must resolve to a remote asset to allow the transcoder to refresh this item of content repeatedly.

13.2.6 ExternalResource

ExternalResource is an abstract interface to a PCF URN. Associations, attributes and shall be as defined in table 34.

Table 34: External resources interface

Associations	PCFSourceDocument: URN values may appear in any PCF source document. They shall be resolved on the transcoder to actual resource information.
	Transcoder: ExternalResources are configured by the transcoder operator to support access by PCF services to specific resources. URN values are supplied by resource providers.
Attributes:	URN: portable URN namespace used to reference external resources. Resolved by transcoder using network-specific information.

13.2.7 Hint and specializations

Hint is an abstract interface to an item of metadata that assists the transcoder in operating on a transaction. Specializations are also described. Associations, attributes and operations shall be as specified in table 35.

Table 35: hints interface

Associations:	ServiceTransaction: Hints exist in the scope of a ServiceTransaction and have the same lifetime.
	TranscoderDirective(specialization): this is an opaque, transcoder-specific instruction that allows the hint set to be extended for a particular transcoder instance. PCF does not specify any syntax for what the opaqueData attribute may contain, except that it shall be of type xsd:string.
	PortableHint(specialization): this is a PCF-defined transcoder hint. The hint type is contained in the type attribute, and a type-dependant value is contained in the value attribute. Annex E defines the current set of portable hint types and values.

13.3 PCF data exchange sequence for transcoder input

Figure 77 illustrates the sequence of actions necessary to deliver a PCF service description to a transcoder. Numbered messages are referred to in the following text.

A service provider shall register a PCF service with a transcoder prior to any other action [1, 2]. This shall return a service registration instance. The service registration must be presented in all subsequent transactions [3, 8, 19].

A PCF transcoder shall maintain the order of all input transactions of equal priority when updating a service registration context.

A PCF transcoder may process transactions of a higher priority value before processing any transactions of a lower priority value. Input order of transactions with different priority values may not be maintained [not shown].

A PCF transcoder shall maintain the atomicity of each transaction between validation and service registration context update, independent of the priority of such transactions.

NOTE: In the illustration, serialization of processing is used to maintain atomicity. Thus the validation of the second transaction [14] is delayed until the first transaction is completed [11]. Alternative strategies may be employed.

In the event of a syntax validation failure or *coherence validation failure*, the affected transaction shall not be applied to the service registration context and an error shall be returned [not shown].

A service provider shall unregister a PCF service after all other actions are complete [19, 20].

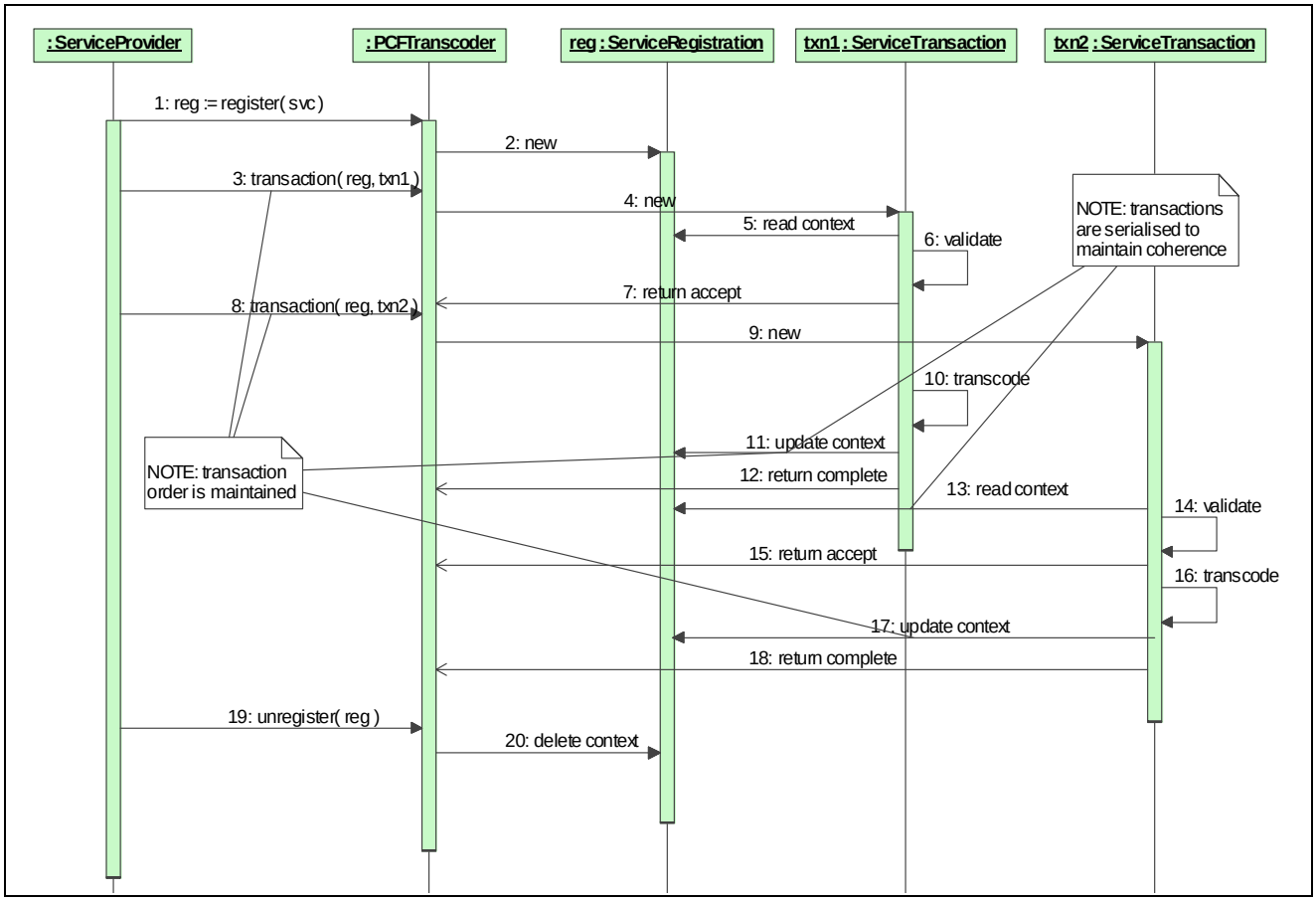

Figure 77: Transcoder operation sequence model

An example implementation of an interface conforming to this model is given using the SOAP [19] standard web services description language in clause G.3.1.

Annex A (normative): Component specifications

A.1 Container components

A.1.1 Layout components

A.1.1.1 Service

The **Service** component shall represent the complete description of an interactive service.

```
    <ComponentSpec provider="dvb.org" name="Service" container="true" serializable="false">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="pcfSpecVersion firstScene referenceScreen
referenceScreenMapping referenceScreenAlignment referenceScreenSurround serviceAspectRatio
videoHandlingPriority"/>
        <Properties>
            <PropertySpec name="pcfSpecVersion" type="string" use="required"
access="initializeOnly"/>
            <PropertySpec name="firstScene" type="uri" use="required" access="initializeOnly"/>
            <PropertySpec name="announcement" type="uri" use="optional" access="initializeOnly"/>
            <PropertySpec name="serviceDigest" type="uri" use="optional" access="initializeOnly"/>
            <PropertySpec name="referenceScreen" type="size" use="optional" access="initializeOnly">
                <pcf:Size name="default" value="720 576"/>
            </PropertySpec>
            <PropertySpec name="referenceScreenMapping" type="enumeration" use="optional"
access="initializeOnly">
                <EnumerationSpec name="mappedTo">
                    <EnumerationItem name="display-anamorphic"/>
                    <EnumerationItem name="display-preserve"/>
                    <EnumerationItem name="pixel"/>
                </EnumerationSpec>
                <pcf:String name="default" value="pixel"/>
            </PropertySpec>
            <PropertySpec name="referenceScreenAlignment" type="enumeration" use="optional"
access="initializeOnly">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="bullseye"/>
            </PropertySpec>
            <PropertySpec name="referenceScreenSurround" type="color" use="optional"
access="initializeOnly">
                <pcf:Color name="default" value="#000000"/>
            </PropertySpec>
            <PropertySpec name="serviceAspectRatio" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="0 0"/>
            </PropertySpec>
            <PropertySpec name="videoHandlingPriority" type="enumeration" use="optional"
access="readWrite">
                <EnumerationSpec name="priority">
                    <EnumerationItem name="aspectRatio"/>
                    <EnumerationItem name="alignment"/>
                </EnumerationSpec>
                <pcf:String name="default" value="aspectRatio"/>
            </PropertySpec>
        </Properties>
    </ComponentSpec>
```

A Service component is a kind of map item, as defined in clause 6.2.5.2. This means that it may contain zero or more instances of any other kind of PCF item, with the following restrictions:

- A Service component shall contain at least one scene component.
- A Service component shall not contain other Service components.

NOTE: An explicit mechanism for nesting services components is beyond the scope of the present document.

The **pcfSpecVersion** property shall identify the version of the PCF specification with which the service description is conformant.

The **firstScene** property shall identify the initially active scene item within the service description to be presented.

The **announcement** property may be used to identify the PCF description of the presentation of an announcement, e.g. "Press Red". This is handled separately from the first scene (as identified by the firstScene property) since such announcements are implemented in different ways on different platforms and not always as part of a delivered interactive service. See annex M.

The **serviceDigest** property may be used to identify the service digest. The service digest provides summary information about the service description, as defined is clause 12. The service digest does not form part of the PCF service description.

The **referenceScreen** property shall be used to define the size of a rectangular co-ordinate system that comprises the reference screen in which all other visible components of the service description shall be located.

The **referenceScreenMapping** property shall be used to define how the co-ordinate system of the reference screen shall be mapped into the co-ordinate system of a target device when the resolution of a target device is different to that of the reference screen. See clause 8.7.2.2.

The **referenceScreenAlignment** property shall be used to position the reference screen within the display of the target device in cases where the resolution of the mapped reference screen is smaller than that of the display of a target device. See clause 8.7.2.2.

The **referenceScreenSurround** property shall be used to indicate the colour to use to fill any part of the display of a target device not defined by the mapped reference screen. See clause 8.7.2.2.

The **serviceAspectRatio** property shall be used to indicate the aspect ratio of the service. The value "0 0" shall mean that the aspect ratio of the service is undefined.

The **videoHandlingPriority** property shall be used to indicate whether the priority is to present video at the correct aspect ratio or to maintain registration of video and graphics in scenarios where both can not be achieved.

The **Service** component is a **StaticELC** (see clause 8.2) with a fixed origin at screen co-ordinate 0,0.

A.1.1.2 Scene

The purpose of the **scene** item is to describe a spatially and temporally co-ordinated viewer experience at a particular point in the service description.

```
<ComponentSpec provider="dvb.org" name="Scene" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties=""/>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="scene_events"/>
        </GeneratedEvents>
</ComponentSpec>
```

A scene component is a kind of map item, as defined in clause 6.2.5.2. This means that it may contain zero or more instances of any other kind of PCF item, with the following restrictions:

- A scene component shall not contain Service components.
- A scene component shall not contain other scene components.

The scene component is a StaticELC (see clause 8.2) with a fixed origin at screen co-ordinate 0,0.

The scene shall generate a SceneLifecycle event when it becomes active, immediately prior to rendering any of its child components. The action property shall indicate if the scene has been entered by a forward or history navigation action.

The scene shall generate a SceneLifecycle event when it becomes inactive, immediately prior to deactivating any of its child components. The action property shall indicate that the scene is exiting.

A.1.1.3 Static explicit layout container specification

```
<ComponentSpec provider="dvb.org" name="StaticELC" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="origin"/>
    <Properties>
        <PropertyGroupRef ref="positioning_properties-absolute" properties="origin"/>
    </Properties>
</ComponentSpec>
```

The **origin** property defines the co-ordinate point within the **ELC**'s parent component that shall be used as the zero co-ordinate point for positioning the **ELC**'s child components.

The origin property is **initializeOnly**.

A.1.1.4 Explicit layout container specification

```
<ComponentSpec provider="dvb.org" name="ELC" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="origin"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components" properties="visible"/>
        <PropertyGroupRef ref="positioning_properties-absolute" properties="origin"/>
    </Properties>
</ComponentSpec>
```

The **origin** property defines the co-ordinate point within the **ELC**'s parent component that shall be used as the zero co-ordinate point for positioning the **ELC**'s child components.

If the **visible** property is set to true, all child components that also have their visible property set to true shall be shown. If the visible property is set to false, all child components shall be hidden.

A.1.1.5 Flow layout container component specifications

A.1.1.5.1 TruncateFlowContainer component

A **TruncateFlowContainer** is a flow layout container that does not provide any paging or scrolling facilities. Child content and components are flowed within the available area. If the child elements, when laid out, overflow the available vertical area they are truncated.

The following XML defines the **StaticTruncateFlowContainer** interface:

```
<ComponentSpec provider="dvb.org" name="StaticTFC" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="visible focusable enabled origin
                                            size bordercolor fillcolor textcolor
                                            h-align v-align content"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="font-properties"/>
        <PropertySpec name="grow" type="boolean" access="initializeOnly"
                      use="optional">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
        <PropertySpec name="max-height" type="integer" access="initializeOnly"
                      use="optional">
            <pcf:Integer name="default" value="0"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The following XML defines the **DynamicTruncateFlowContainer** interface:

```
<ComponentSpec provider="dvb.org" name="TFC" container="true">
    <Overview version="1.0"/>
```

```
    <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
                                            fillcolor textcolor h-align v-align content"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="font-properties"/>
        <PropertySpec name="grow" type="boolean" access="readWrite" use="optional">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
        <PropertySpec name="max-height" type="integer" access="readWrite" use="optional">
            <pcf:Integer name="default" value="0"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

A **StaticTruncateFlowContainer** is a flow layout container that may be initialized with defined content area dimensions, and a set of child content and components. Once initialized, these values cannot be changed during the lifetime of the component.

A **TruncateFlowContainer** provides the same functionality as a static flow container, except the dimensions of its content area, and its set of child content and components may be changed during the lifetime of the component, in which case the component will re-calculate its layout.

The **visible** property defines whether or not the flow container is visible.

The **focusable** property defines whether or not the flow container will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the flow container is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the flow container when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the flow container when it is drawn within an explicit layout container. This value may only be modified after initialization for a dynamic flow container.

The **fillcolor** property shall define the default fill colour of the flow container.

The **textcolor** property shall define the default colour of the text within the flow container.

The **border_properties** property group shall define the colour, style, width and corner-radius of the flow container's border.

The **alignment_properties** property group shall define the default vertical and horizontal alignment of the text within the content area of the flow container.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the flow container is itself a child of a parent flow layout container.

The **font_properties** define the default font to use to display any content text.

If the **grow** property is set to "true" then the height of the flow container shall increase past the height defined by the **size** property, up to the value defined by the **max-height** property, if the flowed content requires this extra height in order to display without being truncated. If the flow layout has grown to its max-height and the content is still too large to fit, then the content shall be truncated.

Child components and content elements shall be laid out using the rules defined within clause 8.3.

A.1.1.5.2 ScrollFlowContainer component

A **ScrollFlowContainer** is a flow layout container that provides scrolling facilities. The height of the box into which child content and components are flowed is potentially limitless. If the height is greater than the height of the **SFC**'s content area then it shall be possible for the user to view the total flow in stages by scrolling over flow using navigation keys.

If the content is large enough to require scrolling, then a scroll bar shall be displayed adjacent to the content area, but within the bounds defined by the **size** property, to show which point within the scrollable area is currently visible.

The following XML defines the **StaticScrollContainer** interface:

```
    <ComponentSpec provider="dvb.org" name="StaticSFC" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor textcolor h-align v-align navigation_properties content scroll-position"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
            <PropertyGroupRef ref="navigation_properties"/>
            <PropertySpec name="scroll-position" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="west"/>
            </PropertySpec>
        </Properties>
    </ComponentSpec>
```

The following XML defines the **DynamicScrollContainer** interface:

```
    <ComponentSpec provider="dvb.org" name="SFC" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor textcolor h-align v-align navigation_properties content scroll-position"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
            <PropertyGroupRef ref="navigation_properties"/>
            <PropertySpec name="scroll-position" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="west"/>
            </PropertySpec>
        </Properties>
    </ComponentSpec>
```

A **StaticScrollContainer** is a scroll container that may be initialized with defined content area dimensions, and a set of child content and components. Once initialized, these values cannot be changed during the lifetime of the component.

A **DynamicScrollContainer** provides the same functionality as a static scroll container, except the dimensions of its content area, and its set of child content and components may be changed during the lifetime of the component, in which case the component will re-calculate its layout.

The **visible** property defines whether or not the scroll container is visible.

The **focusable** property defines whether or not the scroll container will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the scroll container is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the scroll container when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the scroll container when it is drawn within an explicit layout container. This value may only be modified after initialization for a dynamic scroll container.

The **fillcolor** property shall define the default fill colour of the scroll container.

The **textcolor** property shall define the default colour of the text within the scroll container.

The **border_properties** property group shall define the colour, style, width and corner-radius of the scroll container's border.

The **alignment_properties** property group shall define the default vertical and horizontal alignment of the text within the content area of the scroll container.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the scroll container is itself a child of a parent flow layout container.

The **font_properties** define the default font to use to display any content text.

The **next-key** property shall define the key the user may press to scroll to the next line of content.

The **previous-key** property shall define the key the user may press to scroll to the previous line of content.

The **scroll-position** property shall define whether the scroll bar, if required, is displayed to the East or the West of the content area.

The **content** property shall define the content elements to display within the scroll container. Content may be any marked up document fragment that conforms to the rules defined within the supplied XML schema "x-dvb-pcf.xsd". This value may only be modified after initialization for a dynamic flow container.

Child components and content elements shall be laid out using the rules defined within clause 8.3

A.1.1.5.3 PFC component

A **PFC** is a flow layout container that provides paging facilities. The height of the box into which child content and components are flowed is potentially limitless. If the height is greater than the height of the page flow container's content area then it shall be possible for the user to view the total flow in stages by paging through the flow using navigation keys.

The following XML defines the **StaticPFC** interface:

```
    <ComponentSpec provider="dvb.org" name="StaticPFC" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor textcolor h-align v-align navigation_properties content label-format label-text"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
            <PropertyGroupRef ref="navigation_properties"/>
            <PropertySpec name="label-format" type="string" access="initializeOnly"  use="optional">
              <pcf:String name="default" value="Page %p of %t"/>
            </PropertySpec>
            <PropertySpec name="label-text" type="string" access="readOnly"/>
            <PropertySpec name="num-pages" type="integer" access="readOnly"/>
            <PropertySpec name="current-page" type="integer" access="readOnly"/>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="page_container_events"/>
        </GeneratedEvents>
    </ComponentSpec>
```

The following XML defines the **DynamicPFC** interface:

```
    <ComponentSpec provider="dvb.org" name="PFC" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor textcolor h-align v-align navigation_properties content label-format label-text"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
            <PropertyGroupRef ref="navigation_properties"/>
            <PropertySpec name="label-format" type="string" access="readWrite" use="optional">
                <pcf:String name="default" value="Page %p of %t"/>
            </PropertySpec>
            <PropertySpec name="label-text" type="string" access="readOnly"/>
            <PropertySpec name="num-pages" type="integer" access="readOnly"/>
            <PropertySpec name="current-page" type="integer" access="readOnly"/>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="page_container_events"/>
        </GeneratedEvents>
    </ComponentSpec>
```

A **StaticPageContainer** is a page container that may be initialized with defined content area dimensions, and a set of child content and components. Once initialized, these values cannot be changed during the lifetime of the component.

A **DynamicPageContainer** provides the same functionality as a static page container, except the dimensions of its content area, and its set of child content and components may be changed during the lifetime of the component, in which case the component will re-calculate its layout.

The **visible** property defines whether or not the page container is visible.

The **focusable** property defines whether or not the page container will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the page container is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the page container when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the page container when it is drawn within an explicit layout container. This value may only be modified after initialization for a dynamic page container.

The **fillcolor** property shall define the default fill colour of the page container.

The **textcolor** property shall define the default colour of the text within the page container.

The **border_properties** property group shall define the colour, style, width and corner-radius of the page container's border.

The **alignment_properties** property group shall define the default vertical and horizontal alignment of the text within the content area of the page container.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the page container is itself a child of a parent flow layout container.

The **font_properties** define the default font to use to display any content text.

The **next-key** property shall define the key the user may press to move to the next page of content.

The **previous-key** property shall define the key the user may press to move to the previous page of content.

The **label-format** property shall define the format in which page information shall be rendered into the **label-text** information string. This value of this property shall be a plain text string that may contain either or both of the following format flags:

- %c: replace with the current page number;
- %t: replace with the total number of pages.

EXAMPLE: For a page container positioned on page 3 of a total of 6 pages, a label-format value of "Page %c of %t" would result in a value in label-text of "Page 3 of 6".

This value may only be modified after initialization for a dynamic page container.

The **label-text** property is a read only property that shall contain an information string, with a format defined by the **label-format** property, that may be used to provide a report of the current page position within the total flow. This value shall update as the user navigated through the available pages.

The **num-pages** property is a read only property that contains the total number of pages available within this page container.

The **current-page** property is a read only property that contains the page number of the currently visible page.

The **content** property shall define the content elements to display within the page container. Content may be any marked up document fragment that conforms to the rules defined within the supplied XML schema "x-dvb-pcf.xsd". This value may only be modified after initialization for a dynamic flow container.

Child components of a page flow container shall be laid out using the rules defined within clause 8.3.

A page flow container shall generate an **OnPageChanged** event every time the user navigates to a new page.

A.1.2 Flow components

A.1.2.1 Flow

A.1.2.1.1 Introduction

The following fragment of XML defines the Flow component:

```
    <ComponentSpec provider="dvb.org" name="Flow" container="true">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="content"/>
        <Properties>
            <PropertySpec name="content" type="markedUpText" use="optional" access="readWrite"/>
            <PropertySpec name="directionality" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="directionality">
                    <EnumerationItem name="ltr"/>
                    <EnumerationItem name="rtl"/>
                </EnumerationSpec>
                <pcf:String name="default" value="ltr"/>
            </PropertySpec>
        </Properties>
    </ComponentSpec>
```

The Flow component can contain marked up text and other visual elements such as tables and **image**s, that are to be automatically laid-out and presented in one of the three kids of Flow Layout Container components.

The Flow component does not have any origin or size properties. The width for a Flow is derived from the available width of the FlowLayoutContainer in which it is to be presented. The height of a flow is derived the sum of the heights of each line of laid-out content in the Flow.

Flow layout shall always be implemented from the top down.

A.1.2.1.2 The content property

```
<PropertySpec name="content" type="markedUpText" use="optional" access="readWrite"/>
```

The **content** property shall define the content elements to be laid-out within the flow. Content may be any marked up document fragment that conforms to the rules defined within the supplied XML schema "x-dvb-pcf.xsd". This value may only be modified after initialization for a dynamic flow container.

A.1.2.1.3 The directionality property

```
<PropertySpec name="directionality" type="enumeration" access="initializeOnly" use="optional">
                <EnumerationSpec name="directionality">
                    <EnumerationItem name="ltr"/>
                    <EnumerationItem name="rtl"/>
                </EnumerationSpec>
                <pcf:String name="default" value="ltr"/>
            </PropertySpec>
```

The **directionality** property shall define the direction of automatic layout in the Flow.

Default Flow direction is left-to-right.

Flow layout may optionally be implemented in right-to-left direction.

NOTE: Directionality of a Flow applies to the entire Flow. The PCF does not support conflicting directionality for different parts of a single flow.

A.1.2.2 TextFlow

A.1.2.2.1 Introduction

The following fragment of XML defines the Flow component:

```
    <ComponentSpec name="TextFlow" serializable="false">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="content"/>
        <Properties>
            <PropertySpec name="content" type="markedUpText" use="optional" access="readWrite">
                <pcf:MarkedUpText name="default" value=""/>
            </PropertySpec>
            <PropertySpec name="directionality" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="directionality">
                    <EnumerationItem name="ltr"/>
                    <EnumerationItem name="rtl"/>
                </EnumerationSpec>
                <pcf:String name="default" value="ltr"/>
            </PropertySpec>
        </Properties>
    </ComponentSpec>
```

The TextFlow component can contain only marked up text that is to be automatically laid-out and presented in one of the three kinds of Flow Layout Container components.

The TextFlow component does not have any origin or size properties. The rendered width of the content declared within a TextFlow component is derived from the available width of the FlowLayoutContainer in which it is to be presented. The height of a flow is derived the sum of the heights of each line of laid-out content in the TextFlow.

TextFlow layout shall always be implemented from the top down.

A.1.2.2.2 The content property

```
<PropertySpec name="content" type="markedUpText" use="optional" access="readWrite"/>
```

The **content** property shall define the content elements to be laid-out as a flow.

Content may be any marked up document fragment that conforms to the rules defined within the supplied XML schema "x-dvb-pcf.xsd" with the following restrictions:

- Block elements shall be restricted to paragraph elements only.
- Special elements shall be restricted to line break elements and span elements only.

Any other mark-up may be ignored but any text enclosed within it shall be rendered.

This value of this property may only be modified after initialization for a dynamic flow container.

A.1.2.2.3 The directionality property

```
<PropertySpec name="directionality" type="enumeration" access="initializeOnly" use="optional">
                <EnumerationSpec name="directionality">
                    <EnumerationItem name="ltr"/>
                    <EnumerationItem name="rtl"/>
                </EnumerationSpec>
                <pcf:String name="default" value="ltr"/>
            </PropertySpec>
```

The **directionality** property shall define the direction of automatic layout in the Flow.

Default Flow direction is left-to-right.

Flow layout may optionally be implemented in right-to-left direction.

NOTE: Directionality of a Flow applies to the entire Flow. The PCF does not support conflicting directionality for different parts of a single flow.

A.1.2.3 Table components

Similar to HTML 4.0, table layout is "row primary", meaning that service authors specify rows, not columns, in the PCF service description.

A **Table** comprises any number of rows of cells.

Columns are derived after the rows have been defined - the first cell in each row belongs to the first column, the second cell in each row belongs to the second column, and so on. Rows and columns may be grouped, and any such groupings may optionally be reflected in the visual presentation of the table.

EXAMPLE: Styling characteristics such as fill and border colour may be applied row-wise and column-wise to groups of table cells to provide individual formatting for any row or column. This enables authors to highlight table headers and footers, or columns and rows as required to enhance presentation and usability of the table content.

The PCF table model comprises: tables, rows, row groups, columns, columns groups and cells. These are defined using the following elements:

A.1.2.3.1 The Table component

```
<ComponentSpec provider="dvb.org" name="Table" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="table-layout table-columns column-widths bordercolor
fillcolor"/>
    <Properties>
        <PropertySpec name="table-columns" type="integer" use="required" access="initializeOnly"/>
        <PropertySpec name="column-widths" type="string" use="optional" access="initializeOnly"/>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
```

```
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
    </Properties>
</ComponentSpec>
```

A.1.2.3.1.1 The table-layout property

The **table-layout** property specifies the layout algorithm to be used for the table. If the **table-layout** property value is "auto", then the automatic layout algorithm, or other implementation-specific algorithm shall be used. If the **table-layout** property value is "fixed", then the fixed layout algorithm shall be used.

A.1.2.3.1.2 The caption property

The **caption** property specifies a text string that can be used to provide a brief description of the purpose of the table. The **caption** property is optional, and can be used to populate a table heading.

A.1.2.3.1.3 The table-columns property

The **table-columns** property specifies the number of columns in the table.

EXAMPLE: <table table-columns="5">. In this example, a five column table will be drawn. In the absence of a **column-widths** specification, each column will consume an equal amount of the available table width. In this case, each column will consume.

A.1.2.3.1.4 The row-height property

The **row-height** property specifies in pixels the default height for each row in the table. The value of the row-height property may be over ridden by a **TR** or a **TD**.

NOTE: By default, a table cell will wrap to accommodate its content. Where row height has been specified, cells in that row will truncate any content that cannot be accommodated within the specified height.

A.1.2.3.1.5 The columnwidth property

Table column widths may be defined in three ways:

- Absolute column widths - each column is specified in pixels.
- Percentage column widths - each column width is specified as a percentage of the available table width.
- Auto column widths - the number and width of columns is not specified, but is calculated by the PCF implementation.

Column widths may be specified in percentage values. A percentage value specified for a column width shall be resolved relative to the computed table width. If the table has a width property value of "auto", then a percentage represents a constraint on the column's width which the PCF implementation should satisfy.

NOTE 1: Percentage width declarations may not always be possible to satisfy. A column width value of 110 % cannot be satisfied.

NOTE 2: In this algorithm, both rows and columns constrain and are constrained by the dimensions of the cells they contain. Setting column width may impact on the height of a row, and *vice versa*.

EXAMPLE 1: Absolute column widths

```
<table width="100" table-layout="fixed" table-columns="20, 20, 20, 20, 20">
```

NOTE 3: Where specified, absolute **table-column** values must add up to equal **table-width** value.

EXAMPLE 2: Percentage column widths

```
<table width="100" table-layout="fixed" table-columns="20%, 20%, 20%, 20%, 20%">
```

NOTE 4: Where specified, percentage table-column values must add up to 100 %

A.1.2.3.2 Table row group components

Table rows may be grouped into table header, table body and table footer groups, using the **TH**, **TB** and **TF** components respectively. This division enables PCF implementations to support scrolling of table bodies independently of header and footer, or repeat of table header and footer rows across multiple pages when the **Table** is presented in a **PFC**.

When present, the **TH**, **TB** and **TF** each contain a row group. Each row group shall contain at least one **TR**.

A.1.2.3.2.1 The TH component

```
<ComponentSpec provider="dvb.org" name="TH" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties=""/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="tableheader_properties"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
    </Properties>
</ComponentSpec>
```

The **TH** component can contain a row group. Where **TH** exists, it shall contain at least one **TR**.

A.1.2.3.2.2 The TB component

```
<ComponentSpec provider="dvb.org" name="TB" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties=""/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
    </Properties>
</ComponentSpec>
```

The **TB** component can contain a row group. Where **TB** exists, it shall contain at least one **TR**.

The **TB** component shall always be declared, unless the table consists only of a single table body, with no **TH** or **TF**.

A.1.2.3.2.3 The TF component

```
<ComponentSpec provider="dvb.org" name="TF" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties=""/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="tablefooter_properties"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
    </Properties>
</ComponentSpec>
```

The **TF** component can contain a row group. Where **TF** exists, it shall contain at least one **TR**.

A.1.2.3.3 Table column group components

A.1.2.3.3.1 The TC component

```
<ComponentSpec provider="dvb.org" name="TC">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="column-number"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="tablecolumn_properties"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
        <PropertySpec name="span" type="integer" use="optional" access="initializeOnly">
            <pcf:Integer name="default" value="1"/>
        </PropertySpec>
        <PropertySpec name="column-number" type="integer" use="required" access="initializeOnly"/>
    </Properties>
</ComponentSpec>
```

The **TC** component allows grouping of properties to be applied column-wise to table cells. The **TC** component does not group columns structurally.

The column-number property

```
<PropertySpec name=" column-number" type="integer" use="required" access="initializeonly"/>
```

The **column-number** property defines the column number to which the properties specified in **TC** shall be applied.

NOTE: Column numbering is contingent on directionality of Flow. A table in a left-to-right flow shall have column 1 as its left-most column. In a right-to-left flow, the first column shall be the right-most column.

The span property

```
<PropertySpec name="span" type="integer" use="optional" access="initializeonly" default="1"/>
```

The span property specifies the number of columns spanned by the **TC** element. The properties defined for the **TC** are shared for all cells in all columns in the span.

The default value for the span property is 1. If the span property value *n* is greater than 1, then the properties defined in **TC** shall be shared with the next *n*-1 columns.

A.1.2.3.3.2 The TCG component

```
<ComponentSpec provider="dvb.org" name="TCG" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="column-number"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="tablecolumn_properties"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
        <PropertySpec name="span" type="integer" use="optional" access="initializeOnly">
            <pcf:Integer name="default" value="1"/>
        </PropertySpec>
        <PropertySpec name="column-number" type="integer" use="required" access="initializeOnly"/>
    </Properties>
</ComponentSpec>
```

The **TCG** component groups columns together. Tables consist either of a single implicit column group, or of any number of explicitly declared columns groups.

The number of columns in a column group shall be defined in one of two mutually exclusive ways:

- The span property of the column group specifies the number of columns in the group.

- Each column component represents one or more columns in the group.

The column-number property

```
<PropertySpec name=" column-number" type="integer" use="required" access="initializeonly"/>
```

The **column-number** property defines the column number to which the properties specified in **TCG** shall be applied.

NOTE: Column numbering is contingent on directionality of Flow. A table in a left-to-right flow shall have column 1 as its left-most column. In a right-to-left flow, the first column shall be the right-most column.

The span property

```
<PropertySpec name="span" type="integer" use="optional" access="initializeonly" default="1"/>
```

The span property specifies the number of columns spanned by the **TCG** element. The properties defined for the **TCG** are shared for all cells in all columns in the span.

The default value for the span property is 1. If the span property value *n* is greater than 1, then the properties defined in **TC** shall be shared with the next *n*-1 columns.

The span value shall be ignored if the **TCG** contains one or more **TC** components.

A.1.2.3.4 The TR component

```
<ComponentSpec provider="dvb.org" name="TR" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="bordercolor fillcolor"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="tablerow_properties"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
    </Properties>
</ComponentSpec>
```

The **TR** component acts as a contained for a row of cells.

The **TR** component inherits all its styling attributed from its parent table component.

A.1.2.3.5 The TD component

```
<ComponentSpec provider="dvb.org" name="TD" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="bordercolor fillcolor"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="background_properties"/>
        <PropertySpec name="rowspan" type="integer" use="optional" access="initializeOnly">
            <pcf:Integer name="default" value="1"/>
        </PropertySpec>
        <PropertySpec name="colspan" type="integer" use="optional" access="initializeOnly">
            <pcf:Integer name="default" value="1"/>
        </PropertySpec>
        <PropertySpec name="wrap" type="boolean" use="optional" access="initializeOnly">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **TD** component contains content to be laid-out within a tabular structure.

TDs may be empty.

Content within a **TD** shall be laid-out according to the algorithm defined for Flow. By default, content within a table cell shall wrap, and the table cell shall stretch to accommodate the content.

A.1.2.3.5.1 The rowspan property

```
<PropertySpec name="rowspan" type="integer" use="optional" access="initializeonly" default="1" />
```

The rowspan property specifies the number of rows the cell shall span. Default value for rowspan property is "1".

A rowspan property value of "0" specifies that the cell shall span all rows from the row in which the cell is declared, to the last row in the rowgroup in which the cell is specified. If no rowgroup has been specified, then the cell shall span all rows from the row in which it is declared, to the last row in the table.

A.1.2.3.5.2 The colspan property

```
<PropertySpec name="colspan" type="integer" use="optional" access="initializeonly" default="1" />
```

The colspan property specifies the number of columns the cell shall span. Default value for the colspan property is "1".

A colspan property value of "0" specifies that the cell shall span all columns from the column in which the cell is declared, to the last column in the column group in which the cell is specified. If no column group has been specified, then the cell shall span all columns from the column in which it is declared, to the last column in the table.

NOTE: column spanning is contingent on directionality of the Flow in which the table is located.

A.1.2.3.5.3 The wrap property

```
        <PropertySpec name="wrap" type="boolean" use="optional" access="initializeOnly">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
```

The wrap property specifies behaviour of flowed content in a table cell when the content is too large to be accommodated within the cell. Default behaviour is for the content to wrap, and the table cell to grow to the extent required to accomodate the flowed content.

A wrap property value of "false" specifies that overflowing content in the table cell shall not wrap, and shall be truncated instead.

A.2 Visual components

A.2.1 Background

The **Background** component displays an image or a plain fill colour as the background for a scene.

```
    <ComponentSpec name="Background" provider="dvb.org">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="fillcolor"/>
        <Properties>
            <PropertyGroupRef ref="background_properties" properties="fillcolor fillcolor-rendering-
intent image tiling offset stretchToFit"/>
        </Properties>
    </ComponentSpec>
```

The **Background** component displays a static background for a scene. It is always positioned at the rearmost position in the display stack. The extent of the defined background shall always match the extent of the screen.

The **image** property may be used to specify an image resource to display as the background. If an image is specified, then its associated resource should be resolved and this should be displayed according to the **tiling-mode**, **stretch-to-fit** and **offset** properties. If no **image** property is specified or the image resource cannot be resolved, the **fillcolor** property shall provide a solid fill colour for the scene.

The **focusable** and **enabled** properties shall always be "false": the background shall not take focus or have an enabled or disabled state. As such, the **fillcolor-focus**, **fillcolor-disabled**, **fillcolor-active** and **fillcolor-idle** are not relevant and shall be ignored.

The **visible** property shall always be "true": it shall not be possible to make the background invisible.

The **aacolor** property shall be ignored as this has no meaning for a component that is always the rearmost in the z-order.

NOTE: To display video as a background to a scene, a **Video** component should be placed at the bottom of the display stack.

A.2.2 Basic shapes

A.2.2.1 Notes on basic shapes in general (informative)

Apart from the **AxisLine** component, all the basic shape components require some line-art capability. If this is not possible on the target platform it may be possible to pre-render line art at the head end, for display as a bitmap on the receiver.

A.2.2.2 AxisLine

An **AxisLine** component draws a straight, axis-aligned line.

```
<ComponentSpec provider="dvb.org" name="AxisLine">
<Overview version="1.0"/>
<IntendedImplementation coreProperties="color origin length axis"/>
    <Properties>
        <PropertySpec name="length" type="integer" use="required" access="readWrite"/>
        <PropertySpec name="axis" type="enumeration" use="required" access="readWrite">
            <EnumerationSpec name="axisline">
                <EnumerationItem name="vertical"/>
                <EnumerationItem name="horizontal"/>
            </EnumerationSpec>
        </PropertySpec>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="linestyle_properties"/>
        <PropertyGroupRef ref="color_properties-linecolor"/>
        <PropertyGroupRef ref="positioning_properties-absolute" properties="origin"/>
    </Properties>
</ComponentSpec>
```

An **AxisLine** component shall draw an X- or Y-axis aligned, straight line in a specified line colour.

The **origin** property shall define the start point of drawing the line. The **length** and **axis** properties shall be used to determine the end point. For vertical lines, the **length** property shall define the top-to-bottom distance in reference screen pixels from the start point. For horizontal lines, the **length** property shall define the left-to-right distance in reference screen pixels from the start point. A negative **length** property shall indicate that the line shall be drawn from bottom-to-top or right-to-left from the start point respectively.

The **linecolor** property shall determine the foreground colour to be used to draw the line. The **linestyle** property may be used to determine a particular style with which to draw the line.

A.2.2.3 Ellipse

An **Ellipse** component draws an ellipse within a conceptual enclosing rectangle of defined position and size.

```
    <ComponentSpec provider="dvb.org" name="Ellipse">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
        </Properties>
    </ComponentSpec>
```

The **visible** property defines whether or not the **Ellipse** is visible.

The **focusable** property defines whether or not the **Ellipse** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Ellipse** is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the **Ellipse**'s enclosing rectangle.

The **size** property shall define the size of the **Ellipse**'s enclosing rectangle.

The **fillcolor** property shall define the fill colour of the **Ellipse**.

The **border_properties** property group shall define the colour, style and width of the **Ellipse**'s border.

A.2.2.4 Line

A **Line** component draws a straight line between two points.

```
    <ComponentSpec provider="dvb.org" name="Line">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="linecolor origin size"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="linestyle_properties"/>
            <PropertyGroupRef ref="color_properties-linecolor"/>
        </Properties>
    </ComponentSpec>
```

A **Line** component shall draw a straight line between any two points in the reference screen area in a specified colour.

The **origin** property shall define the start point of drawing the line and the **size** property shall define the end point as an offset from the **origin**.

A.2.2.5 Pixel

The **Pixel** component draws a single pixel on the screen.

```
    <ComponentSpec provider="dvb.org" name="Pixel">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled linecolor origin"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="color_properties-linecolor"/>
            <PropertyGroupRef ref="positioning_properties-absolute" properties="origin"/>
        </Properties>
    </ComponentSpec>
```

The **visible** property defines whether or not the **Pixel** is visible.

The **focusable** property defines whether or not the **Pixel** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Pixel** is eligible to receive focus, due to user navigation.

The **origin** property shall define the position on screen where the **Pixel** is drawn.

The **linecolor** property shall define the colour in which the **Pixel** is drawn.

A.2.2.6 Polygon

A **Polygon** component draws an arbitrary polygon with a specified set of vertices.

```
    <ComponentSpec provider="dvb.org" name="Polygon">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor vertex"/>
        <Properties>
```

```
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute" properties="origin"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertySpec name="vertex" type="positionArray" use="required" access="readWrite"/>
        </Properties>
    </ComponentSpec>
```

The **visible** property defines whether or not the **Polygon** is visible.

The **focusable** property defines whether or not the **Polygon** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Polygon** is eligible to receive focus, due to user navigation.

The **origin** property defines the zero point that all vertices are drawn in relation to.

Each **vertex** property defines a vertex of the **Polygon**, relative to the origin.

The **fillcolor** property shall define the fill colour of the **Polygon**.

The **border_properties** property group shall define the colour, style and width of the **Polygon**'s border.

A.2.2.7 Rectangle

A **Rectangle** component draws a vertical and horizontal axis-aligned rectangle with a specified origin and size.

```
    <ComponentSpec provider="dvb.org" name="Rectangle">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin size bordercolor
fillcolor"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
        </Properties>
    </ComponentSpec>
```

The **visible** property defines whether or not the **Rectangle** is visible.

The **focusable** property defines whether or not the **Rectangle** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Rectangle** is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the **Rectangle**.

The **size** property shall define the horizontal and vertical dimensions of the **Rectangle**.

The **fillcolor** property shall define the fill colour of the **Rectangle**.

The **border_properties** property group shall define the colour, style, width and corner-radius of the **Rectangle**'s border.

A.2.3 Clock

The **Clock** component displays an automatically updating textual clock, whose date time format shall be configurable.

```
    <ComponentSpec provider="dvb.org" name="Clock">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties=""/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="color_properties-textcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
```

```
        <PropertySpec name="format" type="string" use="required" access="initializeOnly"/>
    </Properties>
</ComponentSpec>
```

The **visible** property defines whether or not the **Clock** is visible.

The **focusable** property defines whether or not the **Clock** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Clock** is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the **Clock** when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the **Clock**.

The **fillcolor** property shall define the fill colour of the **Clock**.

The **textcolor** property shall define the colour of the text within the **Clock**.

The **border_properties** property group shall define the colour, style, width and corner-radius of the **Clock**'s border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the text within the content area of the **Clock**.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **Clock** is drawn within a flow.

The **font_properties** define the font to use to display the **Clock**'s text.

The **format** property shall define the format in which the date and time are presented. Format is a string, in which certain character combinations shall be replaced with date and time elements. These character combinations shall be as follows:

- %a - replaced with locale specific abbreviated day name (eg Sun, Mon, Tue)
- %A - replaced with locale specific full day name (eg Sunday, Monday)
- %b - replaced with locale specific abbreviated month name (Jan, Feb)
- %B - replaced with locale specific full month name (January, February)
- %c - replaced with locale specific date/time info (in UK: Tue 10 August)
- %d - replaced by date of month (01-31)
- %H - replaced by hour (24 hour clock) (00-23)
- %I - replaced by hour (12 hour clock) (1-12)
- %j - replaced by day of year (001-365)
- %m - replaced by month (01-12)
- %M - replaced by minute (0-59)
- %p - replaced by local specific string for AM and PM
- %S - replaced by second (00-59)
- %U - replaced by week number of year (Monday 1st day of week) (01-53)
- %x - replaced by local date representation in locale specific format: (UK "31/12/04", US "12/31/04")

- %X - replaced by local time representation in local specific format (UK "17:32:16")
- %y - replaced by the year without the century: "04"
- %Y - replaced by the year with the century : "2004"

A.2.4 ConnectStatusImage

A.2.4.1 Introduction

The following XML fragment defines the **ConnectStatusImage** component:

```
<ComponentSpec provider="dvb.org" name="ConnectStatusImage">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible enabled size bordercolor h-align v-align"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="margin_properties"/>
        </Properties>
    </ComponentSpec>
```

The **ConnectStatusImage** component serves to define a location within the reference screen where platform specific graphics may be rendered to represent each of the four states supported by the ReturnPath component: opening, open, closing and closed.

NOTE: Graphics used to represent connection status are platform specific, and are not specified by service authors.

A.2.5 HintTextBox

A.2.5.1 Introduction

A **HintTextBox** component renders plain text within a defined rectangular area relevant to the visible component that currently has focus.

```
    <ComponentSpec provider="dvb.org" name="HintTextBox">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible enabled origin
                                                size bordercolor fillcolor textcolor
                                                h-align v-align"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="color_properties-textcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
        </Properties>
    </ComponentSpec>
```

The **HintTextBox** is essentially identical to the TextBox component, except that the text to be displayed is not defined within the **HintTextBox** component itself. Instead the text to be presented by the **HintTextBox** component is defined by the **HintText** property of the visible component that currently has focus. In this way the **HintTextBox** component is to provide a simple mechanism to deliver context-sensitive information. The text presented shall be updated to reflect any change of focus.

EXAMPLE: An input field component may have a HintText property value of "please enter your phone number and press select". This text will be presented by the **HintTextBox** component when focus rests on that input field.

The **HintTextBox** is not a focusable component.

Only one **HintTextBox** instance shall be active at any point in time.

Text shall be rendered within a **HintTextBox** using the same rules as for a TextBox with the wrapping property set to true.

A.2.5.2 Properties defined elsewhere

- intrinsic_properties-visual_components: clause C.1.1;
- border_properties: clause C.4;
- font_properties: clause C.9;
- color_properties-fillcolor: clause C.3.4;
- positioning_properties-absolute: clause C.7.1;
- alignment_properties: clause C.7.3;
- padding_and_margin_properties: clause C.8;
- background properties: clause C.8.

A.2.6 Image

The **Image** component displays a still image.

```
    <ComponentSpec provider="dvb.org" name="Image">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled size content"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="margin_properties"/>
            <PropertySpec name="content" type="imageData" use="required" access="readWrite"/>
            <PropertySpec name="content-focus" type="imageData" use="optional" access="readWrite"/>
            <PropertySpec name="content-disabled" type="imageData" use="optional"
access="readWrite"/>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="media_events"/>
        </GeneratedEvents>
        <GeneratedErrors>
            <GeneratedErrorGroupRef ref="media_errors"/>
        </GeneratedErrors>
    </ComponentSpec>
```

The **visible** property shall define whether or not the Image component is visible.

The **focusable** property shall define whether or not the Image component will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the Image component is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the Image component when it is drawn within an explicit layout container. This shall correspond to the top-left hand corner of the image.

The **size** property shall define the horizontal and vertical dimensions of the Image component. If the image size is different then the image shall not be scaled to fit. Instead:

- If the image size is larger then the parts that fall outside the boundary of the **Image** component (on the right and bottom boundaries) shall not be rendered.
- If the image size is smaller then the pixels within the boundary of the **Image** component that are not defined by the image shall be set to the colour defined by the **fillcolor** property.

The **fillcolor** property defines the fill colour that shall be used between:

- the padding edge and the content edge;

- the content edge and the image, when the image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **border_properties** property group shall define the colour, style, width and corner-radius of the image's border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the image within the content area of the image when the actual image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the image is drawn within a flow.

The **content** property shall define the image to be displayed when the image component is in its default state.

The **content-focus** property may be used to define an image to be displayed when the image component has focus.

The **content-disabled** property may be used to define an image to be displayed when the image component is disabled.

The statechart in figure 78 defines image loading behaviour.

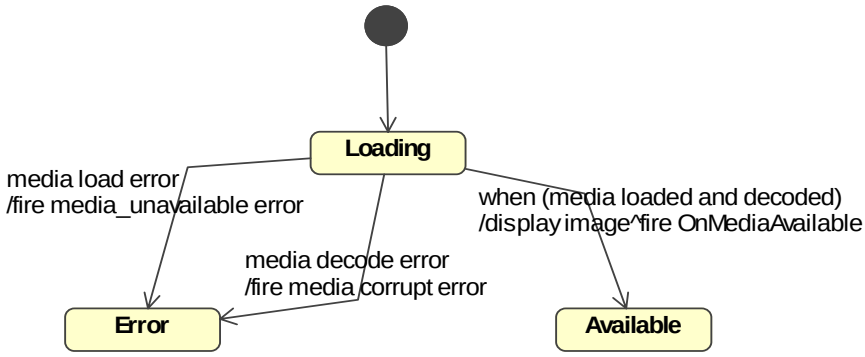

Figure 78: Image loading behaviour

The image is initially loading.

If there is a load error or an image decoding error then a **media_unavailable** or **media_corrupt** error shall be fired respectively.

If the image loads and decodes successfully the image shall become available for display and an **OnMediaAvailable** event shall be fired. The rendering of the image is determined by the **visible** property.

A.2.7 ImageAnimated

The **ImageAnimated** component displays a simple sequence of images.

```
    <ComponentSpec provider="dvb.org" name="ImageAnimated">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled
                                                size bordercolor h-align v-align content pause"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="margin_properties"/>
            <PropertyGroupRef ref="animation_properties"/>
            <PropertySpec name="current-frame" type="integer" use="optional" access="readWrite">
                <pcf:Integer name="default" value="0"/>
            </PropertySpec>
            <PropertySpec name="num-frames" type="integer" access="readOnly"/>
            <PropertySpec name="content" type="imageDataArray" access="initializeOnly"
use="required"/>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="animation_events"/>
```

```
            <GeneratedEventGroupRef ref="media_events"/>
        </GeneratedEvents>
        <GeneratedErrors>
            <GeneratedErrorGroupRef ref="media_errors"/>
        </GeneratedErrors>
    </ComponentSpec>
```

The **visible** property shall define whether or not the ImageAnimated component is visible.

The **focusable** property shall define whether or not the ImageAnimated component will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the ImageAnimated component is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the ImageAnimated component when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the ImageAnimated component. If the dimensions of any image in the sequence are different to the value of the **size** property then it shall not be scaled to fit. Instead:

- If the image size is larger then the parts that fall outside the boundary of the **Image** component (on the right and bottom boundaries) shall not be rendered.
- If the image size is smaller then the areas within the boundary of the **Image** component that are not defined by the image shall be transparent.

The **fillcolor** property defines the fill colour that shall be used between:

- the padding edge and the content edge;
- the content edge and the image, when the image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **border_properties** property group shall define the colour, style, width and corner-radius of the image's border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the image within the content area of the component, when the image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the image is drawn within a flow.

The **animation_properties** property group shall define the way in which the images are animated.

The **frameperiod** property shall define the delay, in milliseconds, between successive images.

The **running** property shall define whether or not the images are currently being animated.

The **number-of-loops** property shall define the number of loops of the animation sequence to show before stopping. If this value is zero then the animation shall loop continuously.

The **pause** property shall define the number of milliseconds to wait between successive loops.

The **current-frame** property, when read, shall return the index of the frame currently being displayed. When written to it shall set the current position within the animation sequence.

The **num-frames** property shall return the total number of frames in the animation sequence.

The **content** property shall define the array of images that make up the animation sequence. The images shall be presented in the order they are defined in this array, starting with the image at the index value of 1.

The statechart in figure 79 illustrates the ImageAnimated component's loading and animation behaviour.

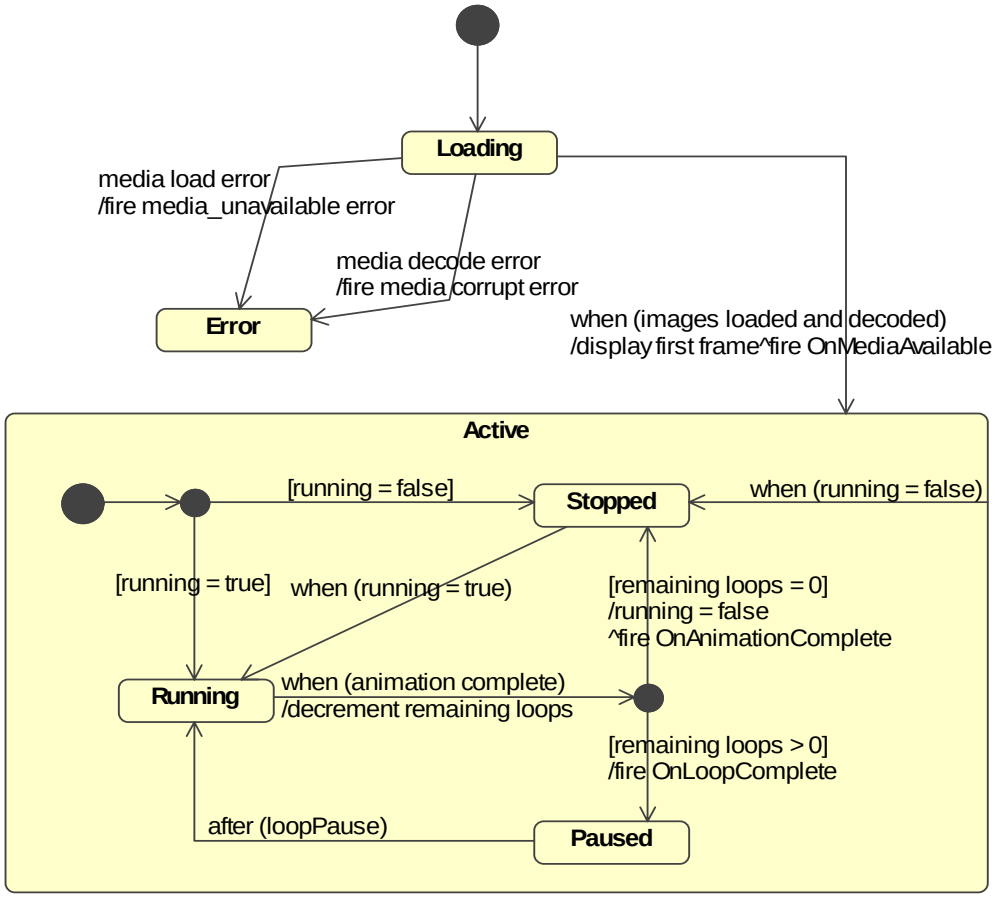

Figure 79: ImageAnimated behaviour

The animation image component is initially in a loading state.

If there is a load error or an image decoding error then a **media_unavailable** or **media_corrupt** error shall be fired respectively.

If all the images load and decode successfully, the first frame of the animation shall become available for display, an **OnMediaAvailable** event shall be fired.

Once available, the ImageAnimated shall be either initially in the running or the stopped state, as defined by the **running** property.

When the animation sequence completes, one of two things shall happen:

- if there are more animation loops to present, as defined by the **number-of-loops** property, the component shall generate an **OnLoopComplete** event and pause for the number of milliseconds defined by the **pause** property, before presenting the next loop in the animation;
- if there are no more animation loops to present the animation shall enter the stopped state with its final frame showing, set its running property to false, and generate an **OnAnimationComplete** event.

A.2.8 ImageScalable

The **ImageScalable** component displays a still image with control over its scaling and presentation within the visible area of the component itself.

```
<ComponentSpec provider="dvb.org" name="ImageScalable">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="visible focusable enabled size content fillcolor "/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
```

```
            <PropertyGroupRef ref="margin_properties"/>
            <PropertySpec name="content" type="imageData" use="required" access="readWrite"/>
            <PropertySpec name="viewport-h-position" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="0 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-v-position" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="0 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-h-size" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-v-size" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="h-scale" type="proportion" use="optional" access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="v-scale" type="proportion" use="optional" access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="anchor" type="enumeration" use="optional" access="readWrite">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="bullseye"/>
            </PropertySpec>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="media_events"/>
        </GeneratedEvents>
        <GeneratedErrors>
            <GeneratedErrorGroupRef ref="media_errors"/>
        </GeneratedErrors>
    </ComponentSpec>
```

The **visible** property shall define whether or not the **ImageScalable** component is visible.

The **focusable** property shall define whether or not the **ImageScalable** component will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the **ImageScalable** component is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the **Image** component when it is drawn within an explicit layout container. This shall correspond to the top-left hand corner of the image.

The **size** property shall define the horizontal and vertical dimensions of the **Image** component. If the image size is different then it shall not be scaled to fit. Instead:

- If the image size is larger then the parts that fall outside the boundary of the **Image** component (on the right and bottom boundaries) shall not be rendered.
- If the image size is smaller then the areas within the boundary of the **Image** component that are not defined by the image shall be transparent.

The **fillcolor** property defines the fill colour that shall be used between:

- the padding edge and the content edge;
- the content edge and the image, when the image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **border_properties** property group shall define the colour, style, width and corner-radius of the image's border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the image within the content area of the image when the actual image dimensions are smaller than the dimensions of the content area and the image is not being scaled.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the image is drawn within a flow.

The **content** property shall define the image to be displayed when the image component is in its default state.

The process by which an area of the decoded image is selected, scaled and subsequently displayed by the ImageScalable component as part of an interactive service is the same as that described for the Video component (see clause A.2.15).

The image loading behaviour for an ImageScalable component is the same as for an Image component (see clause A.2.6).

A.2.9 TextBox

A TextBox component renders plain text within a rectangular area.

```
<ComponentSpec provider="dvb.org" name="TextBox">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="visible focusable enabled origin
                                            size bordercolor fillcolor
                                            textcolor h-align v-align
                                            content wrapping"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="color_properties-textcolor"/>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertyGroupRef ref="font-properties"/>
        <PropertySpec name="content" type="string" use="optional" access="readWrite">
            <pcf:String name="default" value=""/>
        </PropertySpec>
        <PropertySpec name="wrapping" type="boolean" use="optional" access="initializeOnly">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **visible** property defines whether or not the **TextBox** is visible.

The **focusable** property defines whether or not the **TextBox** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **TextBox** is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the **TextBox** when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the **TextBox**.

The **fillcolor** property shall define the fill colour of the **TextBox**.

The **textcolor** property shall define the colour of the text within the **TextBox**.

The **border_properties** property group shall define the colour, style, width and corner-radius of the **TextBox**'s border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the text within the content area of the **TextBox**.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **TextBox** is drawn within a flow.

The **font_properties** define the font to use to display the content text.

The **content** property shall define the textual content to be rendered.

Textual content shall be rendered as a series of lines within the rectangular area defined by the **size** property. The flow of text shall be based on the rules defined in clause 8.3.4 as if the textual content was flow within a paragraph element as defined in annex F. However, the exact way in which the rendering occurs shall be affected by the **wrapping** property as follows:

If the wrapping property is false then:

- The flow of text shall only break onto a new line only where a Carriage Return character (0x0D) is present in the text. Consequently, the number of lines to render shall be equal to the number of Carriage Returns present, plus one.

If the wrapping property is true then:

- In addition to breaking onto a new line where a Carriage Return character (0x0D) is present in the text the flow of text shall automatically break onto successive lines as required to avoid any horizontal truncation of the text content.

Any textual content that flows beyond the bounds of the rectangular area defined by the size property shall be truncated in a platform-specific manner.

A.2.10 Ticker

The **Ticker** component displays a scrolling ticker.

```
    <ComponentSpec provider="dvb.org" name="Ticker">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable enabled origin
                                        bordercolor fillcolor textcolor
                                        h-align v-align loopPause direction content"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="color_properties-textcolor"/>
            <PropertyGroupRef ref="alignment_properties"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertyGroupRef ref="font-properties"/>
            <PropertyGroupRef ref="animation_properties"/>
            <PropertySpec name="content" type="string" use="optional" access="readWrite">
                <pcf:String name="default" value=""/>
            </PropertySpec>
            <PropertySpec name="direction" type="enumeration" use="optional" access="readWrite">
                <EnumerationSpec name="scrollDirection">
                    <EnumerationItem name="left"/>
                    <EnumerationItem name="right"/>
                </EnumerationSpec>
                <pcf:String name="default" value="left"/>
            </PropertySpec>
            <PropertySpec name="startBlank" type="boolean" access="readWrite" use="optional">
                <pcf:Boolean name="default" value="true"/>
            </PropertySpec>
            <PropertySpec name="tickerType" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="tickerTypes">
                    <EnumerationItem name="marquee"/>
                    <EnumerationItem name="reveal"/>
                </EnumerationSpec>
                <pcf:String name="default" value="marquee"/>
            </PropertySpec>
        </Properties>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="animation_events"/>
        </GeneratedEvents>
    </ComponentSpec>
```

The **visible** property defines whether or not the **Ticker** is visible.

The **focusable** property defines whether or not the **Ticker** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property defines whether or not the **Ticker** is eligible to receive focus, due to user navigation.

The **aacolor** property defines the anti-alias colour to use when rendering the text.

The **origin** property shall define the position of the top left-hand corner of the **Ticker** when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the **Ticker**.

The **fillcolor** property shall define the fill colour of the **Ticker**.

The **textcolor** property shall define the colour of the text within the **Ticker**.

The **border_properties** property group shall define the colour, style, width and corner-radius of the **Ticker**'s border.

The **alignment_properties** property group shall define the vertical and horizontal alignment of the text within the content area of the **Ticker**. Only the vertical alignment property shall be used.

The **padding_properties** property group shall define the padding between the content edge and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **Ticker** is drawn within a flow.

The **font_properties** define the font to use to display the content text.

The content property shall define the textual content to display within the **Ticker**. Display of the textual content shall conform to the following rules:

- Line breaks will be ignored.
- Content within a **Ticker** shall not automatically word-wrap. If the **Ticker**Type property is set to "reveal" and the content string is too long to fit into the content area, the rendering behaviour is undefined by the present document.

The **animation_properties** property group shall define the way in which the **Ticker** animates its content.

The **frameperiod** property defines the movement speed in milliseconds per pixel.

NOTE 1: Platforms are free to implement any actual pixel step size provided the perceived movement speed remains as specified by the frameperiod property.

The **running** property defines whether or not the **Ticker** is currently running, i.e. moving.

The **number-of-loops** property defines the number of loops of the **Ticker** content to show before stopping. If this value is zero then the **Ticker** shall loop continuously.

The **loopPause** property defines the number of milliseconds to wait between successive loops. During this pause the content shall remain visible.

The **direction** property shall define the direction in which the content moves as it scrolls.

NOTE 2: The default direction property, "left", is most suitable for left-to-right script in a marquee. Service authors may change this value for right-to-left script or when **Ticker**Type is set to "reveal".

The **startBlank** property shall define whether or not each loop should start with as much content as possible pre-positioned within the **Ticker**, or whether the **Ticker** should begin the loop blank.

NOTE 3: Where the **Ticker**Type property is set to "reveal" the **Ticker** will always start blank.

The statechart in figure 80 defines the animation behaviour for the **Ticker** component.

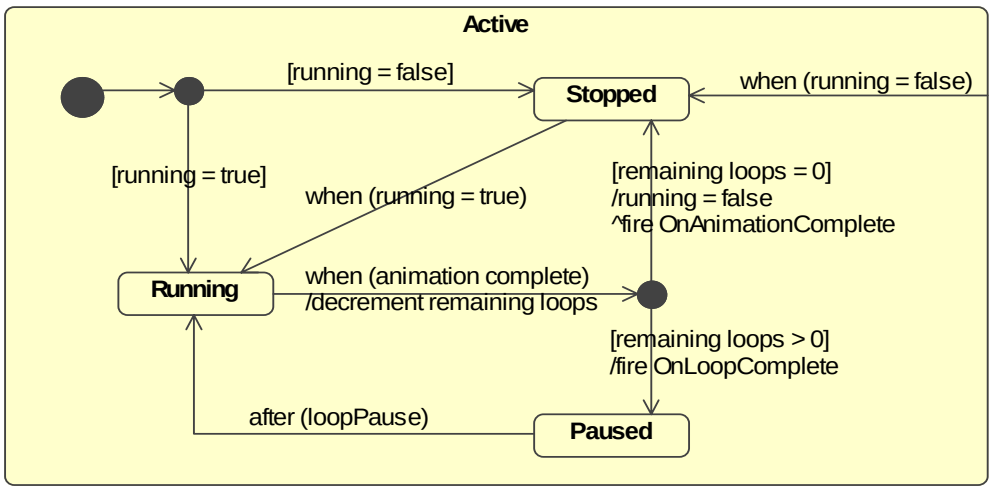

Figure 80: Ticker animation behaviour

This statechart defines the animation-related behaviour when the **Ticker** is **active**, that is, when its parent scene is the active scene. This statechart operates concurrently with the standard visual component statechart.

The **Ticker** is either initially in the running or stopped state, as defined by the **running** property.

When one **Ticker** animation loop sequence completes, one of two things shall happen:

- if there are more animation loops to present, as defined by the **number-of-loops** property, the **Ticker** shall generate an **OnLoopComplete** event and pause for the number of milliseconds defined by the **pause** property, before presenting the next loop in the animation;
- if there are no more animation loops to present the **Ticker** shall enter the stopped state, set its running property to false, and generate an **OnAnimationComplete** event.

NOTE 4: It should be noted that the **paused** state is a transitory state that is entered automatically at the end of each loop, and exited after **loopPause** milliseconds. It should not be confused with the concept of manually pausing a video or audio player transport.

A.2.11 Input components

A.2.11.1 Button

A **Button** component shall draw a vertical and horizontal axis-aligned rectangle with a specified origin and size, and shall superimpose a single line of text and an image if specified.

```
<ComponentSpec provider="dvb.org" name="Button">
        <Overview version="1.01"/>
        <IntendedImplementation coreProperties="visible enabled focusable origin size label image
fillcolor fillcolor-active textcolor selected selectmode hotkey"/>
        <Properties>
            <!--visual properties-->
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="font_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <!--optional button label text-->
            <PropertySpec name="content" type="string" use="optional" access="readWrite"/>
            <!--optional button image-->
            <PropertySpec name="image" type="imageData" use="optional" access="initializeOnly"/>
            <PropertySpec name="image_align" type="enumeration" use="optional"
access="initializeOnly">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="bullseye"/>
            </PropertySpec>
```

```
            <!--behavioural properties-->
            <PropertySpec name="selected" type="boolean" access="readWrite"/>
            <PropertySpec name="selectmode" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="selectmodes">
                    <EnumerationItem name="pushbutton"/>
                    <EnumerationItem name="toggle"/>
                    <EnumerationItem name="oneshot"/>
                </EnumerationSpec>
                <pcf:String name="default" value="pushbutton"/>
            </PropertySpec>
            <PropertySpec name="hotkey" type="userKey" access="initializeOnly" use="optional"/>
        </Properties>
        <HandledEvents>
            <HandledEventSpec name="select" eventtype="KeyEvent" eventclass="user">
                <Qualifier>
                    <pcf:UserKey name="key" value="VK_ENTER"/>
                </Qualifier>
            </HandledEventSpec>
            <HandledEventSpec name="hotkey" eventtype="KeyEvent" eventclass="user">
                <Qualifier>
                    <PropertyRef componentproperty="hotkey" eventproperty="key"/>
                </Qualifier>
            </HandledEventSpec>
        </HandledEvents>
        <HandledActions>
            <HandledActionGroupRef ref="visible-actions"/>
        </HandledActions>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="button_events"/>
        </GeneratedEvents>
        <GeneratedErrors>
            <GeneratedErrorGroupRef ref="basic_errors"/>
            <GeneratedErrorGroupRef ref="media_errors"/>
        </GeneratedErrors>
    </ComponentSpec>
```

The **visible** property shall define whether or not the **Button** is visible.

The **focusable** property shall define whether or not the **Button** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the **Button** is eligible to receive focus, due to user navigation. In addition the **Button** shall be prevented from changing active state while disabled.

The **origin** property shall define the position of the top left-hand corner of the rectangle.

The **size** property shall define the horizontal and vertical dimensions of the rectangle.

The **color_properties-fillcolor** property group shall define the fill colour of the rectangle in at least the idle and active states. Implementations may also use other colours in the group to show other states as described in clause 7.3.3.

The **border_properties** property group shall define the colour, style, width and corner-radius of the rectangle's border.

The **aacolor** property shall define the anti-alias colour to use when rendering the text.

The **textcolor** property shall define the colour of the text within the **Button**.

The **padding_properties** property group shall define the padding between the content edge of the text label and image and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **Button** is drawn within a flow.

The **font_properties** shall define the font to use to display the content text.

The **label** property shall define the textual content to display within the **Button**. Display of the textual content shall conform to the following rules:

- Label text within a **Button** shall not automatically word-wrap. If a line of text is too long to fit into the content area of the **Button** the rendering behaviour is undefined by the present document.

The **image** property may be used to specify an image resource to display in addition to the text label and background rectangle.

The **image-align** property may be used to specify the relative alignment of the image and the text label within the content area of the **Button**. The image may be placed north, south, east or west of the text, or the text may be centrally aligned with the image [default]. The resulting content block is then centered within the content area of the **Button**. There shall be an implicit z-order such that the text appears in front of the image and the image appears in front of the rectangle. This order cannot be changed.

The **selected** property shall enable action language access to the active state of the **Button**.

The **selectmode** property shall control the behaviour of the **Button**. Setting this to *pushbutton* shall cause intermittent action, setting this to *toggle* shall cause the active state to remain until another select event occurs and setting this to *one-shot* shall cause the **Button** to only respond to the first select event.

The **hotkey** property shall allow the **Button** to respond to the key event specified by this value even if the **Button** does not have focus.

A **Button** shall behave according to two separate statemachines; the standard intrinsic focus machine and an activity statemachine that defines the response to the "select" and "hotkey" events.

NOTE: The standard intrinsic focus machine is shared with all other visible, focusable components and is described in clause 7.4.

The activity statemachine is shown in figure 81.

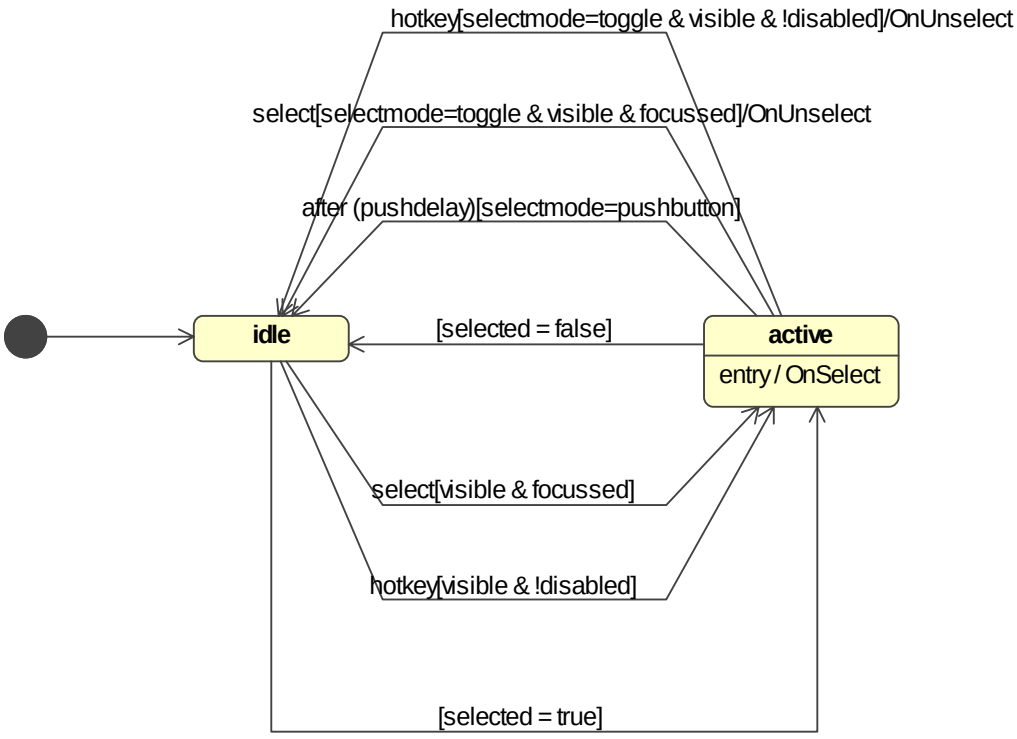

Figure 81: Button statemachine

A **Button** shall always initialize into the **idle** state.

A **Button** shall respond to the select event only when it is visible and has focus.

A **Button** shall respond to the hotkey event only when it is visible and is not disabled.

A **Button** in the active state with selectmode set to "pushbutton" shall become idle after a platform-specific delay called "pushdelay".

A **Button** in the active state with selectmode set to "toggle" shall become idle after a select or hotkey event.

A **Button** in the active state with selectmode set to "oneshot" shall not respond to any further events.

A.2.11.2 PickList

When not active, a **PickList** component shall draw a vertical and horizontal axis-aligned rectangle with a specified origin and size, and shall superimpose a single line of text representing the selected item.

There shall be a visual indication that this value may be adjusted by the user using up and down actions.

When active, a **PickList** component shall draw a pop-up rectangular area anchored to the original display rectangle and at the top of the Z order. This pop-up shall be large enough to contain the defined list of textual items, and highlight the currently selected item. It shall respond to user input to adjust which item in the list is highlighted, and to select the currently highlighted item as the selected item.

```
<ComponentSpec provider="dvb.org" name="PickList">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties=" visible enabled focusable origin size fillcolor-active
textcolor items selected"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="border_properties"/>
        <PropertyGroupRef ref="font_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertySpec name="items" type="stringArray" access="initializeOnly"/>
        <PropertySpec name="selection" type="integer" access="readWrite"/>
        <PropertySpec name="pivot" type="enumeration" access="initializeOnly">
            <EnumerationRef ref="relative-positions"/>
        </PropertySpec>
    </Properties>
    <HandledEvents>
        <HandledEventSpec name="select" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_ENTER"/>
            </Qualifier>
        </HandledEventSpec>
        <HandledEventSpec name="up" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_UP"/>
            </Qualifier>
        </HandledEventSpec>
        <HandledEventSpec name="down" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_DOWN"/>
            </Qualifier>
        </HandledEventSpec>
    </HandledEvents>
    <GeneratedEvents>
        <GeneratedEventGroupRef ref="choice_events"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorGroupRef ref="basic_errors"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **visible** property shall define whether or not the **PickList** is visible.

The **focusable** property shall define whether or not the **PickList** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the **PickList** is eligible to receive focus, due to user navigation. In addition the **PickList** shall be prevented from changing value while disabled.

The **origin** property shall define the position of the top left-hand corner of the rectangle.

The **size** property shall define the horizontal and vertical dimensions of the rectangle.

The **color_properties-fillcolor** property group shall define the fill colour of the rectangle. Implementations may also use other colours in the group to show other states as described in clause 7.3.3.

The **border_properties** property group shall define the colour, style, width and corner-radius of the rectangle's border.

The **aacolor** property shall define the anti-alias colour to use when rendering the text.

The **textcolor** property shall define the colour of the text within the **PickList**.

The **padding_properties** property group shall define the padding between the content edge of the item text and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **PickList** is drawn within a flow.

The **font_properties** shall define the font to use to display the item text.

The **items** property shall define an array of strings. It shall be an error if any of these strings contain line breaks.

The **selection** property is an integer that shall define which item in the items array is currently selected.

Display of the item text shall conform to the following rules:

- If the selected item text is too large to fit into the content area of the **PickList** the rendering behaviour is undefined by the present document.
- If the number of items in the **PickList** is too large to display using a pop-up rectangle the rendering behaviour is undefined by the present document.

The **pivot** property shall define the location of the pop-up relative to the rectangle. The value shall identify which side or corner of the rectangle and pop-up area shall be anchored together.

A **PickList** shall behave according to two separate statemachines; the standard intrinsic focus machine and an activity statemachine that defines the response to the "select", "up" and "down" events.

NOTE: The standard intrinsic focus machine is shared with all other visible, focusable components and is described in clause 7.5.2.

The activity statemachine is shown in figure 82.

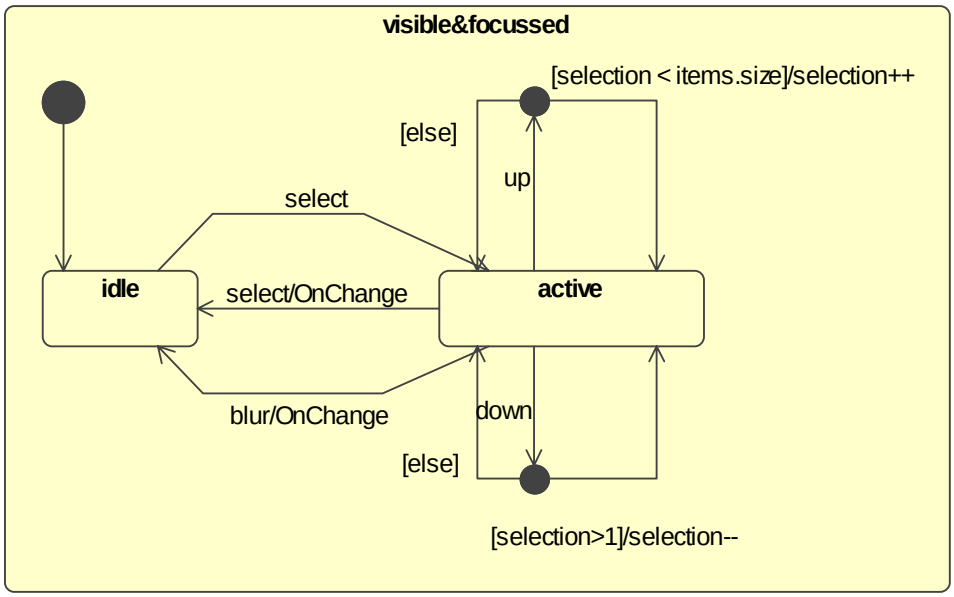

Figure 82: PickList statemachine

A **PickList** shall only respond to events if it both visible and focused.

A **PickList** shall initialize into the idle state.

A **PickList** shall respond to the select event when it is in the idle state.

A **PickList** shall respond to the select, up and down events when it is in the active state.

A **PickList** shall generate OnChange events each time it transitions from active to idle states.

A **PickList** shall transition from active to idle states when it loses focus.

A.2.11.3 RadioButtonGroup

A **RadioButtonGroup** is a specialization of an explicit layout container that contains behaviour appropriate to co-ordinate a collection of child **Button** components such that only one is selected at any time.

```
<ComponentSpec provider="dvb.org" name="RadioButtonGroup" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="origin selected"/>
    <Properties>
        <PropertySpec name="origin" type="position" access="readWrite"/>
        <PropertySpec name="selected" type="string" access="readOnly"/>
    </Properties>
</ComponentSpec>
```

A **RadioButtonGroup** may only contain **Button** components.

A **RadioButtonGroup** does not have any visual representation itself, nor does it consume user events.

The property **selected** shall expose the name of the child **Button** component that is currently selected i.e. it has the value checked.name. This value may be "nil" if no child component is selected.

A **RadioButtonGroup** shall behave according to the statemachine illustrated in figure 83.

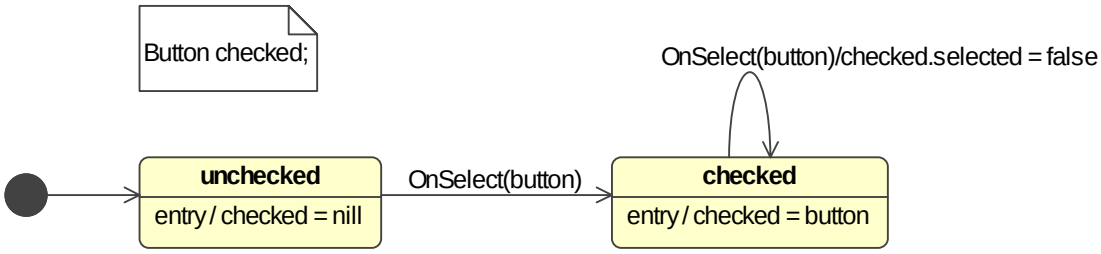

Figure 83: RadioButtonGroup statemachine

A **RadioButtonGroup** shall respond to the OnSelect events generated by the **Button** components contained within it. See clause A.2.11.1 for the definition of the **Button** component.

When initialized, a **RadioButtonGroup** shall set its internal state variable "checked" to the value "nil".

On receipt of an OnSelect event, if the **RadioButtonGroup** is in the unchecked state it shall become checked. If the **RadioButtonGroup** was already in the checked state it shall set the "selected" property of the **Button** referenced by the checked variable to "false".

On entering the checked state the **RadioButtonGroup** shall set the state variable "checked" to the **Button** that generated the OnSelect event.

A.2.11.4 SpinControl

A **SpinControl** component shall draw a vertical and horizontal axis-aligned rectangle with a specified origin and size, and shall superimpose a single line of text representing an integer value. There shall be a visual indication that this value may be adjusted by the user using up and down actions.

```
    <ComponentSpec provider="dvb.org" name="SpinControl">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible enabled focusable origin size min max
fillcolor fillcolor-active textcolor value"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="font_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
```

```
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertySpec name="min" type="integer" use="required" access="readWrite"/>
        <PropertySpec name="max" type="integer" use="required" access="readWrite"/>
        <PropertySpec name="value" type="integer" use="required" access="readWrite"/>
    </Properties>
    <HandledEvents>
        <HandledEventSpec eventtype="KeyEvent" eventclass="user" name="increment">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_UP"/>
            </Qualifier>
        </HandledEventSpec>
        <HandledEventSpec name="decrement" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_DOWN"/>
            </Qualifier>
        </HandledEventSpec>
    </HandledEvents>
    <GeneratedEvents>
        <GeneratedEventGroupRef ref="choice_events"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorGroupRef ref="basic_errors"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **visible** property shall define whether or not the **SpinControl** is visible.

The **focusable** property shall define whether or not the **SpinControl** will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the **SpinControl** is eligible to receive focus, due to user navigation. In addition the **SpinControl** shall be prevented from changing value while disabled.

The **origin** property shall define the position of the top left-hand corner of the rectangle.

The **size** property shall define the horizontal and vertical dimensions of the rectangle.

The **color_properties-fillcolor** property group shall define the fill colour of the rectangle. Implementations may also use other colours in the group to show other states as described in clause 7.3.3.

The **border_properties** property group shall define the colour, style, width and corner-radius of the rectangle's border.

The **aacolor** property shall define the anti-alias colour to use when rendering the text.

The **textcolor** property shall define the colour of the text within the **SpinControl**.

The **padding_properties** property group shall define the padding between the content edge of the text value and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the **SpinControl** is drawn within a flow.

The **font_properties** shall define the font to use to display the value text.

The **min** property shall define the lower bound of the range over which the value may be adjusted by the user. It is an error to attempt to set it larger than the max value.

The **max** property shall define the upper bound of the range over which the value may be adjusted by the user. It shall be an error to attempt to set it smaller than the min value.

The **value** property shall define the integer value displayed within the **SpinControl**. It shall remain between the values defined by the min and max properties. It is an error to attempt to set it outside this range.

NOTE 1: Range errors as described here can be generated during both transcoding and at run-time.

Display of the textual representation of the value property shall conform to the following rules:

- If the text representation of the value is too long to fit into the content area of the **SpinControl** the rendering behaviour is undefined by the present document.

A **SpinControl** shall behave according to two separate statemachines; the standard intrinsic focus machine and an activity statemachine that defines the response to the "up" and "down" events.

NOTE 2: The standard intrinsic focus machine is shared with all other visible, focusable components and is described in clause 7.5.2.

The activity statemachine is shown in figure 84.

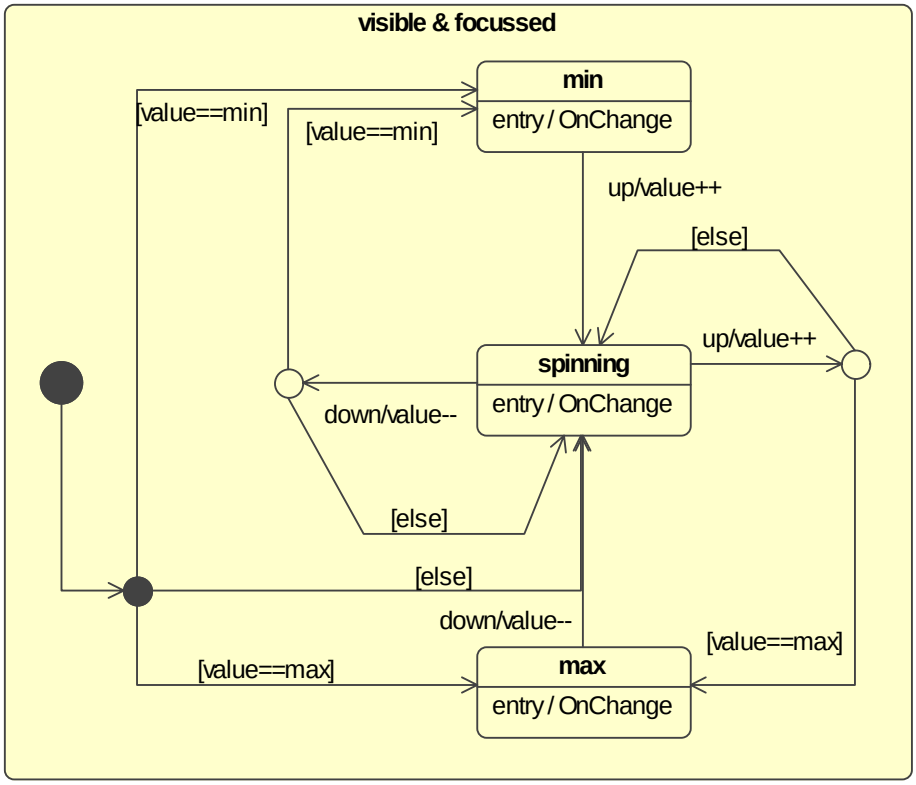

Figure 84: SpinControl statemachine

A **SpinControl** shall only respond to events if it both visible and focused.

A **SpinControl** shall initialize into the appropriate state according to the value in the value property.

A **SpinControl** shall respond to the up and down events when it is in the spinning state.

A **SpinControl** shall respond only to the up event when it is in the min state.

A **SpinControl** shall respond only to the down event when it is in the max state.

NOTE 3: A **SpinControl** shall not consume a VK_UP KeyEvent when in the max state, or a VK_DOWN KeyEvent when in the min state.

A **SpinControl** shall generate OnChange events each time its value property is altered by the user using up and down events.

A.2.11.5 TextInput

A **TextInput** component shall draw a vertical and horizontal axis-aligned rectangle with a specified origin and size, and shall superimpose a single line of text representing the user entered value.

```
<ComponentSpec provider="dvb.org" name="TextInput">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="origin size min max fillcolor value"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
        <PropertyGroupRef ref="border_properties"/>
```

```
        <PropertyGroupRef ref="font_properties"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
        <PropertyGroupRef ref="positioning_properties-absolute"/>
        <PropertyGroupRef ref="padding_and_margin_properties"/>
        <PropertySpec name="value" type="string" use="optional" access="readWrite"/>
        <PropertySpec name="valuesize" type="integer" use="optional" access="initializeOnly"/>
        <PropertySpec name="datatype" type="enumeration" use="required" access="initializeOnly">
            <EnumerationSpec name="textentrytypes">
                <EnumerationItem name="string"/>
                <EnumerationItem name="integer"/>
            </EnumerationSpec>
        </PropertySpec>
        <PropertySpec name="hotkeys" type="userKeyArray" access="initializeOnly"/>
    </Properties>
    <HandledEvents>
        <HandledEventSpec name="userkey" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
    <           pcf:UserKeyArray name="key">
                    <pcf:UserKey value="VK_0"/>
                    <pcf:UserKey value="VK_1"/>
                    <pcf:UserKey value="VK_2"/>
                    <pcf:UserKey value="VK_3"/>
                    <pcf:UserKey value="VK_4"/>
                    <pcf:UserKey value="VK_5"/>
                    <pcf:UserKey value="VK_6"/>
                    <pcf:UserKey value="VK_7"/>
                    <pcf:UserKey value="VK_8"/>
                    <pcf:UserKey value="VK_9"/>
                    <pcf:UserKey value="VK_UNKNOWN"/>
                </pcf:UserKeyArray>
            </Qualifier>
        </HandledEventSpec>
        <HandledEventSpec name="hotkey" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <PropertyRef componentproperty="hotkeys" eventproperty="key"/>
            </Qualifier>
        </HandledEventSpec>
        <HandledEventSpec name="delete" eventtype="KeyEvent" eventclass="user">
            <Qualifier>
                <pcf:UserKey name="key" value="VK_LEFT"/>
            </Qualifier>
        </HandledEventSpec>
    </HandledEvents>
    <GeneratedEvents>
        <GeneratedEventGroupRef ref="text_input_events"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorGroupRef ref="basic_errors"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **visible** property shall define whether or not the textinput is visible.

The **focusable** property shall define whether or not the textinput will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the textinput is eligible to receive focus, due to user navigation. In addition the textinput shall be prevented from changing value while disabled.

The **origin** property shall define the position of the top left-hand corner of the rectangle.

The **size** property shall define the horizontal and vertical dimensions of the rectangle.

The **color_properties-fillcolor** property group shall define the fill colour of the rectangle. Implementations may also use other colours in the group to show other states as described in clause 7.3.3.

The **border_properties** property group shall define the colour, style, width and corner-radius of the rectangle's border.

The **aacolor** property shall define the anti-alias colour to use when rendering the text.

The **textcolor** property shall define the colour of the text within the textinput.

The **padding_properties** property group shall define the padding between the content edge of the text and the border.

The **margin_properties** property group shall define the margin between the border edge and the containing block when the textinput is drawn within a flow.

The **font_properties** shall define the font to use to display the text.

The **value** property shall define and expose to action language the text entered by the user.

The **valuesize** property shall define the maximum number of characters that can be entered by the user.

The **datatype** property shall define the expected format of the entered text and shall prevent invalid characters being entered or consumed.

The **hotkeys** property shall define an array of key codes to which the textinput shall respond if it does not have focus.

Display of the text shall conform to the following rules:

- If the text is too large to fit into the content area of the textinput the rendering behaviour is undefined by the present document.

A textinput shall behave according to two separate statemachines; the standard intrinsic focus machine and an activity statemachine that shall define the response to the user key events.

NOTE: The standard intrinsic focus machine is shared with all other visible, focusable components and is described in clause 7.5.2.

The activity statemachine is shown in figure 85.

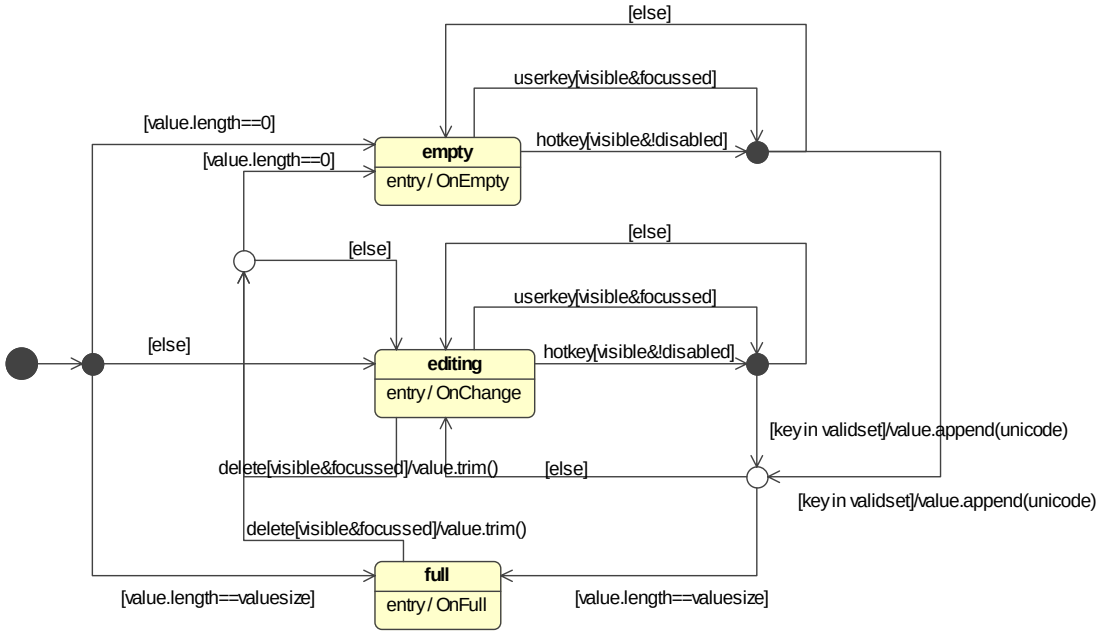

Figure 85: textinput statemachine

Userkey events are any KeyEvent where the key property is in the set VK_0 to VK_9 or has the value VK_UNKNOWN and the Unicode value is an alphanumeric i.e. not a navigation key.

Hotkey events are any KeyEvent where the key property matches one of the values in the hotkeys array property.

A textinput shall respond to delete and userkey events if it both visible and focused.

A textinput shall respond to hotkey events if it is both visible and not disabled.

A textinput shall initialize into the appropriate state depending on the length of the value property.

A textinput shall respond to the userkey and hotkey events if it is not in the full state. If the key carried in the event is valid for the set defined by the datatype property then the events unicode property value shall be appended to the textinput value. KeyEvent valid sets are shown below.

Datatype	Valid KeyEvent values
String	VK_0 to VK_9, VK_UNKNOWN containing a printable Unicode character
Number	VK_0 to VK_9

A textinput shall respond to the delete event if it is not in the empty state. A delete event shall trim the last character from the value property.

A textinput shall generate OnChange events each time the value property is modified by the user.

A textinput shall generate an OnFull event when it reaches the full state.

A textinput shall generate an OnEmpty event when it reaches the empty state.

A.2.12 Menu

A.2.12.1 Introduction

The following XML defines the PCF menu component:

```
<?xml version="1.0" encoding="UTF-8"?>
<ComponentSpecs xmlns="http://www.dvb.org/pcf/components" xmlns:pcf="http://www.dvb.org/pcf/pcf"
xmlns:pcf-types="http://www.dvb.org/pcf/pcf-types" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.dvb.org/pcf/components

component-syntax.xsd">
    <ComponentSpec provider="dvb.org" name="Menu">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="origin size fillcolor textcolor selected
selectmode"/>
        <Properties>
            <!--visual properties-->
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="font_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>
            <PropertySpec name="menuAlign" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="menu_align">
                    <EnumerationItem name="vertical"/>
                    <EnumerationItem name="horizontal"/>
                </EnumerationSpec>
                <pcf:String name="default" value="vertical"/>
            </PropertySpec>
            <PropertySpec name="image" type="imageData" access="initializeOnly" use="optional"/>
            <PropertySpec name="imageAlign" type="enumeration" use="optional"
access="initializeOnly">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="west"/>
            </PropertySpec>
            <PropertySpec name="labelArray" type="stringArray" access="initializeOnly"
use="required"/>
            <!--behavioural properties-->
            <PropertySpec name="targetArray" type="uriArray" access="initializeOnly"
use="required"/>
            <PropertySpec name="initalLabel" type="integer" access="initializeOnly" use="optional"/>
            <PropertySpec name="index" type="integer" access="readOnly"/>
            <PropertySpec name="target" type="uri" access="readOnly"/>

            <PropertySpec name="menuLoop" type="boolean" access="initializeOnly" use="optional">
                <pcf:Boolean name="default" value="true"/>
            </PropertySpec>
            <PropertySpec name="selectmode" type="enumeration" access="initializeOnly"
use="optional">
                <EnumerationSpec name="selectmodes">
                    <EnumerationItem name="pushbutton"/>
                    <EnumerationItem name="toggle"/>
                </EnumerationSpec>
                <pcf:String name="default" value="pushbutton"/>
            </PropertySpec>
```

```
        </Properties>
        <HandledEvents>
            <HandledEventSpec name="prev" eventtype="UserKey" eventclass="user"/>
            <HandledEventSpec name="next" eventtype="UserKey" eventclass="user"/>
            <HandledEventSpec name="select" eventtype="UserKey" eventclass="user">
                <Qualifier>
                    <pcf:UserKey name="key" value="VK_ENTER"/>
                </Qualifier>
            </HandledEventSpec>
        </HandledEvents>
        <GeneratedEvents>
            <GeneratedEventGroupRef ref="button_events"/>
        </GeneratedEvents>
        <GeneratedErrors>
            <GeneratedErrorGroupRef ref="media_errors"/>
        </GeneratedErrors>
    </ComponentSpec>
</ComponentSpecs>
```

The **menu** component provides a mechanism for users to navigate and select one of a set of options presented in a visually coherent, ordered list. Only one option may be highlighted or selected at a time.

The menu consists of an array of label items that are represented by text strings and optionally a background image that tracks the presently highlighted label**.**

The **labelArray** shall define the text strings to be presented for each label item in the menu. If any of these strings cannot be rendered on one line within the declared size of the menu, the PCF does not specify the rendering behaviour.

Menus may be vertically or horizontally aligned. Labels in a menu are grouped in a table-like structure, consisting of a single column or a single row. Menus shall have a specified **origin**, and may optionally have a specified **size**.

NOTE 1: If no size is specified, the platform should use the labels' text strings and font to determine a suitable dimension.

Spacing of labels within a menu is not explicitly defined. This shall be determined instead from the text font specifications.

Label items within a menu shall be ordered from first to last according to their order of their declaration. This shall correspond to top to bottom or right to left ordering when rendered.

Initial highlight within a menu shall fall on the first label, unless an **initialLabel** property value has been defined for the menu and lies within the range of the number of items declared in the labelArray.

NOTE 2: Only the highlighted label shall be rendered using any **-focus** related text and colour properties if defined. Other labels shall be rendered with the default values. In addition only the highlighted label shall be rendered using any **-active** or **-idle** properties.

By default, navigation within a menu shall loop, meaning that in a vertically aligned menu, a down arrow navigation action when the highlight is on the bottom most label will result in the highlight moving to the topmost label. Service authors may control this behaviour with the **menuLoop** property.

A.2.12.2 Properties defined elsewhere

- intrinsic_properties-visual_components: clause C.1.1;
- border_properties: clause C.4;
- font_properties: clause C.9;
- color_properties-fillcolor: clause C.3.4;
- positioning_properties-absolute: clause C.7.1;
- alignment_properties: clause C.7.3;
- padding_and_margin_properties: clause C.8.

A.2.12.3 The labelArray property

The **labelArray** string array specifies the set of text strings to be presented in the menu, ordered from first to last.

A.2.12.4 The targetArray property

The **targetArray** property shall define an array of URI values containing navigation targets. Each entry in the targetArray property shall have a corresponding entry in the labelArray property.

If the length of the labelArray property is not equal to the length of the targetArray property the behaviour of **Menu** is undefined by the present document.

A.2.12.5 The initialLabel property

The **initialLabel** property defines an integer that indicates which menu label shall be highlighted upon initialization of the menu, where zero corresponds to the first label. If no value is specified then, by default, the first item in the labelArray shall be highlighted.

A.2.12.6 The index property

The **index** property provides for run-time indication of the currently highlighted label, where zero corresponds to the first label.

A.2.12.7 The target property

The **target** property provides a read only value of the target value that corresponds to the currently highlighted menu label.

A.2.12.8 The menuAlign property

The **menuAlign** property specifies whether the menu should be aligned horizontally or vertically. The default value is "vertical" for a vertically aligned menu consisting of a single column of labels.

A.2.12.9 The menuLoop property

The **menuLoop** property specifies menu loop navigation behaviour. Where the menuLoop property has a value of "true", highlight behaviour shall loop around the menu.

EXAMPLE: In a vertically aligned menu with the default menuLoop property value of "true", the down button shall move highlight to the next label in the menu. When the highlight reaches the last label in the menu, a down button press shall move highlight to the first label in the menu.

A.2.12.10 The selectmode property

The selectmode property sets the select behaviour for all labels of the menu component. This is identical to that defined for the button component.

See clause A.2.11.1 for description of the button selectmode property.

A.2.12.11 The image property

The **image** property specifies a background image to be presented behind or next to the text of the highlighted menu label. See also **imageAlign**.

A.2.12.12 The imageAlign property

The imageAlign property sets the alignment of the image (if any) with the label text in the menu, as it does for the button component.

NOTE: Only the highlighted label will be rendered with an image. Other labels will only render their text values.

See clause A.2.11.1 for description of the button imageAlign property.

A.2.12.13 Behaviour specification

The menu behaviour is specified in the statechart displayed as figure 86.

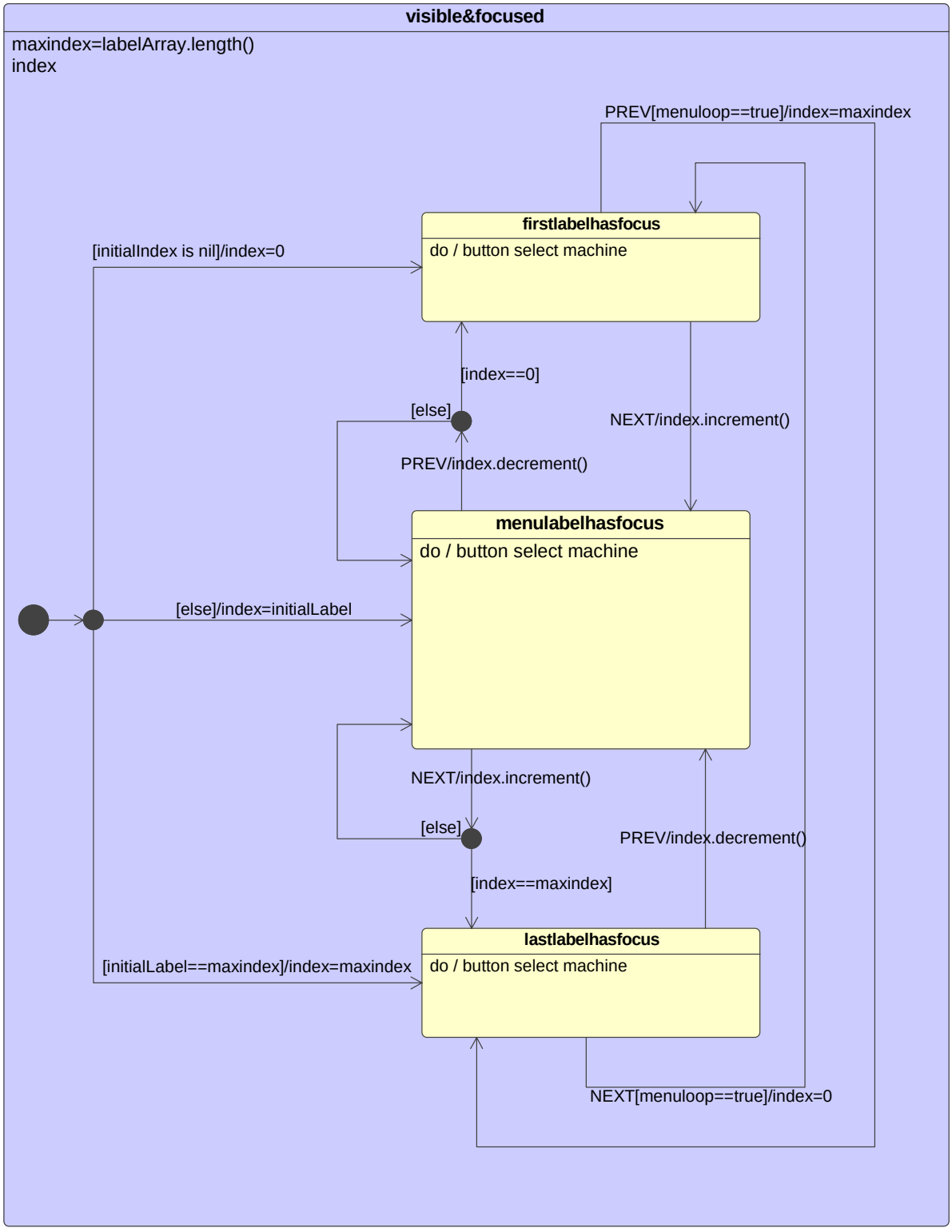

Figure 86: Menu component statechart

On entry to the visible and focused state, the menu shall highlight the appropriate label as required by the initialLabel property.

On navigation actions the menu shall move the highlight as required by the menuLoop property, or shall release the focus.

On select action the menu shall behave as specified for the button component. In addition if there is a non-nil value in the targetArray[index] field the component shall invoke a navigation action using the URI value. This shall be equivalent to invoking the following action language statement:

```
sceneNavigate(targetArray[index], "remember" , nil);
```

This action shall occur in place of generating an OnSelect event.

A.2.13 NumericNavigator

A.2.13.1 Introduction

A **NumericNavigator** component provides a mechanism for users to navigate through a service, or to trigger events using numeric keys.

The following XML fragment defines the PCF **NumericNavigator** component:

```
<ComponentSpec provider="dvb.org" name="NumericNavigator">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="origin size fillcolor value valueSize valueArray
targetArray"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="border_properties"/>
            <PropertyGroupRef ref="font_properties"/>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertyGroupRef ref="padding_and_margin_properties"/>

            <PropertySpec name="value" type="integer" access="readWrite" use="required"/>
            <PropertySpec name="valueSize" type="integer" access="initializeOnly" use="required"/>
            <PropertySpec name="valueArray" type="integerArray" access="initializeOnly"
use="required"/>
            <PropertySpec name="targetArray" type="uriArray" access="initializeOnly"
use="required"/>
            <PropertySpec name="descriptionArray" type="stringArray" access="initializeOnly"
use="optional"/>
            <PropertySpec name="target" type="uri" access="readOnly" use="required"/>
            <PropertySpec name="invalidMessage" type="string" access="final" use="optional">
                <pcf:String name="default" value=""/>
            </PropertySpec>
            <PropertySpec name="description" type="string" access="readOnly" use="optional"/>
        </Properties>

        <HandledEvents>
            <HandledEventSpec name="numberKey" eventtype="KeyEvent" eventclass="user">
            </HandledEventSpec>
            <HandledEventSpec name="select" eventtype="KeyEvent" eventclass="user">
                <Qualifier>
                    <pcf:UserKey name="key" value="VK_ENTER" />
                </Qualifier>
            </HandledEventSpec>
            <HandledEventSpec name="delete" eventtype="KeyEvent" eventclass="user">
                <Qualifier>
                    <pcf:UserKey name="key" value="VK_DELETE" />
                </Qualifier>
            </HandledEventSpec>
        </HandledEvents>

        <GeneratedEvents>
            <GeneratedEventSpec eventtype="OnValid"/>
            <GeneratedEventSpec eventtype="OnInvalid"/>
            <GeneratedEventSpec eventtype="OnSelect"/>
        </GeneratedEvents>
```

```
        <GeneratedErrors>
            <GeneratedErrorSpec errortype="unknown_error"/>
        </GeneratedErrors>
</ComponentSpec>
```

A.2.13.2 Properties defined elsewhere

- intrinsic_properties-visual_components: clause C.1;
- border_properties: clause C.4;
- font_properties: clause C.9;
- color_properties-fillcolor: clause C.3.4;
- positioning_properties-absolute: clause C.7.1;
- padding_and_margin_properties: clause C.8.

A.2.13.3 The value property

```
<PropertySpec name="value" type="integer" access="readWrite" use="required" default="0"/>
```

The **value** property shall define and expose to action language the text entered by the user and shall prevent invalid characters being entered or consumed.

A.2.13.4 The valueSize property

```
<PropertySpec name="valueSize" type="integer" access="initializeOnly" use="required"/>
```

The **valueSize** property shall define the maximum number of numbers that can be entered by the user.

A.2.13.5 The valueArray property

```
<PropertySpec name="valueArray" type="IntegerArray" access="initializeOnly" use="required"/>
```

The **valueArray** property shall define an array of integers representing the valid set of navigation numbers. Each entry in the valueArray property shall have a corresponding entry in the targetArray property.

A.2.13.6 The targetArray property

```
<PropertySpec name="targetArray" type="URIArray" access="initializeOnly" use="required"/>
```

The **targetArray** property shall define an array of URI values containing navigation targets. Each entry in the targetArray property shall have a corresponding entry in the valueArray property.

If the length of the valueArray property is not equal to the length of the targetArray property the behaviour of **NumericNavigator** is undefined by the present document.

A.2.13.7 The descriptionArray property

```
<PropertySpec name="descriptionArray" type="StringArray" access="initializeOnly" use="optional"
default=""/>
```

The **descriptionArray** property may define textual descriptions of each of the values defined in the valueArray property.

A.2.13.8 The target property

```
<PropertySpec name="target" type="URI" access="readOnly" use="required"/>
```

The **target** property shall define and expose to action language the target URI chosen by the user.

A.2.13.9 The invalidMessage property

```
<PropertySpec name="invalidMessage" type="string" access="final" use="optional" default=""/>
```

The **invalidMessage** property may define and expose to action language the string to display when an invalid value has been entered.

A.2.13.10 The description property

```
<PropertySpec name="description" type="string" access="readOnly" use="optional" default=""/>
```

The **description** property may define and expose to action language a textual description of the value entered.

A.2.13.11 Behaviour specification

When active, a **NumericNavigator** component shall draw a rectangle with a specified origin and size. This rectangle shall be large enough to contain a single line of text representing the user entered value.

NOTE 1: If the text is too large to fit into the rectangle of the **NumericNavigator** the rendering behaviour is undefined by the present document.

A **NumericNavigator** shall behave according to two separate statemachines; the standard intrinsic focus machine and an activity statemachine that shall define the response to the user key events.

NOTE 2: The standard intrinsic focus machine is shared with all other visible, focusable components and is described in clause 7.5.2.

The activity statemachine is shown in figure 87.

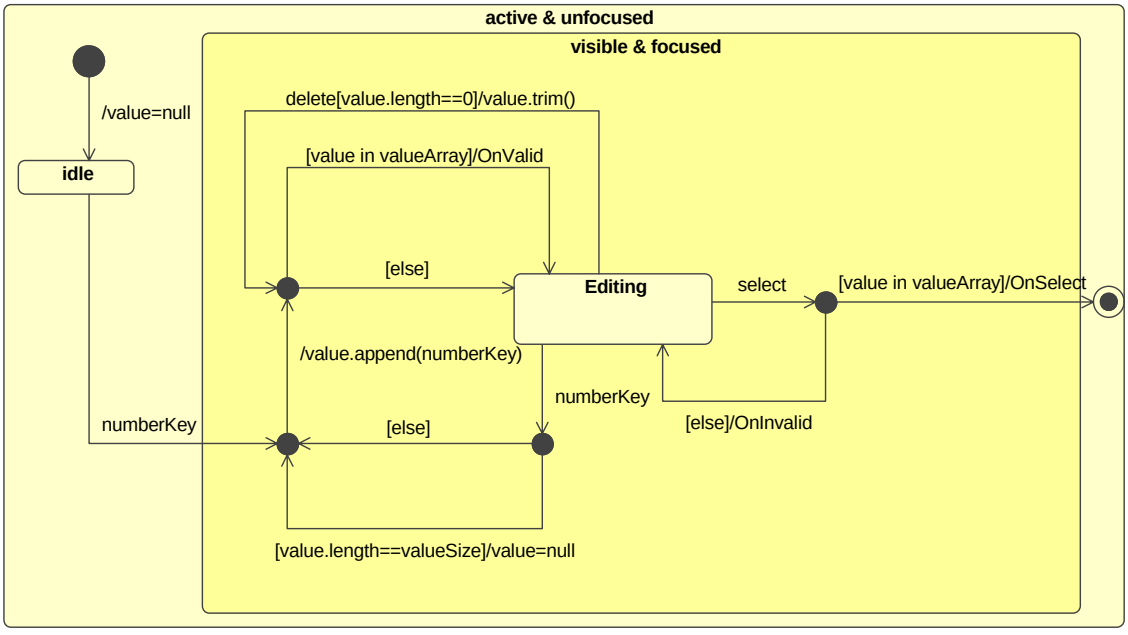

Figure 87: NumericNavigator statemachine

Numberkey events are any KeyEvent where the key property is in the set VK_0 to VK_9.

A **NumericNavigator** shall initialize into the appropriate state depending on the length of the value property.

A **NumericNavigator** shall respond to numberKey events if it is active.

A **NumericNavigator** shall respond to numberKey and select events if it is both visible and not disabled.

A **NumericNavigator** shall respond to delete key if the value property is not empty and **NumericNavigator** is visible and not disabled. A delete event shall trim the last character from the value property.

A **NumericNavigator** shall generate an OnValid event each time the value property is modified by the user and the new value property exists in valueArray.

A **NumericNavigator** shall generate an OnSelect event in response to a select event if the value property is in valueArray. In addition if there is a non-nil value in the target property the component shall invoke a navigation action using this URI value. This shall be equivalent to invoking the following action language statement:

```
sceneNavigate(target], "remember" , nil);
```

This action shall occur in place of generating an OnSelect event.

A **NumericNavigator** shall generate OnInvalid events in response to a select event if the value property is not in valueArray.

A **NumericNavigator** shall respond to numberKey events and initialize the value property if the size of the value property is equal to the valueSize property.

A **NumericNavigator** shall respond to numberKey events and append the key to the value property.

A.2.14 Subtitles

The **Subtitles** component controls the presentation of a subtitles elementary stream located within the composition managed by the enclosing **Stream** component.

```
<ComponentSpec provider="dvb.org" name="Subtitles">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties=""/>
    <Properties/>
</ComponentSpec>
```

In the present document the **Subtitles** component is only used to provide a means to switch on and off the presentation of any subtitles that the viewer has requested. In effect this allows the interactive service to either or stop the presentation of subtitles should they already be running. However, it does not provide a means to start the presentation of subtitles if this has not be requested by the viewer. Nor does it provide a means to select the subtitles elementary stream to present.

NOTE: It is assumed that the subtitles elementary stream to present will be determined in a platform-specific manner based on the video elementary stream being presented and any viewer language preference.

The subtitles elementary stream shall only be presented when the **Subtitles** component, the enclosing parent **Video** component and the latter's enclosing parent **Stream** component are active.

A.2.15 Video

The **Video** component controls the presentation of a video elementary stream located within the composition managed by the enclosing **Stream** component. The **Video** component can be used to scale the decoded video frame and select an area of the decoded, scaled video frame to present in the reference screen area.

```
<ComponentSpec provider="dvb.org" name="Video">
        <Overview version="1.0"/>
        <IntendedImplementation coreProperties="visible focusable size content termination
fillcolor"/>
        <Properties>
            <PropertyGroupRef ref="intrinsic_properties-visible_components"/>
            <PropertyGroupRef ref="positioning_properties-absolute"/>
            <PropertySpec name="content" type="uri" use="optional" access="readWrite">
                <pcf:URI name="default" value="urn:dvb-pcf::default"/>
            </PropertySpec>
            <PropertySpec name="termination" type="enumeration" use="optional"
access="initializeOnly">
                <EnumerationSpec name="lastFrame">
                    <EnumerationItem name="disappear"/>
                    <EnumerationItem name="freeze"/>
                </EnumerationSpec>
                <pcf:String name="default" value="disappear"/>
            </PropertySpec>
            <PropertySpec name="viewport-h-position" type="proportion" use="optional"
access="readWrite">
```

```
                <pcf:Proportion name="default" value="0 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-v-position" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="0 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-h-size" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="viewport-v-size" type="proportion" use="optional"
access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="h-scale" type="proportion" use="optional" access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="v-scale" type="proportion" use="optional" access="readWrite">
                <pcf:Proportion name="default" value="1 1"/>
            </PropertySpec>
            <PropertySpec name="anchor" type="enumeration" use="optional" access="readWrite">
                <EnumerationRef ref="relative-positions"/>
                <pcf:String name="default" value="bullseye"/>
            </PropertySpec>
            <PropertyGroupRef ref="color_properties-fillcolor"/>
        </Properties>
    </ComponentSpec>
```

The **visible** property shall define whether or not the **Video** component is visible.

The **focusable** property shall define whether or not the **Video** component will ever be able to enter a focused state.

If the focusable property is set to true, the **enabled** property shall define whether or not the **Video** component is eligible to receive focus, due to user navigation.

The **origin** property shall define the position of the top left-hand corner of the **Video** component when it is drawn within an explicit layout container.

The **size** property shall define the horizontal and vertical dimensions of the **Video** component. This defines the visible area of the component.

The **content** property shall identify a video elementary stream located within the composition managed by the enclosing parent **Stream** component. The value for this property may be a URL or a URN as appropriate given the definition of the composition within the enclosing **Stream** component. The value for this property may be a URL or a standard PCF URN (defined in annex Q) as appropriate given the definition of the composition within the enclosing Stream component.

A special case of the content property is when the video elementary stream to be identified is the default for the composition as determined by the target platform. This shall be defined by setting value of the content property to "urn:x-dvb-pcf::default".

NOTE 1: The video elementary stream will only be presented when both the **Audio** component and the enclosing parent **Stream** component are active.

The **termination** property shall indicate whether the last video frame shall disappear (and be replaced with the colour #000000) or whether it shall freeze when the composition managed by the enclosing parent **Stream** component is either paused or reaches its inherent end.

NOTE 2: The video elementary stream will only be presented when both the **Video** component and the enclosing parent **Stream** component are active.

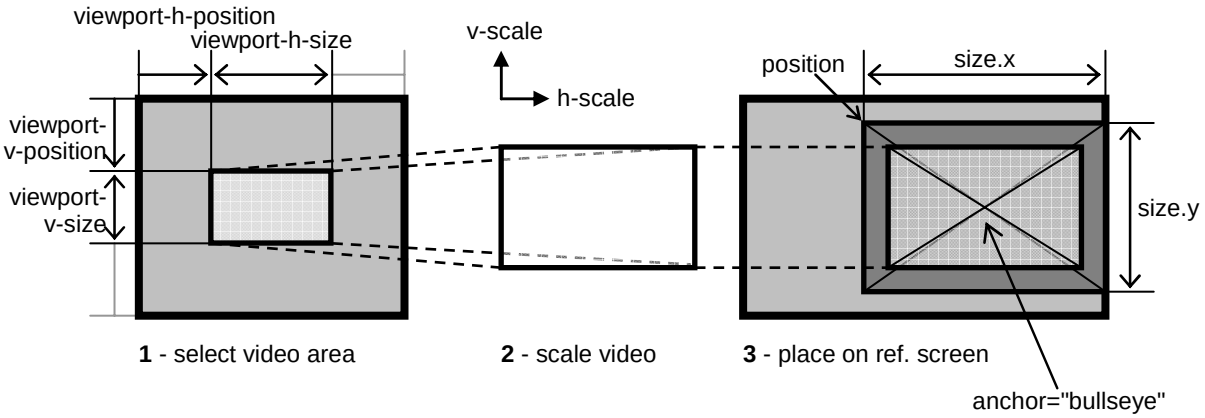

Figure 88: Video selection, scaling and positioning

Figure 88 illustrates how an area of the decoded video is selected, scaled and subsequently displayed as part of an interactive service. This shall take place in the following three stages:

1) A viewport area of the decoded video shall be defined by a box with dimensions set by the **viewport-h-size** and **viewport-v-size** properties that has its top-left hand corner located at a position corresponding to the decoded video pixel defined by the **viewport-h-position** and **viewport-v-position** properties. All properties shall be specified as a proportion of the respective horizontal and vertical dimensions of the frames in the video elementary stream.

2) The selected viewport may be scaled in proportion to its current size using the **h-scale** and **v-scale** properties.

3) The scaled video shall be presented within the area defined by the **origin** and **size** properties, which are relative to the current reference screen. The scaled video shall be aligned within this area according to the **anchor** property. If the scaled video is larger than the available rectangular area in any direction, it shall be clipped in that dimension to fit the available space. If the scaled video is smaller in any direction than the available rectangular area, the gap between the video's edge and the edge of the video window shall be filled with the colour specified by the **fillcolor** property in that dimension

EXAMPLE: If the **anchor** property is set to "bullseye", the central pixel of the viewport shall line up with the central pixel of the rectangle defined by the position and size properties. If anchor is set to "north-west", the top-left most pixel of the video shall line up with the top-left most pixel of the rectangle defined by the position and size properties.

NOTE 3: The process of selecting, scaling and positioning source video in a video window is designed to assist an author with defining video windows where they do not know much about the dimensions or aspect ratio of the source video. Through the use of proportions, it is more likely that when the video is displayed to a user, it does not appear to be distorted. In scenarios where an author knows the dimension and aspect ratio of the source material and want more precise control, the viewport properties can all be specified as proportions of the source video's dimensions, for example "360 720".

A.3 Non-visual components

A.3.1 Audio

The **Audio** component controls the presentation of an audio elementary stream located within the composition managed by the enclosing **Stream** component.

```
<ComponentSpec provider="dvb.org" name="Audio">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="content volume"/>
    <Properties>
        <PropertySpec name="content" type="uri" use="optional" access="readWrite">
            <pcf:URI name="default" value="urn:dvb-pcf::default"/>
        </PropertySpec>
        <PropertySpec name="volume" type="integer" use="optional" access="readWrite">
```

```
            <pcf:Integer name="default" value="0"/>
        </PropertySpec>
        <PropertySpec name="active" type="boolean" use="optional" access="readWrite">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **content** property shall identify an audio elementary stream located within the composition managed by the enclosing parent **Stream** component. The value for this property may be a URL or a URN as appropriate given the definition of the composition within the enclosing **Stream** component. The value for this property may be a URL or a standard PCF URN (defined in annex Q) as appropriate given the definition of the composition within the enclosing Stream component.

A special case of the content property is when the audio elementary stream to be identified is the default for the composition as determined by the target platform. This shall be defined by setting value of the content property to "urn:x-dvb-pcf::default".

The **active** property shall indicate whether the component is currently presenting audio.

The **volume** property of an **Audio** component shall define the volume at which the audio elementary stream should be presented. This value describes a scaling of the original elementary stream volume in decibels.

- 0 dB: Leave the volume unchanged.
- > 0 dB: Shall be implemented as 0 dB or louder.
  May be approximated to 0 dB.
- < 0 dB: Shall be implemented as quieter than 0 dB.
  May be approximated to -256 dB.
- -256 dB: Shall mute the audio elementary stream.

The audio elementary stream will only be presented when both the **Audio** component and the enclosing parent **Stream** component are active.

A.3.2 Cookie variables

A.3.2.1 BooleanCookie

A **BooleanCookie** component shall represent a run-time-accessible value of the PCF Boolean data type, where the value shall persist for the lifetime of a session rather than the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="BooleanCookie">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="boolean" access="readWrite">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a **BooleanCookie** component is described. The value of the cookie shall persist throughout the lifetime of a user's session and can be read and set when the variable is in scope, as defined in clause 0.

A.3.2.2 DateTimeCookie

A **DateTimeCookie** component shall represent a run-time-accessible value of the PCF dateTime data type, where the value shall persist for the lifetime of a session rather than the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="DateTimeCookie">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
```

```
        <PropertySpec name="value" type="dateTime" access="readWrite">
            <pcf:DateTime name="default" value="2000-01-01T00:00:00"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a **DateTimeCookie** component is described. The value of the cookie shall persist throughout the lifetime of a user's session and can be read and set when the variable is in scope, as defined in clause 0.

A.3.2.3 IntegerCookie

An **IntegerCookie** component shall represent a run-time-accessible value of the PCF integer data type, where the value shall persist for the lifetime of a session rather than the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="IntegerCookie">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="integer" access="readWrite">
            <pcf:Integer name="default" value="0"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when an **IntegerCookie** component is described. The value of the cookie shall persist throughout the lifetime of a user's session and can be read and set when the variable is in scope, as defined in clause 0.

A.3.2.4 String cookie

A **string cookie** component shall represent a run-time-accessible value of the PCF string data type, where the value shall persist for the lifetime of a session rather than the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="StringCookie">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="string" access="readWrite">
            <pcf:String name="default" value=""/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a string cookie component is described. The value of the cookie shall persist throughout the lifetime of a user's session and can be read and set when the variable is in scope, as defined in clause 0.

A.3.3 CurrentTime

The **CurrentTime** component shall provide the current system date and time.

```
<ComponentSpec provider="dvb.org" name="CurrentTime">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible_components"/>
        <PropertySpec name="value" type="dateTime" access="readOnly"/>
    </Properties>
</ComponentSpec>
```

The **value** property of shall provide the current system date and time on each read. The minimum granularity of change for the **value** property shall be a second.

NOTE: On some platforms, date and time values will not necessarily advance in millisecond steps.

A.3.4 Random

The **Random** component is a random number generator that has a read-only value.

```
<ComponentSpec provider="dvb.org" name="Random">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value max-result"/>
    <Properties>
        <PropertySpec name="value" type="integer" access="readOnly"/>
        <PropertySpec name="max-result" type="integer" use="required" access="initializeOnly"/>
    </Properties>
</ComponentSpec>
```

The **value** property shall change each time it is read to produce a platform's best attempt at a truly random number. The resulting random series shall be bounded and inclusive of 0 up to the **max-result** property.

NOTE: If necessary, the random series may be seeded transparently by the platform to achieve the most random result.

A.3.5 Return path components

In addition to a transfer collection, the functionality of the return path is encapsulated in four non-visible components; the **ReturnPath** Component, the **Transaction** Component, the Indicate Component and the **SecureReturnPath** Component.

A.3.5.1 Indicate

On a dial up platform the **Indicate** Component would make a return path connection and then close immediately, with no transfer of data. On an always on broadband platform the **Indicate** Component could simply touch a web page.

```
<ComponentSpec provider="dvb.org" name="Indicate">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="connectionTarget state"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible-components"/>
        <PropertySpec name="connectionTarget" type="uri" access="readWrite"/>
        <PropertySpec name="state" type="enumeration" access="readOnly">
            <EnumerationSpec name="indicate_state">
                <EnumerationItem name="idle"/>
                <EnumerationItem name="indicating"/>
            </EnumerationSpec>
        </PropertySpec>
    </Properties>
    <HandledActions>
        <HandledActionSpec name="indicate"/>
        <HandledActionSpec name="abort"/>
    </HandledActions>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnComplete"/>
        <GeneratedEventSpec eventtype="OnAbort"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorSpec errortype="OnError"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **connectionTarget** property shall define the target application server to which the indicate component will establish its connection. Its value is an abstract URI that shall be resolved by a transcoder on a target platform. For example a dial up capable platform could resolve to a telephone number whereas a broadband platform could resolve to a URL.

The **state** property reports the status of the indicate component as depicted in the indicate component statechart (reference to be included).

A.3.5.2 ReturnPath

The **ReturnPath** component embodies the return path itself

```
<ComponentSpec provider="dvb.org" name="ReturnPath">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="connectionTarget auto-connect state"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible-components"/>
        <PropertySpec name="connectionTarget" type="uri" access="initializeOnly" use="required"/>
        <PropertySpec name="auto-connect" type="boolean" access="initializeOnly" use="optional">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
        <PropertySpec name="state" type="enumeration" access="readOnly">
            <EnumerationSpec name="return-path-state">
                <EnumerationItem name="closed"/>
                <EnumerationItem name="opening"/>
                <EnumerationItem name="open"/>
                <EnumerationItem name="closing"/>
            </EnumerationSpec>
        </PropertySpec>
    </Properties>
    <HandledActions>
        <HandledActionSpec name="connect"/>
        <HandledActionSpec name="disconnect"/>
        <HandledActionSpec name="transfer"/>
    </HandledActions>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnOpening"/>
        <GeneratedEventSpec eventtype="OnOpen"/>
        <GeneratedEventSpec eventtype="OnClosing"/>
        <GeneratedEventSpec eventtype="OnClose"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorSpec errortype="OnError"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **connectionTarget** property shall define the target application server to which the **ReturnPath** component will establish its connection. Its value is an abstract URI that shall be resolved by a transcoder on a target platform. For example a dial up capable platform could resolve to a telephone number whereas a broadband platform could resolve to a URL.

The **auto-connect** property defines the start up behaviour of the **ReturnPath** component, which when set to true requires that the **ReturnPath** component starts opening its connection as soon as it is in scope, otherwise it initiates into the closed state.

The **state** property reports the status of the **ReturnPath** component as depicted in the **ReturnPath** component statechart in clause A.3.5.2.

A.3.5.3 SecureReturnPath

The **SecureReturnPath** component embodies the return path itself:

```
<ComponentSpec provider="dvb.org" name="SecureReturnPath">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="connectionTarget auto-connect state"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible-components"/>
        <PropertySpec name="connectionTarget" type="uri" access="initializeOnly" use="required"/>
        <PropertySpec name="auto-connect" type="boolean" access="initializeOnly" use="optional">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
        <PropertySpec name="state" type="enumeration" access="readOnly">
            <EnumerationSpec name="secure-return-path-state">
                <EnumerationItem name="closed"/>
                <EnumerationItem name="opening"/>
                <EnumerationItem name="open"/>
                <EnumerationItem name="closing"/>
            </EnumerationSpec>
        </PropertySpec>
    </Properties>
    <HandledActions>
```

```
        <HandledActionSpec name="connect"/>
        <HandledActionSpec name="disconnect"/>
        <HandledActionSpec name="transfer"/>
    </HandledActions>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnOpen"/>
        <GeneratedEventSpec eventtype="OnClose"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorSpec errortype="OnError"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **connectionTarget** property shall define the target application server to which the **SecureReturnPath** component will establish its connection. Its value is an abstract URI that shall be resolved by a transcoder on a target platform. For example a dial up capable platform could resolve to a telephone number whereas a broadband platform could resolve to a URL.

The **auto-connect** property defines the start up behaviour of the **SecureReturnPath** component, which when set to true requires that the **SecureReturnPath** component starts opening its connection as soon as it is in scope, otherwise it initiates into the closed state.

The **state** property reports the status of the **SecureReturnPath** component as depicted in clause A.3.5.3.

A.3.5.4 Transaction component

The **Transaction** component embodies the status of an actual data transfer exchange process.

```
<ComponentSpec provider="dvb.org" name="TransactionComponent">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="state timeout"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible_components"/>
        <PropertySpec name="state" type="enumeration" access="readOnly">
            <EnumerationSpec name="transaction-state">
                <EnumerationItem name="idle"/>
                <EnumerationItem name="busy"/>
            </EnumerationSpec>
        </PropertySpec>
        <PropertySpec name="timeout" type="integer" use="optional" access="readWrite">
            <pcf:Integer name="default" value="3000"/>
        </PropertySpec>
    </Properties>
    <HandledActions>
        <HandledActionSpec name="abort"/>
    </HandledActions>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnStart"/>
        <GeneratedEventSpec eventtype="OnComplete"/>
        <GeneratedEventSpec eventtype="OnAbort"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorSpec errortype="OnError"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **timeout** property defines the number of milliseconds the **Transaction** component shall wait before assuming that the transaction has failed and an on-error event should be triggered.

The **state** property reports the status or the **Transaction** component as depicted in the **Transaction** component statechart (reference to be included). This shall be enumerated as "idle" or "busy".

A.3.6 Stream

The **Stream** component controls the connection to a composition of one or more elementary media streams. The composition may consist of one or more individual media streams of varying type, e.g. audio, video, subtitle, real-time graphics.

```
<ComponentSpec provider="dvb.org" name="Stream" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="content looping speed"/>
```

```
    <Properties>
        <PropertySpec name="content" type="streamData" use="required" access="readWrite"/>
        <PropertySpec name="looping" type="integer" use="optional" access="readWrite">
            <pcf:Integer name="default" value="1"/>
        </PropertySpec>
        <PropertySpec name="speed" type="proportion" use="optional" access="readWrite">
            <pcf:Proportion name="default" value="1 1"/>
        </PropertySpec>
        <PropertySpec name="counterPosition" type="timecode" use="optional" access="readWrite">
            <pcf:Timecode name="default" value="00:00:00:00"/>
        </PropertySpec>
        <PropertySpec name="endPosition" type="timecode" use="optional" access="readWrite">
            <pcf:Timecode name="default" value="23:59:59:00"/>
        </PropertySpec>
        <PropertySpec name="active" type="boolean" use="optional" access="readWrite">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnStreamPlaying"/>
        <GeneratedEventSpec eventtype="OnStreamStopped"/>
    </GeneratedEvents>
    <GeneratedErrors>
        <GeneratedErrorSpec errortype="OnStreamError"/>
    </GeneratedErrors>
</ComponentSpec>
```

The **Stream** component can be used to manage connections to compositions encoded using different formats and obtained from different kinds of location. For example, an MPEG Program (DVB Service) within an broadcast MPEG-2 Transport Stream or an audio clip in the form of an MPEG-2 Elementary Stream that exist within the receiver in memory or on disk.

The relationships between the **Stream** component and other related components are illustrated in figure 89.

NOTE 1: In common with other parts of the present document this class hierarchy does not exist with in the PCF. The figure, including the abstract classes StreamBase and ElementaryStream, is provided to illustrate the relationship between various components.

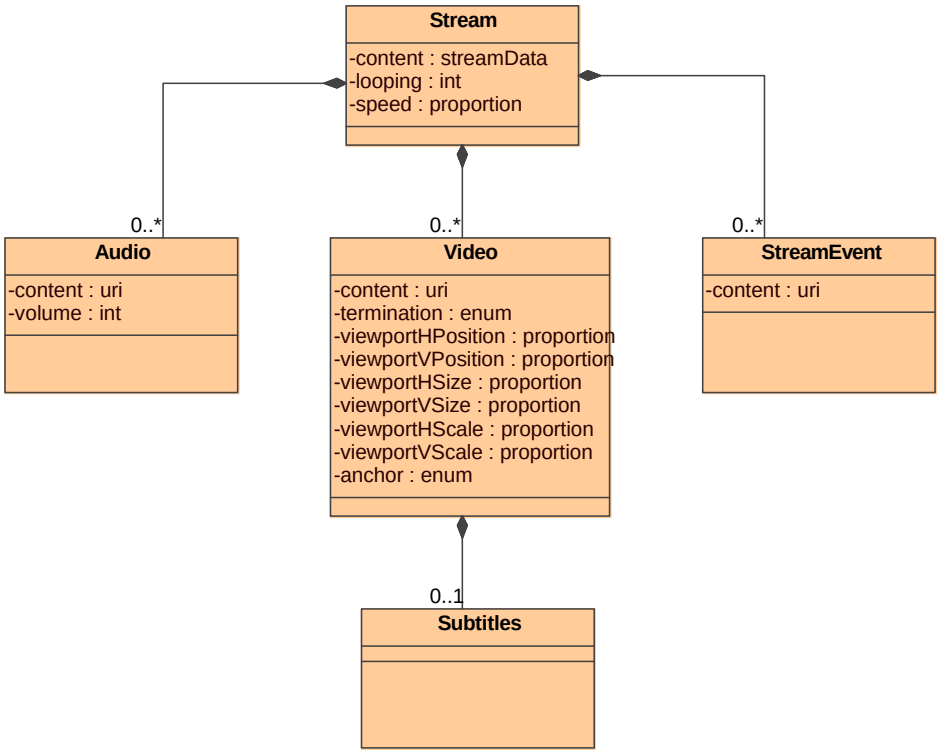

Figure 89: Stream control and presentation components

What figure 89 shows is that to present a specific elementary media streams from within a composition a component of the corresponding type, e.g. Video, shall be declared within the relevant **Stream** component. See annex R for an example.

The **content** property shall define the composition of elementary media streams to be managed by the **Stream** component. This composition may be embedded within the service description or be referenced as an external body.

A special case of an external body item is when the composition to be managed is the default for the context in which the transcoded service description is running. For example, to present the video and audio for whatever TV service the interactive service forms part of. This shall be described using an external body item with the uri property set to "urn:x-dvb-pcf::default".

EXAMPLE: Set the composition to the default in the current context.

```
<StreamData name="content">
    <ExternalBody content-type="application/octet-stream" uri="urn:x-dvb-pcf::default"/>
</StreamData>
```

The **looping** property shall define the number of times to loop the composition, as follows:

0: Loop indefinitely

1: Play once

>1 (n): Play (n) times

The **speed** property shall define the rate of advancement of the composition, as follows:

0 1: Stop

1 1: Play at normal speed

NOTE 2: The **Stream** component does not support trick-modes, such as reverse playback, faster than normal speed playback.

The **counterPosition** property shall define the current position within the composition. When a composition starts running it will start playing from the point defined by the counter position. If the counterPosition property is updated playback will skip to this new position within the composition. If the new value for the counterPosition property is beyond the endPosition then counterPosition shall be set to the value of endPosition.

If this property is not supported by a target platform then updating its value will have no effect and if read it shall return a null value.

The **endPosition** property shall define the position within the composition at which to stop playing. When a running composition reaches this point it shall stop and change its running flag to "false". If endPosition is set to a timecode that is before the current running position then the stream will immediately stop and set its running flag to "false". If endPosition is set to a timecode that is after the end timecode of the composition then the composition shall stop playing if and when it reaches its inherent end.

If this property is not supported by a target platform then updating its value will have no effect and if read it shall return a null value.

The **active** property shall indicate whether the component is currently an active source of stream events.

The intrinsic behaviour of a **Stream** component and any child components is described in figures 90 and 91.

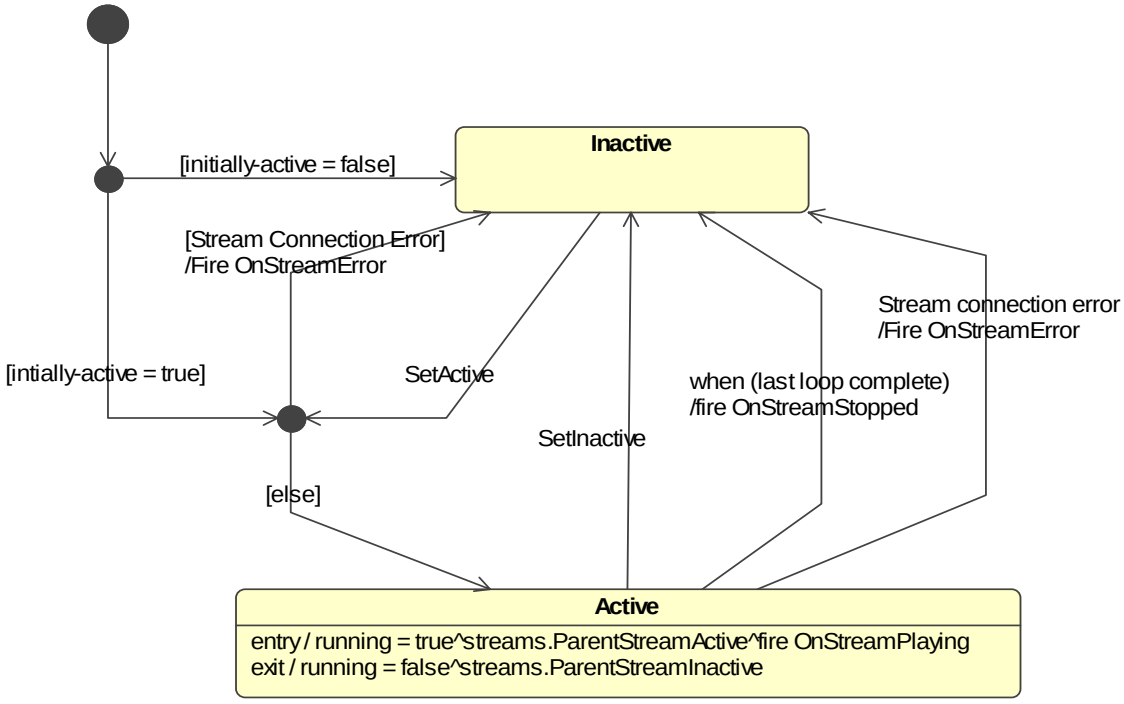

Figure 90: Stream behaviour

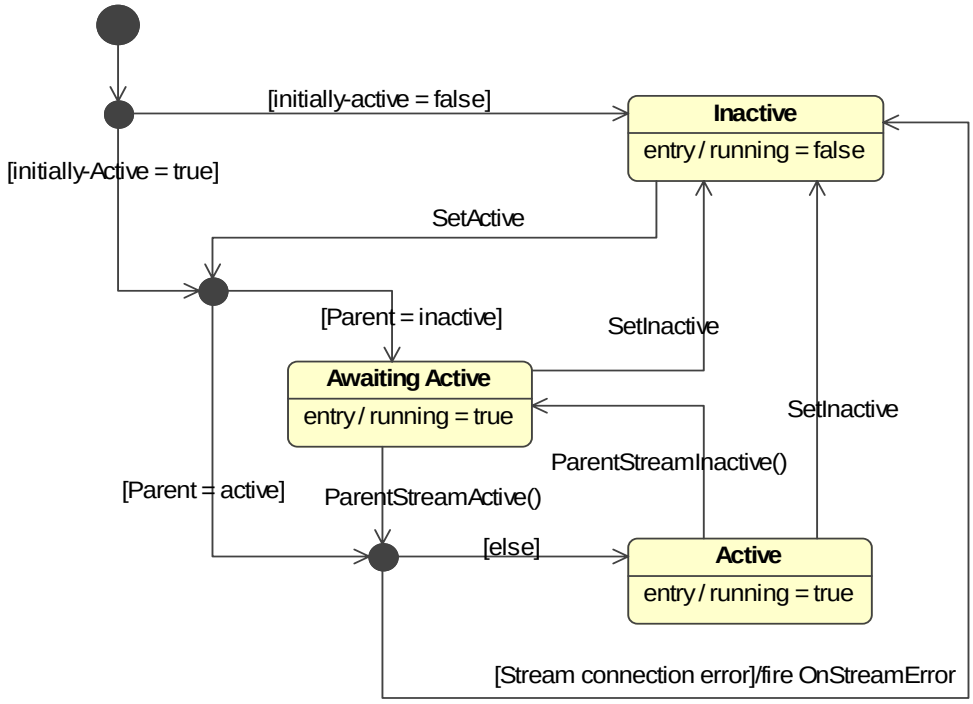

Figure 91: Elementary Stream Behaviour

Multiple **Stream** components may be active at the same time. However, when running **Stream** components and any child components will normally consume platform resources (audio/video decoders etc). The management of such resource is not described by the present document.

NOTE 2: For explicit control a service author may need to deactivate active **Stream** component and/or child components before activating another **Stream** component and/or child component that will consume shared resources.

A.3.7 StreamEvent

The **StreamEvent** component controls the generation of stream events from a data elementary stream within the composition managed by the enclosing **Stream** component.

```
<ComponentSpec provider="dvb.org" name="StreamEvent">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="content"/>
    <Properties>
        <PropertySpec name="content" type="uri" use="optional" access="readWrite">
            <pcf:URI name="default" value="urn:dvb-pcf::default"/>
        </PropertySpec>
        <PropertySpec name="active" type="boolean" use="optional" access="readWrite">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **content** property shall identify a data elementary stream used to deliver stream events located within the composition managed by the enclosing **Stream** component.

The **active** property shall indicate whether the component is currently an active source of elementary media streams.

A special case of the content property is when the data elementary stream to be identified is the default for the composition as determined by the target platform. This shall be defined by setting value of the content property to "urn:x-dvb-pcf::default".

The data elementary stream shall only be a source of stream events when both the StreamEvent component and the enclosing parent **Stream** component are active.

A.3.8 Timer

The **Timer** component implements a timer that may be used to generate timed events.

```
<ComponentSpec provider="dvb.org" name="Timer">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="period continuous"/>
    <Properties>
        <PropertyGroupRef ref="intrinsic_properties-nonvisible-components"/>
        <PropertySpec name="period" type="integer" access="readWrite"/>
        <PropertySpec name="continuous" type="boolean" access="readWrite" use="optional">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
    </Properties>
    <HandledActions>
        <HandledActionSpec name="start"/>
        <HandledActionSpec name="stop"/>
    </HandledActions>
    <GeneratedEvents>
        <GeneratedEventSpec eventtype="OnTimer"/>
    </GeneratedEvents>
</ComponentSpec>
```

The **period** property defines the number of milliseconds until the **Timer** fires its **timeout** event.

The **continuous** property defines what the **Timer** should do once it times out: if this value is set to true the **Timer** will restart, and will therefore generate further timeout events at regular intervals; if this value is false the **Timer** will reset and require restarting before it generates any further events.

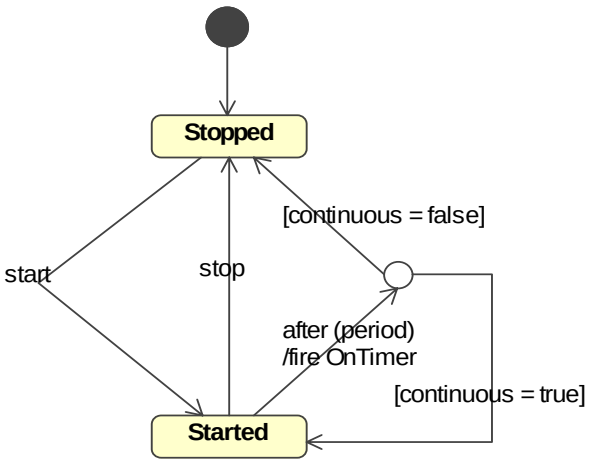

Figure 92: Timer behaviour

A **Timer** is initially stopped.

On receipt of a **start** action the **Timer** starts.

If the **Timer** is started and it receives a **stop** action the **Timer** stops and no **Timer** event shall be generated.

An **OnTimer** event shall be fired **period** milliseconds after entering the started state. At this point, if the **continuous** property is false then the **Timer** shall revert to a stopped state. If **continuous** is set to true the **Timer** shall re-enter the started state.

A.3.9 Transient variables

A.3.9.1 BooleanVar

The **BooleanVar** component shall represent a run-time-accessible value of the PCF Boolean data type, where the value persists only for the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="BooleanVar" container="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="boolean" access="readWrite">
            <pcf:Boolean name="default" value="true"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a **BooleanVar** component is described, and can be read and set during the variable's scoped lifetime, as defined in clause 7.5.3.2.

A.3.9.2 DateTimeVar

The **DateTimeVar** component shall represent a run-time-accessible value of the PCF dateTime data type, where the value persists only for the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="DateTimeVar">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="dateTime" access="readWrite">
            <pcf:DateTime name="default" value="2000-01-01T00:00:00"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a **DateTimeVar** component is described, and can be read and set during the variable's scoped lifetime, as defined in clause 7.5.3.2.

A.3.9.3 IntegerVar

The **IntegerVar** component shall represent a run-time-accessible value of the PCF integer data type, where the value persists only for the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="IntegerVar" serializable="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="integer" access="readWrite">
            <pcf:Integer name="default" value="0"/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when an **IntegerVar** component is described, and can be read and set during the variable's scoped lifetime as defined in clause 7.5.3.2.

A.3.9.4 StringVar

The **StringVar** component shall represent a run-time-accessible value of the PCF string data type, where the value persists only for the lifetime of the component's container.

```
<ComponentSpec provider="dvb.org" name="StringVar" serializable="true">
    <Overview version="1.0"/>
    <IntendedImplementation coreProperties="value"/>
    <Properties>
        <PropertySpec name="value" type="string" access="readWrite">
            <pcf:String name="default" value=""/>
        </PropertySpec>
    </Properties>
</ComponentSpec>
```

The **value** property shall represent the current value of the variable. This value can be initialized when a **StringVar** component is described, and can be read and set during the variable's scoped lifetime, as defined in clause 7.5.3.2.

Annex B (normative): Events and errors

B.1 System events

B.1.1 Service event

A **service lifecycle event** provides for temporal synchronization to the start and end of a service session.

```
<EventSpec name="ServiceLifecycle" class="system">
    <PropertySpec name="action" type="enumeration" access="readOnly">
        <EnumerationSpec name="lifecycleaction">
            <EnumerationItem name="load"/>
            <EnumerationItem name="exit"/>
        </EnumerationSpec>
    </PropertySpec>
</EventSpec>
```

A service lifecycle event contains an **action** property to indicate if this is the start or end of a service session.

- A "load" event shall occur when the service session is ready to begin execution, immediately prior to activating the initial scene component i.e. all components of a service have been acquired but no rendering has yet taken place.
- An "exit" event shall occur immediately after the request to navigate away from the current service and before discarding any components of the service or deactivating the currently active scene.

B.1.2 Stream event

A **stream event** provides for temporal synchronization points within a playing media stream.

NOTE 1: PCF service description permits several media streams to be playing simultaneously. Events from all playing streams shall contribute to the timeline of stream events that occur during a service session.

```
<EventSpec name="StreamEvent" class="system">
    <PropertySpec name="streamId" type="uri" access="readOnly"/>
    <PropertySpec name="eventName" type="string" access="readOnly"/>
    <PropertySpec name="payload" type="string" access="readOnly"/>
</EventSpec>
```

A stream event shall contain a **streamId** property which identifies the Stream component instance that manages the media stream composition in which the StreamEvent was delivered.

A stream event shall contain an **eventName** property that is used to classify the event in a service-specific manner. This value shall not be null and shall be defined within the context of a single media stream. See also annex O.

NOTE 2: each event is located at a specific temporal position within the timeline of the media stream with which it is associated. A stream event with a given **streamId** and **eventName** may occur more than once during the service session because several instances of it may be defined at distinct temporal locations in the same media stream.

NOTE 3: PCF does not specify any pre-roll of events. In the logical PCF behaviour model, stream events occur at the exact moment which their temporal location represents in the associated media stream. Platform implementations should ensure that the rendering of the effects caused by a stream event occur as close as possible to this moment. Thus a temporal location value is not required in the context of the event itself.

A **stream event** shall contain a **payload** property that carries optional event data, represented as a string.

Stream events are delivered to a platform independently of the PCF service description which uses them, thus this mechanism is outside the scope of PCF.

Event timelines may be delivered by binding them to media items associated with the PCF service or in a separate standalone document. The present document recommends a set of informative alternative bindings to achieve this in annex O.

B.1.3 ProgramChange event

A **program change event** provides for temporal synchronization to program boundaries within a media stream. It is intended to directly model DVB EIT Present/Following transitions or their equivalent.

```
<EventSpec name="ProgramChange" class="system">
    <PropertySpec name="streamId" type="uri" access="readOnly"/>
    <PropertySpec name="eventId" type="uri" access="readOnly"/>
    <PropertySpec name="startTime" type="dateTime" access="readOnly"/>
    <PropertySpec name="duration" type="time" access="readOnly"/>
    <PropertySpec name="language" type="iso639" access="readOnly"/>
    <PropertySpec name="name" type="string" access="readOnly"/>
    <PropertySpec name="shortDescription" type="string" access="readOnly"/>
</EventSpec>
```

A program change event shall contain a **streamId** property which identifies the Stream component instance that manages the media stream composition in which the program change occurred.

A program change event shall contain an **eventId** property that shall uniquely identify the program within the media stream.

A program change event shall contain a **startTime** property that shall contain the expected start date and time for the program.

A program change event shall contain a **duration** property that shall contain the expected duration for the program.

A program change event shall contain a **language** property that shall contain the language for the program name and description.

A program change event shall contain a **name** property that shall contain the title for the program.

A program change event shall contain a **shortDescription** property that shall contain a synopsis for the program.

B.1.4 RunningStatus event

A **running status** event provides for temporal synchronization with the start and end of transmission within a media stream. It is intended to directly model DVB Service Running Status transitions or their equivalent.

```
<EventSpec name="RunningStatus" class="system">
    <PropertySpec name="streamId" type="uri" access="readOnly"/>
    <PropertySpec name="status" type="enumeration" access="readOnly">
        <EnumerationSpec name="rst">
            <EnumerationItem name="stopped"/>
            <EnumerationItem name="starting"/>
            <EnumerationItem name="running"/>
            <EnumerationItem name="stopping"/>
        </EnumerationSpec>
    </PropertySpec>
</EventSpec>
```

A running status event shall contain a **streamId** property which identifies the Stream component instance that manages the media stream composition in which the transition occurred.

A running status event shall contain a **status** property which indicates the state into which the media stream has transitioned. These states correspond directly with those defined for DVB Service Information [27].

B.2 User events

B.2.1 Key event

A **key event** shall be generated by the run-time environment when one of a specific set of user input device keys is activated. This key must be in the set that is mapped to the values defined in clause K.1 by the platform implementation.

```
<EventSpec name="KeyEvent" class="user">
    <PropertySpec name="key" type="userKey" access="readOnly"/>
</EventSpec>
```

A key event shall contain a **key** property which indicates which virtual key has been activated.

NOTE: Key events shall only occur once per activation of a specific key. There is no automatic key repeat indication, or key down/up indication. See Raw Key Event for these features.

B.2.2 RawKey event

A **raw key event** may be generated in a platform-specific manner to provide more detailed information regarding the nature of the user input device key activation.

NOTE: Raw key events may not be portable or even available on all platforms.

```
<EventSpec name="RawKeyEvent" class="user">
    <PropertySpec name="keycode" type="integer" access="readOnly"/>
    <PropertySpec name="keytype" type="enumeration" access="readOnly">
        <EnumerationSpec name="rawkeytype">
            <EnumerationItem name="keydown"/>
            <EnumerationItem name="keyup"/>
            <EnumerationItem name="keyrepeat"/>
        </EnumerationSpec>
    </PropertySpec>
</EventSpec>
```

A raw key event shall contain a **keycode** property which indicates which physical device key has been activated. The value contained in this property is platform-specific and not defined by the present document.

EXAMPLE: A keycode may allow a service author to distinguish between a front panel key and a remote key of the same nominal virtual key value, or to access an alternative user input device such as a game controller.

A raw key event shall contain a **keytype** property which indicates which action a user is invoking on a physical device key. Actions that may be distinguished are: key down, key up and key repeat.

B.3 Component events

B.3.1 Navigation events

Navigation events shall be generated by a component when the intrinsic focus behaviour alters the focus state of a component.

```
<EventSpec class="component" name="OnFocus"/>
<EventSpec class="component" name="OnBlur"/>
```

An **OnFocus** event shall occur when a component receives focus.

An **OnBlur** event shall occur when a component loses focus.

There is a GeneratedEventGroup for Navigation events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup name="navigation_events">
    <GeneratedEventSpec eventtype="OnFocus"/>
```

```
    <GeneratedEventSpec eventtype="OnBlur"/>
 </GeneratedEventGroup>
```

B.3.2 Scene events

Scene events, **enterforward** and **enterhistory** shall be generated when a Scene component becomes active. Scene event **exit** shall be generated when the scene becomes inactive.

```
<EventSpec name="SceneLifecycle" class="component">
 <PropertySpec name="action" type="enumeration" access="readOnly">
     <EnumerationSpec name="lifecycleaction">
         <EnumerationItem name="enterforward"/>
         <EnumerationItem name="enterhistory"/>
         <EnumerationItem name="exit"/>
     </EnumerationSpec>
 </PropertySpec>
</EventSpec>
```

There is a GeneratedEventGroup for Scene events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup name="scene_events">
     <GeneratedEventSpec eventtype="SceneLifecycle"/>
</GeneratedEventGroup>
```

B.3.3 Button events

Button events may be generated when any selectable input control is operated.

NOTE: Button events include navigation events.

```
 <EventSpec class="component" name="OnSelect"/>
 <EventSpec class="component" name="OnUnselect"/>
```

An **OnSelect** event shall occur when the control is activated.

An **OnUnselect** event shall occur if the control is positively deactivated e.g. a radio button is unchecked.

There is a GeneratedEventGroup for Button events, which some components may use. It is declared as follows:

```
 <GeneratedEventGroup name="button_events">
     <GeneratedEventGroupRef ref="navigation_events"/>
     <GeneratedEventSpec eventtype="OnSelect"/>
     <GeneratedEventSpec eventtype="OnUnselect"/>
 </GeneratedEventGroup>
```

B.3.4 Choice events

Choice events may be generated when any editable input control is altered by the user.

```
 <EventSpec class="component" name="OnChange"/>
```

An **OnChange** event shall occur when an edit operation is completed on the control e.g. a character is entered into a text input directly, or an on-screen keyboard is used to edit and then save the text value.

There is a GeneratedEventGroup for Choice events, which some components may use. It is declared as follows:

```
 <GeneratedEventGroup name="choice_events">
     <GeneratedEventGroupRef ref="navigation_events"/>
     <GeneratedEventSpec eventtype="OnChange"/>
 </GeneratedEventGroup>
```

B.3.5 Media events

Media events may be generated by any component that handles media content in a dynamic manner.

```
 <EventSpec class="component" name="OnMediaAvailable"/>
```

An **OnMediaAvailable** event shall be generated when the component has successfully acquired its media content and is presenting or is ready to present it, e.g. an image has been acquired and decoded in a Background component.

There is a GeneratedEventGroup for Media events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup name="media_events">
    <GeneratedEventSpec eventtype="OnMediaAvailable"/>
</GeneratedEventGroup>
```

B.3.6 TextInput events

TextInput events may be generated by input controls that support free text entry.

```
<EventSpec class="component"  name="OnEmpty"/>
<EventSpec class="component"  name="OnFull"/>
<EventSpec class="component" name="OnUnprintableCharacter"/>
```

An **OnEmpty** event shall be generated when a delete edit is requested on an already empty text field.

An **OnFull** event shall be generated when an increase edit is requested on an already full text field.

An **OnUnprintableCharacter** event shall be generated when an edit is requested that contains characters not suitable for the text field, e.g. an onscreen keyboard is used to enter an alpha characters into a numeric only text field.

There is a GeneratedEventGroup for TextInput events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup  name="text_input_events">
    <GeneratedEventGroupRef ref="choice_events"/>
    <GeneratedEventSpec eventtype="OnEmpty"/>
    <GeneratedEventSpec eventtype="OnFull"/>
    <GeneratedEventSpec eventtype="OnUnprintableCharacter"/>
</GeneratedEventGroup>
```

B.3.7 Animation events

Animation events may be generated by components that display an animation.

```
<EventSpec class="component" name="OnLoopComplete"/>
<EventSpec class="component" name="OnAnimationComplete"/>
```

An **OnLoopComplete** event shall be generated when the animation reaches the end of the sequence and has more loops to complete.

An **OnAnimationComplete** event shall be generated when the animation reaches the end of the sequence and there are no more loops to present.

There is a GeneratedEventGroup for Animation events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup name="animation_events">
    <GeneratedEventSpec eventtype="OnLoopComplete"/>
    <GeneratedEventSpec eventtype="OnAnimationComplete"/>
</GeneratedEventGroup>
```

B.3.8 PageContainer events

Page Container events may be generated by page container components.

```
<EventSpec class="component" name="OnPageChanged"/>
```

An **OnPageChanged** event shall be generated when the user navigates to a different page.

There is a GeneratedEventGroup for PageContainer events, which some components may use. It is declared as follows:

```
<GeneratedEventGroup name="page_container_events">
    <GeneratedEventSpec eventtype="OnPageChanged"/>
</GeneratedEventGroup>
```

B.3.9 Timer event

A **Timer event** is generated by the **Timer** component.

```
<EventSpec class="component" name="OnTimer">
    <PropertySpec name="name" type="string" access="readOnly"/>
</EventSpec>
```

B.3.10 NumericNavigator events

OnValid and **OnInvalid** events are generated by the NumericNavigator component.

```
<EventSpec class="component" name="OnValid"/>
<EventSpec class="component" name="OnInvalid"/>
```

B.3.11 Stream events

OnStreamPlaying and **OnStreamStopped** events are generated by the NumericNavigator component.

```
<EventSpec class="component" name="OnStreamPlaying"/>
<EventSpec class="component" name="OnStreamStopped"/>
```

B.3.12 ReturnPath events

ReturnPath events can be generated by the components which control the return path.

```
<EventSpec name="OnOpening" class="component"/>
<EventSpec name="OnOpen" class="component"/>
<EventSpec name="OnClosing" class="component"/>
<EventSpec name="OnClose" class="component"/>
<EventSpec name="OnStart" class="component"/>
<EventSpec name="OnComplete" class="component"/>
<EventSpec name="OnAbort" class="component"/>
```

An **OnOpening** event shall be generated when an attempt is made to open a return path connection.

An **OnOpen** event shall be generated when a return path connection has been successfully opened.

An **OnClosing** event shall be generated when an attempt is made to close the return path connection.

An **OnClose** event shall be generated when the return path connection has been successfully closed.

An **OnStart** event shall be generated when a transmission over the return path has been started.

An **OnComplete** event shall be generated when a transmission over the return path has been successfully completed.

An **OnAbort** event shall be generated when a transmission was aborted before completion by the application.

B.4 Errors

Errors are instances of the Error Event which have the event properties set by values defined according to the ErrorSpec for the error. Clause B.4.1 declares the Error Event. Individual Errors are then declared in the remaining clauses.

B.4.1 Error events

An **Error event** defines a standard means for propagating error conditions within a service description.

```
<EventSpec name="Error" class="component">
    <PropertySpec name="name" type="string" access="readOnly"/>
    <PropertySpec name="level" type="enumeration" access="readOnly">
        <EnumerationSpec name="errorlevel">
            <EnumerationItem name="notice"/>
```

```
            <EnumerationItem name="warning"/>
            <EnumerationItem name="error"/>
        </EnumerationSpec>
    </PropertySpec>
    <PropertySpec name="errorstring" type="string" access="readOnly"/>
</EventSpec>
```

Error events are not directly referenced by component specifications. Instead **ErrorSpec** items declare Errors with properties of this event provided. Components shall then use **GeneretedErrorSpec** items to reference the ErrorSpec which defines the error they use.

An error event shall contain a **name** property as specified by the **name** attribute of an ErrorSpec item.

An error event shall contain an **errorlevel** property as specified by the **level** attribute of an ErrorSpec item.

An error event shall contain an **errorstring** property that contains a human readable description for the error event which may be platform specific and is not defined in the present document.

B.4.2 Basic errors

All components may generate this **basic error**.

```
<ErrorSpec name="OnUnknownError" level="error"/>
```

There is a GeneratedErrorGroup for Basic Errors, which some components may use. It is declared as follows:

```
<GeneratedErrorGroup name="basic_errors">
    <GeneratedErrorSpec errortype="OnUnknownError"/>
</GeneratedErrorGroup>
```

B.4.3 Media errors

Components that handle dynamic media content in any form may generate these media errors.

```
<ErrorSpec name="OnUnknownError" level="error"/>
<ErrorSpec name="OnInvalidMediaType" level="warning"/>
<ErrorSpec name="OnMediaUnavailable" level="warning"/>
<ErrorSpec name="OnMediaCorrupt" level="warning"/>
<ErrorSpec name="OnMediaEncodingNotSupported" level="warning"/>
```

An **OnInvalidMediaType** error shall be generated if the type of media does not match a type supported by the component, e.g. video/mpeg for an Image component which supports only image/* types.

An **OnMediaUnavailable** error shall be generated if the media content item cannot be acquired in a reasonable time.

NOTE: What constitutes a reasonable time shall be platform-specific and is not defined in the present document.

An **OnMediaCorrupt** error shall be generated if the media content item fails to complete any decoding required before rendering it.

An **OnMediaEncodingNotSupported** error shall be generated if the media content item is not packaged using an encoding supported by the component, e.g. GZIP compressed content is acquired when only uncompressed content is supported.

There is a GeneratedErrorGroup for Media Errors, which some components may use. It is declared as follows:

```
<GeneratedErrorGroup name="media_errors">
    <GeneratedErrorSpec errortype="OnInvalidMediaType"/>
    <GeneratedErrorSpec errortype="OnMediaUnavailable"/>
    <GeneratedErrorSpec errortype="OnMediaCorrupt"/>
    <GeneratedErrorSpec errortype="OnMediaEncodingNotSupported"/>
</GeneratedErrorGroup>
```

B.4.4 Stream errors

The following error is generated by Stream components.

```
<ErrorSpec level="error" name="OnStreamError"/>
```

B.4.5 ReturnPath errors

The following error is generated by Return Path components.

```
<ErrorSpec name="OnError" level="warning"/>
```

B.4.6 Execution errors

The following errors are generated by execution of action language:

```
<ErrorSpec name="ReferenceError" level="error"/>
<ErrorSpec name="ExecutionError" level="error"/>
<ErrorSpec name="ExecutionError" level="warning"/>
<ErrorSpec name="ExecutionError" level="notice"/>
```

NOTE: Execution error level is determined by the class of failure as defined in clause 9.4.6.7.

Annex C (normative): Property Groups

C.1 Intrinsic properties

Intrinsic properties are applied to all components, where each component is either classified as visible or nonvisible.

C.1.1 Visible components

C.1.1.1 Enumeration

```
<EnumerationSpec name="update-values">
    <EnumerationItem name="auto"/>
    <EnumerationItem name="onrefresh"/>
    <EnumerationItem name="none"/>
</EnumerationSpec>
```

Enumeration values shall be defined as follows:

- **auto** - content shall immediately refresh when an update is received.
- **onrefresh** - when updated content has been received, it shall be displayed only after a refresh or redraw event.
- **none** - updated content shall not be displayed within the scope of the current **Scene**.

C.1.1.2 PropertyGroup

```
<PropertyGroup name="instrinisic_properties-visible_components">
    <PropertySpec name="enabled" type="boolean" default="true"
        writeable="true"/>
    <PropertySpec name="focusable" type="boolean" default="true"
        writeable="false"/>
    <PropertySpec name="reactivetext" type="string" use="optional" writeable="true"/>
    <PropertySpec name="visible" type="boolean" default="true"
        writeable="true"/>
    <PropertySpec name="tabindex" type="integer" use="optional" writeable="false"/>
    <PropertySpec name="update" type="enumeration" default="auto" writeable="false">
        <EnumerationRef ref="update-values"/>
    </PropertySpec>
    <PropertySpec name="aacolor" type="color" use="optional" writeable="false"/>
</PropertyGroup>
```

The properties in this property group shall be defined as follows:

- **enabled** - set to "true" if a component is currently enabled, as defined in clause 7.5.2, and "false" if it is disabled.
- **focusable** - set to "true" if a component can receive focus, as defined in clause 7.5.2, and "false" if it cannot.
- **reactivetext** - text associated with a component that may be displayed to a user when the component has focus.
- **visible** - set to "true" if a component is currently visible and "false" if not.
- **tabindex** - used to indicate the order of a component within a sequence of other components that a user can navigate focus between.
- **update** - controls the behaviour of a component when updated content is available.
- **aacolor** - anti-alias colour to use when drawing a component on top of other components or Background components.

C.2 Background properties

C.2.1 Background_properties-images

```
<PropertyGroup name="background_properties-images">
        <PropertySpec name="image" type="imageData" use="optional" access="readWrite"/>
        <PropertySpec name="tiling" type="enumeration" access="initializeOnly" use="optional">
            <EnumerationSpec name="tiling-mode">
                <EnumerationItem name="none"/>
                <EnumerationItem name="horizontal"/>
                <EnumerationItem name="vertical"/>
                <EnumerationItem name="both"/>
            </EnumerationSpec>
            <pcf:String name="default" value="none"/>
        </PropertySpec>
        <PropertySpec name="offset" type="position" use="optional" access="initializeOnly">
            <pcf:Position name="default" value="0 0"/>
        </PropertySpec>
        <PropertySpec name="stretchToFit" type="boolean" use="optional" access="initializeOnly">
            <pcf:Boolean name="default" value="false"/>
        </PropertySpec>
    </PropertyGroup>
```

The properties in this property group shall be defined as follows:

- **src** - the source of the background image.
- **tiling** - controls direction and allowability of background images.
- **offset** - controls offset of top left corner of image from origin of component.
- **stretch-to-fit** - set to "true" if background image shall be stretched to fill background area of component, "false" if not.

C.2.2 Background_properties

```
    </PropertyGroup>
    <PropertyGroup name="background_properties">
        <PropertyGroupRef ref="background_properties-images"/>
        <PropertyGroupRef ref="color_properties-fillcolor"/>
    </PropertyGroup>
```

C.2.2.1 Properties defined elsewhere

- Background properties - images: clause C.2.
- Color_properties-fillcolor: clause C.3.4.

C.3 Color properties

C.3.1 Color_properties

```
    <PropertyGroup name="color_properties">
        <PropertySpec name="color" type="color" use="required" access="readWrite"/>
        <PropertySpec name="color-focus" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="color-disabled" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="color-active" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="color-idle" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="shadowcolor" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="highlightcolor" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="aacolor" type="color" use="optional" access="initializeOnly"/>
        <PropertySpec name="rendering-intent" type="enumeration" use="optional" default="auto"
access="readWrite">
```

```
            <EnumerationRef ref="rendering-intent"/>
        </PropertySpec>
    </PropertyGroup>
```

The color_properties group defines the generic set of color-related properties for use in PCF components.

The properties in this property group shall be defined as follows:

- **color** - a sequence of four integer values between 0 and 255, expressed in hexadecimal, representing red, green, blue and transparency colour values respectively;
- **color-focused** - a **color** value to be displayed when the component is in a focused state;
- **color-disabled** - a **color** value to be displayed when the component is in a disabled state;
- **color-active** - a **color** value to be displayed when the component is in an active state;
- **color-idle** - a **color** value to be displayed when the component is in an idle state;
- **color-shadow** - a **color** value to be used for shadows;
- **color-highlight** - a **color** value to be used for highlights;
- **color-aa** - a **color** value to be used in calculation of anti-alias colours;
- **rendering-intent** -as specified in Extensible Stylesheet Language (XSL) 1.0 [25] clause 17.3.

C.3.2 Color_properties-linecolor

```
    </PropertyGroup>
    <PropertyGroup name="color_properties-linecolor">
        <PropertySpec name="linecolor" type="color" use="required" access="readWrite"/>
        <PropertySpec name="linecolor-focus" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-disabled" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-active" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-idle" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-shadow" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-highlight" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="linecolor-rendering-intent" type="enumeration" use="optional"
access="initializeOnly">
            <EnumerationRef ref="rendering-intent"/>
        </PropertySpec>
    </PropertyGroup>
```

The color_properties-linecolor properties group defines the set of linecolor-related properties for use in PCF components.

The properties in this property group shall be defined as follows:

- **linecolor** - a sequence of four integer values between 0 and 255, expressed in hexadecimal, representing red, green, blue and transparency colour values respectively;
- **linecolor-focused** - a **color** value to be displayed when the component is in a focused state;
- **linecolor-disabled** - a **color** value to be displayed when the component is in a disabled state;
- **linecolor-active** - a **color** value to be displayed when the component is in an active state;
- **linecolor-idle** - a **color** value to be displayed when the component is in an idle state;
- **linecolor-shadow** - a **color** value to be used for shadows;
- **linecolor-highlight** - a **color** value to be used for highlights;
- **linecolor-rendering-intent** -as specified in Extensible Stylesheet Language (XSL) 1.0 [25] clause 17.3.

C.3.3 Color_properties-bordercolor

```
    </PropertyGroup>
    <PropertyGroup name="color_properties-bordercolor">
        <PropertySpec name="bordercolor" type="color" use="required" access="readWrite"/>
        <PropertySpec name="bordercolor-focus" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-disabled" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-active" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-idle" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-shadow" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-highlight" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="bordercolor-rendering-intent" type="enumeration" use="optional"
access="initializeOnly">
            <EnumerationRef ref="rendering-intent"/>
        </PropertySpec>
    </PropertyGroup>
```

The color_properties-fillcolor properties group defines the set of fillcolor-related properties for use in PCF components.

The properties in this property group shall be defined as follows:

- **bordercolor** - a sequence of four integer values between 0 and 255, expressed in hexadecimal, representing red, green, blue and transparency colour values respectively;
- **bordercolor-focused** - a **color** value to be displayed when the component is in a focused state;
- **bordercolor-disabled** - a **color** value to be displayed when the component is in a disabled state;
- **bordercolor-active** - a **color** value to be displayed when the component is in an active state;
- **bordercolor-idle** - a **color** value to be displayed when the component is in an idle state;
- **bordercolor-rendering-intent** -as specified in Extensible Stylesheet Language (XSL) 1.0 [25] clause 17.3.

C.3.4 Color_properties-fillcolor

```
    </PropertyGroup>
    <PropertyGroup name="color_properties-fillcolor">
        <PropertySpec name="fillcolor" type="color" use="required" access="readWrite"/>
        <PropertySpec name="fillcolor-focus" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="fillcolor-disabled" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="fillcolor-active" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="fillcolor-idle" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="fillcolor-rendering-intent" type="enumeration" use="optional"
access="initializeOnly">
            <EnumerationRef ref="rendering-intent"/>
        </PropertySpec>
    </PropertyGroup>
```

The color_properties-fillcolor properties group defines the set of fillcolor-related properties for use in PCF components.

The properties in this property group shall be defined as follows:

- **fillcolor** - a sequence of four integer values between 0 and 255, expressed in hexadecimal, representing red, green, blue and transparency colour values respectively;
- **fillcolor-focused** - a **color** value to be displayed when the component is in a focused state;
- **fillcolor-disabled** - a **color** value to be displayed when the component is in a disabled state;
- **fillcolor-active** - a **color** value to be displayed when the component is in an active state;
- **fillcolor-idle** - a **color** value to be displayed when the component is in an idle state;
- **fillcolor-shadow** - a **color** value to be used for shadows;
- **fillcolor-highlight** - a **color** value to be used for highlights;
- **fillcolor-rendering-intent** -as specified in Extensible Stylesheet Language (XSL) 1.0 [25] clause 17.3.

C.3.5 Color_properties-textcolor

```
    </PropertyGroup>
    <PropertyGroup name="color_properties-textcolor">
        <PropertySpec name="textcolor" type="color" use="required" access="readWrite"/>
        <PropertySpec name="textcolor-focus" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-disabled" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-active" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-idle" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-shadow" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-highlight" type="color" use="optional" access="readWrite"/>
        <PropertySpec name="textcolor-rendering-intent" type="enumeration" use="optional"
access="initializeOnly">
            <EnumerationRef ref="rendering-intent"/>
        </PropertySpec>
    </PropertyGroup>
```

The color_properties-textcolor properties group defines the set of textcolor-related properties for use in PCF components.

The properties in this property group shall be defined as follows:

- **textcolor** - a sequence of four integer values between 0 and 255, expressed in hexadecimal, representing red, green, blue and transparency colour values respectively;
- **textcolor-focused** - a **color** value to be displayed when the component is in a focused state;
- **textcolor-disabled** - a **color** value to be displayed when the component is in a disabled state;
- **textcolor-active** - a **color** value to be displayed when the component is in an active state;
- **textcolor-idle** - a **color** value to be displayed when the component is in an idle state;
- **textcolor-rendering-intent** -as specified in Extensible Stylesheet Language (XSL) 1.0 [25] clause 17.3.

C.4 Border properties

```
<EnumerationSpec name="border-singlecolor-linestyles">
    <EnumerationItem name="solid"/>
    <EnumerationItem name="dashed"/>
    <EnumerationItem name="dotted"/>
    <EnumerationItem name="double"/>
</EnumerationSpec>
<EnumerationSpec name="border-multicolor-linestyles">
    <EnumerationItem name="groove"/>
    <EnumerationItem name="ridge"/>
    <EnumerationItem name="bevelled-outset"/>
    <EnumerationItem name="bevelled-inset"/>
</EnumerationSpec>
<PropertyGroup name="border_properties">
    <PropertyGroupRef ref="color_properties-bordercolor"/>
    <PropertyGroupRef ref="linestyle_properties"/>
    <PropertyGroupRef ref="cornerradius_properties"/>
    <PropertySpec name="border-width" type="integer" default="2" access="readWrite"/>
    <PropertySpec name="border-top-linestyle-singlecolor" type="enumeration" default="solid"
access="initializeOnly">
        <EnumerationRef ref="border-singlecolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-top-linestyle-multicolor" type="enumeration" use="optional"
access="initializeOnly">
        <EnumerationRef ref="border-multicolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-top-width" type="integer" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-top-color" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-top-focuscolor" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-top-disabledcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-top-activecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-top-idlecolor" type="color" use="optional" access="initializeOnly"/>
```

```
    <PropertySpec name="border-top-shadowcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-top-highlightcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-linestyle-singlecolor" type="enumeration" default="solid"
access="initializeOnly">
        <EnumerationRef ref="border-singlecolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-bottom-linestyle-multicolor" type="enumeration" use="optional"
access="initializeOnly">
        <EnumerationRef ref="border-multicolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-bottom-width" type="integer" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-bottom-color" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-bottom-focuscolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-disabledcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-activecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-idlecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-shadowcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-bottom-highlightcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-left-linestyle-singlecolor" type="enumeration" default="solid"
access="initializeOnly">
        <EnumerationRef ref="border-singlecolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-left-linestyle-multicolor" type="enumeration" use="optional"
access="initializeOnly">
        <EnumerationRef ref="border-multicolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-left-width" type="integer" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-left-color" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-left-focuscolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-left-disabledcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-left-activecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-left-idlecolor" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-left-shadowcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-left-highlightcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-linestyle-singlecolor" type="enumeration" default="solid"
access="initializeOnly">
        <EnumerationRef ref="border-singlecolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-right-linestyle-multicolor" type="enumeration" use="optional"
access="initializeOnly">
        <EnumerationRef ref="border-multicolor-linestyles"/>
    </PropertySpec>
    <PropertySpec name="border-right-width" type="integer" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-right-color" type="color" use="optional" access="initializeOnly"/>
    <PropertySpec name="border-right-focuscolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-disabledcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-activecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-idlecolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-shadowcolor" type="color" use="optional"
access="initializeOnly"/>
    <PropertySpec name="border-right-highlightcolor" type="color" use="optional"
access="initializeOnly"/>
</PropertyGroup>
```

The BorderProperties properties group defines the set of border-related properties for use in PCF components.

C.4.1 BorderProperties enumerations

C.4.1.1 BorderSingleColorLinestyle enumeration

The enumerations used in the BorderProperties group shall be defined as follows:

- **BorderSinglecolorLinestyles** - specifies the single-color linestyle for the border of the component. Enumerated values are:
  - "solid" - the boder is drawn as a singlecolor solid line;
  - "dashed" - the border is drawn as a singlecolor dashed line;
  - "dotted" - the border is drawn as a singlecolor dotted line;
  - "double" - the border is drawn as a singlecolor double line.

C.4.1.2 BordermulticolorLinestyle enumeration

- **BorderMulticolorLinestyles** - specifies the multi-color linestyle for the border of the component. Enumerated values are:
  - groove - the border looks as though it were carved into the canvas;
  - ridge - the opposite of groove. The border looks as though it were coming out of the canvas;
  - bevelled-outset - the border makes the entire component look as though it were embedded in the canvas;
  - bevelled-inset - the border makes it look as though the entire component were coming out of the canvas.

C.4.2 BorderProperties specification

C.4.2.1 Border-Width

```
<PropertySpec name="border-width" type="integer" default="2" access="readWrite"/>
```

Specifies in pixels the width of the border. Default value is "2".

C.4.2.2 Side-specific property application

All borderproperties can be specified as generic border properties to be applied to all sides of a component, or as a border property specific to a single side of a component.

Any border property may also be applied only to a single side of a component. The side shall be declared as Left, Right, Top or Bottom.

EXAMPLE: The **Border-Width** property specifies border width for all sides of a visible component. The **BorderTopWidth** property specifies border width only for the top border of the component.

C.4.3 BorderProperties defined elsewhere

The generic set of border properties are defined in the property groups listed below.

- ColorPropertiesBordercolor: clause C.3.3.
- LinestyleProperties: clause C.6.
- CornerradiusProperties: clause C.5.

C.5 CornerRadius properties

```
    <PropertyGroup name="CornerRadius_properties">
        <PropertySpec name="CornerRadius" type="integer" use="optional" default="0"
access="initializeOnly"/>
        <PropertySpec name="CornerRadius-NW" type="integer" default="0" access="initializeOnly"/>
        <PropertySpec name="CornerRadius-NE" type="integer" default="0" access="initializeOnly"/>
        <PropertySpec name="CornerRadius-SW" type="integer" default="0" access="initializeOnly"/>
        <PropertySpec name="CornerRadius-SE" type="integer" default="0" access="initializeOnly"/>
    </PropertyGroup>
```

The CornerRadius property group defines in pixels the radius to which corners of visible components are to be rounded. The default value of "0" means no rounding shall be applied to the corners of the component.

A single CorneRradius specification can be made within a component specification to apply CornerRadius uniformly to each corner of the component.

CornerRadius can also be applied individually to single corners of a component. The corners shall be declared as NW, NE, SW and SE.

EXAMPLE: The CornerRadius property specifies corner radius for all corners of a visible component. The CornerRadius NW property specified CornerRadius only for the top-left corner of the component.

C.6 LineStyle properties

```
    <EnumerationSpec name="linestyles">
        <EnumerationItem name="solid"/>
        <EnumerationItem name="dashed"/>
        <EnumerationItem name="dotted"/>
        <EnumerationItem name="double"/>
        <EnumerationItem name="groove"/>
        <EnumerationItem name="ridge"/>
        <EnumerationItem name="bevelled-inset"/>
        <EnumerationItem name="bevelled-outset"/>
    </EnumerationSpec>
    <PropertyGroup name="linestyle_properties">
        <PropertySpec name="linestyle" type="enumeration" default="solid" access="initializeOnly">
            <EnumerationRef ref="linestyles"/>
        </PropertySpec>
        <PropertySpec name="line-width" type="integer" default="2" access="initializeOnly"/>
    </PropertyGroup>
```

The **LineStyleProperties** property group specifies the set of linestyle related properties for use in PCF components.

C.6.1 Linestyle enumerations

```
<EnumerationSpec name="linestyles">
        <EnumerationItem name="solid"/>
        <EnumerationItem name="dashed"/>
        <EnumerationItem name="dotted"/>
        <EnumerationItem name="double"/>
        <EnumerationItem name="groove"/>
        <EnumerationItem name="ridge"/>
    </EnumerationSpec>
```

The enumerations used in the LinestyleProperties group shall be defined as follows:

- Solid - the line shall be drawn as a solid line.
- Dashed - the line shall be drawn as a dashed line.
- Dotted - the line shall be drawn as a dotted line.
- Double - the line shall be drawn as a double line.

- Groove - the line shall be drawn such that it appears to be cut into the canvas.
- Ridge - the opposite to groove. The line shall be drawn such that it appears to come out of the canvas.

C.6.2 LineStyle properties specification

C.6.2.1 Linestyle

```
<PropertySpec name="linestyle" type="enumeration" default="solid" access="initializeOnly">
    <EnumerationRef ref="linestyles"/>
</PropertySpec>
```

The linestyle property specifies the linestyle to be applied the line. Linestyles are defined in linestyle enumerations, clause C.6.1.

C.6.2.2 Linewidth

```
<PropertySpec name="line-width" type="integer" default="2" access="initializeOnly"/>
```

The LineWidth property specifies in pixels the width of the line. Default value is 2 pixels.

C.7 Positioning and layout properties

C.7.1 PositioningPropertiesAbsolute

```
<PropertyGroup name="positioning_properties-absolute">
    <PropertySpec name="origin" type="position" use="optional" access="readWrite"/>
    <PropertySpec name="size" type="size" use="optional" access="readWrite"/>
</PropertyGroup>
```

The PositioningPropertiesAbsolute group specifies absolute positions for component in the PCF.

C.7.1.1 Origin

```
<PropertySpec name="origin" type="position" use="optional" access="readWrite"/>
```

The Origin property specifies the pixel location to be used as the origin of the component. The Origin specification is made within the content of the parent container.

EXAMPLE: An Origin value of "100 100" for a visible component nested directly within a **Scene** means that the component will be drawn such that the top left corner will be 100 pixels below and 100 pixels to the right of the top left corner of the reference screen area. A visible component with an Origin value of "100, 100" declared within an **ELC** will be drawn such that the top left corner of the component will be offset 100 pixels to the right and 100 pixels below the Origin of the **ELC**.

C.7.1.2 Size

```
<PropertySpec name="Size" type="size" use="optional" access="readWrite"/>
```

The Size property specifies in pixels the X and Y dimensions of the component. The Size specification is made within the reference screen area.

EXAMPLE: A component with a Size definitions of "100 200" shall be drawn 100 pixels wide and 200 pixels deep.

C.7.2 Flow layout properties

```
<PropertyGroup name="layout_properties-flow">
        <PropertySpec access="initializeOnly" name="white-space" type="enumeration">
            <EnumerationSpec name="white-space-handling">
                <EnumerationItem name="normal"/>
                <EnumerationItem name="pre"/>
                <EnumerationItem name="no-wrap"/>
            </EnumerationSpec>
        </PropertySpec>
        <PropertyGroupRef ref="alignment_properties"/>
        <PropertyGroupRef ref="padding_properties"/>
        <PropertyGroupRef ref="margin_properties"/>
        <PropertyGroupRef ref="border_properties"/>
    </PropertyGroup>
```

The LayoutPropertiesFlow property group specifies the flow layout related properties for PCF components.

C.7.2.1 WhiteSpaceHandling property

```
        <PropertySpec access="initializeOnly" name="white-space" type="enumeration">
            <EnumerationSpec name="white-space-handling">
                <EnumerationItem name="normal"/>
                <EnumerationItem name="pre"/>
                <EnumerationItem name="no-wrap"/>
            </EnumerationSpec>
        </PropertySpec>
```

The WhiteSpaceHandling enumeration shall be defined as follows:

- **normal** - This value directs PCF implementations to collapse sequences of white space, and to break lines as necessary to fill line boxes. Additional line breaks may be created by occurrences of the
 element.
- **pre** - This value prevents PCF implementations from collapsing sequences of white space. Lines are only broken at newlines in the source, or at occurrences of the
 element.
- **nowrap** - This value collapses whitespace as for "normal", but suppresses line breaks within text except for those created by the
 element.

C.7.2.2 Flow layout properties defined elsewhere

The flow layout properties are defined in the property groups below:

- Alignment properties, clause C.7.3.
- Padding properties, clause C.8.1.
- Margin properties, clause C.8.2.
- Border properties, clause C.4.2.

C.7.3 Alignment-properties

```
    <EnumerationSpec name="horizontal-alignments">
        <EnumerationItem name="left"/>
        <EnumerationItem name="center"/>
        <EnumerationItem name="right"/>
        <EnumerationItem name="justify"/>
    </EnumerationSpec>
    <EnumerationSpec name="vertical-alignments">
        <EnumerationItem name="top"/>
        <EnumerationItem name="middle"/>
        <EnumerationItem name="baseline"/>
        <EnumerationItem name="bottom"/>
        <EnumerationItem name="sub"/>
        <EnumerationItem name="super"/>
        <EnumerationItem name="text-top"/>
        <EnumerationItem name="text-bottom"/>
```

```
    </EnumerationSpec>
    <PropertyGroup name="alignment_properties">
        <PropertySpec name="h-align" type="enumeration" access="initializeOnly" default="left">
            <EnumerationRef ref="horizontal-alignments"/>
        </PropertySpec>
        <PropertySpec name="v-align" type="enumeration" default="top" access="initializeOnly">
            <EnumerationRef ref="vertical-alignments"/>
        </PropertySpec>
    </PropertyGroup>
```

C.7.3.1 Alignment properties enumerations

C.7.3.1.1 Horizontal alignment enumeration

```
<EnumerationSpec name="horizontal-alignments">
        <EnumerationItem name="left"/>
        <EnumerationItem name="center"/>
        <EnumerationItem name="right"/>
        <EnumerationItem name="justify"/>
</EnumerationSpec>
```

The horizontal alignment enumeration shall be defined as follows:

- left - left aligned;
- right - right aligned;
- centre - centre aligned;
- justify - justified aligned.

C.7.3.1.2 Vertical alignment enumeration

```
    <EnumerationSpec name="vertical-alignments">
        <EnumerationItem name="top"/>
        <EnumerationItem name="middle"/>
        <EnumerationItem name="baseline"/>
        <EnumerationItem name="bottom"/>
        <EnumerationItem name="sub"/>
        <EnumerationItem name="super"/>
        <EnumerationItem name="text-top"/>
        <EnumerationItem name="text-bottom"/>
    </EnumerationSpec>
```

The vertical alignment enumeration shall be defined as follows:

- Top: align the top of the box with the top of the line box.
- Middle: align the vertical midpoint of the box with the baseline of the parent box plus half the x-height of the parent.
- Baseline: align the baseline of the box with the baseline of the parent box. If the box does not have a baseline, align the bottom of the box with the parent's baseline.
- Bottom: align the bottom of the box with the bottom of the line box.
- Sub: lower the baseline of the box to the proper position for subscripts of the parent's box.

NOTE 1: This value has no effect on the font size of the element's text.

- super: raise the baseline of the box to the proper position for superscripts of the parent's box.

NOTE 2: This value has no effect on the font size of the element's text.

- TextTop: align the top of the box with the top of the parent element's font.
- TextBottom: align the bottom of the box with the bottom of the parent element's font.

C.7.3.2 H-align property

```
<PropertySpec name="h-align" type="enumeration" access="initializeOnly" default="left">
    <EnumerationRef ref="horizontal-alignments"/>
</PropertySpec>
```

The H-align property specifies the horizontal alignment for content within a component.

H-align values are defined in the horizontal-alignments enumeration, clause C.7.3.1.1.

C.7.3.3 V-align property

```
<PropertySpec name="v-align" type="enumeration" default="top" access="initializeOnly">
    <EnumerationRef ref="vertical-alignments"/>
</PropertySpec>
```

The V-align property specifies the vertical alignment for content within a component.

V-align values are defined in the vertical-alignments enumeration, clause C.7.3.1.2.

C.8 Padding and margin properties

C.8.1 Padding properties

```
<PropertyGroup name="padding_properties">
    <PropertySpec name="padding" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="padding-top" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="padding-bottom" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="padding-left" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="padding-right" type="integer" default="0" access="initializeOnly"/>
</PropertyGroup>
```

The padding properties group defines padding to be applied between component edge and contained content.

C.8.1.1 Padding

```
<PropertySpec name="padding" type="integer" default="0" access="initializeOnly"/>
```

The padding property specifies in pixels the amount of space to be left between a component border and the content it contains. Default value is "0".

C.8.1.2 Side-specific padding application

Padding can be specified as a generic property to be applied to all sides of a component, or as a property specific to a single side of a component.

Where padding is to be applied to a single side of a component, the side shall be declared as Left, Right, Top or Bottom.

EXAMPLE: The **Padding** property specifies padding for all sides of a visible component. The **PaddingTop** property specifies padding only for the top border of the component.

C.8.2 Margin properties

```
<PropertyGroup name="margin_properties">
    <PropertySpec name="margin" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="margin-top" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="margin-bottom" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="margin-left" type="integer" default="0" access="initializeOnly"/>
    <PropertySpec name="margin-right" type="integer" default="0" access="initializeOnly"/>
</PropertyGroup>
```

The margin property group defines margin to be applied around the component edge.

C.8.2.1 Margin

```
<PropertySpec name="margin" type="integer" default="0" access="initializeOnly"/>
```
The margin property specifies in pixels the margin to be applied around the component edge.; default value is "0".

C.8.2.2 Side-specific margin application

Margin can be specified as a generic property to be applied to all sides of a component, or as a property specific to a single side of a component.

Where margin is to be applied to a single side of a component, the side shall be declared as Left, Right, Top or Bottom.

EXAMPLE: The **Margin** property specifies margin for all sides of a visible component. The **MarginTop** property specifies margin only for the top border of the component.

C.9 Font properties

C.9.1 Font family

```
<PropertySpec name="font-family" type="font-family" use="optional" default="sans-serif"/>
```

The font-family property specification for PCF shall be as specified in clause 15.2.2 of CSS2 [22].

C.9.2 Font emphasis

```
<PropertySpec name="font-style type="enumeration" use="optional" default="normal">
    <EnumerationSpec name="font-emphasis">
        <EnumerationItem name="em"/>
        <EnumerationItem name="strong"/>
        <EnumerationItem name="i"/>
        <EnumerationItem name="b"/>
        <EnumerationItem name="u"/>
        <EnumerationItem name="big"/>
        <EnumerationItem name="small"/>
    </EnumerationSpec>
</PropertySpec>
```

The font-emphasis property specification for PCF shall be as specified in clause 11.8.2 of WML 1.3 [24].

C.9.3 Font style

```
<PropertySpec name="font-style type="enumeration" use="optional" default="normal">
    <EnumerationSpec name="font-style">
        <EnumerationItem name="normal"/>
        <EnumerationItem name="italic"/>
        <EnumerationItem name="oblique"/>
        <EnumerationItem name="inherit"/>
    </EnumerationSpec>
</PropertySpec>
```

The font-style property specification for PCF shall be as specified in clause 15.2.3 of CSS2 [22].

C.9.4 Font variant

```
<PropertySpec name="font-variant type="enumeration" use="optional" default="normal">
    <EnumerationSpec name="font-variant">
        <EnumerationItem name="normal"/>
        <EnumerationItem name="small-caps"/>
        <EnumerationItem name="inherit"/>
    </EnumerationSpec>
</PropertySpec>
```

The font-variant property specification for PCF shall be as specified in clause 15.2.3 of CSS2 [22].

C.9.5 Font weight

```
<PropertySpec name="font-weight type="enumeration" use="optional" default="normal">
    <EnumerationSpec name="font-weight">
        <EnumerationItem name="normal"/>
        <EnumerationItem name="bold"/>
        <EnumerationItem name="bolder"/>
        <EnumerationItem name="light"/>
        <EnumerationItem name="lighter"/>
        <EnumerationItem name="100"/>
        <EnumerationItem name="200"/>
        <EnumerationItem name="300"/>
        <EnumerationItem name="400"/>
        <EnumerationItem name="500"/>
        <EnumerationItem name="600"/>
        <EnumerationItem name="700"/>
        <EnumerationItem name="800"/>
        <EnumerationItem name="900"/>
        <EnumerationItem name="inherit"/>
    </EnumerationSpec>
</PropertySpec>
```

The font-weight property specification for PCF shall be as specified in clause 15.2.3 of CSS2 [22].

C.9.6 Font stretch

```
<PropertySpec name="font-stretch type="enumeration" use="optional" default="normal">
    <EnumerationSpec name="font-stretch">
        <EnumerationItem name="normal"/>
        <EnumerationItem name="wider"/>
        <EnumerationItem name="narrower"/>
        <EnumerationItem name="ultra-condensed"/>
        <EnumerationItem name="extra-condensed"/>
        <EnumerationItem name="condensed"/>
        <EnumerationItem name="semi-condensed"/>
        <EnumerationItem name="semi-expanded"/>
        <EnumerationItem name="expanded"/>
        <EnumerationItem name="extra-expanded"/>
        <EnumerationItem name="ultra-expanded"/>
        <EnumerationItem name="inherit"/>
    </EnumerationSpec>
</PropertySpec>
```

The font-stretch property specification for PCF shall be as specified in clause 15.2.3 of CSS2 [22].

C.9.7 Font size

```
<PropertySpec name="font-size type="font-size" use="optional" default="medium"/>
```

The font-size property specification for PCF shall be as specified in clause 15.2.4 of CSS2 [22].

C.9.8 Font-size-adjust

```
<PropertySpec name="font-size-adjust type="integer" nil="true" use="optional" default="nil"/>
```

The font-size property specification for PCF shall be as specified in clause 15.2.4 of CSS2 [22].

C.10 The LabelProperties property group

```
<PropertyGroup name="label_properties">
    <PropertySpec name="label" type="string" use="optional" default="" access="readWrite"/>
    <PropertySpec name="label-focused" type="string" use="optional" default="" access="readWrite"/>
    <PropertySpec name="label-disabled" type="string" use="optional" default="" access="readWrite"/>
    <PropertySpec name="label-active" type="string" use="optional" default="" access="readWrite"/>
```

```
    <PropertySpec name="label-idle" type="string" use="optional" default="" access="readWrite"/>
</PropertyGroup>
```

The LabelProperties property group defines a generic label string, as well as label string values to be used in each of the focus states supported by **Button** components: focused, disabled, active and idle.

C.11 The ImageProperties property group

```
<PropertyGroup name="image_properties">
    <PropertySpec name="imageURL" type="uri" use="optional" default="" access="readWrite"/>
    <PropertySpec name="imageURL-focused" type="uri" use="optional" default="" access="readWrite"/>
    <PropertySpec name="imageURL-disabled" type="uri" use="optional" default="" access="readWrite"/>
    <PropertySpec name="imageURL-active" type="uri" use="optional" default="" access="readWrite"/>
    <PropertySpec name="imageURL-idle" type="uri" use="optional" default="" access="readWrite"/>
</PropertyGroup>
```

The ImageProperties property group defines a generic **imageURL** for image source, as well as alternate image locations to satisfy each of the focus states supported by the Image component: focused, disabled, active and idle.

C.12 The AnimationProperties property group

```
<PropertyGroup name="animation_properties">
    <PropertySpec name="framePeriod" type="integer" use="required" access="readWrite"/>
    <PropertySpec name="running" type="boolean" use="optional" default="true" access="readWrite"/>
    <PropertySpec name="numberOfLoops" type="integer" use="optional" default="0"
access="readWrite"/>
    <PropertySpec name="loopPause" type="integer" use="optional" default="0" access="readWrite"/>
</PropertyGroup>
```

The AnimationProperties property group defines properties to be used in ImageAnimated and **Ticker** components.

C.12.1 The FramePeriod property

```
<PropertySpec name="framePeriod" type="integer" use="required" access="readWrite"/>
```

The **frameperiod** property defines the movement speed in milliseconds per pixel.

C.12.2 The Running property

```
<PropertySpec name="running" type="boolean" use="optional" default="true" access="readWrite"/>
```

The **running** property defines whether or not the **Ticker** or ImageAnimated is currently running, i.e. moving.

C.12.3 TheNumberOfLoops component

```
<PropertySpec name="numberOfLoops" type="integer" use="optional" default="0" access="readWrite"/>
```

The **number-of-loops** property defines the number of loops of the **Ticker** or ImageAnimated content to show before stopping. If this value is zero then the **Ticker** or ImageAnimated shall loop continuously.

C.12.4 The LoopPause component

```
<PropertySpec name="loopPause" type="integer" use="optional" default="0" access="readWrite"/>
```

The **loopPause** property defines the number of milliseconds to wait between successive loops. During this pause the content shall remain visible.

Annex D (normative): Profile specifications

D.1 DVB profiles

D.1.1 Introduction

The following clauses define a number of standard DVB profiles that reflect significant boundaries in the capabilities of digital television platforms and/or PCF transcoder complexity.

It is fairly apparent that none of these profiles contains the **ReturnPath** profile package (see clause C.1.7). However, this profile package can be easily included where required using the standard mechanism for declaring a profile as an extension to another existing profile. For example:

```
<profileDef name="mycom.com/myrpprofile">
  <BaseProfile id="dvb.org/pcf/profiles/basic"/>
  <package id="dvb.org/pcf/package/returnpath" level="1"/>
</profileDef>
```

D.1.2 dvb.org/pcf/profile/basic

D.1.2.1 Overview

This profile has been defined so as to encapsulate what is believed to be the minimum useful sub-set of PCF features. Consequently a service described using PCF features restricted to this profile is expected to offer a high degree of portability, whilst requiring the minimum of PCF support within a target platform.

D.1.2.2 Definition

This profile shall be defined as follows:

```
<profileDef name="dvb.org/pcf/profiles/basic">
  <package id="dvb.org/pcf/package/architecture" level="1"/>
  <package id="dvb.org/pcf/package/behaviour" level="1"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="1"/>
  <package id="dvb.org/pcf/package/layout/text" level="1"/>
</profileDef>
```

D.1.3 dvb.org/pcf/profile/core

D.1.3.1 Overview

This profile has been defined so as to encapsulate what is believed to be the maximum sub-set of PCF features that is still expected to offer a high degree of portability. This is likely to require a considerable degree of PCF support within a target platform.

D.1.3.2 Definition

This profile shall be defined by either of the following, which are equivalent and equally valid:

```
<profileDef name="dvb.org/pcf/profiles/core">
  <package id="dvb.org/pcf/package/architecture" level="2"/>
  <package id="dvb.org/pcf/package/behaviour" level="3"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="1"/>
  <package id="dvb.org/pcf/package/layout/flow" level="1"/>
  <package id="dvb.org/pcf/package/layout/text" level="1"/>
```

```
</profileDef>

<profileDef name="dvb.org/pcf/profiles/core">
  <baseProfile id=" dvb.org/pcf/profiles/basic"/>
  <package id="dvb.org/pcf/package/architecture" level="2"/>
  <package id="dvb.org/pcf/package/behaviour" level="3"/>
  <package id="dvb.org/pcf/package/layout/flow" level="1"/>
</profileDef>
```

D.1.4 dvb.org/pcf/profile/full

D.1.4.1 Overview

This profile has been defined so as to encapsulate all of the features of the present document, with the exception of the **ReturnPath**.

D.1.4.2 Definition

This profile shall be defined by either of the following, which are equivalent and equally valid:

```
<profileDef name="dvb.org/pcf/profiles/full">
  <package id="dvb.org/pcf/package/architecture" level="2"/>
  <package id="dvb.org/pcf/package/behaviour" level="4"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="2"/>
  <package id="dvb.org/pcf/package/layout/flow" level="2"/>
  <package id="dvb.org/pcf/package/layout/text" level="2"/>
</profileDef>

<profileDef name="dvb.org/pcf/profiles/full">
  <baseProfile id=" dvb.org/pcf/profiles/core"/>
  <package id="dvb.org/pcf/package/behaviour" level="4"/>
  <package id="dvb.org/pcf/package/layout/explicit" level="2"/>
  <package id="dvb.org/pcf/package/layout/flow" level="2"/>
  <package id="dvb.org/pcf/package/layout/text" level="2"/>
</profileDef>
```

Annex E (normative): Portable hints for PCF data exchange

type	values	description
priority	low	marks a transaction to be serviced with a lower priority than those marked with "normal" or "high" priority.
	normal	default priority. Marks a transaction to be serviced with a priority between those marked with "low" and "high" priorities.
	high	marks a transaction to be serviced with a priority above those marked with "low" and "normal" priorities.
validity_period	[0..MAX_INT]	gives a number of seconds for which any volatile assets in the transaction may be cached by a network or transcoder. The value 0 may be used to indicate no caching is allowed. This allows an author to request that the asset be kept as current as possible by the platform. Actual behaviour will be platform-specific.

Annex F (normative): Marked up text format

The elements that make up the marked up text format are described in clauses F.1 to F.6.

F.1 Block elements

The root of all marked up text items is the **body element** ("body"). Body elements shall contain a sequence of zero or more **block elements**. Block elements are: **paragraph elements** ("p"); **horizontal rule elements** ("hr"); **table elements** ("table"). Body and block elements are illustrated in figure 93.

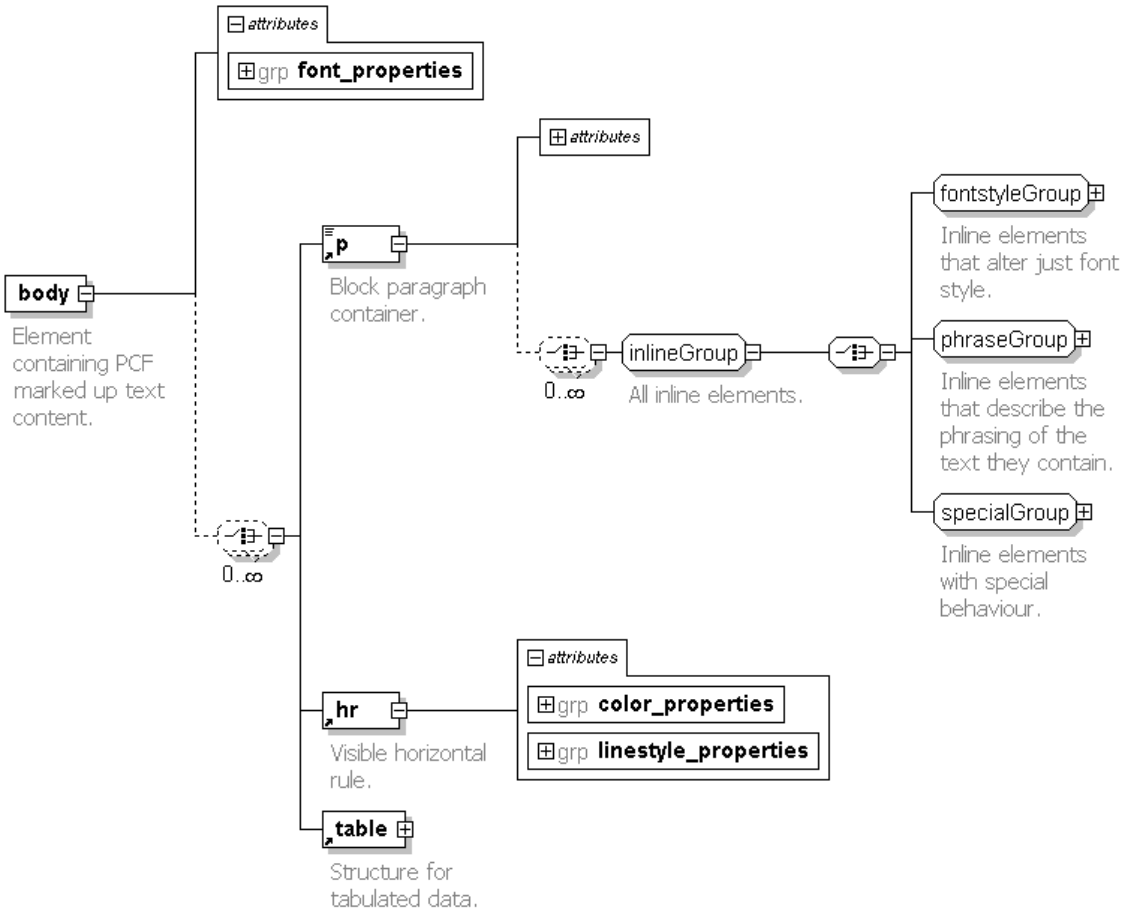

Figure 93: Marked up text block elements

Body elements can have font properties that specify font characteristics that can be applied to all text within the marked up text item.

Paragraph element ("p") is a mixed content element that shall contain text interleaved with a sequence of zero or more elements that are members of the inline element group. The inline element group itself consists of three other element groups:

- **font style group** - elements that change the font style of the text they contain, as described in clause F.2;
- **phrase group** - elements that change the phrasing of the text they contain, as described in clause F.3;
- **special group** - elements with a specialized effect to the content they contain, as described in clause F.4.

Paragraph elements may have font properties specified that can be applied to alter the style of any text contained within the paragraph, overriding any styles specified in the paragraph's enclosing tag.

Horizontal rule ("hr") elements draw horizontal lines that span the width of the container of the marked up text, between other block elements. Horizontal rules can have colour properties and line style properties specified.

Table elements contain structured table content, as described in clause F.5.

EXAMPLE: The simplest possible PCF marked up content element consists of an empty body tag, as shown below:

```
<body xmlns="http://www.dvb.org/pcf/x-dvb-pcf"/>
```

F.2 Font style elements

Font style elements alter the font style of all text they contain. The PCF currently has two font style elements **big** and **small**, as illustrated in figure 94. Rendering of font style elements depends on the user agent.

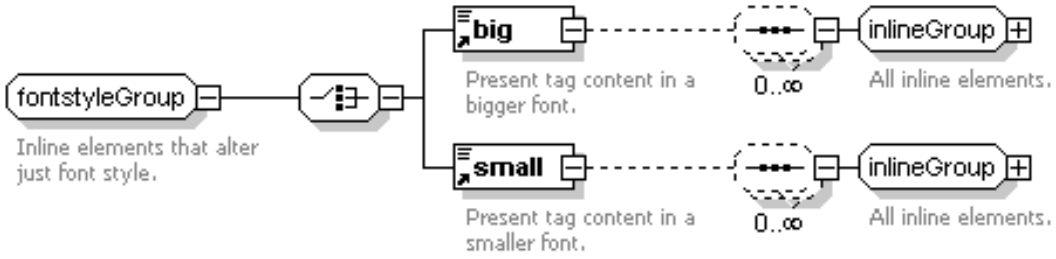

Figure 94: Marked up text font style inline elements

The following is informative description of how to render the font style elements:

- Big elements present the text content they contain with a larger font than that of the font size before the big element.
- Small elements present the text content they contain with a smaller font than that of the font size before the big element.

F.3 Phrase elements

Phrase elements add structural information to the text they contain. The PCF has two phrase elements as illustrated in figure 95: **emphasis element** ("em"); **stronger emphasis element** ("strong"). The rendering of font style elements depends on the user agent.

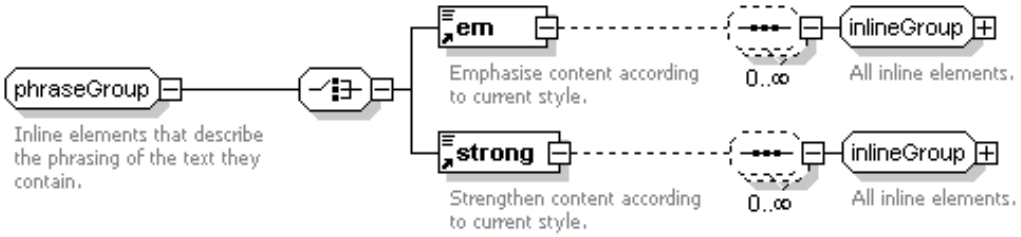

Figure 95: Marked up text phrase inline elements

The following is informative description of how to render phrase elements:

- "em" elements may cause the text they contain to be rendered in an italicized font;

- "strong" elements may cause the text they contain to be rendered in an emboldened font.

F.4 Special elements

The special elements of PCF allow special effects, such as font styling, to be applied to textual content. This includes:

- **anchor elements** ("a") - contain inline navigation references to other PCF **Scene**s or content;
- **line break elements** ("br") - insert a line break between text within a paragraph;
- **image elements** ("img") - insert an inline image by URI reference in the flow of text content;
- **span elements** ("span") - apply a font styling to the contained content.

The special elements are shown in figure 96.

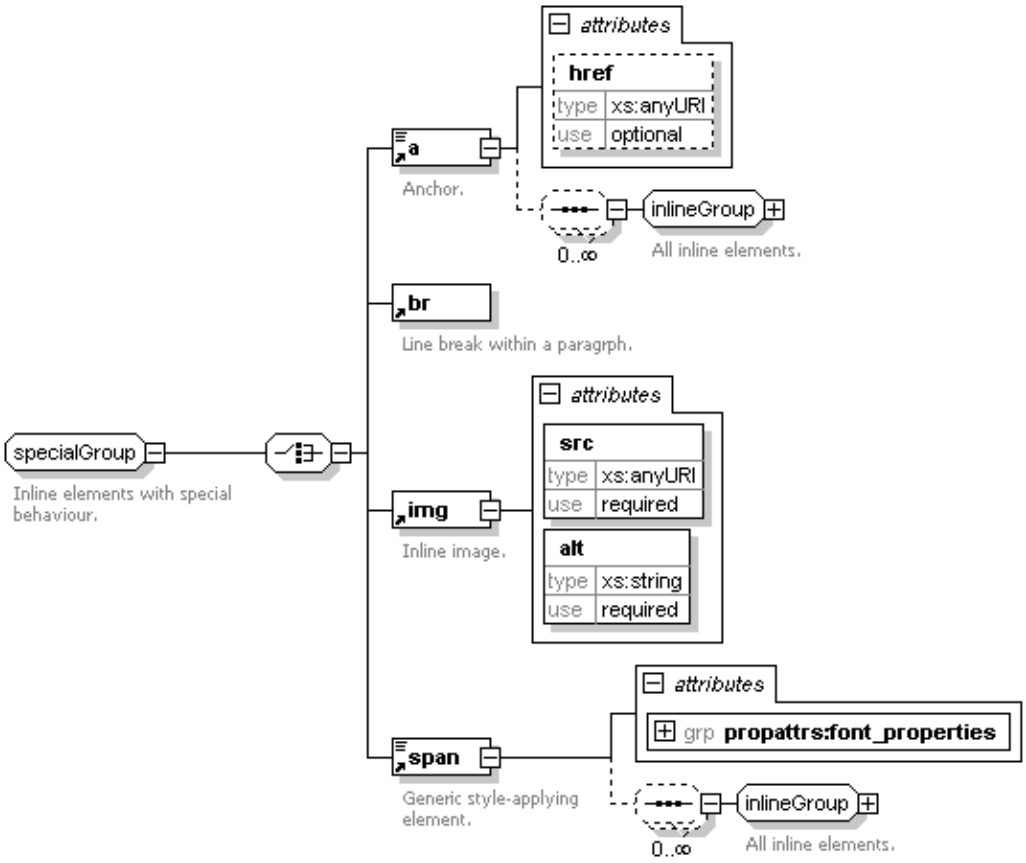

Figure 96: Marked up text special inline elements

The anchor element has a hypertext reference attribute ("href") that is a URI link. When the content is presented to a user, the presentation of the content within the anchor may in a different style to indicate the presence of a link to a user. A user may also be able to navigate to the link and select it to navigate to the content that it references.

NOTE 1: The way that the textual content of the anchor element is presented to the user is not specified here. Neither is the means in which a user can focus on the link, select it and navigate to the content it references. Component specifications that make use of the anchor element should provide the details of the presentation and behaviour of hypertext links.

The line break element shall cause a line break to be inserted into the text at the position where the element is located, i.e. any text after a line break element shall be presented to the user on a new line.

The image element may be replaced by the image content that is referenced by the URI contained in the source attribute ("src") it contains. A user may be presented with the text contained in the alternative attribute ("alt") as part of the presentation of the text content or the behaviour of the component that is presenting the component.

NOTE 2: The presentation of the image in relation to the text around it is described in clause 8.3.4 with the flow layout detail of the PCF specification.

The span element allows an author to specify any allowable PCF font style property that may be applied in the presentation of the text contained within the element.

NOTE 3: The presentation of the text is not required to be in the exact style specified by the span element. A transcoder may choose to try to match with the closest font available on a particular platform or ignore the span element altogether.

EXAMPLE: The following fragment of a marked up text item shows how a different font style can be applied to a word within a sentence. A transcoder should try to apply as many of the styles as possible to the word.

```
<p>Press the
    <span font-family="Helvetica" font-weight="bolder"
        textcolor="#af0000" font-stretch="wider">RED</span>
    button for more information.
</p>
```

F.5 Table structures

Table elements are block elements that represent the structured textual content of two-dimensional tabulated information. Table structures can be of any size and the table element and its sub-elements can be used to specify properties for the presentation of the text within tables. The elements of a table structure are illustrated in figure 97.

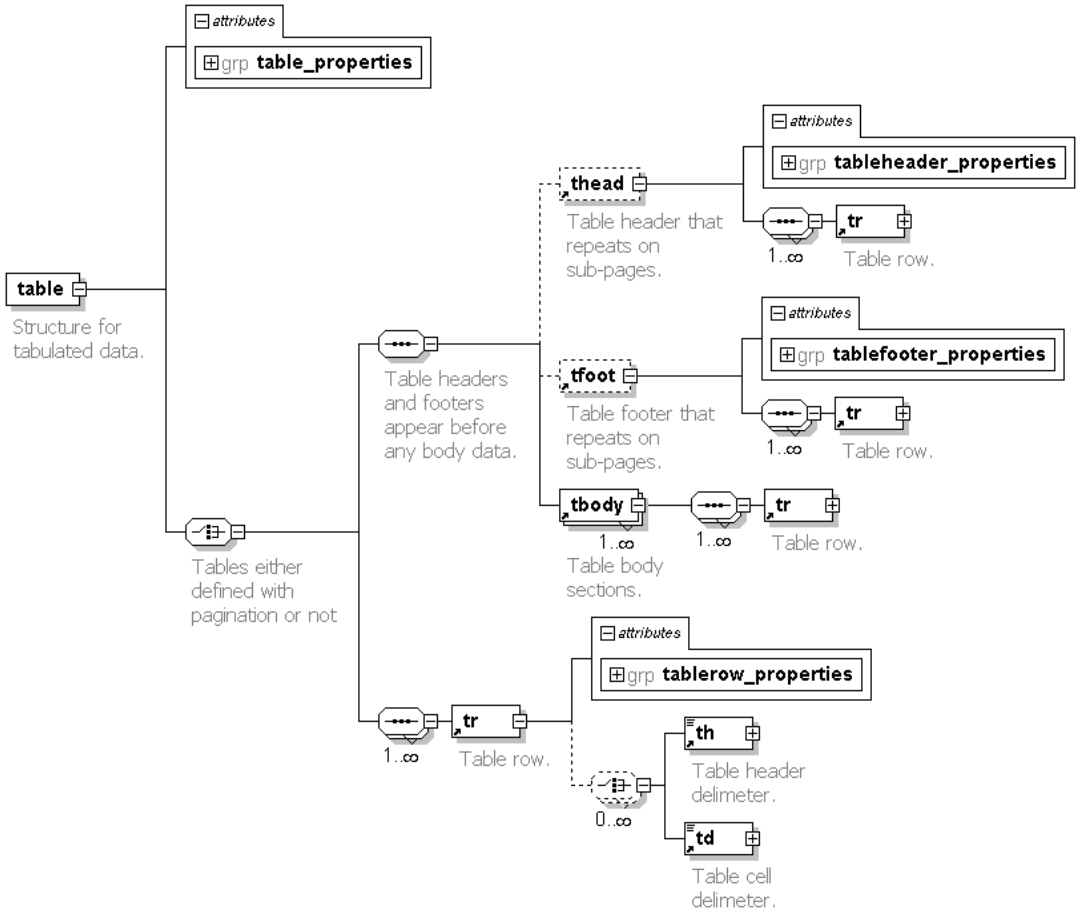

Figure 97: Marked up text table structures

Table elements shall contain either a sequence a paginated table item, as described below, or **TR** items, as described in clause F.6. Table items may have table property attributes specified to indicate how the table and any items it contains should be presented.

A paginated table item enables authors to describe larger amounts tabulated data that it may be necessary to split into pages in order for a user to access all of it. Paginated table items consist of: header and footer **TR**s that are displayed on every page; **TB** sections that collect together information that should be displayed together on the same page. A paginated table item shall consist of a sequence of the following elements:

- **table header element** ("thead") - an optional table header that shall contain a sequence of one or more table row items and may have some table header styling property attributes;
- **table footer element** ("tfoot") - an optional table footer that shall contain a sequence of one or more table row items and may have some table footer styling property attributes;
- **table body elements** ("tbody") - one or more table bodies, each of which contains a sequence of one or more table rows that describe the text content for the body of a page of tabulated information.

NOTE: The presentation of a paginated table is platform dependent and shall depend on the component used to display the tabulated information. The behaviour that allows a user to navigate between pages of tabulated content may be specified by a component and will also be platform dependent. Table bodies can be joined together by a transcoder but, where possible, the table bodies should not be split across multiple pages.

EXAMPLE: The following fragment of a marked up text item shows how the text content for a table containing details a rugby championship may be split across two pages, each page having the same table header.

```
<table>
    <thead>
        <tr> <th>Team</th> <th>Played</th> <th>Points</th> </tr>
    </thead>
    <tbody>
        <tr> <td>Wales</td> <td>4</td> <td>8</td> </tr>
        <tr> <td>Ireland</td> <td>4</td> <td>6</td> </tr>
        <tr> <td>France</td> <td>4</td> <td>6</td> </tr>
    </tbody>
    <tbody>
        <tr> <td>England</td> <td>4</td> <td>2</td> </tr>
        <tr> <td>Scotland</td> <td>4</td> <td>2</td> </tr>
        <tr> <td>Italy</td> <td>4</td> <td>0</td> </tr>
    </tbody>
</table>
```

F.6 Table rows

Table row elements contain a sequence of zero or more cells of text content. The cells are either **table header cell elements** ("th") or **standard table cell elements** ("td"), as shown in figure 98.

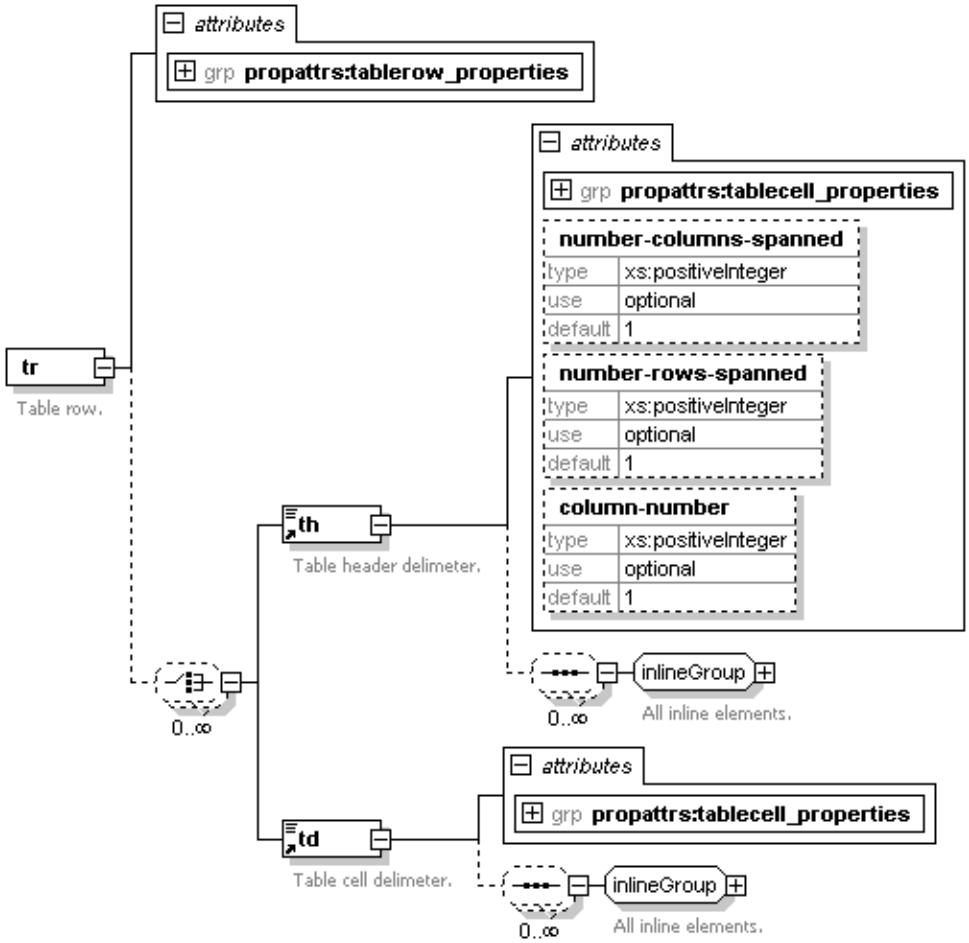

Figure 98: Marked up text table rows

Table header cell elements and standard table cell elements shall both contain text and elements from the inline content element group, as defined in clauses F.1 to F.4. Both may have table cell style attributes applied to them.

Table header cell elements may be presented with a table header cell styling to distinguish them from standard table cell elements.

Annex G (normative): XML resources

There are a number of resources that can be used in the authoring and validation of a PCF description of an interactive service. These take the form of XML schemas and documents. Version 1.0 of the schemas accompany the present document and are contained in archive ts_102523v010101p0.zip.

A brief description of each is provided here.

G.1 PCF syntax

G.1.1 pcf.xsd

This is the core PCF schema.

NOTE: **behaviour.xsd** and **schemacomponents.xsd** are included into **pcf.xsd** schema and so share the same namespace.

G.1.2 behaviour.xsd

This schema describes behaviour including statemachines, OnEvents and ActionLanguage.

G.1.3 schemacomponents.xsd

This schema describes the schema components.

G.1.4 pcf-types.xsd

This schema describes the PCF data types and abstract structures.

G.1.5 x-dvb-pcf.xsd

This schema describes PCF marked up text.

G.1.6 servicedigest.xsd

This schema describes information about, and profiling details of a PCF service.

G.2 Component definition syntax

G.2.1 component-syntax.xsd

This schema describes the component definition syntax.

G.2.2 components.xml

This xml file describes the PCF components, including their properties and behaviour.

G.2.3 propertygroups.xml

This xml file describes groups of properties that form part of the component descriptions in **components.xml**.

G.2.4 eventgroups.xml

This xml file describes groups of events that form part of the component descriptions in **components.xml**.

G.2.5 events.xml

This xml file describes the events (which includes errors) recognized within the PCF.

G.3 Transport and packaging

G.3.1 transport.wsdl

This web service description language (wsdl) document specifies the PCF transport and packaging SOAP interface.

Annex H (normative): Action language expression function library

A library of functions that is available within expressions of the PCF action language.

H.1 Type conversion functions

Type conversion functions allow values of one run-time data type to be converted to values of another. All the available run-time data types conversions are enumerated in clause H.1.1 and then described in detail in clauses H.1.2 to H.1.12.

H.1.1 Available type conversions

Table 36 shows type conversion functions that are available within the PCF action language.

Table 36: Available type conversions

Type conversion	boolean	color	component	currency	date	dateTime	integer	markedUp Text	position	size	string	time	timecode	URI
boolean		X	X	X	X	X	1	X	X	X	1	X	X	X
color	X		X	X	X	X	>4	X	X	X	1	X	X	X
component	X	X		X	X	X	X	X	X	X	X	X	X	X
currency	X	X	X		X	X	1	X	X	X	1	X	X	X
date	X	X	X	X		1/*	>4	X	X	X	1	X	X	X
dateTime	X	X	X	X	1		>9	X	X	X	1	1	X	X
integer	1	3-4>	X	X	3-5>	6-9>		X	2>	2>	1	3-6>	X	X
markedUpText	X	X	X	X	X	X	X		X	X	1	X	X	X
position	X	X	X	X	X	X	>2	X		X	1	X	X	X
size	X	X	X	X	X	X	>2	X	X		1	X	X	X
string	X	X	X	1	X	X	1	1	X	X		X	X	X
time	X	X	X	X	X	*	>6	X	X	X	1		X	X
timecode	X	X	X	X	X	X	>4	X	X	X	1	X		X
URI	X	X	X	X	X	X	X	X	X	X	X	X	X	

Information in the table shall be interpreted according to the following rules:

- To determine whether a type conversion function exists between values of two data types, the table shall be read row type name first, then column type name. These shall be known as the first value type and second value type respectively.

EXAMPLE 1: To determine whether it is possible to convert from a value of type **color** to a value of type **string,** look to the contents of the cell where the first value type (**color)** row meets the second value type (**string)** column.

The contents of the associated cell shall be interpreted as follows:

- **X** - No type conversion function exists from the first value type to the second value type.
- **1** - A type conversion function exists that creates exactly one value of the second type from one value of the first type.
- >***number*** - A type conversion function exists that creates *number* values of the second value type from one value of the first value type.

- ***number***> - A type conversion function exists that creates one value of the second value type from *number* values of the first value type.
- ***** - A hybrid type conversion function exists that takes values of two different types to make the value of another type.

EXAMPLE 2: The contents of the cell indicating conversion from **position** to **string** contains the value "1". This indicates that one value of type **position** can be converted to one value of type **string**. However, conversion from **position** to **integer** is characterized by ">2". This indicates that the conversion of a single value of type **position** shall produce two values of type **integer**.

EXAMPLE 3: The contents of the cells indicating a conversion from **time** and **date** to **dateTime** contain the hybrid conversion function *. This indicates that a value of type **dateTime** can be created from a value of type **time** and a value of type **date**.

NOTE: The existence of a type conversion between types is not associative. If a type conversion function exists between **type A** and **type B**, this does not imply that a type conversion function exists between **type B** and **type A**.

- The *number* of the first value type may be expressed as a range: "*number1-number2*". A number range indicates that the conversion function to the second value type shall accept a variable number of parameters of the first value type. The number of parameters shall lie within this inclusive range.

EXAMPLE 4: The table shows a conversion between values of type integer to a value of type color. The cell contents "3-4>" indicate that 3 integers can be converted to a value of type color (RGB values) and that 4 integers can be converted to a value of type color (RGB + transparency).

H.1.2 From boolean

H.1.2.1 Boolean to integer

Converts a boolean value to an integer value of 0 for false and 1 for true.

The signature of the function is:

```
integer booleanToInteger(boolean)
```

This function always succeeds.

H.1.2.2 Boolean to string

Converts a boolean value to a string value, where a false value is converted to the string "false" and true is converted to the string "true".

The signature of the function is:

```
string booleanToString(boolean)
```

The function always succeeds.

H.1.3 From color

H.1.3.1 Color to integer parts

Converts a color value to its four constituent integer parts.

The signature of the function is:

```
integerArray[4] colorToInteger(color)
```

The function always succeeds.

H.1.3.2 Color to string

Converts a color value to a string value.

The signature of the function is:

```
string colorToString(color)
```

The function always succeeds.

H.1.4 From currency

H.1.4.1 Currency to integer

Converts a currency value to an integer, removing the decimal point.

The signature of the function is:

```
integer currencyToInteger(currency)
```

The function always succeeds.

H.1.4.2 Currency to string

Converts a currency value to a string value, retaining the decimal point.

The signature of the function is:

```
string currencyToString(currency)
```

The function always succeeds.

H.1.5 From date

H.1.5.1 Date to dateTime

Converts a date value to a dateTime value, where the time is set to 00:00:00.000.

EXAMPLE: Conversion of the date value "2000-01-01+11:00" to a dateTime value would result in "2000-01-01T00:00:00.000+1100".

The signature of the function is:

```
dateTime dateToDateTime(date)
```

The function always succeeds.

H.1.5.2 Date to integer parts

Converts a date value to its constituent integer parts.

Five integers result from a date value that includes the (optional) time zone information.

The signature of the function is:

```
integerArray[5] dateAllToInteger(date)
```

Three integers result from a date value that does not include the (optional) time zone information.

The signature of the function is:

```
integerArray[3] dateToInteger(date)
```

The function always succeeds.

H.1.5.3 Date to string

Converts a date value to a string value.

The signature of the function is:

```
string dateToString(date)
```

The function always succeeds.

H.1.6 From dateTime

H.1.6.1 DateTime to date

Converts a dateTime value to a date value, omitting the time component.

The signature of the function is:

```
date dateTimeTodate(dateTime)
```

The function always succeeds.

H.1.6.2 DateTime to integer parts

Converts a dateTime value to its constituent integer parts.

EXAMPLE: The dateTime value "2000-10-11T12:34:56.789-12:30" converts to integer values of "2000", "10", "11", "12", "34", "56", "789", "12" and 30".

Nine integers result from a dateTime that includes all optional components.

The signature of the function is:

```
integerArray[9] dateTimeAllToInteger(dateTime)
```

Eight integers result from a dateTime that includes the (optional) time zone information but does not include the (optional) decimal fraction of a second.

The signature of the function is:

```
integerArray[8] dateTimeAllDateToInteger(dateTime)
```

Seven integers result from a dateTime that includes the (optional) decimal fraction of a second but does not include the (optional) time zone information.

The signature of the function is:

```
integerArray[7] dateTimeAllTimeToInteger(dateTime)
```

Six integers result from a dateTime that does not include the (optional) decimal fraction of a second nor the (optional) time zone information.

The signature of the function is:

```
integerArray[6] dateTimeToInteger(dateTime)
```

The function always succeeds.

H.1.6.3 DateTime to string

Converts a dateTime value to a string value.

The signature of the function is:

```
string dateTimeToString(dateTime)
```

The function always succeeds.

H.1.6.4 DateTime to time

Converts a dateTime value to a time value, omitting the date component.

The signature of the function is:

```
time dateTimeToTime(dateTime)
```

The function always succeeds.

H.1.7 From integer

H.1.7.1 Integer to boolean

Converts an integer to a boolean where the integer value 0 is converted to false and all other integers to true.

The signature of the function is:

```
boolean integerToBoolean(integer)
```

The function always succeeds.

H.1.7.2 Integer parts to color

Converts three or four integer values to a single color value. Each integer shall be converted to its hexadecimal representation.

If four integers are supplied then these shall determine the red, green, blue and transparency components respectively.

The signature of the function is:

```
color integerToColor(integerArray[4])
```

If only three integers are supplied then these shall determine the red, green and blue components; transparency shall be set to its default value.

The signature of the function is:

```
color integerToColor(integerArray[3])
```

The function shall succeed if each integer falls between 0 and 255 so that successful conversion to a hexadecimal representation is possible.

H.1.7.3 Integer parts to date

Converts three or five integer values to a single date value.

If five integers are supplied then these shall determine the year, month, day and time zone information.

EXAMPLE 1: The integer values "2000", "12", "25", "12" and "30" convert to a date value of 2000-12-25+12:30.

The signature of the function is:

```
date integerToDate(integerArray[5])
```

If only three integers are supplied then these shall determine the year, month, and day.

EXAMPLE 2: The integer values "2000", "12" and "25" convert to a date value of 2000-12-25

The signature of the function is:

```
color integerToColor(integerArray[3])
```

The function shall succeed if each integer obeys the constraints on each component part as described in clause 6.2.3.3.

H.1.7.4 Integer parts to dateTime

Converts six to nine integer values to a single dateTime value.

If nine integers are supplied then these shall determine all parts of the dateTime data type.

EXAMPLE: The integer values "2000", "12", "25", "12", "34", "56", "789", "-12" and "30" convert to a dateTime value of 2000-12-25T12:34:56.789-12:30.

The signature of the function is:

```
dateTime integerToDateTime(integerArray[9])
```

If eight integers are supplied then these shall determine the year, month, day, hour, minutes, seconds and timezone parts.

EXAMPLE: The integer values "2000", "12", "25", "12", "34", "56", "12" and "30" convert to a dateTime value of 2000-12-25T12:34:56+12:30.

The signature of the function is:

```
dateTime integerToDateTime(integerArray[8])
```

If seven integers are supplied then these shall determine the year, month, day, hour, minutes, seconds and milliseconds.

EXAMPLE: The integer values "2000", "12", "25", "12", "34", "56" and "789" convert to a dateTime value of 2000-12-25T12:34:56.789.

The signature of the function is:

```
dateTime integerToDateTime(integerArray[7])
```

If only six integers are supplied then these shall determine the year, month, day, hour, minutes and seconds.

EXAMPLE: The integer values "2000", "12", "25", "12", "34" and "56" convert to a dateTime value of 2000-12-25T12:34:56.

The signature of the function is:

```
dateTime integerToDateTime(integerArray[6])
```

The function shall succeed if each integer obeys the constraints on each component part as described in clause 6.2.3.3.

H.1.7.5 Integer parts to position

Converts two integer values to a single position value, where the first integer becomes the horizontal component of the position value and the second integer becomes the vertical component of the position value.

The signature of the function is:

```
position integerToPosition(integerArray[2])
```

The function shall always succeed.

H.1.7.6 Integer parts to size

Converts two integer values to a single size value, where the first integer becomes the width component of the size value and the second integer becomes the height component of the size value.

The signature of the function is:

```
size integerToSize(integerArray[2])
```

The function shall always succeed.

H.1.7.7 Integer parts to string

Converts an integer value to a string value.

The signature of the function is:

```
string integerToString(integer)
```

The function always succeeds.

H.1.7.8 Integer parts to time

Converts three to six integer values to a single time value.

If six integers are supplied then these shall determine all components of the time data type.

EXAMPLE 1: The integer values "12", "34", "56", "789", "-12" and "30" convert to a time value of 12:34:56.789-12:30.

The signature of the function is:

```
time integerToTime(integerArray[6])
```

If five integers are supplied then these shall determine the hour, minutes, seconds and timezone parts. The milliseconds part shall take its default value.

EXAMPLE 2: The integer values "12", "34", "56", "12" and "30" convert to a time value of 12:34:56+12:30.

The signature of the function is:

```
time integerToTime(integerArray[5])
```

If four integers are supplied then these shall determine the hour, minutes, seconds and milliseconds.

EXAMPLE 3: The integer values "12", "34", "56" and "789" convert to a time value of 12:34:56.789.

The signature of the function is:

```
time integerToTime(integerArray[4])
```

If only three integers are supplied then these shall determine the hour, minutes, seconds.

EXAMPLE 4: The integer values "12", "34" and "56" convert to a time value of 12:34:56.

The signature of the function is:

```
time integerToTime(integerArray[3])
```

The function shall succeed only if each integer obeys the constraints on each component part as described in clause 6.2.3.12.

H.1.8 From marked up text

H.1.8.1 Marked up text to string

Converts a marked up text value to a string value.

The signature of the function is:

```
string markedUpTextToString(markedUpText)
```

The function always succeeds.

H.1.9 From position

H.1.9.1 Position to integer parts

Converts a position value to its composite integer parts, where the first integer value represents the horizontal component of the position value and the second integer value represents the vertical component of the position value.

The signature of the function is:

```
integerArray[2] positionToInteger(position)
```

The function shall always succeed.

H.1.9.2 Position to string

Converts a position value to a string value.

The signature of the function is:

```
string positionToString(position)
```

The function always succeeds.

H.1.10 From size

H.1.10.1 String to currency

Converts a string value to a currency value.

The signature of the function is:

```
currency stringToCurrency(string)
```

The function shall succeed if the string obeys the constraints of a currency value as described in clause 6.2.3.2.

H.1.10.2 String to integer

Converts a string value to an integer value.

The signature of the function is:

```
integer stringToInteger(string)
```

The function shall succeed if the string obeys the constraints of an integer value as described in clause 6.2.2.2.

H.1.10.3 String to markedUpText

Converts a string value to a markedUpText value.

The signature of the function is:

```
markedUpText stringToMarkedUpText(string)
```

The function shall succeed if the string obeys the constraints of a markedUpText value as described in clause 6.2.3.7.

H.1.11 From time

H.1.11.1 Time and date to dateTime

Converts a time value and a date value to a single dateTime value.

EXAMPLE: The time value "12:34:56.789-12:30" and date value "2005-12-25-12:30" converts to dateTime values of "2005-12-25T12:34:56.789-12:30".

The signature of the function is:

```
dateTime timeDateToDateTime(time, date)
```

The function shall fail if time and date have different values for the timezone part.

H.1.11.2 Time to integer parts

Converts a time value to its six constituent integer parts.

EXAMPLE: The time value "12:34:56.789-12:30" converts to integer values of "12", "34", "56", "789", "-12", "30".

Six integers result from a time value that includes the (optional) decimal fraction of a second information.

The signature of the function is:

```
integerArray[6] timeAllToInteger(time)
```

Five integers result from a time value that does not include the (optional) decimal fraction of a second information.

The signature of the function is:

```
integerArray[5] timeToInteger(time)
```

The function always succeeds.

H.1.11.3 Time to string

Converts a time value to a string value.

The signature of the function is:

```
string timeToString(time)
```

The function always succeeds.

H.1.12 From timecode

H.1.12.1 Timecode to integer parts

Converts a timecode value to its four constituent integer parts.

EXAMPLE: The timecode value "23:59:59.29" converts to integer values of "23", "59", "59", "29".

The signature of the function is:

```
integerArray[4] timecodeToInteger(timecode)
```

The function always succeeds.

H.1.12.2 Timecode to string

Converts a timecode value to a string value.

```
string timecodeToString(timecode)
```

The function always succeeds.

H.2 Arithmetic functions

There are no arithmetic functions beyond those provided as operators in the Action Language grammar defined in the present document.

H.3 Array functions

A set of functions to manipulate array data types.

H.3.1 Array length

Returns an integer value representing the number of records in the array.

The signature of the function is:

```
integer length(<array>)
```

where <array> represents any identifier declared as an array type either in the action language or as an array property of a component.

For multi-dimensional arrays, the total number of records is returned i.e. the product of the array sizes in each dimension.

The function always succeeds.

H.4 String functions

A set of functions to manipulate string data types.

H.4.1 String length

Returns an integer value representing the number of characters in the string.

The signature of the function is:

```
integer strlen(string s)
```

The function always succeeds.

H.4.2 String compare

Returns an integer value representing the lexical difference between characters in two strings.

The signature of the function is:

```
integer strncmp(string s1, string s2, integer n)
```

Characters at locations 0 to [n-1] in s1 and s2 are compared. The comparison will stop when a difference in the character values is encountered or the location reaches the value of n.

The return value is -1 if the value of the character encountered in s1 is less than that in s2.

The return value is +1 if the value of the character encountered in s1 is greater than that in s2.

The return value is 0 if all characters encountered are the same.

The function always succeeds.

H.4.3 String contains

Returns an integer value representing the location of the first occurrence of one string in another.

The signature of the function is:

```
integer strstr(string s1, string s2)
```

If the string s2 matches a sequence of characters in s1, the return value is the location in s1 of the first character of the sequence s2.

If the string s2 is not found in s1 the return value is -1.

The function always succeeds.

H.4.4 String extract

Returns a portion of a string.

The signature of the function is:

```
string substr(string s, integer start, integer length)
```

The return value is a string composed of *length* characters from the string *s* beginning at the location *start*.

NOTE: The returned string is not necessarily independent of the input string. This is to avoid dynamic allocation of storage for a new string value.

The function will succeed if there are enough characters in *s* to fulfil the request.

Annex I (normative): Action language notation syntax

I.1 Grammar introduction

This annex contains format definition of the PCF action languages. The grammatical productions in this annex shall define the syntactical grammar productions of the PCF action language. The goal production shall be "ActionLanguage" as defined in clause I.3.1The semantics of the action language shall be as defined by the action language model in clause 9.4.

All spaces and new line characters shall be ignored in the default grammar state apart from where defined as a valid part of a literal's production.

The notation in represented in Extended Bachus-Naur Form (EBNF). All characters are considered as represented according to the Unicode standard [6].

I.2 Literals

I.2.1 Numeric literals

```
IntegerLiteral      ::= DecimalLiteral | HexLiteral | OctalLiteral
DecimalLiteral      ::= [1-9] ([0-9])*
HexLiteral          ::= '0' [xX] ([0-9a-fA-F])+
OctalLiteral        ::= '0' ([0-7])*

CurrencyLiteral     ::= '<currency>' ([0-9]) '.' [0-9] [0-9] '</currency>'

ColorLiteral        ::= '#' HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
                            (HexDigit HexDigit)?
HexDigit            ::= [0-9a-fA-F]
```

NOTE: The literal representation for the productions "IntergerLiteral", "DecimalLiteral", "HexLiteral" and "OctalLiteral" are compatible with ECMA-script (see bibliography).

I.2.2 Date and time literals

```
DateTimeLiteral     ::= '<dateTime>' DatePart 'T' TimePart (Timezone)? '</dateTime>'
TimeLiteral         ::= '<time>' TimePart (Timezone)? '</time>'
DateLiteral         ::= '<date>' DatePart (Timezone)? '</date>'
DatePart            ::= [0-9] [0-9] [0-9] [0-9] '-' [01] [0-9]'-' [0-3] [0-9]
TimePart            ::= [012] [0-9] ':' [0-5] [0-9] ':' [0-5] [0-9]
                            ('.' [0-9] ([0-9])? ([0-9])? )?
Timezone            ::= ([+-] [01] [0-3] [0-5] [0-9]) | 'UTC'
TimecodeLiteral     ::= '<timecode>' [012] [0-9] ':' [0-5] [0-9] ':' [0-5] [0-9]
                            '.' [0-5] [0-9] '</timecode>'
```

I.2.3 Character-based literals

```
StringLiteral       ::= '"' ( [^"\#x0a#x0d] |
                            '\' ([ntbrf\'"] | [0-7] ([0-7])? | [0-3] [0-7] [0-7]))*
                            '"'
MarkedUpTextLiteral ::= '<body>' ([^<])* '</body>'
URILiteral          ::= '<uri>' ([^<])* '</uri>'
EnumerationLiteral  ::= '<enum>' Identifier '</enum>'
```

I.2.4 Geometric literals

```
BoundsLiteral       ::= '<bounds>' SignedInteger ListSpace SignedInteger ListSpace
                        SignedInteger ListSpace SignedInteger '</Bounds>'   SignedInteger
::= ([+-] ([#x20#x09]*)? IntegerLiteral
ListSpace           ::= [#x20#x09] /* Space and tab */
PositionLiteral     ::= '<position>' SingedInteger ListSpace SignedInteger
                        '</position>'
SizeLiteral         ::= '<size>' SignedInteger ListSpace SignedInteger '</size>'
ProportionLiteral   ::= '<proportion>' SignedInteger ListSpace SignedInteger
                        '</proportion>'
```

I.2.5 Literal production

```
NilLiteral          ::= 'nil'
Literal             ::= IntegerLiteral | CurrencyLiteral | ColorLiteral |
                        BoundsLiteral | PositionLiteral | SizeLiteral |
                        StringLiteral | DateLiteral | TimeLiteral |
                        DateTimeLiteral | BooleanLiteral | MarkedUpTextLiteral |
                        URILiteral | TimecodeLiteral | EnumerationLiteral |
                        ProportionLiteral | NilLiteral
```

I.2.6 Identifiers

```
Identifier            ::= IdentifierStart | Identifier IdentifierPart
IdentifierPart        ::= IdentifierStart | UnicodeCombiningMark | UnicodeDigit |
                            "\" UnicodeEscapeSequence
UnicodeEscapeSequence ::= 'u' HexDigit HexDigit HexDigit HexDigit
ReservedTokens        ::= TypeName | 'if' | 'for' | 'else' | 'break' | 'continue'
```

The "IdentifierStart" production shall match the "_" character or any character from the following Unicode categories: "Uppercase letter (Lu)", "Lowercase letter (Ll)", "Titlecase letter (Lt)", "Other letter (Lo)", or "Letter number (Nl)".

The "UnicodeCombiningPart" production shall match any character from the following Unicode categories: "Non-spacing mark (Mn)" or "Combining spacing mark (Mc)".

The "UnicodeDigit" production shall match any character from the Unicode Category "Decimal number (Nd)".

The "Identifier" production shall not match any of the following:

- valid matches for the "ReservedTokens";
- system action call names as defined in annex J;
- expression names as defined in annex G.

NOTE: The character categories listed here form an acceptable set in the intersection of identifiers defined for both NCNames in XML [5] and the "IndentifierName" production in ECMAScript (see bibliography).

I.3 Structure and statements

I.3.1 Goal production and statements

```
ActionLanguage      ::= (Statement)*
Statement           ::= Assignment ';' | Declaration ';' | Loop ';' | Break ';'
                        Continue ';' | Conditional ';' | Action ';' | ';' |
                        Block
Block               ::= '{' (Statement)* '}'
```

I.3.2 Assignment

```
Assignment          ::= PathReference ('=' | '+=' | '-=' | '*=' | '/=' ) Expression |
                            PathReference ('++' | '--')
PathReference       ::= NameWithIndex ('.' NameWithIndex )*
NameWithIndex       ::= Identifier (Index)*
Index               ::= '[' Expression ']'
```

I.3.3 Action call

```
Action              ::= PathReference Arguments
Arguments           ::= '(' (Expression (',' Expression)*)? ')'
```

I.3.4 Declaration

```
Declaration         ::= TypeName (ArrayDeclaration)? Identifier ('=' Expression)? |
                            TypeName Identifier ArrayDeclaration ('=' Expression)?
TypeName            ::= 'boolean' | 'bounds' | 'color' | 'component' |
                            'currency' | 'date' | 'datetime' | 'integer' |
                            'markeduptext' | 'position' | 'size' | 'string' |
                            'time' | 'timecode' | 'URI'
ArrayDeclaration    ::= ('[' IntegerLiteral ']')+
```

NOTE: Array declarations can only be declared to be of a fixed size and shall not be dynamically determined by the evaluation of an expression.

I.3.5 Conditional

```
Conditional         ::= 'if' '(' Expression ')' Statement ('else' Statement)?
```

I.3.6 Loop

```
Loop                ::= 'for' '(' (Declaration | Assignment)? ';'
                            (Expression)? ';' (Assignment)? ')' Statement
Break               ::= 'break'
Continue            ::= 'continue'
```

I.4 Expressions

I.5 Ternary operator

```
Expression          ::= ConditionalOrExpr ('?' Expression ':' Expression)?
```

I.5.1 Logical and relative expressions

```
Condition           ::= ConditionalOrExpr
ConditionalOrExpr   ::= ConditionalAndExpr ('||' ConditionalAndExpr )*
ConditionalAndExpr  ::= EqualityExpr ('&&' EqualityExpr)*
EqualityExpr        ::= RelationalExpr ('==' RelationalExpr | '!=' RelationalExpr)*
RelationalExpr      ::= AdditiveExpr ('<' AdditiveExpr | '<=' AdditiveExpr |
                                     '>' AdditiveExpr | '>=' AdditiveExpr)*
```

I.5.2 Arithmetic expressions

```
AdditiveExpr        ::= MultiplicativeExpr ('+' MultiplicativeExpr |
                                            '-' MultiplicativeExpr)*
MultiplicativeExpr  ::= UnaryExpr ('*' UnaryExpr | '/' UnaryExpr | '%' UnaryExpr)*
```

I.5.3 Unary expressions

```
UnaryExpr           ::= '+' UnaryExpr | '-' UnaryExpr |
                        '++' PrimaryExpr | '--' PrimaryExpr |
                        UnaryNotPlusMinExpr
UnaryNotPlusMinExpr ::= '!' UnaryExpr | PostfixExpr
```

I.5.4 Primary expressions

```
PostfixExpr         ::= PrimaryExpr ('++' | '--')?
PrimaryExpr         ::= PrimaryPrefix (Arguments)?
PrimaryPrefix       ::= Literal | '(' Expression ')' | PathReference | ArrayExpr
ArrayExpr           ::= '[' (Expression (',' Expression)*)? ']'
```

Annex J (normative): System action library

J.1 Transitions

J.1.1 Forward navigate to another scene

Navigates from the current (source) scene to another (target) scene. Provides control of history stack entry.

The signature of the function is:

```
nil sceneNavigate(URI target,
                  enum<forget,remember,mark> type,
                  URI substitute);
```

The parameter "target" is a URI of the scene that becomes active as a result of this action.

The parameter "type" is an enumeration that determines whether any information is recorded on the history stack as a result of the action. The value "remember" indicates that a scene is to be placed on the history stack as a result of this navigation. If this is a critical navigation, then it may be appropriate to use the value "mark". The value "forget" indicates that no scene is placed on the stack. If no value is given, the default is "forget".

The parameter "substitute" is a URI of an alternative scene that shall be placed on the history stack instead of the source scene. This parameter shall be used in conjunction with the type parameter.

EXAMPLE: `sceneNavigate(<uri>#~/scene2</uri>, <enum>mark</enum>, nil);`
will navigate to the scene named "scene2" in the current service, recording the source scene on the history stack and marking it.

The function shall succeed if the target scene exists in the run-time environment.

Errors in forward navigation can be captured using the OnForwardNavigation error event.

NOTE: An OnForwardNavigation error shall result in service termination unless an author explicitly describes how this should be handled.

J.1.2 Historical navigate to a previous scene

Navigates from the current (source) scene to a scene recorded on the history stack.

EXAMPLE: `historyNavigate(true);`
will navigate to the nearest scene recorded in the history stack as marked, and erase the history stack down to and including this recorded scene.

The signature of the function is:

```
nil historyNavigate(boolean mark);
```

The parameter "mark" is a Boolean that indicates whether to navigate to the last recorded scene or the last marked scene in the history stack.

When the mark parameter has a value of "false" then this describes navigation to the last scene placed on the history stack. The last scene on the stack shall be deleted as a result of this navigation.

When the mark parameter has a value of "true" then this describes navigation to the last *marked* scene on the history stack. All scenes at the end of the stack shall be deleted up to and including this last marked scene as a result of this navigation.

The function shall succeed if the required recorded scene is found on the stack and exists in the run-time environment.

Errors in history navigate can be captured using the OnInvalidHistory error event.

NOTE: An OnInvalidHistory error shall result in service termination unless an author explicitly describes how this should be handled.

J.1.3 Erasing the history stack

Provides for erasure of items from the history stack.

EXAMPLE: `historyDelete(all);`
will remove all recorded items from the history stack.

The signature of the function is:

```
nil historyDelete(enum<all,marked,last> deltype);
```

The "deltype" enumeration shall be one of the 3 literals "all, "marked" or "last" and controls how much of the history stack is erased; all items, all down to and including the nearest marked item or the last item recorded respectively. If no marked item is found then "marked" removes all items.

The function shall always succeed.

J.1.4 Reading the history stack

Provides for access to items recorded on the history stack.

EXAMPLE: `target = historyRead(true);`
will retrieve the nearest marked item from the history stack.

The signature of the function is:

```
component historyRead(Boolean marked);
```

The "marked" Boolean indicates whether to retrieve the last recorded scene or the last marked scene from the history stack.

The function shall always succeed. If the required recorded scene is not found on the stack the function shall return a "nil" value.

J.1.5 Navigate to another service

Navigates from the current (source) service to another (target) service. The history stack shall be deleted on navigation to a new service.

EXAMPLE: `serviceNavigate(<uri>#quiz.xml#Tuesday</uri>, nil);`
will navigate to the service named "service2", and delete all scenes on the history stack.

The signature of the function is:

```
nil serviceNavigate(URI target,
                    URI entrypoint);
```

The "target" component shall resolve to a PCF service item.

The function shall succeed if the target service exists in the run-time environment.

J.2 Exit

J.2.1 Exiting the service session

Provides a means to exit the service session.

EXAMPLE: `exit();`
will exit the service session.

The signature of the function is:

```
nil exit();
```

The function shall always succeed. Behaviour if the service entry point cannot be found is not defined by the PCF.

J.3 Presentation

J.3.1 Controlling rendering

Provides a means to block scene rendering activity during the service session.

EXAMPLE: `redraw(false);`
will block scene rendering from this point forwards.

The signature of the function is:

```
nil redraw(boolean enable);
```

The "enable" boolean block and unblocks scene rendering activity. Subsequent calls to this function with the same value will have no effect.

The function shall always succeed.

J.4 Environment

J.4.1 Getting the time

Provides a means to access the system clock.

EXAMPLE: `time = getSystemTime();`
will retrieve the system clock value.

The signature of the function is:

```
time getSystemTime();
```

The function shall return the system clock value from the receiver environment.

The function shall always succeed.

J.4.2 Getting the platform identifier

Provides a means to access a unique identifier for this platform.

EXAMPLE: `platform = getPlatformID();`
will retrieve the platform identifier value.

The signature of the function is:

```
string getPlatformID();
```

The function shall return the platform identifier value from the receiver environment.

The function shall always succeed.

J.4.3 Getting the receiver identifier

Provides a means to access a receiver identifier that is unique within the platform.

EXAMPLE: `myid = getReceiverID();`
will retrieve the receiver identifier value.

The signature of the function is:

```
string getReceiverID();
```

The function shall return the receiver identifier value from the receiver environment.

The function shall always succeed.

J.4.4 Getting the default language

Provides a means to access the default language setting.

EXAMPLE: `lang = getDefaultLanguage();`
will retrieve the default language value.

The signature of the function is:

```
iso639 getDefaultLanguage();
```

The function shall return the default language value from the receiver environment.

The function shall always succeed.

J.5 XML Shortcuts

In a PCF profile where action language is excluded, it shall still be possible to express the following system API calls using an alternative "shortcut" XML syntax.

J.5.1 Scene navigate shortcut

A scene navigate item is a shortcut for the sceneNavigate action (see clause J.1.1).

The properties of a scene navigate item include:

- a name that is unique within the scope of the shortcut;
- a type - an enumeration of either "remember", "forget" or "mark".

The description of a scene navigate item shall contain:

- exactly one URI item called "trigger" to indicate the new PCF scene;
- at most one URI item called "substitute".

EXAMPLE: A scene navigate item is represented in the PCF XML using an element with name "". This example describes a scene navigate where a substitute scene is placed on the history stack as a result of the navigation.

```
<SceneNavigate type="remember" substitute="#../otherscene">
    <URI name="target" value="#../scene1"/>
<SceneNavigate>
```

J.5.2 Service navigate shortcut

A service navigate item is a shortcut for the serviceNavigate action (see clause J.1.5).

The description of a service navigate item shall contain:

- exactly one PCF URI item called "target" to indicate the new PCF service.
- At most one PCF string item called "entrypoint" to indicate a component entry point other than the service's default first scene.

EXAMPLE: A service navigate item is represented in the PCF XML using an element with name "ServiceNavigate".

```
<ServiceNavigate>
    <URI name="target" value="#quiz.xml"/>
    <String name="entrypoint" value=" #/startscene"/>
</ServiceNavigate>
```

J.5.3 History navigate shortcut

A history navigate item is a shortcut for the historyNavigate action (see clause J.1.2).

The properties of a history navigate item include:

- a Boolean property item called "mark" - which has a default value "false".

EXAMPLE: A history navigate is represented in the PCF XML using an element with name "HistoryNavigate".

```
<HistoryNavigate mark="true"/>
```

J.5.4 Exit action shortcut

An exit action item describes an action that terminates the currently active service.

An exit action item has no properties.

EXAMPLE: An exit action is represented in the PCF XML using an element with name "Exit".

```
<ExitAction/>
```

Annex K (normative): User keys

Table 37 shows all virtual user key codes supported by PCF. Each user key code is then described in more detail in clause K.2.

K.1 Virtual key codes

The virtual key codes as enumerated in table 37 are the acceptable values for the PCF user key data type described in clause 6.2.3.15.

Table 37: Virtual key codes

VK_0	VK_ACCEPT	VK_COLORED_KEY_5	VK_PNP_TOGGLE
VK_1	VK_CANCEL	VK_DELETE	VK_PREV
VK_2	VK_CHANNEL_DOWN	VK_DOWN	VK_RESET
VK_3	VK_CHANNEL_UP	VK_ENTER	VK_RIGHT
VK_4	VK_CLEAR	VK_GUIDE	VK_SPLIT_SCREEN_TOGGLE
VK_5	VK_COLORED_KEY_0	VK_HELP	VK_TAB
VK_6	VK_COLORED_KEY_1	VK_HOME	VK_TELETEXT
VK_7	VK_COLORED_KEY_2	VK_INFO	VK_UNKNOWN
VK_8	VK_COLORED_KEY_3	VK_LEFT	VK_UP
VK_9	VK_COLORED_KEY_4	VK_OPTIONS	

NOTE: Virtual key codes are mapped to platform-specific keys or functionality. Virtual keys may correspond to a sequence of key presses or some platform-specific representation such as an overlaid menu.

K.2 Key code descriptions

K.2.1 Numeric keys

VK_0 … VK_9

K.2.2 Navigation keys

VK_UP, VK_DOWN, VK_LEFT, VK_RIGHT

K.2.3 Coloured keys

VK_COLORED_KEY_0/5

K.2.4 Service-level control keys

VK_CHANNEL_UP/DOWN

VK_TELETEXT

VK_PNP_TOGGLE

VK_SPLIT_SCREEN_TOGGLE

K.2.5 Unknown key

VK_UNKNOWN

Annex L (informative): Scalability techniques

The PCF provides a powerful referencing mechanism that allows a PCF service description to be flexibly partitioned into a number of fragments. Furthermore, there is nothing that prevents a fragment from being referenced as part of more than one service description. This means that, for example, two variants of a particular service can easily be described that contain of common core based on shared fragments with the variant-specific aspects based on variant-specific fragments. The referencing of such generic fragments not only reduces the total data involved in the description of the two variants but also means that any change to such a fragment will be inherently reflected in both variants.

There is no mechanism within the present document to allow a PCF service description to contain more than one description for a particular aspect of the service. Such a scalability technique would allow a more editorially desirable (but perhaps less portable) alternative to be provided in addition to a description within the bounds of the minimum profile associated with the service description. For example, ideally display this spinning flaming logo but if this is not possible display this static bitmap.

NOTE: In fact the profile association mechanism (see clause 11.3) already anticipates such an "alternative description" mechanism, since it allows more than one profile to be associated with a particular service description, i.e. one for each description.

Annex M (normative): Service announcement and boot

The availability of an interactive service is generally promoted to the viewer using some kind of on screen "call to action" or "announcement", e.g. "Press Red". The description of such an announcement may be captured as PCF **Scene** item. However, such a **Scene** item needs careful handling due to the way in which the implementation of announcements varies from platform to platform. In some platforms interactive applications auto-boot and so the announcement must form part of the application itself. In other platforms the announcement is provided by some means outside of the interactive application, with the application only booted following viewer input.

To accommodate these different models a **Scene** item that describes the viewer experience of an announcement (the "announcement **Scene**") shall be identified independently of the initially active **Scene** within the core service description (the "first **Scene**") using the **announcement** and **firstScene** properties of the service item respectively.

NOTE 1: This is only required when maximizing the portability of a service description. If a service description is authored with a particular target platform in mind then the description of the announcement may be incorporated into the core service description or not described at all as appropriate.

NOTE 2: Due to the platform specific nature of announcements there is no guarantee as to how or even if a described announcement **Scene** is rendered.

The description within the announcement **Scene** of **Scene** navigation to the first **Scene** (typically in response to viewer input) shall make use of the defined PCF value item **firstScene** as opposed to making an explicit reference to the **Scene** that has been identified as the first **Scene**. This means that the description of navigation from the announcement **Scene** will remain valid even if the **firstScene** property of the service item is changed.

A service description shall contain no explicit scene navigation to a **Scene** identified as the announcement **Scene**. Instead it shall be assumed that only on exiting the interactive service will the announcement **Scene** be returned (navigated) to. How this return to the announcement **Scene** is implemented in the transcoded output will vary greatly from platform to platform.

These rules are illustrated in the following example:

```
<Service name="my_service">
    <URI name="firstScene" value="#home"/>
    <URI name="announcement" value="#press_red"/>

    <Scene name="press_red">
        ...
        <OnEvent name="red_button_nav">
            <Trigger eventtype="KeyEvent">
                <UserKey name="key" value="VK_COLORED_KEY_0"/>
            </Trigger>
            <ServiceNavigate>
                <URI name="target" href="#~/firstScene" context="derived"/>
            </ServiceNavigate>
        </OnEvent>
    </Scene>

    <Scene name="home">
        ...
        <OnEvent name="exit">
            <Trigger eventtype="KeyEvent">
                <UserKey name="key" value="VK_PREV"/>
            </Trigger>
            <ExitAction/>
        </OnEvent>
    </Scene>
</Service>
```

Annex N (informative): Independent authoring of elements of a service description

The PCF provides a powerful referencing mechanism that allows a PCF service description to be flexibly partitioned into a number of fragments. There is nothing inherent within the PCF that prevents fragments from being authored independently from one another. This might be useful when different parties are responsible for different parts of a service.

EXAMPLE: An interactive service might allow the viewer to browse and buy books. One party looks after the presentation of the available stock and the selection of items to buy, whilst a second party looks after the payment transaction.

Using the PCF referencing model, two (or more) parties can maintain the relevant aspects of a service description independently while providing a unified viewer experience. However, this does require an "interface" to be agreed in advance to ensure that the PCF referencing mechanism will work across the boundary between these independently authored fragments. However, this is not unreasonable as such a "contract" is almost certainly required simply to deal with issues of on-screen real-estate and the passing of information from one fragment to the other, e.g. total cost of books in the shopping basket.

NOTE: The nesting of **Service** components is not allowed by the present document. It is possible that this (or some more powerful mechanism along the lines of HTML Frames) may be defined by a future revision of the PCF specification to provide a more explicit means of combining independently authored service fragments.

Annex O (informative): StreamEvent bindings

O.1 XML document binding

The following XML schema may be used to represent a sequence of **StreamEvent**s for ingest by a platform.

Each StreamEvent element has attributes to represent the event name and timecode for firing. **StreamEvent**s may also contain an optional payload string.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://www.dvb.org/pcf/stream-events" xmlns:pcf-
types="http://www.dvb.org/pcf/pcf-types" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns="http://www.dvb.org/pcf/stream-events" elementFormDefault="qualified"
attributeFormDefault="unqualified">
    <xs:import namespace="http://www.dvb.org/pcf/pcf-types" schemaLocation="pcf-types.xsd"/>
    <xs:element name="StreamEvents">
        <xs:annotation>
            <xs:documentation>list of events for a single item of material e.g. broadcast program,
PVR recording, etc.</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="StreamEvent">
                    <xs:complexType>
                        <xs:simpleContent>
                            <xs:extension base="pcf-types:string">
                                <xs:attribute name="eventname" type="pcf-types:string"
use="required"/>
                                <xs:attribute name="timecode" type="pcf-types:timecode"
use="required"/>
                            </xs:extension>
                        </xs:simpleContent>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="material_id" type="pcf-types:string" use="optional"/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

EXAMPLE: The following simple example shows the above schema in use:

```
<?xml version="1.0" encoding="UTF-8"?>
<StreamEvents xmlns="http://www.dvb.org/pcf/stream-events"
xmlns:pcf-types="http://www.dvb.org/pcf/pcf-types" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://www.dvb.org/pcf/stream-events stream-events.xsd"
material_id="Q033-2298-BANZAI">
    <StreamEvent eventname="question" timecode="00:10:55:08">1</StreamEvent>
    <StreamEvent eventname="question" timecode="00:13:35:02">2</StreamEvent>
    <StreamEvent eventname="answer" timecode="00:22:15:12">1</StreamEvent>
    <StreamEvent eventname="answer" timecode="00:11:11:17">2</StreamEvent>
    <StreamEvent eventname="score" timecode="00:27:21:09">call if you have scored more than:
6</StreamEvent>
</StreamEvents>
```

O.2 MXF document binding

MFX (see bibliography) documents may carry one or more timelines of events in addition to streamed media content. This annex defines the binding between the MFX object types and values and their interpretation as PCF **StreamEvent**s.

Table 38: MXF document binding

MFX object type	MFX property name	PCF stream event property
Shot	Shot Description	eventname
Key Point	Keypoint Position	timecode
Key Point	Keypoint Value	payload

NOTE: It is recommended to agree a Keypoint Kind value to represent PCF **StreamEvent**s within the MXF data dictionary. This serves to separate these key frames from any others.

O.3 AAF document binding

AAF (see bibliography) documents may describe one or more timelines of events in addition to streamed media content. This annex defines the binding between the AAF object types and properties and their interpretation as PCF **StreamEvent**s.

Table 39: AAF document binding

AAF object type	AAF property name	PCF stream event property
EventMobSlot	SlotName [if defined] else SlotID	eventname
Event	Position	timecode
Event	Comment	payload

Annex P (informative): Receiver handling of aspect ratio

Video is produced in a variety of aspect ratios, most commonly 4:3 or 16:9. This video is then in turn presented on a mixture of displays of either 4:3 or 16:9 aspect ratio. The upshot is that it is common for video produced in one aspect ratio to be presented on a display of different aspect ratio. This makes it necessary for the combination of receiver and display to provide suitable processing to ensure that the viewer is presented with video in the correct aspect ratio.

When considering video only, such "aspect ratio handling" is well supported by current digital television receiver implementations. For example, when presenting 16:9 video on a 4:3 display aspect ratio handling techniques, such as "centre cut-out" and "letter box" as shown below, are readily available.

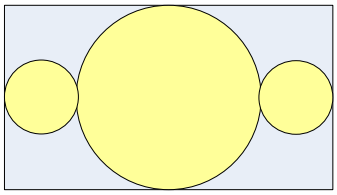

Figure 99: 16:9 source video

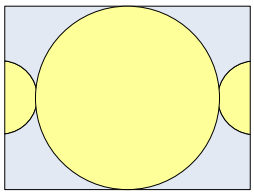

Figure 100: 16:9 source video on 4:3 display, with "centre cut-out" compensation

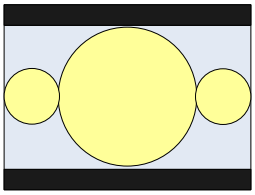

Figure 101: 16:9 source video on 4:3 display, with "letter-box" compensation

However, problems can occur when trying to ensure the exact placement or "alignment" of OSD graphics with specific points in the video.

In an idealized receiver implementation aspect ratio handling will not have any impact on graphics registration. In such an implementation any processing performed on the video will also be performed on the graphics so spatial relationships (and hence registration) will be preserved. However, in reality few (if any) actual implementations are based on this. In reality physical hardware architectures can mean that aspect ratio handling can be applied to the video prior to the overlay of any graphics with the result that any registration is lost.

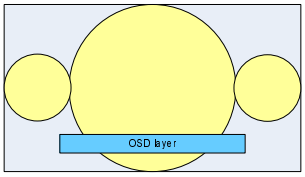

Figure 102: 16:9 source video presented at 16:9 with graphics box

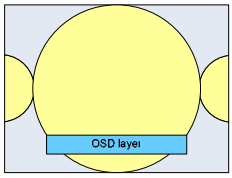

Figure 103: 16:9 source video presented at 4:3 with centre cut-out compensation, with graphics box

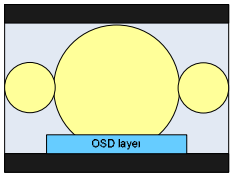

Figure 104: 16:9 source video presented at 4:3 with letter-box compensation, with graphics box

Of course one solution to this would be to only allow aspect ratio handling to take place after the video and graphics had been combined. Whilst this would ensure that registration would be preserved, it could result in the presentation of video (and indeed graphics) at an incorrect aspect ratio.

In summary there are scenarios where the requirements of achieving the correct aspect ratio and maintaining registration can not be met. Furthermore, under such circumstances there is no single rule that will always deliver the most optimal behaviour. Instead, only the service author is likely to know which of the requirements has the greatest priority.

Annex Q (normative): Standard PCF URNs

Table 40 lists URNs have been defined as standard within the PCF.

NOTE: The lack of any explicit provider identifier implies that this part of the URN should be assumed to be "dvb.org".

Table 40: URNs in PCF

URN	Description
urn:x-dvb-pcf::default	A default value with declared meaning within the context of its use.
urn:x-dvb-pcf::dvb-ctag<NN>	Identifies an elementary media stream within the context of DVB Service (MPEG Program) assigned the DVB component tag value <NN>, where NN represents an 8-bit value in a hexadecimal format. For example, "urn:x-dvb-pcf::dvb-ctag64", for the component tag value 0x64.
urn:x-dvb-pcf::mpeg-pid<NN>	Identifies an elementary media stream within the context of an MPEG Transport Stream assigned the MPEG packet identifier (PID) value <NN>, where NN represents an 8-bit value in a hexadecimal format. For example, "urn:x-dvb-pcf::mpeg-pid64", for the PID value 0x64.

Annex R (informative): Example PCF service descriptions

R.1 "Hello World!"

R.1.1 Simplest possible description

In the simplest possible example the complete description of a service is provide in a single PCF source document.

```
-- source document: main.xml --

<PCF>
    <Service name="hello_world">
        <URI name="firstScene" value="#hi"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Scene name="hi">
            <TextBox name="caption">
                <Position name="origin" value="100 100"/>
                <Size name="size" value="200 30"/>
                <String name="content" value="Hello, world!"/>
            </TextBox>
        </Scene>
    </Service>
</PCF>
```

R.1.2 External content

This example provides an alternative way of describing the same viewer experience captured by the example in annex R.1.1 by referencing out from the PCF source document to an external resource, in this case a text file.

```
-- source document: main.xml --

<PCF>
    <Service name="hello_world">
        <URI name="firstScene" value="#hi"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Scene name="hi">
            <TextBox name="caption">
                <Position name="origin" value="100 100"/>
                <Size name="size" value="200 30"/>
                <String name="content">
                    <ExternalBody content-type="text/plain" uri="myfile.txt"/>
                </String>
            </TextBox>
        </Scene>
    </Service>
</PCF>
```

Where the content of file `myfile.txt` is:

```
Hello, world!
```

This allows the text to be displayed to be authored independently of the rest of the service description. It also means that updates to the text to be displayed can be made without having to modify any PCF source document.

NOTE: The partitioning of the service description in this manner does imply that the transcoded output will consist of two corresponding data entities. It would, for example, be perfectly reasonable for the transcoded output to consist of just a single data entity.

R.1.3 Service and scene defined in separate source documents

This example illustrates how a service description can be partitioned into more than one PCF source document. This allows different parties to author different parts. It does however require some agreement at the "interface": in this case the name of the **Scene** and the source document in which it is declared.

```
-- source document: main.xml --

<PCF>
    <Service name="hello_world">
        <URI name="firstScene" value="#hi"/>
        <String name="pcfSpecVersion" value="1.0"/>

        <Scene name="hi" href="#scene_def.xml#/hello"/>
    </Service>
</PCF>

-- source document: scene_def.xml --

<PCF>
    <Scene name="hello">
        <TextBox name="caption">
            <Position name="origin" value="100 100"/>
            <Size name="size" value="200 30"/>
            <String name="content" value="Hello, world!"/>
        </TextBox>
    </Scene>
</PCF>
```

R.2 Templated authoring

R.2.1 Scene item defined using a template

A common authoring technique is to define scenes as an instance of some kind of presentation template. The PCF does not provide an explicit "template item". However, the PCF does provide features to support a template-based approach to service authoring.

In this example the service description is partitioned into two source documents. The first source document declares the service item. This contains two Scene components that are based on a "template" declared in the second source document in the form of a static explicit layout container component.

NOTE: Such a "template" could be defined using other container components.

```
-- source document: main.xml --

<PCF>
    <Service name="quick_march">
        <URI name="firstScene" value="#one"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Scene name="one">
            <String name="dynamic_caption" value="1..."/>
            <URI name="next_scene" value="#../two"/>
            <StaticELC name="template" href="#template_def.xml#template1"/>
        </Scene>
        <Scene name="two">
            <String name="dynamic_caption" value="2..."/>
            <URI name="next_scene" value="#../one"/>
            <StaticELC name="template" href="#template_def.xml#template1"/>
        </Scene>
    </Service>
</PCF>

-- source document: template_def.xml --

<PCF>
    <StaticELC name="template1">
        <TextBox name="static_caption">
            <Position name="origin" value="100 100"/>
            <Size name="size" value="200 30"/>
            <String name="content" value="Quick, march!"/>
```

```
        </TextBox>
        <TextBox name="dynamic_caption">
            <Position name="origin" value="100 200"/>
            <Size name="size" value="200 30"/>
            <String name="content" href="#../../dynamic_caption" context="derived"/>
        </TextBox>
        <OnEvent name="press_red">
        <Trigger eventtype="KeyEvent">
            <UserKey name="key" value="VK_COLORED_KEY_0"/>
        </Trigger>
            <SceneNavigate>
                <URI name="target" href="#../../../next_scene" context="derived"/>
            </SceneNavigate>
        </OnEvent>
    </StaticELC>
</PCF>
```

R.2.2 Scene item defined using multiple templates

An extension to the example in clause R.2.1 is one in which a scene item is defined by reference to two "templates", in this case two static explicit layout container components. However, there is a potential problem in that there could be a namespace clash between the two referenced components such that resolution can become ambiguous. This creates an error since there are now two items with the same name, i.e. "text".

```
-- source document: main.xml --

<PCF>
    <Service name="broken">
        <URI name="firstScene" value="#ooops"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Scene name="ooops">
            <String name="text" value="This is some text"/>
            <StaticELC name="template1" href="template1.xml#template1"/>
            <!-- this next String element causes ambiguity - duplicate name -->
            <String name="text" value="This is some more text"/>
            <StaticELC name="template2" href="template2.xml#template2"/>
        </Scene>
    </Service>
</PCF>

-- source document: template1.xml --

<PCF>
    <StaticELC name="template1">
        <TextBox name="headline">
            <Position name="origin" value="100 100"/>
            <Size name="size" value="200 30"/>
            <String name="content" href="#/../../text" context="derived"/>
        </TextBox>
    </StaticELC>
</PCF>

-- source document: template2.xml --

<PCF>
    <StaticELC name="template2">
        <TextBox name="body">
            <Position name="origin" value="100 200"/>
            <Size name="size" value="200 30"/>
            <String name="content" href="#/../../text" context="derived"/>
        </TextBox>
    </StaticELC>
</PCF>
```

Where the risk of such a namespace clash cannot be removed through management of the template declaration, e.g. a third party "library" declaration, it will be necessary to clearly identify the context of any association through the use of a collection item.

```
-- source document: main.xml --

<PCF>
    <Service name="not_broken">
        <URI name="firstScene" value="#ahhh"/>
        <String name="pcfSpecVersion" value="1.0"/>
```

```
        <Scene name="ooops">
            <Collection name="c1">
                <String name="text" value="This is some text"/>
                <StaticELC name="template1" href="#template1.xml#template1"/>
            </Collection>
            <Collection name="c2">
                <String name="text" value="This is some more text"/>
                <StaticELC name="template2" href="#template2.xml#template2"/>
            </Collection>
        </Scene>
    </Service>
</PCF>
```

Where the source documents `template1.xml` and `template2.xml` are as before.

In a second, slightly more specialized example a **Scene** item is defined by reference to the same "template" more than once. This inherently causes a clash in reference names and so must be dealt with in the manner described above.

R.3 Presenting streamed content

R.3.1 Default elementary media streams using a URN

This example shows how to present the default video and audio, and (if available and selected for presentation by the viewer) subtitles elementary media streams for the context in which the transcoded service description is running. All references are provided as URNs that must be resolved in a platform-specific manner by the relevant PCF transcoder.

```
-- source document: main.xml --

<PCF>
    <Service name="watch_tv">
        <URI name="firstScene" value="#my_scene"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Stream name="tv">
            <StreamData name="content">
                <ExternalBody content-type="application/octet-stream" uri="urn:x-dvb-pcf::default"/>
            </StreamData>
            <Video name="default_fullscreen_video">
                <Subtitles name="allow_subs_presentation"/>
            </Video>
            <Audio name="default_audio"/>
        </Stream>

        <Scene name="my_scene">
            <!-- Scene def here -->
        </Scene>
    </Service>
</PCF>
```

R.3.2 From specific broadcast service using a URN

This example shows how to present the default video and a specific audio elementary media streams from a specific TV service identified via a URN.

NOTE: In this example (and others that follow) there is no Subtitles component so no subtitles elementary media stream will be presented, even if available and selected for presentation by the viewer.

Since the elementary media stream composition representing the TV service is identified using a URN any references to elementary media streams within it must also be made using a URN.

```
-- source document: main.xml --

<PCF>
    <Service name="watch_tv">
        <URI name="firstScene" value="#my_scene"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Stream name="tv">
            <StreamData name="content">
```

```
                <ExternalBody content-type="application/octet-stream" uri="urn:x-dvb-
pcf:bbc.co.uk:golf"/>
            </StreamData>
            <Video name="mosaic_video"/>
            <Audio name="audio_for_highlighted_mosaic_item">
                <URI name="content" value="urn:x-dvb-pcf:bbc.co.uk:audio1"/>
            </Audio>
        </Stream>

        <Scene name="my_scene">
            <!-- Scene def here -->
        </Scene>
    </Service>
</PCF>
```

R.3.3 From specific broadcast service using a URL

This example shows how to present the default video and a specific audio elementary media streams from a specific TV service identified via a URL.

Since the elementary media stream composition representing the TV service is identified using a URL any references to elementary media streams within it can be made using a URN or a URL. In this case the DVB URL format is used to identify an elementary media stream composition with the DVB service ID `0x5678` and an audio elementary media stream within it with the DVB component tag `0x64`. The PCF default URN is used to identify the default video within the composition.

```
-- source document: main.xml --

<PCF>
    <Service name="watch_tv">
        <URI name="firstScene" value="#my_scene"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Stream name="tv">
            <StreamData name="content">
                <ExternalBody content-type="video/x-MP2T-P" uri="dvb://1234..5678"/>
            </StreamData>
            <Video name="default_fullscreen_video"/>
            <Audio name="audio">
                <URI name="content" value="dvb://1234..5678.64"/>
            </Audio>
        </Stream>

        <Scene name="my_scene">
            <!-- Scene def here -->
        </Scene>
    </Service>
</PCF>
```

An equivalent description would be one where the value for the content property of the **Audio** component is set to the standard PCF URN urn:x-dvb-pcf::dvb-ctag64 (see annex Q).

R.3.4 From local file using a URL

In this example a file `my_special_beep.mpg` contains MPEG coded audio in an MPEG-2 elementary stream. The file is referenced by the **Stream** component with the content-type property set to `video/x-MP2ES` to indicate that the streamed media composition is provided as an MPEG-2 elementary stream, but at this point of undefined encoded content within. The subsequent declaration of the **Audio** component has the content-type property set to `audio/mpeg` to indicate that the contents of the MPEG-2 elementary stream is MPEG coded audio. Since this is the only elementary media stream inside this type of composition the use of the PCF default URN is sufficient.

```
-- source document: scene_defs.xml --

<PCF>
    <Scene name="some_other_scene">
        <Stream name="audio_clip">
            <StreamData name="content">
                <ExternalBody content-type="video/x-MP2ES" uri="my_special_beep.mpg"/>
            </StreamData>
            <Audio name="default_audio">
                <URI name="content" value="urn:x-dvb-pcf::default"/>
```

```
            </Audio>
        </Stream>
    </Scene>
</PCF>
```

R.4 Miscellaneous examples

R.4.1 Service item contains "boilerplate" visible components

In some services a number of visible components are common to most (or even all) scenes. Furthermore it may be necessary to preserve the state of such components between **Scene**s. To achieve this, components may be placed with a service item. A good example of this would be "boilerplate" components such as a logo or a video window.

```
-- source document: main.xml --

<PCF>
    <Service name="followme">
        <URI name="firstScene" value="#press"/>
        <String name="pcfSpecVersion" value="1.0"/>
        <Rectangle name="background">
            <Position name="origin" value="0 0"/>
            <Size name="size" value="720 576"/>
            <Color name="fillcolor" value=""/>
        </Rectangle>
        <Image name="logo">
            <Position name="origin" value="100 550"/>
            <Size name="size" value="100 70"/>
            <ImageData name="content">
                <ExternalBody content-type="image/png" uri="logo.png"/>
            </ImageData>
        </Image>
        <Scene name="press">
            <String name="instructions" value="Press Red"/>
            <URI name="next_scene" value="#../again"/>
            <StaticELC name="template" href="#template_def#template1"/>
        </Scene>
        <Scene name="again">
            <String name="instructions" value="Go on, do it again"/>
            <URI name="next_scene" value="#../press"/>
            <StaticELC name="template" href="#template_def#template1"/>
        </Scene>
    </Service>
</PCF>

-- source document: template_def.xml --

<PCF>
    <StaticELC name="template1">
        <TextBox name="text">
            <Position name="origin" value="100 200"/>
            <Size name="size" value="200 30"/>
            <String name="content" href="#../../instructions" context="derived"/>
        </TextBox>
        <OnEvent name="press_red">
            <Trigger eventtype="KeyEvent">
                <UserKey name="key" value="VK_COLORED_KEY_0"/>
            </Trigger>
            <SceneNavigate>
                <URI name="target" value="#../../next_scene"/>
            </SceneNavigate>
        </OnEvent>
    </StaticELC>
</PCF>
```

Annex S (informative): Bibliography

- ISO/IEC 9899 (1999): "The C Programming Language".
- Java Language Specification: "The Java Language Specification by James Gosling, Bill Joy and Guy Steele". ISBN 0-201-63451-1. http://java.sun.com/docs/books/jls/index.html
- ISO 8601 (2000): "Data elements and interchange formats - Information interchange - Representation of dates and times".
- ISO/IEC 9075 (1999 and 2003): "Information technology - database languages - SQL".
- ETSI TS 102 322: "Specification for a Lightweight Microbrowser for interactive tv applications, based on and compatible with WML".
- T.H. Nelson: "Literary Machines". Mindful Press, Sausalito, CA, 1982.
- NewsML 1.2: "News Markup Language". Details at http://www.newsml.org/
- ECMA-262 / ISO/IEC 16262: "ECMAScript Language Specification 3rd edition", December 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm
- ISO/IEC 19501: "Unified modelling language 2.0". http://www.omg.org/technology/documents/formal/uml.htm
- SMPTE 377M: MXF.
- AAF Association: "Advanced Authoring Format (AAF) Object Specification v1.1". April 2005. http://www.aafassociation.org/html/specs/aafobjectspec-v1.1.pdf

History

Document history		
V1.1.1	September 2006	Publication