Conch – Conformance checking for media files

Technical and Architectural Report

Project Acronym: PREFORMA

Grant Agreement number: 619568

Project Title: PREservation FORMATs for culture information/e-archives

Prepared by: MediaArea.net SARL

- Jérôme Martinez
- Dave Rice
- Tessa Fallon
- Ashley Blewer
- Erik Piil
- Guillaume Roques

Prepared for:

Date: February 28, 2015

Summary:

- Policy checker
- Reporter
- User interface
- Transport layer
  - Conch: File on disk or direct memory mapping
  - Plugin integration proof of concept: libcURL
- Container/Wrapper implementation checker
  - Conch: Matroska checker
  - Plugin integration proof of concept: mkvalidator
- Container/Wrapper Demultiplexing
  - Conch
  - Plugin integration proof of concept: FFmpeg
- Stream/Essence implementation checker
  - Conch:
  - Plugin integration proof of concept: jpylyzer
  - Plugin integration proof of concept: DV Analyzer
  - Optional
- Stream/Essence decoder
  - Conch
  - Plugin integration proof of concept: FFmpeg
  - Plugin integration proof of concept: OpenJPEG
- Container/Wrapper vs Stream/Essence Coherency Check
  - Conch
- Baseband Analyzer
  - Conch
  - Playback and Playback Analysis (through plugin)
  - Plugin integration proof of concept: QCTools
- Implementation Checker
  - Introduction
  - Design & Functional Requirements
- Policy Checker - Command Line Interface
  - Functional Overview
  - Design and Functional Requirements
- Policy Checker - Web Interface
- Metadata Fixer – Graphical User Interface
  - Introduction
  - Design & Functional Requirements
- Metadata Fixer – Command Line Interface
  - Functional Overview
- Metadata Fixer – Web Interface
- Reporter
  - Functional Overview

Introduction to Architecture

This serves as a roadmap for the technical components of the project, split into two categories: Global architecture and Checker architecture. The global architecture schema defines the context in which the PreFormaMediaInfo software is situated and gives a high-level understanding of the software. The Checker Architecture details the structural components of the implementation checker and metadata-grabbing module.

The implementation checker's goals are based on the following core principles:

Portability

The checker has the capability to be packaged and run as an executable on a computer running any common operating system. For this reason, the shell has plans to be integrated into three platforms: Command line, graphical user interface, and a web-based (server-client) platform. Qt was chosen as the core toolkit for the development of the graphical user interface because of its flexibility and ability to be deployed across many different operating system platforms.

The developed software will also have the capability to function as a micro-service application alongside other digital preservation systems defined by the OAIS Reference Model, such as Archivematica. With its micro-service approach, Archivematica serves as a wrapper for related task-specific software and is an open source system that also works to maintain standards based on providing access to digital collections. The implementation checker suits this systems' design as an integrated micro-service suite, allowing for it to run alongside other third-party software tools that also serve to process digital objects and help to standardize a preservation focused ingest-to-access workflow. Within this design, various micro-services, including the implementation checker, can be built together into customized workflows and deployed across all operating systems and platforms.

As the shell will be developed to integrate into and perform within a web-based platform, the web user interface (UI) will function across 3 major web-based browsers. These platforms are to include Firefox, Google Chrome, and Internet Explorer. The software's UI and capabilities will function similarly across all of these relevant web-based environments.

Scalability

Similar to the way in which MediaInfo can be built and expanded upon in archival institutions to perform media analysis at scale, the implementation checker can be integrated into scripts and systems that can validate files en masse and deliver computer-readable and human-readable metadata via standard XML reports.

The scalability of each built component of the implementation checker will allow the software to scale horizontally throughout multiple computers and servers in order to potentially meet a heavy and increased workload. The software can be deployed on more computers for added speed, dexterity and reliability, distributing and partitioning tasks to a cluster of machines in order to improve performance and increase the working capacity of the software. Built within this architecture, the implementation checker can be expanded upon and integrated into various workflow environments with the added ability to increase its number of file requests.

Distribution

The source code will reside on an open development platform (Github) in order to provide easy access and distribution during all stages of development. It will also have the ability to extract nightly builds of the code and deploy using continuous integration. The subsequent builds and software releases can then be downloaded, built and run on any system.

Modularity

MediaArea plans to collaborate with the other PREFORMA teams to support optimal interoperability with each other as well as with third-party developers of additional implementation checkers that will utilize the implementation checker shells. The checker's API allows for the checker to be integrated alongside other components and the future development of plug-in features. The architecture's modularity will also allow the software's features ongoing and improved maintenance as well as offering reliability as a freestanding shell.

The open source project will be programmed and provided in C++ but will be functionally compatible with other programming languages through the use of bindings. Other than C++, the team will support bindings to C, Python, Java, C# and other languages upon open request and feedback. The bindings will function as wrappers to allow the original code to be explicitly used and supported within other common languages. Supporting this capability will allow the API to be fully integrated within the open source community with the full potential of interoperability.

Deployment

The shell will be deployed on the PREFORMA website, as a stand alone shell, networked with other PREFORMA shells currently in development and used within test environments. Due to the levels of interoperability set up as an integral component of the implementation checker, it can be utilized within legacy systems as well as future systems. The shell will also be released to run on all relevant deployment platforms, implying that any user will be able to download and operate the most up-to-date version of the implementation checker.

Interoperability

MediaArea's API will allow for the full integration and interoperability between all software systems. The API will be developed to operate between the targeted systems while offering the ability to run and behave similarly across each, whether the user is operating within MS Windows, Mac OSX, or Linux distributions.

To enable growth within an open source environment, it is essential for the API and all corresponding data structures and software releases to be able to comply to and function within an interoperable architecture across different deployment platforms and software systems. System-to-system interoperability can be enhanced to meet user's potential needs, and the implementation checker's API will help implement an effective exchange and integration of the required data and information. This integration will suit other software and will work across whichever system the user is running the software. The implementation checker's shell will also operate within legacy and future systems.

Global Architecture

This includes the following:

- Technical specifications
- Open source conformance compliancy
- Relationships between frameworks
- Framework traversal patterns

This is broken down by category and the categories are as follows:

- Common Elements
- Core
- Database
- Scheduler
- New file daemon
- implementation checker and metadata grabbing module
- Policy Checker
- Reporter
- Shell(s) ** CLI (Command line interface) ** GUI (Graphical user interface) ** Web (Web based interface)

Checker Architecture

This includes the following:

- Technical specifications
- Relationships between each structural component
- Relationships between plugins

This is broken down by category and the categories are as follows:

- Transport interface
- Container/wrapper implementation
- Container/wrapper demuxer
- Stream/essence implementation

- Stream/essence decoder (through plugin)
- Stream/container coherency check
- Baseband analyzer (optional, through plugin)# PreForma Global Architecture

PreForma strives to offer access and ease to users with a structure that operates similarly and efficiently cross-platform and functions in both an online and offline capacity within different interfaces.

Architecture schema

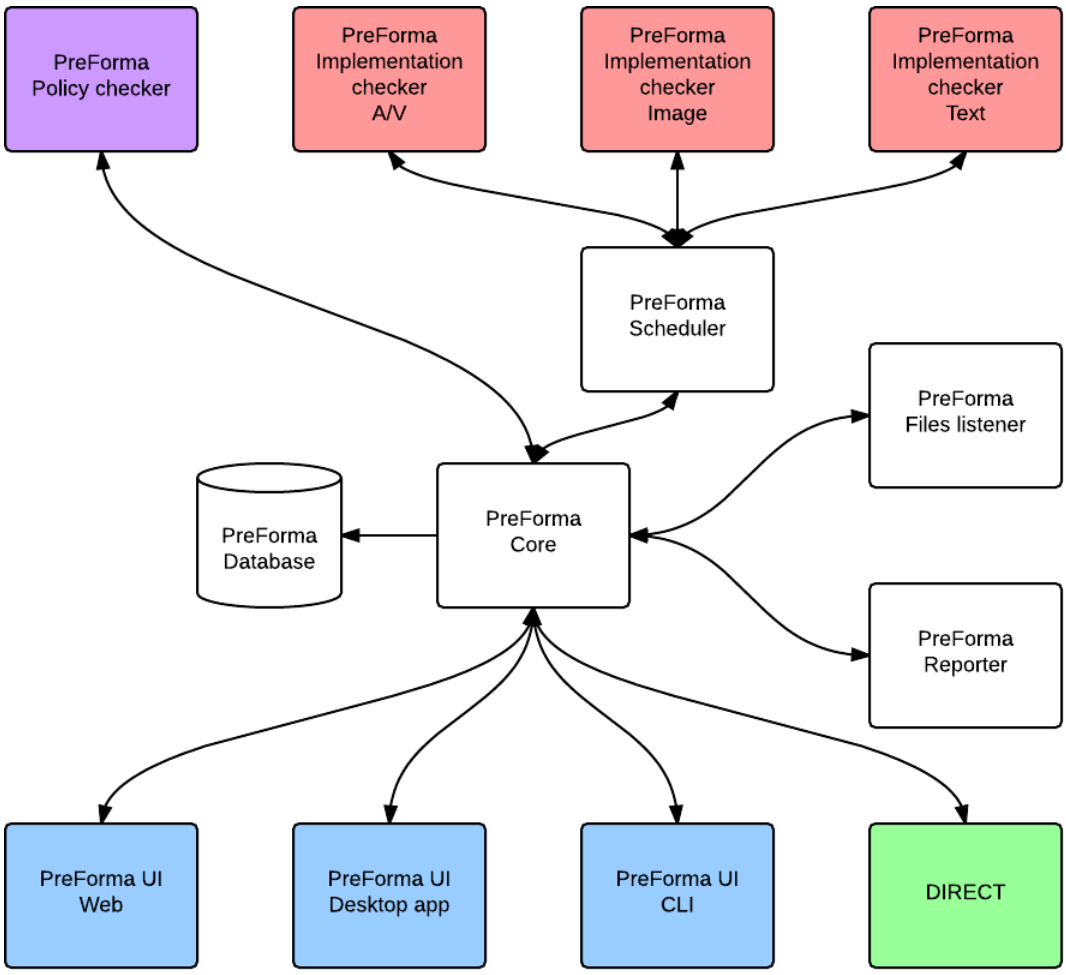

Figure 1: Global Architecture Schema

Common to all elements

All elements can be installed on the same server or on different servers. This will be dependent on the expected workload and anticipated points of access. The flexibility of the architecture allows for a single user to either use the shell on one computer or to use it on multiple computers, depending on scalability and need. A distributed system can be set up to allow the implementation checker to process large files more quickly than when working on a single machine. This architecture offers this ability with relative ease for the user. In addition to added speed, this build of scalability within the implementation checker increases its capacity to function reliably within various anticipated workflows.

File access

File access using CLI To place a file into the checking system, a user may execute the CLI name file manually or they can asynchronously run the file and test for its readiness. If the files are passed in a batch format, the reports can be later retrieved by manually requesting updates from the Core.

File access using GUI To place and use a file through the checker's GUI, both technical and non-technical users will work within windows and dialog boxes that enable them to navigate to, open, and load system files directly within the interface. A standard open dialog box will search for and load files for processing within the software. The GUI runs within a human-readable, interactive environment with visually intuitive access to the file checking process. In addition to allowing direct access to checking individual files, scheduled batch file checking via special folders residing on the user's computer/network can also be enabled directly within the GUI.

File access using Web To place and check files through the web-based browser platform, the process will be similar to working within the implementation checker's GUI. In this case, file loading will function visually, as with the GUI platform, but within the specific web browser's user interface (UI) and without the user having to directly download the software. Local network access will power the checker within the browser (Firefox, Chrome, or IE). Similar to the CLI and GUI, the web-based (server-client) platform allows for individual file loading and checking (via a visually intuitive designed interface) as well as enabling the capability of calling up batch-scheduled checks directly via the UI.

Common to all Files can be copied to a special folder somewhere on the user's computer or local network. This folder can then be set to regularly check for new files at defined intervals and process the files accordingly. This function can be set up within any of the three checker access methods (CLI, GUI, and Web UI).

File processing

The architecture allows for two modes of file processing: Direct access and asynchronous. Through enabling both processes, users can choose to have the checker handle individual files or have it process them in batches. Direct access allows for files to be processed one at a time, while asynchronous allows for files to be processed in batches and returned to at a later time.

Internet Access

The architecture does not require direct internet access. Users can work with any version (CLI, GUI, and Web UI) offline as long as it is installed on a user's machine (or distributed machines). In this case, for each deployment platform the version is installed and being used on, the user will be able to create an executable even when they are not connected to the internet. The web-browser UI can be run using a local network access.

Automation

PreForma includes the option of automated checking, allowing for users to receive notifications when new files are placed or come into the system.

A user can set up a system on the Core which will configure the system to run scheduled automatic checks for new files and batches. This system can be accessed and configured within the user interface to run on a timed and defined schedule.

Batching

A user also has the option to set up batch file checking and validating and schedule it to be sent through to the checker at any later point in time. These large tasks can be set up to perform according to a defined schedule. All versions of checker access (CLI, GUI, and Web UI) will have the capacity to enable this function, either through the command line or directly through the interface.

Prioritization

A user can also prioritize the checker to queue individual items and scheduled checks based on a defined priority level, with a lower priority placed on periodical checks. This function will be available through all versions of the checker (CLI, GUI, and Web UI). Priority levels for checks can be divided by High (for checks requested by user), Normal (for automated checks), Low (for periodical checks) requests.

PreForma allows users to load and edit configurations within a REST API configuration that will run via HTTP.

The Web-based system allows users to work both on and outside of the network as well as locally if they choose. If the user chooses to work locally, they will have access to the application server directly. This operation runs via a HTTP Daemon.

Users will have access to this web-based user interface for basic management. This will allow users to see the list of files, as well as the processes and checking happening to each of these files. Users will also be shown demarcation of which files pass testing successfully. The web user interface (UI) will function across 3 major web-based browsers (Firefox, Google Chrome, and Internet Explorer).

Core (Controller)

The Core serves as communication between all plugins within and outside of the Conch system and between all layers. The Core is the main service and runs in a passive, background mode. For example, if a user updates The Core, it will have no effect on the functionality of other systems. If a user begins using MySQL while running the implementation checker and decided to change to PostgreSQL, the Core could be adapted to address such a change. In essence, while components shift, the Core functions to present the data to all databases consistently and similarly and can adapt to different components.

The Core has several major functions:

- controls the checkers and manages data for the User Interface
- waits for commands from the Files listener and User Interface
- sends commands to the scheduler for file-checking
- launch periodical checks
- communicates with the database to store and retrieve data from the checkers
- sends data to DIRECT

The Core supports the following requirements:

- Scheduling
- Statistics
- Reporting
- User management

- Policies management

Interface :

- Scheduler : Advanced Message Queuing Protocol
- Policy checker / Files listener / User Interface / DIRECT : REST API
- Database : native driver

Programming language : C++

Database

The Database is responsible for storing the associated metadata and results of the implementation checker, including the policy checker rules and the user rights management. All specific technical metadata and conformance checking results for each type of file format are sent back to the Core before being stored within the database.

- store metadata and results of the implementation checker
- store the policy checker rules
- user rights management
- trace (optionally)

Interface :

- Core : native driver

Software :

As the main purpose of the software build is to store flat datas, it's more suitable to use a document oriented database (NoSQL). However, a more traditional relational database can also be used.

There are various potential database management system options, contingent upon the open source licensing requirements:

- Relational database : MySQL (GPLv2) / PostgreSQL (PostgreSQL License) / SQLite (Public domain)
- Non relational database : MongoDB (AGPLv3) / Elasticsearch (Apache license 2)

Scheduler

The Scheduler element is a form of software "middleware" that distributes the files to be checked across the implementation checkers by using a message broker interface. It translates the file data into one unified language for access within all aspects of the software. The Scheduler also controls priority levels for file checking.

- distributes files
- translates file data into unified language
- batch processing
- priority

The scheduler can take care of the priority function within the implementation checkers : * high : for checks requested by user * normal : for automated checks * low : for periodical checks

Interface :

- Core : Advanced Message Queuing Protocol
- Checkers : Advanced Message Queuing Protocol

Software :

RabbitMQ (MPL 1.1) / Gearman (BSD) / ZeroMQ (LGPL v3)

Files listener

The Files listener is a background process that listens for new files available for checking and validating. Automated checking, set up through the software, will notify when new files come into the system. Each time a new file is available, or if a file is modified, an event is sent to the Core which automatically requests a check.

Different solutions can be implemented depending on the file's storage and operating system. Implemented solutions can include an inotify notification system API for a Linux kernel or kqueue/kevent for a BSD kernel or files directory scanning.

- Automated checks

Programming language : C++

implementation checker and Metadata Grabbing Module

This module can utilize one or more checkers for each media type. As the implementation checker's process could potentially run for a long time, we use an asynchronous system based on a messaging system in order to not lock up the whole process. Metadata and conformance checking results for each file are sent back to the Core to be stored within the database.

- runs the conformance tests for the different types of media files
- grabs metadata (used for policy checking)

See Checker Architecture for more details.

Interface :

- Scheduler : Advanced Message Queuing Protocol

Programming language : C++ for MediaArea, depends on other participants for JPEG 2000, TIFF, PDF.

Policy checker

The policy checker serves to run tests on all metadata grabbed by the implementation checker and metadata grabbing module (the Checker). A vocabulary of technical metadata for each file format and media type will be created for the policy checker's functions.

- runs the policy tests for the different type of media files.

Interface :

- Core : REST API

Programming language : C++

Reporter

Within each of the developed user interfaces there will be ways to export raw metadata and human-readable JSON/XML reports after the conformance checking process. The reporter will define and express how a file's checked metadata corresponds to the validation result standards.

- exports a machine readable report, including preservation metadata for each file checked
- exports a report that allows external software agents to further process the file
- exports a human readable report
- exports a "fool-proof" report which also indicates what should be done to fix the non-conformances

The machine readable report will be produced using a standard XML format, implemented by all implementation checkers working within the PreForma ecosystem. This allows the reported module to combine output from multiple checker components into one report while also including sub-elements within the report that will address each conformity check. The report will be based on a standard output format that will be made by the consortium.

The human readable report summarizes the preservation status of a batch of files as a whole, reporting to a non-expert audience whether a file is compliant with the standard specifications of the format or institution while also addressing improvements in the creation/digitisation workflow process.

Interface :

- Core : REST API

Programming language :

- CLI : C++
- GUI : C++ / Qt (LGPLv3+)
- Web : PHP/Symfony (MIT)

User interface

The User interface (UI) is the shell component that allows direct interaction between users (or other systems) and the PreForma components:

- displays test results
- controls the Core
- allows metadata (descriptive and structural) to be edited
- edit configuration (periodical checks, policy checker, user rights)

Conch will provide three different options for a human interface in order to introduce maximum user interaction and flexibility within the implementation checker. These three interfaces are:

- CLI (Command line interface)

A command line interface will be functional on nearly all types of operating systems and platforms, including those with very little graphical interface support. CLI use allows for integration into a batch-mode processing workflow for analyzing files at scale. This interface is more intended for technical and expert users and for non-human interaction.

- GUI (Graphical user interface)

A Graphical user interface (GUI) will be developed and provided for both expert and non-expert users. The GUI, being based on Qt, has the strength of being versatile between operating systems and does not require additional development time to provide support for multiple platforms. The GUI can function similarly across all deployment platforms.

- Web UI (server/client)

An optional Web-based user interface (UI) will also be provided for both expert and non-expert users. In order to run this option, an internet access will not be needed or required. Local network access will power the checker within the user's chosen web-browser. The web interface will provide access to conformance checks without having to directly download and install the software.

Interface :

- Core : REST API

Programming language :

- CLI : C++
- GUI : C++ / Qt (LGPLv3+)
- Web : PHP/Symfony (MIT)# Checker Architectural Layers

The design of the implementation checker portion of the Conch application will be comprised of several layers which will communicate via a Core controller. The layers shall include:

- Transport interface
- Container/wrapper implementation
- Container/wrapper demuxer
- Stream/essence implementation
- Stream/essence decoder (optional, through plugin)
- Stream/container coherency check
- Baseband analyzer (optional, through plugin)

Transport layer

Conch: File on disk or direct memory mapping

Conch uses the native file API for each operating system to enable direct file access, including files that are still in the process or being written. The inclusion of MediaInfo also offers features for direct memory mapping which will be useful for third-party development or plugins.

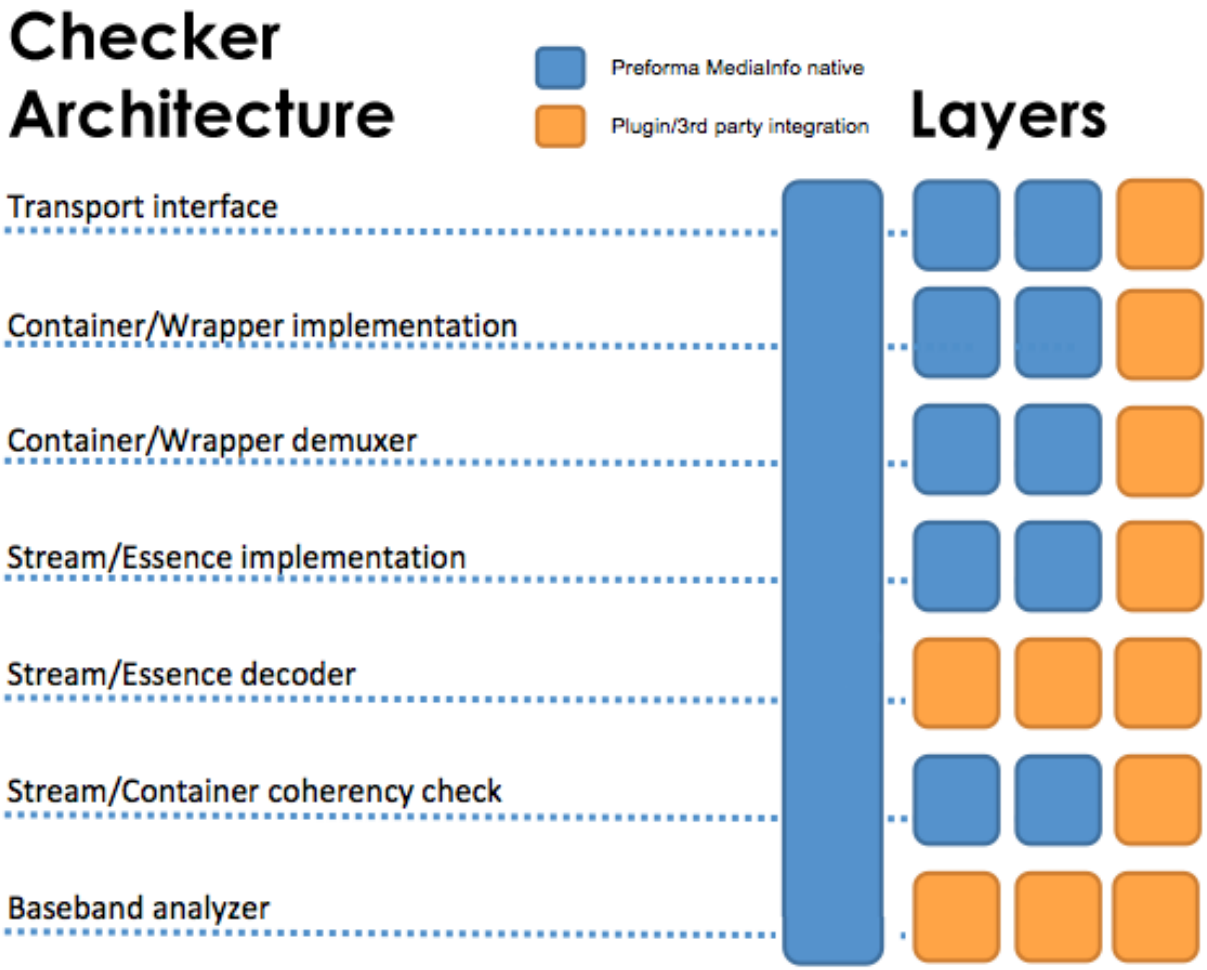

Figure 2: software architecture layers

Plugin integration proof of concept: libcURL

libcURL is licensed under an MIT license that is compatible with both GPLv3+ and MPLv2+. We can relicense to be under GPLv3+ and MPLv2+. curl offers extensive support for transferring data through many protocols. By incorporating curl into Conch the tool will be able to assess files that may be accessible online by providing a URL (or list of URLs) in place of a filepath.

Since we will be generating a library of reference and sample files that will include large audiovisual files, users will be able to assess reference files without necessarily needing to download them.

Used as a proof of concept of plugin integration: HTTP/HTTPS/FTP/FTPS support via MediaInfo open source GPLv3+/MPL2+ and libcurl (MIT license, compatible with GPLv3+/MPL2+)

Container/Wrapper implementation checker

Conch: Matroska checker

Plugin integration proof of concept: mkvalidator

mkvalidator is a basic and no more maintained Matroska checker (BSD license, compatible with GPLv3+/MPL2+) which will be used mostly for demonstration of the plugin integration.

Container/Wrapper Demultiplexing

Conch

Conch will utilize MediaInfo's existing demuxing libraries (can be relicensed under GPL....) which will allow for PreForma's selected video codecs, FFV1 and JPEG2000, to be assessed from within many formats founds within archives although these container formats themselves aren't the focus of the current PreForma project. Through discovery interviews with archives and vendors we have found FFV1's archival implementations to use a variety of container formats such as AVI and QuickTime as well as Matroska. In order to allow developed tools to support FFV1 even if not contained within Matroska, Conch will support the following formats for demuxing (though not necessarily for conformity (yet)):

- MXF (commonly found within memory institutions)
- MOV/MP4 (often found containing FFV1, JPEG2000, and LPCM)
- DV (video stream format which uses LPCM)
- AVI (used with FFV1 by DV Profession, NOA, Austria Mediathek)
- WAV (a common container for LPCM)
- WAVE64 (64-bit extensions of WAV for 2GB+ files)
- RF64 (64-bit extensions of WAV for 2GB+ files)

By supporting the demultiplexing of these formats through MediaInfo, the developed tools will be applicable to a wide variety of files that contain PreForma's selected codecs: FFV1, JPEG2000, and LPCM. This demultiplexing support can be available through MediaInfo's existing libraries in a manner that is compatible with PreForma's licensing requirements.

Plugin integration proof of concept: FFmpeg

FFmpeg is one of the most ubitiquous, comprehensive, and open tools for demultiplexing and decoding audiovisual data; however, although FFmpeg's GPLv2+ license is compatible with PreForma's selected

GPLv3+ license, it is not compatible with PreForma's other selected license, MPLv2+. As the PreForma conformance project evolves to support additional formats and codecs through plugins the use of FFmpeg's features are expected to becoming more and more appealing.

Although Conch won't incorporate FFmpeg in order to comply with the MPLv2+ licensing requirement, we would like to design plugin support for FFmpeg. In this way a memory institution using Conch could separately download FFmpeg and link the two together to enable additional tools such as:

Stream/Essence implementation checker

Plugin

Conch:

- FFV1
- PCM (including D-10 Audio, AES3)

Plugin integration proof of concept: jpylyzer

For JPEG 2000 (GPLv3+ license, compatible with GPLv3+ but not with MPL2+)

Plugin integration proof of concept: DV Analyzer

For DV (BSD license, compatible with GPLv3+ and MPL2+)

Optional

(Not part of the original PreForma tender but can potentially be added upon request after in context of professional services)

- MPEG-1/2 Video (including IMX, AS-07, D-10 Video, FIMS. . .)
- H.264/AVC (including AS-07)
- Dirac
- AC-3 (including AS-07)
- MPEG 1/2 Audio
- AAC
- Any other essence format on sponsor request (we have skills in DV, VC-1, VC-3, MPEG-4 Visual, H.263, H.265/HEVC, FLAC, Musepack, Wavepack, , BMP, DPX, EXR, JPEG, PNG, SubRip, WebVTT, N19/STL, TTML. . .)

Stream/Essence decoder

(Not part of the original PreForma tender but can potentially be added upon request after in context of professional services)

Conch

- PCM (including D-10 Audio, AES3)

Plugin integration proof of concept: FFmpeg

FFmpeg decoder (GPLv2+ license, compatible with GPLv3+ but not with MPL2+)

For instance the integration of FFmpeg can provide integration of very comprehensive decoding and demultiplexing support beyond what can be easily provided with MediaInfo's demuxing libraries. FFmpeg's libavfilter library also provides access to waveform monitoring, vectorscope, audio meters, and other essential audiovisual inspection tools.

- Video Waveform Monitor
- Vectorscope
- Ability to inspect luminance and chroma planes separately
- Audio Meters

We anticipate that the implementation of FFmpeg plugin support will substantially simplify the development of other plugins for broader codec and format support so that an entire decoder or demuxer does not need to be written from scratch in order to extend support.

Plugin integration proof of concept: OpenJPEG

OpenJPEG decoder (BSD license, compatible with GPLv3+/MPL2+)

Container/Wrapper vs Stream/Essence Coherency Check

Conch

Conch will support the coherency check between all suppoted formats (see Container/Wrapper implementation checker and Stream/Essence implementation checker parts)

Baseband Analyzer

Conch

- None (only creation of the API)

Playback and Playback Analysis (through plugin)

Note that the PreForma tender does not require decoding or subsequent baseband analysis or playback; however, from our experience in implementation checker design with DV Analyzer and QCTools and through discovery interviews, we've found that users are quick to require some form of playback in order to facilitate decision-making, response, and strategies for fixing. For instance if the implementation checker warns that the Matroska container and FFV1 codec note contradictory aspect ratios or a single FFV1 frame registers a CRC mismatch it is intuitive that the user would need to decode the video to determine which aspect ratio is correct or to assess the impact of the CRC mismatch. These layers can be supporting by designing a implementation checker and shell that is prepared to utilize FFmpeg as an optional plugin to enable additional analysis features and playback. Our overall proposal is not dependent on supporting an FFmpeg plugin but we believe that preparing a implementation checker that could support FFmpeg as an optional plugin could create a more intuitive, comprehensive, and informed user experience.

We propose incorporating several compatible utilities into Conch to extend functionality and add immediate convenience for users. Each component is built as a plugin and can be replaced by a third party tool.

Plugin integration proof of concept: QCTools

QCTools graphs (report on and graph data documenting video signal loss, flag errors in digitization, identify which errors and artifacts are in original format and which resulted from the digital transfer based on all the data collected in the past.) ## Implementation Checker

The implementation checker receives the incoming data from the file added to the system to be inspected and the desired specification chosen within the Policy Checker and verifies that they match or that the file's parameters fit within the range of the policy's rule set.

The results of this check is sent to the Reporter (via the Core), which then translates the errors into the expected output.

An API call is required for this component only, for use in batch processing.## Policy Checker - Graphical User Interface

Introduction

The policy checker graphical user interface will allow set of policy rules to be created and configured to specfic workflows and then applied to selected collections of files or file-queries across a file system. Although predefined presets are available, users shall be able to use the interface to create or edit policy check presets for reuse. A policy check preset will include a list will essentially be a recipe for evaluating the technical aspects of a set of files. Although MediaArea's implementation here will focus specifically on the selected formats of Matroska, FFV1, and LPCM, the policy checker will be able to preform checks on any type of file supported by MediaInfo.

MediaArea currently manages a registry of terminology to express a diverse set of technical metadata characteristics, including information about containers, streams, contents, and file attributes. Such information is derived from container demuxing and specialized bitstream analysis. The approach of MediaInfo's design is to trace through the entire structure of a file and interpret all data in a manner that categorizes it to the content of the utilized formats underlying specifications. This interpretation of audiovisual data is comprehensive and specification based, but to supported more generalized use within media workflows, the information is then aligned to a pre-defined set of technical metadata terminology.

MediaArea will extend its matroska demultiplexer to focus on the complete potential expression allowed by Matroska 3 finalized and 1 experimental version as well as common Matroska format profiles such as webM. Additionally MediaInfo will continue work on its FFV1 bitstream filter to efficiently analyze single frame, partial streams, or whole streams. Such work will distinguish between contextual and technical metadata unique to the 4 existing versions of ffv1.

Within the use of the policy checker the user shall be able to comprehensive and conditional policy checks to apply to one or many files. The policy checker may be used to assess vendor deliverables, consistency in digitization efforts, obsolescence or quality monitoring, in order to better control and manage digital preservationc collections.

Design & Functional Requirements

Standardizing Policy Expressions To faciliate interoperability between Preforma's developed Policy Checkers and the Shells of all suppliers, MediaArea proposes that all suppliers collaborate to define a common data expression for policy. The policy expression standard to cover a shared registry of technical metadata that shares overlapping scope amongst all supplier policy checkers (for example: policy expressions to test against file size or file name should express those technical concepts identically). Additional policies expressions from all Suppliers should share a concept list of operators such as 'must equal', 'greater than', 'matches regex', etc. A common policy expression shall also share methods to define conditional policy checks, for instance to conditionally test if the frame size is 720x576 is the frame rate is 25 fps and to test if the frame size is 720x486 if the frame rate is 30000/1001.

MediaArea recommends the policy expressions be standardizing with an XML Schema co-defined by all Suppliers, managed with version control in a common GitHub account, and accessible through the documentation of the Open Source Portal.

Policy Designer Interface The policy designer interface enables users to create, edit, and share sets of policies. The functions of the policy designer interface shall include allowing the user to:

- create, edit, and remove policy sets
- name and describe policy sets
- import, export, and validate XML files of policy sets
- allow new policies to be added, edited, or removed from policy sets
- synchronize a vocabulary between the terminology registered in MediaInfo and the user interface
- use a list of policy test operators as defined in the policy expression XSD
- allow policies to be encapsulate within over conditional tests with an if/else approach

The policy designer interface will inform the user with a general indicator as to how time-consuming the test if. For instance to test that all selected Matroska files are version 3 or higher may be done with a quick header parse, but to test that all FFV1 frames have valid embedded CRCs would take a lot more time. Although policy sets may provide an indicator as to the amount of time required (i.e. which sets are quick and which are intensive). This information may be used to set up task scheduling efficiently.

File Selection Interface The file selection interface shall allow the operator to load files through such methods as drag-and-drop, selection through system dialogs, or file system queries. Such sets of files or queries to identify files may then be saved, name, and described. Thus an operator may define a particular directory as an entry point for acquisitions, a queue for quality control assessment, or archival storage.

Policy Test Interface Within the Policy Test Interface, the user may associate policy sets and file selections to form a policy test. Policy tests may contain name and descriptions which will be passed through to the resulting report to provide context. The policy tests may then be run directly or assigned to the task scheduler of the Shell. Within the Policy Checker GUI the results of the policy tests may then viewed or analysis within the Reporter GUI.

The policy test interface will provide access to a log of policy tests, the context of each test, and any errors in running the policy test (such as the file selection being unavailable).

Policy Checker - Command Line Interface

Functional Overview

In the context of the Command Line Interface the Policy Checker shall allow for the Policy Set XML and the file or files to be tested to be inputs to the command line utility. The output of the utility will be a report compliant with PreForma reporting format for the expression of conformance, policy and metadata information. The command line will provide various levels of verbosity in order to show progress in processing through multiple files and allow the policy results to be shown as they are assessed.

The Policy Checker will support exit codes to programmically inform to the output of the process.

Design and Functional Requirements

Users of the command line interface will be prompted with several usage options to call specific project APIs, including the Reporter, implementation checker, Policy Checker, and Metadata Fixer. An additional call for "–Help" will prompt an extended options menu. The functions of the command line interface shall include allowing the user to:

- create, edit, and remove policy sets
- name and describe policy sets
- import, export, and validate XML files of policy sets
- allow new policies to be added, edited, or removed from policy sets
- allow policies to be encapsulated within over conditional tests with an if/else approach

Batch Policy Checking The command line interface allows for efficient batch policy checking through calls made to the API.

Policy Checker - Web Interface

Metadata Fixer – Graphical User Interface

Introduction

Although many audiovisual formats contain comprehensive support for metadata, archivists are eventually faced with a dilemma regarding its application. On one hand, a bit-by-bit preservation of the original data comprising the object is a significant objective. On the other hand, archivists prioritize having archival objects be as self-descriptive as possible. While the OAIS model aims to mitigate such a dilemma through the creation of distinct Information Packages, this is not often the case. For an institution managing digital files as objects for preservation, a change to the file's metadata is a change to the object itself; significant attributes such as file size and checksum are irreversibly altered. Because such revisions to the object prevent fixity functions, the decision on whether or not to fix or add metadata within the OAIS structure is oftentimes complex.

The intrinsic design of the Matroska file format aims to find a balance between these two considerations. Rather than relying on an external checksumming process to validate the fixity of the file, Matroska provides a mechanism for doing so internally. The CRC elements of Matroska may be used within any Element to document the subsequent data of the parent element's payload. With this feature a Matroska file may be edited in one particular section while the other sections maintain their ability to be easily validated. Thus in addition to (or possibly in lieu of) generating a file checksum during acquisition, a archive may use PreForma's Matroska tools to embed CRC elements if they do not already exist. When a Matroska file is internally protected by CRCs, the sections of the file may be edited or fixed while maintaining a function to verify the un-edited functions.

Because of these Matroska features, we are very interested in how archivists may work more actively with internal file metadata through various parts of the OAIS framework. For instance reports on file edits, repairs, and outcomes of preservation events may be added directly to the file. With such tools as proposed by this project, archivists and repository systems may work with living Matroska preservation objects which internally define the context and lifecycle of themselves over time while maintaining the fixity features of the contained audiovisual data which is the essence of the overall preservation focus and what the Matroska container may be used to describe, validate, and support.

Although the Metadata Fixer can provide comprehensive levels of control over metadata creation and editing, the central objective of the metadata fixer is to facilitate repair procedures for conformance or policy issues. The Tag validation status will be presented in detail or summarization dependent on the active layout and

related problematic aspects of the file with designed repair solutions. Because metadata fixes or repairs will alter a preservation file, MediaArea has dedicated a signficiant level of caution to the design of these operations. Learning more our similar experience with BWF MetaEdit, such designs will be based off a thorough programmatic understanding of the file, the actions to move the file towards a greater level of conformity, and the risks associated with doing so. The interfaces designed here will serve to intuitively relate file issues, with programatically proposed fixes, and inform to provide the user with an understand of the context and risk of the fix.

Design & Functional Requirements

File List Layout The GUI version of the metadata fixer will provide an interface to see a table of summarized metadata for one or many open files. The intent is to go allow files to be sorted by particular technical qualities or the content of embedded metadata. A table-based presentation will also allow the inconsistancies of technical metadata to be easily revealed and repaired.

MediaArea has developed such interfaces in other conformance- and metadata-focused projects such as BWF MetaEdit and QCTools and plans to use the File List Layout as an interface center for batch file metadata operations.

Customizable Sections The contents of the File List will be configurable according to the metadata values indexed by MediaInfo during a file parse. In the case of Matroska files these metadata values will also be categorized according to their enclosing Matroska section. These sections include:

```
- Header
- Meta Seek
- Segments
- Tracks
- Chapters
- Clusters
- Cueing Data
- Attachment
- Tagging
```

In additional to Matroska sections a category of file attribute data will also be provided to show information such as file size, file name, etc. Additional a 'global' section is provided to show summarization of the file's status and structure.

A toolbar in the File List Layout will enable the user to select one or many sections to allow for focus on a particular section.

As an example, checking to show the columns associated with the Matroska Header shall reveal columns such as:

```
- File format (Matroska, Webm, etc)
- Format version (version of Matroska, etc)
- Minimum read version
```

A global section would provide informational columns such as:

```
- Amount of VOID data with the Matroska file
- Percentage of CRC coverage with the Matroska file
- Number of metadata tags
- Number of chapters
- Number of attachments
```

As metadata tags may vary substantially, the tagging section of the File List Layout will show selected level 4 metadata tags as well as a column to summarize what level 4 metadata tags are unshown. Columns values which show level 4 metadata tags which contain child elements shall note visually when that tag contains child tags and reveal a summarization of child values over mouse-over. Further interaction with of metadata tags in level 5 and below can be better found in the (Metadata Editor Layout)[####mkv-metadata-editor-layout] which shall be linked from each row of the File List Layout. Within the tag setion of the of the File List Layout the shown Level 4 metadata tags may be edited directly.

The order and selection of viewed columns within the File List Layout may be saved and labelled to configure the display. This feature will allow users to design and configure layouts for particular metadata workflows. MediaArea plans to provide specific layouts in accordance with the objective of particularly OAIS functions, such as to supply contextual metadata about a digitization or acquisition event.

Managing State of Metadata Edits/Fixes A toggle within the toolbar will switch the table's editable entries from read-only to editable, to help prevent inadvertant edits. Each row of the File List Layout shall contain a visual status icon (File Edit State Icon) to depict the state of the file's metadata state. The File Edit State Icon will show if the file has been edited through the UI to different values than the file actually has; for instance, if the file must be saved before the shown changes are written back to the file. Metadata values within editable layouts shall appear in a different font, style or color depending on if they show what is actually stored or altered data that has not yet been saved back to the file. By selected a row which has an edited but unsaved state, the user shall be able to selet a toolbar option to revert the file's record back to its original saved state (to undo the unsaved edit).

Relational to Conformance / Policy Layouts The File List Layout shall contain a column to summarize conformance and policy issues with each open file and link back to the associated sections to reveal more information about these issues.

MKV Metadata Editor Layout The Metadata Editor Layout is designed to efficiency create, edit, of fix metadata on a file-by-file basis. The interface will show the contents of the Matroska tag section and provide various UI to facilitate guided metadata operations, such as providing a date and time interface to provide expresses for temporal fields, but also allowing text string expressions for all string tags as Matroska allows.

- Provide a table to show one row per metadata tag
- Provide columns with following values
  - Hierarchy
    - A relator to link tags to one another in parent/child relationships
    - A UI toggle to show or hide child metadata tags
  - Target Section
    - TargetTypeValue
    - TargetType
    - TargetSummarization of track, edition, chapter, and attachment targets
  - Metadata Section
    - TagName
    - TagLanguage
    - TagDefault (boolean)
    - TagContent (combination of TagString, TagStringFormatted, and TagBinary UI)
  - Tag Status Section
    - Tag validation status (alert on tags adherance to specification rules, logical positioning, and formatting recommendation)

Interface Notes Hierarchy

Each row of the metadata tag table may be freely dragged and dropped into a new position. Although this is usually semantically meaningless, the user should be able to organize the metadata tags into a preferred storage order. An example of this express in the UI could be that the Hierarchy column shall contain a positioner icon that the user may grab with the mouse to position the row in a different order. The positioner icon should also be able to be dragged left or right to affect the neighbor or child relationship to the metadata tag positioned above. For instance if there are two tags in the table at the same level called with the first called ARTIST and the second URL then both the ARTIST value and URL value refer to the declared target. However if the positioner icon of the URL tag row is moved to the right then the UI should indictate that the URL tag is now a child of the ARTIST tag, and thus the URL documents the URL of the ARTIST rather than the target.

Target Summarization

Each metadata tag may be associated with the context of the whole file or many specfic targets. For instance a DESCRIPTION may refer to one or many attachments or a particular chapter or a particular track, etc. In order to show the targets concisely the UI should present a coded summary to show one value that indicates the type and number of related target. The TargetSummarization may how "A3" to indicate that it refers to the third attachment, or "T4" to indicate that it refers to the fourth track. When the TargetSummarization is moused over a popup should reveal a list of associated targets with the UID and perteninet details of each target as well as a link to jump to a focus of that target within its corresponding layout (such as the Chapter Layout or Attachment Layout).

Tag Content Bahavior

Matroska tags may contain either a TagString or TagBinary element. When single-clicking or tabbing into a TagContent field then if the TagContent is a TagString it shall be directly editable and if the TagContent is a TagBinary the TagContentModalWindow shall appear selected to the Binary tab with a guided hex editor.

Tag Content Modal Window

The TagContent Modal Window is a UI designed to accommodate editing of TagContent or TagBinary values. The UI shall contain three tabs:

- String Editor
- Formatted String Editor
- Hexadecimal Editor

When creating a new metadata tag in a matroska file. If the TagName corresponds to a binary type it will open the Tag Content Modal Window to allow to the binary data to be provided, else it will default to allowing the metadata tag value to be edited within the string box of the layout in which the metadata tag was created. If the tag name of the newly created metadata tag corresponds to a binary type then the Hex Editor tab of the Tag Content Modal Window will be used.

The Hex Editor tab of the Tag Content Modal Window will allow for hexadecimal editing, allow data to be loaded to TagBinary from a selected file, or saved out to a new file.

When doubling clicking on an existing metadata tag in an editing or file list layout the Tag Content Modal Window shall open to reveal the most appropriate editing tab. If TagBinary is used then the Modal Window shall open to the Hex Editor tab. If TagString is used than it should use the Formatted String Editor tab if the data complies with the formatting rules, else use the String Editor.

Validation Status

The validation status indicators and associated procedures are central to the objectives of the Metadata Fixer. The Matroska specifications is rich with precise formatting rules and recommendations that are intended to facilitate the predictability and inter-operability of the file format; however, many Matroska tools and workflows make it easy to inadvertantly violate the specifications or cause conformance issues. The Metadata

Fixer layouts will provide a visual indicator of validation status issues, so that when files are opened any validation issues are clearly show in relation to the invalid section and linked to appropriate documentation to contextual the issue. Additionally is an operator makes a modification that is consider a validation issues, the user will be informed to this issue during the edit and before the save. If the users tries to save metadata edits back to a Matroska file while their metadata edit contains validation issues, the user must confirm that this is intended and that the result will be invalid.

In many cases repairs to well defined validation issues are repairable programmatically. The Validation Status section of layouts will show related repair procedures (if defined) and summarize (to the extent feasible) the before-and-after effects on the file.

Metadata Import / Export Both the GUI and CLI of the Metadata fixer will allow Matroska metadata tags to be imported into or exported from a Matroska file using Matroska existing XML tagging form. In addition to information typically found in Matoroska's XML tag format, information on validation status will be included.

Layout Preferences

- Checkboxes to disable appearance of columns in File List Layout
- Functions to allow the currently selected File List Layout options to be saved and labelled
- An ability to load pre-designed or user-created File List Layout options
- A list to specify level 4 Matroska tags to appear in File List Layout as a column
- Default value to use for the default value of TagLanguage on new metadata tags

Metadata Fixer – Command Line Interface

Functional Overview

The Matroska Metadata Fixer command line interface will provide repository systems with a means to automatically assess what potential fixes may be performed, selectively perform them, add or changes files metadata, or preform structural changes to the file.

Overall all the features documented in the Metadata Fixer GUI are also feasible within the CLI, although some scripting may be necessary around the CLI to emulate a fully programmatic performance of all anticipated GUI workflows.

Functional Requirements include:

- accept one or many Matorska files as well as one or many PreForma policy speciications (via xml) as an input
- generate a text based representation of the EBML structure in json or xml, which identifies and categories EBML sections and which includes attributes to associate sections of the EBML structure (or the file itself) with registered conformance or policy errors
- validate a Matroska file against conformance or policy errors and generate a text based output which summarizes errors with associated fixes
- preform and log identified metadata fixes
- add, replace, or remove Matroska metadata values based on a developed EBML equivalent of XPath

Metadata Fixer – Web Interface

Reporter

Functional Overview

The Reporter presents human and machine-readable information related to metadata readouts of individual files, policy checking errors, conformance checking errors, and metadata fixing documentation. This information is derived from multiple APIs passed through the PreForma core and finally combined and transformed into a desired output.

Reports may include an option to report verbose bit traces of individual files, as well as information on preventative measures for nonconformed files. Batch reporting features the option to nest specific objects.

Design and Functional Requirements

Reporter - Graphical User Interface

The General Reporter Interface displays human and machine-readable data to the end user. Through a drop-down menu, a user can select the following out formats:

Human-readable formats: PDF, TXT Machine-readable formats: XML, JSON Optional machine-readable formats: CSV/TSV

On the command line interface, a user would be able to export a report using flags designating output (eg., "-o") and output format (eg., "-of").

Several of these formats will allow for external software agents to futher process this information. This information can be saved in batches and later consulted and provide a historical context for each file.# Style Guide

Source Code Guide

Portability

Excelling in cross-platform open source development, MediaArea will utilize tools throughout the development phase in order to provide users with a downloadable source code that offers functional portability between the different deployment platforms (MS Windows 7, Mac OSX, and Linux).

The source code can be shared and used between the targeted platforms and will run and behave similarly across different users' machines. The various releases will be implemented with adaptable interoperability between these platforms and the software will run dependent on which downloadable source code and executable the user's platform requires. This means MediaArea's open source project, throughout the development and deployment phases, will continue to always provide downloadable access to software and support for each of these individual platforms.

Modularity

MediaArea's regularly released source codes will be developed within a modularized architecture in order to create an unrestricted atmosphere for the improvement of the project's maintainability and assembly. Conforming to this development technique and creating distinct, interchangeable modules will allow the software, and its corresponding open source community, to remain sustainable in its growth and facilitate ongoing collaborative feedback throughout the development phase. Each module will have a documented interface (API) that defines its function and interactive nature within the software.

The construct and eventual structure of modularity within this project will be key to the health and sustainability of the project's potential success as a fully integrated implementation checker. The regular release of source code for this project, built within this architecture, will enable better feedback and issue tracking from both users and memory institutions utilizing the software.

Deployment

MediaArea will develop a implementation checker that is designed to allow for deployment in the five following types of infrastructures and environments:

- PreForma's website
- Within an evaluation framework
- Within a stand-alone system (MS Windows 7, Mac OSX, and Linux)
- Within a network-based system (server or cloud)
- Within various legacy systems

Access release for the intended targeted users will be supported within all of these environments. In order to demonstrate the project's successful deployment within these infrastructures, MediaArea will, in addition to supplying the standard corresponding technical documentation for each, undertake the following considerations:

The project's necessary PreForma website, including centralized links to all open source materials and community outreach, will be considered as the official deliverable for the entire project. The associated and required documentation, tools and instructional feedback will also be provided and accessible on the PreForma project website as well as the open source platform.

The implementation checker will also fulfill the requirement of being deployed within the direct infrastructure to facilitate evaluation and use within the PCP system.

For stand-alone users or smaller institutions, the implementation checker will be fitted with the capability to be packaged, downloaded and run as an executable on machine's running any form of a standard operating system (MS Windows 7, Mac OSX, and Linux).

The implementation checker will be developed to deploy within different network-based solutions and environments (including dedicated servers and cloud solutions) hosting the memory institutions' digital repositories.

Via written API integration, the implementation checker will also be able to properly function when plugged into various legacy systems.

APIs

Via APIs, the developed and deployed implementation checker will be designed to interface and integrate with other software systems. Programming tools and software standards and practices will be upheld in order to allow for a potential software-to-software interface. A long-term sustainable usage and presence within the open source community will be further enhanced via correct implementation of this successful API integration.

The outcome of the ability to interface with other software systems is that the software and technical documentation will have ongoing support and integration within the individual workflows of the memory institutions' preservation plans.

Open Source Practices

Development

MediaArea's open source software development within the PreForma project will establish and uphold open source work practices and standards. These practices will abide by the following project rules:

- Use of nightly builds: A nightly build is an automated build that reflects the most up-to-date version of developed software's source code. Users will have access to these nightly builds as their release will allow for collaborative groups of developers and users to work together and continuously gain immediate feedback and fixes to the most current state of the software. With access to the absolute latest versions, MediaArea and all open source collaborators will more readily gain insight into potential bugs and issues that could arise during the development phase. Programmers will be able to determine if they "broke the build", making the software inoperable with their latest code. Immediate access to fixing these issues can be made more efficiently.
- Use of software configuration management systems: Operating with a software configuration management system (Git) will allow MediaArea easy version control as well as knowledge of the revisions needed. This is an essential part of the open source community that allows developers to be able to work together collaboratively. A version control system allows multiple people to work on the same or similar sections of the source code base, simultaneously and at the same time, with awareness and prevention of overlapping or conflicting work. Git will be used as the software configuration management system for this project.
- Use of an open platform for open development: MediaArea will operate within an open source platform on which to develop software and better facilitate the open development of that software. Public visibility that allows anyone the ability to contribute to the software's development allows for sturdier, more reliable outcomes. Feedback is more easily sought and more readily provided with the use of an open platform. Github will be used as the open platform for open development of this project.

Open Source Platforms

All software development as well as the development of all relevant and corresponding digital assets and tools created by MediaArea during the PreForma project will exist and function as an open source project within an open source platform (Github). This open source development platform will offer full transparency and traceability throughout the development phase and facilitate a functional collaborative environment with developers, users, stakeholders, and institutions.

Source code, issue tracking, documentation, updates, release and various forms and channels of public outreach will be centralized within the open development platform and linked to within the PreForma project page.

Contribution Guide

File Naming Conventions

All project files related to documentation regarding the Preforma project will be named using CamelCase. Project documentation's actual sample data will be shared using the snake_case. These objects should carry a suitably descriptive file name that elaborates on the contents of the file and follow the standard practices and expectations of their corresponding naming conventions and specifications. File naming conventions and rules will be upheld in order to implement an efficient database of document and file releases within the open source community. In regards to the required conventions for commit messages on the open source platform, all messages should be concise and clear and effectively summarize each contribution to the project. If more than one change was made, users should not create one commit message to cover all feedback and changes.

New individual commits should be made to cover each individual change made to the relevant file being altered. Effective commit messages, covering context of a change, will enable MediaArea to work within a speedier, more efficient review process and better alter development around this feedback.

Rules for Qt/C++ code

MediaArea's open source project will be programmed in C++ and will use the Qt application framework.

Guideline for Qt is as follows:

MediaArea will follow the applicable rules for programming within the Qt cross-platform application development framework.

Attention to detail will be given to the following rules/guidelines:

Indentation: - Four spaces to should be given for indentation (not tabs)

Variables: - Each variable should be declared on separate lines, only at the moment they are needed - Avoid short names, abbreviations and single character names (only used for counters and temporaries) - Follow the case conventions for naming

Whitespaces: - Use only one blank line and use when grouping statements as suited. Do not put multiple statements onto one line. - Also use a new line for the body of a control flow statement - Follow the specific single space conventions when needed

Braces: - Attached braces should be used (follow guidelines for rules and exceptions) - Curly braces are used only when the body of a conditional statement contains 1+ line or when body of a conditional statement is empty (follow guidelines for rules and exceptions)

Parenthesis: - Parenthesis should be used to group expressions

Switch Statements and Jump Statements: - Case labels are in the same column as the switch - Each case should have a break statement at the end or a comment to indicate there is no intentional break - Do not use 'else' at the end of Jump Statements unless for symmetry purposes

Line Breaks: - Lines should kept under 100 characters - Wrap text if necessary - Use commas at the end of wrapped text and operators at the beginning of new lines

Exceptions: - Always try to achieve functional, consistent and readable code. When code does not look good, exceptions to the rules may pertain to fixing this situation.

For more specific rules, examples, exceptions and guidelines, please refer to the Qt Coding Style guide: http://qt-project.org/wiki/Qt_Coding_Style

Guidelines for C++ code is as follows:

Manageability and productivity within the C++ coding atmosphere will be preserved by upholding to the Style and Consistency rules necessary for creating a readable and controlled code base. Attention to detail will be given to the rules governing the creation of a workable open source code in the following areas:

- Headers
- Scoping
- Classes
- Naming
- Comments
- Formatting
- Specific Features/Abilities of C++
- Relevant Exceptions

For a detailed account of specific rules, examples and guidelines for each section, please refer to the Google guide on C++: http://google-styleguide.googlecode.com/svn/trunk/cppguide.html

Rules for contributing code

Contributions of code or additions to MediaArea's Preforma project documentation must be written with Qt (following the advised standards and practices) and must be made in the form of a branch submitted as a pull request.

- Create your feature branch (git checkout -b my-new-feature)
- Commit your changes (git commit -am 'adds some feature')
- Push to the branch (git push origin my-new-feature)
- Create a new Pull Request with a more verbose description of the proposed changes

Link to github repository: (https://github.com/MediaArea/PreFormaMediaInfo/fork)

Rules for contributing feedback

Feedback of all kind is encouraged and can either be made through opening an issue on Github or by contacting the team directly at info@mediaarea.net.

Issue tracking and feedback will be encouraged directly through the open source platform (Github) around which, in addition to PreForma's Open Source Portal, MediaArea will function and centralize the anticipated infrastructure for a collaborative community environments. In addition, contributions and feedback can be left via either the IRC channel or the mailing lists pertaining to the project.

Linking

MediaArea will implement linking throughout the open source community in order to create a sustainable and documented infrastructure that facilitates clarity in the progression of the project. In order to produce an environment that offers both the users and MediaArea a space for descriptive feedback, intuitive discoveries within the code, and the ability to resolve issues, linking will function through the released source code, the corresponding software documentation, a ticketing system, general feedback, and potential commit messages. In one such example, as the registry itemizes user's individual conformance checks and tests, these tests will subsequently link to code blocks and commits so that the software can continue to be developed and associated to that conformance check.

If this habit is implemented efficiently, MediaArea will create an open source community that enables ease in interacting and reviewing with both user-friendly and human-readable descriptions of conformity checks combined with their related programmatic results.

Test Files

MediaArea's test files and media will exist in the SampleTestFiles folder within the open source platform. This designated sample folder will also be broken down into separate folders for each relevant file format and the separate specification parameters set for testing.

It is anticipated that a large library of reference media and test files will be created to highlight the different outcomes associated with issues and errors that may arise in regards to certain files and specifications push through the software. The test files will either be self-created, solicited, or pulled from a variety of online reference libraries.

This curated selection of tests will include the following:

- files that conform to the relevant file format's technical specifications
- files that do not conform and therefore deliberately deviate from the file format's technical specifications (in association with the appropriately coded error messages)
- specific files that originate from and/or adhere to the technical specifications of the file formats from participating memory institutions (including examples that both conform to and deviate from the requirements)

Because it is crucial to the stimulation of a sustained and well documented open sourced community, the resulting issues and feedback from the testing of these created and solicited files will also be documented and will contain information on the relevant version of software used for the test.

Release Schedule

MediaArea intends to release various versions of all relevant source codes and executables for each of the deployment platforms that the project will be configured to perform upon successfully. For stable versions of the software, new downloads and rolling releases will be provided and made available on a monthly basis. Stable versions will take into account software fixes, updates, and bug reports throughout the development phase and additionally will have gone through a QA process during that time.

Certain deployed (LTS) versions, upholding the build of the stable versions, will be provided and released during the required delivery stages of the PreForma project and will be developed as sustainable for a long period of time within the open source project.

New nightly builds and updates of the source code will also be made available to download during all stages. This ensures that all users and organizations will have access to downloading the most up-to-date version of code that exists throughout the project.

Downloads will be made available through a public repository with a functioning issue tracker (Github). In conjunction with the releases, a roadmap will be created in order to track these updates publically and encourage open collaborative usage and issue feedback. Both the older and more recent development, stable, and deployed (LTS) versions will be made available to users of any level, throughout these multiple platforms, for the entirety of the project. If a user wishes to download an older version of the source code or executable, MediaArea will have this option available.

All source codes and updates will be made accessible on the following platforms:

- MS Windows
- Mac OSX
- Linux (Ubuntu, Fedora, Debian, and Suse)

In regards to the nightly source code builds and monthly stable version releases, MediaArea will facilitate easy access for users to download each of these different versions with the creation and upkeep of a single file that contains all of the necessary open source tools. Taking into account the varying deployment platforms, a different file (containing all relevant documents) will be created for each. The expected and differing standard procedures and patterns between the different platforms, as well as their individual configurability rules, will be upheld throughout the development and release phases. Support will be accessible via all of these platforms.

Along with the downloadable codes and tools, full supplementary documentation, developed to suit users functioning on each platform, will be included within the release schedule and will stay up-to-date with those releases. The necessary steps required in downloading and extracting MediaArea's source code and software build, in order to created a directly executable object on the user's machine, will be fully documented for every type of user. We intend on establishing and releasing informational documentation including detailed, step-by-step instructions for both a non-technical and highly technical user or institution.

License

All software releases and digital assets delivered by MediaArea will be produced and made available under the following Intellectual Property Rights (IPR) conditions:

- All software developed by MediaArea during the PreForma project will be provided under the following two open source licenses:
  - GNU General Public License 3.0 (GPLv3 or later)
  - Mozilla Public License (MPLv2 or later)
- All source code for all software developed by MediaArea during the PreForma project will always be identical and functional between these two specific open source licenses ("GPLv3 or later" and "MPLv2 or later").
- All open source digital assets for the software developed by MediaArea during the Preforma project will be made available under the open access license: Creative Commons license attribution – Sharealike 4.0 International (CC BY-SA v4.0). All assets will exist in open file formats within an open platform (an open standard as defined in the European Interoperability Framework for Pan-European eGovernment Service (version 1.0 2004)).